UNIVERSITY OF CALIFORNIA,
IRVINE

Towards a Library for Deterministic Failure Testing of Distributed Systems

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Armin Balalaie

Thesis Committee:
Associate Professor James A. Jones, Chair
Professor Cristina V. Lopes
Professor Michael J. Carey

2020

DEDICATION

To Sara, my love, without whom and her patience, this wasn't possible ..

and

My mom without her help, dedication and sacrifices, I wouldn't be where I am today ..

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF LISTINGS

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor James A. Jones, for his guidance and patience with me during my graduate study. Thank you Jim for giving me the freedom to work on the topics that I was passionate about and thanks for always being a good listener and a positive person.

I would also like to thank my committee members, Professor Crista V. Lopes and Professor Michael J. Carey for taking the time to review my thesis and for their help and guidance to make this happen.

ABSTRACT OF THE THESIS

Towards a Library for Deterministic Failure Testing of Distributed Systems

By

Armin Balalaie

Master of Science in Software Engineering

University of California, Irvine, 2020

Associate Professor James A. Jones, Chair

Distributed systems are widespread today, and they are being used to serve millions of customers and process huge amounts of data. These systems run on commodity hardware and in an environment with many uncertainties, e.g., partial network failures and race condition between nodes. Testing distributed systems requires new test libraries that take into account these uncertainties and can reproduce scenarios with specific timing constraints in a programming-language-agnostic way. To this end, we present Failify, a cross-platform, programming-language-agnostic and deterministic failure testing library for distributed systems, which can be seamlessly integrated into different build systems. Failify, as an infrastructure, can also facilitate research in testing distributed systems in various ways. We evaluated the compactness of Failify's deployment API with six open-source distributed systems. Our results indicate that, in average, the most reliable deployment architecture for these systems can be defined in less that 17 lines of code. We also use re-wrote an HDFS test case to demonstrate potential scenarios where Failify's deterministic environmental manipulation and failure injection API can be effective.

Chapter 1

Introduction

Today's customer-facing software companies are widely using distributed systems to serve their customers and process their data. These systems run on hundreds of machines and use complex algorithms to coordinate among themselves. Separating the business logic of an application into different processes residing in different nodes requires an upfront design for fault tolerance. One of the main sources of faults in distributed systems is the environmental uncertainties, specially the fact that the underlying network is unreliable [44]. Despite the vast amount of literature on designing fault tolerant distributed systems, cloud outages are still a fact [33], and distributed systems still can malfunction, lose data and be inconsistent in certain scenarios and during environmental failures [9]. Failures of this kind can impact availability of the system, and consequently, lead to customer dissatisfaction and financial loss for the companies. One of the important reasons behind this phenomenon is the fact that most of these systems only contain automated test cases, which at best test the system in a clustered environment. However, they are rarely being tested automatically and deterministically against environmental failures, e.g., network partitions. In particular their failure recovery code needs more rigorous failure testing [33, 32, 26, 27].

A deeper look into distributed systems can better reveal the problem. Distributed systems provide higher guarantees in terms of availability and scalability using redundancy in the system. When moving from a single process to a distributed system (multiple processes running on different nodes), the communication mechanism will be changed from simple method or function calls to message passing over computer networks, and different parts of the system can reside in different physical machines. However, some of the environmental assumptions change in this new setting, namely the underlying network, is unreliable and the system should deal with non-deterministic network delays and partial network failures; individual nodes and processes can fail independently, but the system should work properly as a whole; there is no global clock shared between the nodes which affects the way the system orders messages; and since the system is formed as a set of independent processes, there is a good chance that it has to deal with concurrency problems but in a distributed context and under the mentioned environmental assumptions [28]. These new properties in distributed systems that were not present in previous non-distributed ones, call for new test frameworks that take into account these environmental assumptions.

Existing approaches for testing distributed systems are either focused only on the deployment of the system in the test environment [23, 3, 40], focused on a specific type of failure (e.g. network partition) [24, 31], systematically inject failures [35, 32], use randomness to inject failures [9, 15], or use model-based approaches to explore the state-space of failure injection as much as possible [36, 43, 37, 42, 41, 39]. Most of the approaches [23, 3, 40, 34, 15] do not support required failures in a distributed environment like network partition, only Jepsen [9] supports clock drift, and [35, 32] use instrumentation to inject failures, which makes them unsuitable for using with different programming languages. Further, though good for bug identification, randomized failure injection can lead to flaky test cases. Systematic failure injection and model-based approaches can also help with bug identification and may be an ideal choice for before-release comprehensive testing, but they are not time-efficient and not good for regular development regression testing. None of these tools can

be used to develop cross-platform, programming-language-agnostic, and deterministic test cases (including scenarios with specific timing constraints) with failure injections for testing implemented distributed algorithms and failure recovery code in a time-efficient way so developers can run them regularly after each change.

Apart from the mentioned approaches, the state-of-the-practice is that different distributed systems have their own distributed testing framework. Cassandra [2], Hadoop [6] and HBase [7] have dtest, Hadoop MiniCluster and HBase MiniCluster frameworks, respectively, for distributed testing. They simply try to be as similar as possible to current testing frameworks like JUnit [10] for Java and Nose [16] for Python, and at the same time, reduce the burden of the deployment of the system in a cluster and asserting certain states and behaviors for the developers. Despite the fact that all the mentioned frameworks have common portions, each of them are implementing a solution that is tightly coupled to the system they are testing. Basically they are reinventing a wheel, which cannot be reused by the others. Further, they rarely have any support to deterministically inject environmental failures.

In an attempt to overcome the mentioned limitations, in this thesis, we present Failify; a cross-platform, programming-language-agnostic, and deterministic failure testing library for distributed systems where it is possible to develop non-flaky test-cases even for scenarios with specific timing constraints (e.g. when race condition between nodes is possible). The main contributions of this work are:

- We designed and implemented a compact and programmable DSL to define the deployment architecture of a distributed system.
- We designed and implemented an environmental manipulation and failure injection API to control a deployed environment during a test case execution.
- We designed and implemented a deterministic, programmable, and programming-language-agnostic scenario reproduction engine, which can reproduce scenarios with

specific timing constraints.

- We evaluated Failify with 6 different distributed systems written in different programming languages and from diverse categories and developed deployment definitions to show that its deployment API is compact and supports a wide variety of distributed systems. Our results indicate that, in average, the most reliable deployment architecture for these systems can be defined in less that 17 lines of code.

- We re-wrote an already existing test case for the lease recovery process in HDFS to demonstrate the potential scenarios that Failify deterministic environmental manipulation and failure injection API can be effective.

Also, partial results of this work has been published in the proceedings of ACM Symposium on Cloud Computing 2019 (SoCC'19) [25]

Chapter 2

Existing Work

In this chapter, we go over existing work in the area of failure testing of distributed systems. We discuss various existing test frameworks and approaches for testing distributed systems.

2.1 Deployment-Focused Testing Frameworks

In recent years, a set of testing frameworks for distributed systems has emerged which focuses mostly on the deployment of distributed systems for the purpose of end-to-end testing without paying much attention to failure testing.

Ducktape [3], developed by Confluent Inc., is a test framework that facilitates setting up and tearing down required nodes for a distributed test case in different environments (e.g., local, Vagrant, cluster), but only supports node failure injection. Zopkio [23], a testing framework for distributed systems developed by Linkedin, provides similar capabilities that Ducktape offers, but it does not support any kind of failure injection and is more focused on performance testing. Dfuntest [40] is another tool, which is only focused on the deployment of the system.

This thesis extends deployment-focused testing frameworks with the ability to inject different kind of failures during the test case execution.

2.2 Random Failure Injection

A lot of attention has been paid to random failure injection for testing distributed systems in industry. This approach is popular mostly because it can potentially reveal hidden and unknown bugs in the system.

Jepsen [9] is a framework for testing mostly the consistency claims of distributed systems and in particular distributed data stores. To use this tool, it is needed to define the setup and tear down process the system, and how it can invoke read, write and compare and set (cas) operations. Then, Jepsen will provide the user with an input generator which can mix read, write and cas operations. After starting the distributed system and a bunch of clients, a model will record all of the operations start and end time. Then, a checker will be used to check the model against a consistency model; e.g. linearizability. It is also possible to define a so called nemesis to simulate different kinds of failures in the system during the execution of input workload; e.g. a network partition, network delay and clock drifts.

An advantage of Jepsen is that it recognizes the client as part of the distributed system which can be necessary for the consistency analysis. Jepsen models the system in a way that is tied to the database domain, and as such, may not be generalizable to all kinds of distributed systems. It removes the burden of creating a workload for testing by generating it. Also, Jepsen injects failures in a non-deterministic way, and due to this fact, test cases may not always fail and the result would be flaky test cases, which may not be suitable to be included in an automated test suite. Jepsen can be seen more as a bug identification framework than a testing framework.

Namazu [15] (formerly named Earthquake) is a programmable fuzzy scheduler for testing real implementations of distributed system. It has a set of inspectors where most of them simulate an environmental resource like disk or network and generate and send an event to an orchestrator. It also has inspectors for instrumentation of Java programs to send out enter into and exit from methods events. There is a queue in the orchestrator which has an explore policy. Based on the policy the order of the events in the queue can be changed. Each inspector will do a blocking poll after sending an event in order to receive an action from the orchestrator. There is a chance that the returning action from the orchestrator is an fault action which can cause failures in the environment. The overall process consists of init, run, validate and clean steps. All of the setting up and running of processes for the distributed systems should be handled by the test developer inside the init and run steps. Also the validation process should be handled by the developer and can either exit normally or with an error which is the indicator of the test result.

Namazu is similar to Jepsen in the sense that it also injects failures randomly and is non-deterministic. It can be used to reveal hidden bugs in the system, and it may not always reveal the hidden bugs in the first pass. So, it may be needed to run the test multiple times to potentially find a bug.

This thesis proposes a more deterministic approach in testing distributed systems, and it is more suited for regression testing and testing common failure scenarios in the system. This approach is, however, orthogonal to random failure injection because it cannot help with identifying unknown bugs in the system.

2.3 Systematic Failure Injection

Researchers have proposed systematic failure injection for distributed systems where a testing engine tries to exhaust the state-space for all of the possible scenarios and different kind of environmental failures than can happen.

Fate and Destini[32] is a test framework for systematic injection of environmental failures, and it also proposes a way of asserting correct behavior of the system in a declarative way. The failures are specifically injected in places in code where an IO operation (network, disk, etc.) is being performed. At each injection point, a failure server will be contacted to inquiry about injecting a failure. Injecting a failure, may cause new code paths to be exercised and may create new failure injection points. The tool runs a specific workload again and again and tries to do it until the state-space is exhausted. They have also proposed strategies to reduce the exploration space. PreFail [35] adds on Fate and Destini [32] by adding the support of failure injection filters, which allows for more control on the types of failures to be injected and injection time. Both of these approaches use instrumentation for failure injection, and thus, are only usable for Java-based distributed systems. Further, systematic failure injection can be very time-consuming and not suitable for running after each change in the code base.

This thesis proposes an approach where failures can be injected for different programming languages and only very specific and common scenarios can be defined and tested during regression testing.

2.4 Failure-specific Frameworks

Researchers have paid special attention to specific failures that happen more often with distributed systems, namely network partitions and disk failures.

In [24], in order to understand the effect of network failures in distributed systems, a selected set of bug reports on network related issues have been studied. The result is network-partition-related bugs can be catastrophic. They have proposed NEAT which is a test framework that allows the developer to define the startup and shutdown process for the system and then allows for injecting different kinds of network partitions during a test case execution. NEAT uses OpenFlow and iptables to accomplish this.

CORDS [31] is fault injection framework for file systems. It consists of two components, namely errfs and errbench. errfs is a user-level FUSE file system that systematically injects filesystem faults. errbench is a suite of system-specific workloads which drives systems to interact with their local storage. For each injected fault, CORDS automatically observes resultant system behavior to find potential malfunction.

This thesis extends NEAT [24] and make network partition injection more deterministic. File system fault injection has not been studied as part of this thesis, but can be done as a future work.

2.5 Model-based Approaches

Implementation-based model checkers have gained momentum in recent years for testing distributed systems. All of these model checkers have a interposition layer that intercepts the IO workload of the system under test (e.g. network messages and disk IO) and then try to reorder events in all possible ways. They also have the capability to inject failures like

crashes and network partitions in between of the events. The most challenging problem here is obviously state-space explosion.

To solve this problem, MaceMC [36] combines DFS and random walk using prioritized events labeled by the testers. CrystalBall [42], MODIST [43], and dBug[41] propose using DPOR independence [29] in a black-box manner without domain-specific knowledge. SAMC [37] enhances DPOR by leveraging white-box information and employing symmetry-based reduction. FlyMC [39] incorporates static analysis to enable more powerful independence and symmetry based reductions and a better prioritization strategy.

The problem with the model-based approaches is that it takes a lot of time to go through all of the possible scenarios, and they are not a good candidate for regular regression testing.

Chapter 3

Example Failify Test Case for HDFS

In this chapter, we demonstrate a very simple example of using Failify to create a JUnit test case for a HDFS [6] cluster in Java.

There are two simple steps when using Failify, namely (1) defining the environment and (2) controlling it in the test case. Each distributed system can have different kinds of nodes. In HDFS, there are Name Nodes and Data Nodes. A typical setup for HDFS has one Name Node and two Data Nodes. In Failify, we call different kinds of nodes a service. Services should be defined first, and then nodes can be defined out of services. Each service contains a set of application paths where the binaries or configuration files to run the service exist, a set of environment variables and the shell commands to start and stop the service. Also, as Failify uses Docker behind the scenes, each service needs to define a Dockerfile to create a docker image for the service. In this file, it is possible to install the required packages for the service.

HDFS only needs Java to be installed. Also iptables and iproute packages are needed for network partitioning and imposing network delays. Listing 3.1 shows the corresponding Dockerfile for the service.

```
1 FROM openjdk:8−stretch
2 RUN apt update && apt install iptables iproute2
```

Listing 3.1: Hello World Program Test Case Dockerfile

Assuming existence of a Hadoop 3.1.2 binary in the current working directory, required configuration files in "etc" directory for a HDFS cluster with a Name Node and two Data Nodes, Listing 3.2 shows a Failify test case for this cluster.

In this example, first, a "hadoop-base" service is defined which contains the common configuration for both "nn" (Name Node) and "dn" (Data Node) services. This service sets up the required path mappings (*PathAttr.COMPRESSED* tells Failify to first decompress the given path and then add it to the specified target path in the nodes), the working directory, Dockerfile address and corresponding built Docker image name and tag, the logs directory to be collected, required environment variables and programming language of the system. The "nn" and "dn" services are defined later by extending "hadoop-base" service and adding required init command (executed only in the first start of a node and not in restarts) and start command to initiate and start the service. Then, an instance of "nn" service and two instances of "dn" are defined. Finally, the deployment definition is built and started. At this stage, the test case can use the runner object to easily control the environment. In this example, the test case applies a network partition, creates a HDFS client using the IP of the name node coming from the runner object, applies a workload against the cluster, removes the applied network partition, and finally asserts the correct state or behavior of the system.

```
Class HelloWorldIT {
@Test
public void test() throws RuntimeEngineException {
    FailifyRunner runner = Deployment.builder("hdfs")
    .withService("hadoop-base")
        .appPath("hadoop.tar.gz", "/hadoop", PathAttr.COMPRESSED)
        .appPath("etc", "/hadoop/hadoop-3.1.2/etc")
        .workDir("/hadoop/hadoop-3.1.2")
        .env("HADOOP_HOME", "/hadoop/hadoop-3.1.2")
        .dockerImg("failify/hadoop:1.0")
        .dockerFile("docker/Dockerfile", true)
        .logDir("/hadoop/hadoop-3.1.2/logs")
        .serviceType(ServiceType.JAVA).and()
    .withService("nn", "hadoop-base")
        .initCmd("bin/hdfs namenode -format")
        .startCmd("bin/hdfs --daemon start namenode").and()
    .withService("dn", "hadoop-base")
        .startCmd("bin/hdfs --daemon start datanode").and()
    .withNode("nn1", "nn").and()
    .nodeInstances(2, "dn", "dn", false).build().start();

    NetPart netPart = NetPart.partitions("nn1,dn1","dn2")
    runner.runtime().networkPartition(netPart1);
    Configuration conf = new Configuration();
    conf.set("fs.default.name", "hdfs://" + runner.runtime().ip("nn1") + ":8020");
    FileSystem fileSystem = FileSystem.get(conf);
    // Do something under network partition
     ...
    runner.runtime().removeNetworkPartition(netPart);
    // Assert everything went fine
     ...
    runner.stop()
    }
}
```

Listing 3.2: Hello World Program Test Case

Chapter 4

Design Goals

To guide the development of Failify, we developed a set of design goals. Realizing these goals ensures that Failify is both usable and applicable for both practitioners and academics. This chapter discusses the goals and the way we achieved them.

4.1 Minimum Learning Curve

In today's software development projects, normally there are a handful of tools a developer needs to learn to get started with the project and be productive. As a result, software developers are not willing to try new tools with a high learning curve. One of the goals in designing Failify was to use familiar concepts to make developers more comfortable about the tool and minimize the learning curve. We chose Java for writing test cases for a couple of reasons, namely: (1) it is pervasive and all of JVM-based languages can use Java libraries as well; (2) it is strong typed, which can help with better auto-completion when using a modern IDE; (3) several famous open source distributed systems, like Hadoop [6] and Spark [20] are JVM-based, and as such, can easily integrate Failify into their test cases.

There are two simple steps when using Failify, namely (1) defining the environment and (2) controlling it in the test case. It is important to pay special attention to the second step. It is the test case that controls the environment and the test case execution. This is really important as the developer can think of Failify as another library to use for developing the test cases and not as a new test framework. This equally means that Failify can be used by any existing test frameworks in JVM-based languages like JUnit [10] or ScalaTest [19].

For the environment definition, we intentionally did not use configuration files to keep the developers engaged with what they know the best, the source code. We used a Hierarchical version of Builder Pattern [30] to keep the configuration compact and readable even in a verbose language like Java. For using Failify, a test developer only needs to know about four Java classes (Deployment, FailifyRunner, NetPart and NetOp), and the rest of the test case definition will be based on a set of method calls that have proper JavaDocs available and can be easily known using IDE auto-completion thanks to the Java strong typing. Also, hierarchies help in reducing the confusion in using the API by only exposing the developer to what is available in each level of the hierarchy. There is a simple rule in using the hierarchies: methods that start with "with" will take the control one level down the hierarchy and the "and" method will pull the control one level up into the hierarchy.

Additionally, we have a handful of examples in the Failify repository [5] and a minimal documentation [4] to demonstrate the best ways that Failify can be used.

4.2 Easy and Deterministic Failure Injection and Environment Manipulation

One of the main goals in designing Failify was to be able to inject environmental failures during test case executions. After defining and starting the test environment using Failify, it

is possible to easily inject environmental failures using a method call. Failify supports node failure, network delay, network loss, network partition and clock drift. It is also possible to run shell commands inside each of the nodes, which opens up the possibility of injecting custom failures that are not supported by the latest version of Failify.

Although the mentioned way of injecting failures is sufficient for many scenarios, there are still scenarios with specific timing constraints that happen only if a set of specific events in different nodes happen in a specific order [24]. In order to be able to develop deterministic test cases involving failure injections for these scenarios, we are using a slightly modified version of the approach taken in [38]. This enables the developer to order events in different nodes in their desired way to reproduce a scenario. For example, it is possible to inject failures before or after a method where a specific stack trace is present. More details about this feature is available in Section 5.

4.3 Cross-Platform and Multi Language

It is important for the Failify to be cross-platform so developers can use it on different platforms. The choice of Java for developing Failify and careful use of filesystem manipulation operations, and using Docker for the deployment in the background made this possible. It is important to note that although Failify can be used in different platforms, the operating system used for deploying services is limited to the Linux-based operating systems in the current implementation.

The deterministic failure injection feature requires minimal binary instrumentation for the affected nodes. As instrumentation is language-specific and we wanted to support as many programming languages as possible, Failify creates abstract instrumentation definitions for each of the affected nodes and then passes these definitions to the specific instrumentor

for different programming languages. At the time of writing this paper, Java and Scala programming languages are supported.

4.4 Seamless Integration

As Failify only uses the built artifacts of the system under test and does not require source code even for instrumentation, it can be seamlessly integrated with any build system. This can help with adoption of Failify by real world systems.

4.5 Multiple Runtime Engines

Deployment definition and the runtime engine that deploys the definition are separate concerns. As such, the runtime engine interface is separated from the deployment definition interface in Failify. At the time of writing this paper, a single node runtime engine is supported to allow developers to at least run the test cases in their own machines. However, every time a new feature is going to be added to the runtime engine, we assess its feasibility for multi-node runtime engines. Failify can potentially have runtime engines for IaaS systems like EC2 [1] or container orchestration systems like Swarm [21] and Kubernetes [12].

4.6 Research Infrastructure

While software testing and debugging research community has made enormous contributions to the practice of software testing and debugging in recent years, most of these contributions are either only applicable to single node software systems or their applications to distributed systems are unknown. Failify can be used as an infrastructure to facilitate this kind of re-

search as it provides the basics to work with distributed systems in a single node and can be extended in various ways including a few provided extension points. Failify can be useful in facilitating research for: log analysis for distributed systems as it can collect logs; automatic detection of consistency violations in distributed systems through adding input generators and custom instrumentors and analyzers; fault localization through extending current runtime engine and adding new instrumentors in order to create multi-node execution traces; systematic and random fault injection for distributed systems; and proving correctness of distributed systems through implementing new instrumentation engines and analyzers, as examples.

Chapter 5

Architecture and Implementation

This chapter describes the architecture and implementation details of Failify. Failify consists of five main components: a Java-based DSL to define the deployment architecture of the system and adding additional events to reproduce specific scenarios in a specific order which then will be fed to the rest of the components; a verification engine that verifies the deployment definition for any errors; a workspace manager that creates a separate workspace for each of the nodes in the system and copies over necessary paths to each workspace based on the deployment definition; an instrumentation engine that is able to instrument the nodes' binaries if necessary; and a runtime engine that deploys the system and allows on-demand manipulation of the deployed environment. The overall architecture of Failify is shown in Figure 5.1. The following sections describe each component in more detail.

5.1 Java-based DSL

The Java-based DSL uses the hierarchical version of the Builder design pattern [30], which makes it easy to define a deployment definition by calling a set of methods. Each deployment

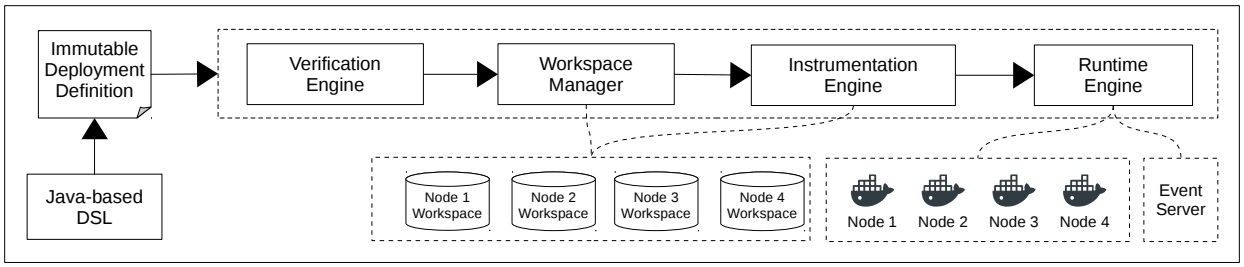

Figure 5.1: Failify's overall architecture

definition consists of a hierarchy of deployment entities e.g., a node. In each level of the hierarchy, it is possible to add a set of these entities. Any method starting with "with" indicates creating a child deployment entity in the hierarchy. In each level, there are other methods to configure the current deployment entity in the hierarchy and calling the "and" method indicates the configuration is done and it is desired to move back to the parent entity's hierarchy level. There are also some shortcut methods that facilitate creating child deployment entities. All the methods in the hierarchy that can cause the creation of a deployment entity require a unique name as their first argument, and thus, all the deployment entities are named.

Any distributed system has a few types of components and a number of nodes running instances of those component types in its deployment architecture. Different types of components are called *service* in the DSL. Any deployment definition should have at least one service definition and at least one node definition out of a service definition.

For each service, it is required to define: (1) a Dockerfile address that will be used to create a docker image for the corresponding service wherein it is possible to install required libraries or applications for the service; (2) the service binaries and dependencies to be added to the service and be bound to a specific target path where it is possible to mark a path as: a library to be used in the instrumentation process; changeable to be copied over to the node's workspace; or compressed to be decompressed before being added to the service; (3) for instrumentation purposes, the programming language of the service and specific paths that

E → E '*' T ❙ E '|' T ❙ T
T → event ❙ (E)

Figure 5.2: Run Sequence Grammar

may need to be instrumented (this should be a target path and not a local path); and (4) the shell commands to start the service. Optionally, it is possible to define the shell command to stop the service, the working directory, a set of environment variables, custom ulimit values, the log files or directories to be collected for the service, the application paths that may be changed through instrumentation to be used later in the instrumentation process, and the UDP and TCP ports to be exposed for the deployment, which can be later used by the test case using the deployment. For each node defined out of a service, it is possible to add additional paths such as configuration files, environment variables, TCP and UPD ports, and log files or directories.

Some distributed systems may need to share files between some of their nodes — e.g., Hadoop 2.9.2 can be configured to use NFS for HDFS high availability deployment. To facilitate this need, it is possible to define shared directories through DSL to be shared between all the services.

5.1.1 Deterministic Failure Injection

The other important part of the DSL is to define a specific scenario to be reproduced for the test case. We used a slightly modified version of ReproLite [38] to achieve this. The DSL allows the definition of a set of internal events (stack trace, block before/after a stack trace, unblock before/after a stack trace and garbage collection), test case events, and a run sequence to accomplish this. Each of these events are named and can be bound together using and (*), or (|) and parentheses through the run sequence. In this context, and (*) means running sequentially and or (|) means running in parallel. So, the run sequence follows a

```
1  ...
2  .withNode("n1", "service1")
3      .withStackTraceEvent("e1")
4          .trace("a.Class1.method1")
5          .trace("b.Class2.method2")
6          .blockAfter().and()
7      .stackTrace("e2", "c.Class3.method3")
8  .and()
9  .withNode("n2", "service1")
10     .stackTrace("e3", "d.Class4.method4, e.Class5.method5").and()
11     .withSchedulingEvent("bbe3")
12         .operation(SchedulingOperation.BLOCK)
13         .before("e3").and()
14     .unblockBefore("ube3", "e3")
15     .garbageCollection("g1")
16 .and()
17 .runSequence("e1 * bbe3 * (e2 | g1) * ube3")
18 ...
```

Listing 5.1: Run Sequence Example

simple expression with parenthesis grammar, where the operations are and (*) and or (|), as shown in Figure 5.2.

Listing 5.1 shows an example of how stack trace events can be defined for different nodes and how they can form a scenario through run sequence. Lines 7, 10, 14, and 15 are examples of how shortcut methods can be used to make the definition even more compact. In this example, all the threads in node n1 calling **c.Class3.method3** (the stack trace of event e2) will be blocked immediately. After a thread in node n1 calls **b.Class2.Method2** where **a.Class1.method1** is present in the stack trace (the stack trace of event e1), every thread in node n2 calling **e.Class5.method5** where **d.Class4.method4** is present in the stack trace will be blocked (event bbe3). Then, the blocked thread in node n1 having the stack trace of event e2 will be unblocked (event e2). Concurrently, a garbage collection operation will be performed in node n2 (event g1). Finally, the blocked threads in node n2 with the stack trace of event e3 will be unblocked (event ube3).

The main differences between this DSL and the one from ReproLite [38] are: (1) in the way the events are defined (by calling methods in a familiar language, Java, and with the

help of an IDE, e.g., IntelliJ, instead of a using a brand new language); (2) addition of the test-case events and removing the external events (e.g., node failure) from the the DSL, and giving the test case the ability to impose them by just calling a method that transfers the test execution control from the deployment engine to the test case itself and allows the use of assertion capabilities of existing testing frameworks; (3) removing the workload events and letting the test case generate the workload, which is the regular way developers write their test cases; and (4) the addition of network delay and loss, network partition, and clock drift failure injections. Test-case events in this DSL are just event names that can be included in the run sequence and later enforced by the test case, itself. The Failify's runtime engine provides adequate methods for the test case to synchronize itself with the running run sequence (e.g., waiting for an event to be satisfied) and enforcing the defined test case events where it is possible to wait for the events' dependencies to be satisfied, do something in the test case such as injecting a failure, and then, marking the test case event as satisfied.

5.2 Verification Engine

The verification engine runs a set of verifiers against the deployment definition to make sure it is error-free. At the time of writing of the paper, three verifiers were implemented. The internal references verifier ensures no deployment entity refers to an undefined deployment entity. The scheduling-operation verifier ensures that block and unblock events appear as pairs in the run sequence and in a valid order. The run-sequence verifier parses the run sequence for syntax errors and also identifies the dependencies to be satisfied and the blocking condition for each individual event in the run sequence to be used later by the runtime engine.

5.3 Workspace Manager

This component is responsible for decompressing compressed application paths, creating shared directories and creating the nodes' workspaces. A node workspace contains empty log files and directories that need to be collected, application paths that are marked as changeable, instrumentable paths, and Failify-required internal files. The output of this component for each node will be a set of local workspace paths to be used later by the instrumentation engine and a set of path mappings which will be used later by the runtime engine.

5.4 Instrumentation Engine

The instrumentation engine is one of the extension points of Failify. Given the deployment definition, this component instruments the nodes' binaries. At the time of writing this paper, a language-agnostic run sequence instrumentation engine was implemented. However, it is possible to add new instrumentation engines to change the nodes binaries in a desired way, and being language-agnostic is not a requirement for an instrumentation engine.

The run sequence instrumentation engine is a language-agnostic instrumentor that adds necessary code to enforce the defined run sequence. It first creates a set of abstract instrumentation definitions based on the internal event definitions and the run sequence and unifies the definitions for each instrumentation point into one instrumentation definition, and then, delegates the actual instrumentation to the proper instrumentor based on the service's programming language. At the time of writing this paper, Java and Scala instrumentors were implemented using AspectJ. Future implementations may include native binary instrumentation using Pin [17].

Each instrumentation definition consists of an instrumentation point and a list of instrumentation operations. The instrumentation point can be before or after a method. An instrumentation operation consists of an operation name from a predefined list including *allow blocking, enforce order*, and *garbage collection*, and a list of string parameters.

The *enforce* order operation takes an event name as input and will block the running thread if the blocking condition (one or more events) is satisfied and will be unblocked when the input event dependencies (one or more events) are satisfied. Before returning, it will mark the input event as satisfied. The blocking condition is important for scheduling events where, unlike stack trace events, the blocking does not start in the beginning. Also, when there are multiple blocking events (scheduling or stack trace) with the same stack trace in the run sequence, the blocking condition helps the blocking to happen based on the order of events' appearance in the run sequence.

The *allow* blocking operation sets a thread-local flag to true and will be added as the first instrumentation operation for each method that has blocking instrumentation. This is because blocking is only allowed once in each passage of a method and for each thread, and only happens when this flag is true. After a blocking happens and is unblocked, this flag will be set to false and no other blocking will happen in the same method passage.

5.5 Runtime Engine

The runtime engine is another extension point in Failify. Given the deployment definition and the created nodes' workspaces, this component deploys the system and provides proper interfaces to manipulate the deployed environment. Although, at the time of writing this paper, only a single node runtime engine was implemented, the required interfaces are in place, and it is possible to develop other kinds of runtime engines, for example to deploy the

system on multiple nodes on IaaS systems like Amazon EC2 [1], or on container orchestration systems like Swarm [21] or Kubernetes [12].

The common tasks including starting and stopping the event server, adding new nodes during test case execution, waiting for an event in the run sequence to be satisfied, enforcing order for a test case event in the run sequence, imposing and removing a network partition, and imposing and removing network delay and loss in an individual node are done through an abstract common runtime engine. A concrete implementation of this common runtime engine should implement capabilities for starting and stopping a file sharing service, starting and stopping the nodes, start/restart/kill/stopping of an individual node, running a shell command in an individual node, providing information including the IP addresses for the nodes and possible port mappings, and applying a clock drift in an individual node.

The event server is started when the deployment definition contains a run sequence. It knows about the blocking condition and dependencies of all the events in the run sequence and the current status of the events, and as such, it acts as the source of truth for all the involved nodes and external events, and it provides a REST API to query this information and mark an event as satisfied.

The network partition functionality is implemented using the Linux *"iptables"* program. The network delay functionality is implemented using the Linux *"tc"* program.

The implemented single node runtime engine uses Docker in the background to deploy the nodes. For each instance of the deployment, it creates a separate docker network. In case the client itself is being run in Docker, the client's container will be added to the created Docker network, which makes it possible to run the client and the deployed system in the same network namespace where the nodes can be accessed directly by the client through their IP address and looked up by the client through the DNS and using their defined name in the deployment definition. This is important in Windows and Mac (unlike Linux), as

Docker containers cannot be accessed using their local IP addresses and are only accessible through port mapping, which is why port mapping information should be provided by the runtime engines. This feature is useful when it is desired to run CI jobs inside a container, and when the client needs to be a member of the distributed system or have direct access to the members through their IP addresses or hostnames, e.g., in Cassandra.

Although Linux provides the capability of having separate namespaces for network and filesystem for the processes, it does not provide a separate time namespace. As such, changing time in one of the containers will change the time in the host and consequently in all the running containers. To resolve this issue, libfaketime [13] library is used. This library intercepts date and time related system calls and reports faked dates and times as specified by the runtime engine.

For each node, all the application paths are shared by the node's container using Docker bind mounting and provided path mappings from the workspace manager. Also, the log files, and the log and shared directories are shared in the same way with the containers, which provides log collection and file sharing without any additional effort.

Chapter 6

Evaluation

We performed evaluation to show that Failify's deployment definition API is compact and can support a wide variety of distributed systems. Also, to demonstrate potential scenarios where Failify's deterministic environmental manipulation and failure injection API can be effective, we created a test case for the lease recovery process in HDFS. We then discuss why the Failify's performance footprint is minimal. Finally, we talk about the limitations of Failify and threats to validity.

6.1 Compactness and Generality of the Deployment API

To evaluate the compactness of Failify's deployment API and its ability to support a wide variety of distributed systems, we experimented with six open-source distributed systems, including distributed databases (Cassandra [2] (Java), RethinkDB [18] (C++), and TiDB [22] (Go and Rust)), messaging systems (Kafka [11] (Scala)), file systems (HDFS [6] (Java)), data-processing frameworks (Spark [20] (Java and Scala)). These systems were carefully hand-

HDFS	Spark	Kafka	RethinkDB	TiDB	Cassandra
20	15	15	11	25	15

Table 6.1: Lines of code needed to define the most reliable deployment architecture using Failify

picked to cover a set of widely used programming languages for development of distributed systems (Java, Scala, C++, Go and Rust) and different applicability categories. For each of these systems, we chose the latest stable version, built it on Debian, and included the build artifacts in the corresponding system's Failify repository. This has been done to focus the example repositories on the Failify test cases and not on the individual systems' build process. All the deployment definitions developed for these systems are available in the Failify's repository [5].

For each of these systems, we developed a *FailifyHelper* class that has a *getDeployment* method, which given a set of parameters (such as the number of nodes) would return a Failify deployment definition instance. This helper class later is used by the test cases to easily define their needed environment. We were able to define the most reliable deployment architecture (i.e. one that doesn't have a single point of failure) of all the picked systems using Failify. Table 6.1 shows the number of lines in the *getDeployment* method for each of the systems. The numbers in the table acknowledge the compactness of Failify's API. In average, the most reliable deployment architecture for these systems can be defined in less that 17 lines of code. It should be noted that the deployment definition can be potentially completed in fewer lines, but the numbers represent a well-indented and readable definition. We only discuss TiDB's deployment definition here. Interested readers are urged to look at Failify's repository [5] for more examples.

TiDB is a distributed scalable Hybrid Transactional and Analytical Processing (HTAP) database built by PingCAP. TiDB platform is comprised of three main components: the TiDB server, the Placement Driver (PD) server and the TiKV server. The TiDB server is

```
1  private static String getPdString(int numOfPds, boolean http) {
2      StringJoiner joiner = new StringJoiner(",");
3      for (int i=1; i<=numOfPds; i++) joiner.add("pd" + i + (http ? "=http://pd" + i + ":2380" :
           ":2379"));
4      return joiner.toString();
5  }
6  public static Deployment getDeployment(int numOfPds, int numOfKvs, int numOfDbs) {
7      String pdStr = getPdString(numOfPds, false), pdInitialClusterStr = getPdString(numOfPds,
           true);
8      return Deployment.builder("example-tidb")
9          .withService("common").dockerImg("failify/tidb:1.0").dockerFile("docker/Dockerfile",
               false).and()
10         .withService("tidb", "common")
11             .appPath("../tidb-2.1.8-bin", "/tidb")
12             .startCmd("/tidb/tidb-server --store=tikv --path=\"" + pdStr +
                   "\"").tcpPort(4000)
13             .and().nodeInstances(numOfDbs, "tidb", "tidb", true)
14         .withService("tikv", "common")
15             .appPath("../tikv-2.1.8-bin", "/tikv").disableClockDrift()
16             .startCmd("/tikv/tikv-server --addr=\"0.0.0.0:20160\"
                       --advertise-addr=\"$(hostname):20160\" " +
17                 "--pd=\"" + pdStr + "\" --data-dir=/data")
18             .and().nodeInstances(numOfKvs, "tikv", "tikv", true)
19         .withService("pd", "common")
20             .appPath("../pd-2.1.8-bin", "/pd").tcpPort(2379)
21             .startCmd("/pd/pd-server --name=\"$(hostname)\"
                       --client-urls=\"http://0.0.0.0:2379\" --data-dir=/data " +
22                 "--advertise-client-urls=\"http://$(hostname):2379\"
                           --peer-urls=\"http://0.0.0.0:2380\" " +
23                 "--advertise-peer-urls=\"http://$(hostname):2380\" --initial-cluster=\"" +
                       pdInitialClusterStr + "\"")
24             .and().nodeInstances(numOfPds, "pd", "pd", false).build();
25 }
```

Listing 6.1: The Deployment Definition for TiDB

a stateless component that processes SQL queries and stores and retrieves data in and from TiKV by accessing the metadata in PD nodes. PD is responsible for storing the metadata for the cluster, leader election, region balancing and unique transaction ID allocation. TiKV is a distributed transactional key-value storage engine.

Listing 6.1 depicts the *getDeployment* method for TiDB. The *getPDString* method generates the PD connection string for RPC and HTTP access and is used later in the start command for all the services. Three Failify services, namely tidb, tikv and pd, are defined. The number of instances from each service is given through parameters, and the method uses the

nodeInstances method to define the required nodes. The clock drift capability is disabled in the tikv nodes as the tikv nodes would crash with libfaketime enabled in the environment. Also tikv and tidb nodes are off on startup and will not start when the deployment is started. This is because tikv and tidb nodes require the PD cluster to be online to get started. Another helper method, called *startNodesInOrder*, checks for the availability of the PD cluster and then starts the rest of the nodes. A TiDB cluster of any size can be easily defined using the 25 lines of code in the *getDeployment* method, which shows the ability of Failify's API to define complex deployment architectures in a compact form.

6.2 Effectiveness of Deterministic Environment Manipulation API

To demonstrate potential scenarios where Failify's deterministic environmental manipulation and failure injection API can be effective, we re-wrote an already existing test case in HDFS [6] for the lease recovery process. To perform this evaluation, we used v3.1.2 of HDFS. In the rest of this section, we first try to introduce the HDFS architecture. Then, we explain the lease recovery process and go over an already existing test case for this process in HDFS test suite. Then, we explain how we re-wrote this test case using Failify. Finally, we discuss how Failify can be improved to better handle such scenarios.

6.2.1 HDFS Architecture

Hadoop Distributed File System (HDFS) is an append-only distributed file system and part of the Hadoop project [6] acting as the underlying filesystem for MapReduce framework. HDFS is fault tolerant, designed to run on commodity hardware, and is written in Java programming language. The overall architecture of HDFS is depicted in Figure 6.1

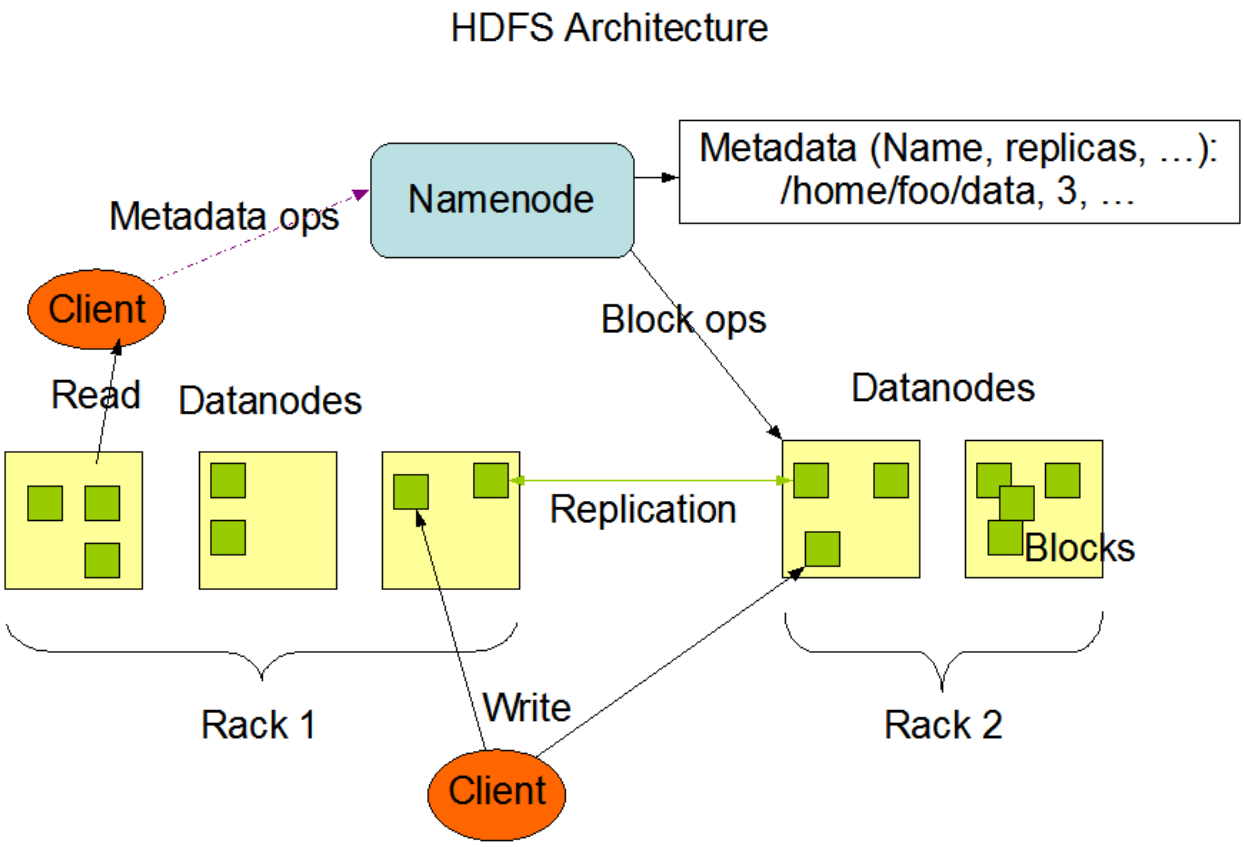

Figure 6.1: HDFS overall architecture [6]

HDFS has a master/slave architecture. An HDFS cluster consists of one or potentially more NameNodes (NN) (one active and multiple standby nodes). NNs are responsible for the management of the file system namespace and regulation of client accesses to the files. Further, each cluster has a set of DataNodes (DN) where the actual reads and writes to files happen. Each file in HDFS, is divided into one or more blocks distributed among different DNs. Also, to be fault-tolerant, multiple replicas of the same block is stored in different DNs. The number of replicas is called the replication factor.

NN only stores the meta-data. It determines the mapping of blocks to DNs, and also, handles operations like closing, opening and renaming of files and directories. DNs do the actual reads and writes, and they also handle block replication. More information about HDFS architecure is available at [6].

6.2.2 Lease Recovery Process and Test Case

In HDFS, file access follows multi-reader, single-writer semantics. As such, in order to write to a file, a client first needs to obtain a lease (i.e. lock) to ensure the single-writer requirement. The client must renew the lease within a predefined period of time. Clients renew the lease for all of their files opened for write at once and not individually. The lease will expire if the client doesn't renew it, or the client dies. When this happens, HDFS closes the corresponding file(s) and releases the lease so that other clients can access the file(s). This process is called lease recovery.

During the lease recovery process, the following process happens for each file opened for write. The active NN finds all of the DNs containing the last block of the file and assigns one of them as the primary node to perform the recovery. The primary DN queries all of the other DNs to get the block info and computes the minimum block length. Consequently, the primary DN asks the other DNs to update their replica to only contain up to the minimum length. Finally, the primary DN commits the new block info to the active NN, and NN updates its meta-data accordingly.

The test case we chose to re-write for the lease recovery process is in the *TestPipelinesFailover* class and the test method *testFailoverRightBeforeCommitSynchronization*. This test case tests the scenario where the active NN fails over after the primary DN sends the commit request to the active NN, but before it is actually committed in NN. When this happens, the primary DN should fail to commit and a later retry will succeed. The source code for this case is depicted in Listing 6.2 (unrelated details have been removed from the test case).

HDFS uses *MiniDFSCluster* utility to deploy a HDFS cluster in one JVM process. Using this utility it is possible to deploy a custom HDFS cluster and perform some actions on top it. For this test case, A cluster with 3 NNs and 3 DNs is being created. The failover mechanism for this cluster is manual. This is being done using the *newMiniCluster* method

```
1  public void testFailoverRightBeforeCommitSynchronization() throws Exception {
2      final Configuration conf = new Configuration();
3      FSDataOutputStream stm = null;
4      final MiniDFSCluster cluster = newMiniCluster(conf, 3);
5      try {
6          cluster.waitActive();
7          cluster.transitionToActive(0);
8          FileSystem fs = HATestUtil.configureFailoverFs(cluster, conf);
9          stm = fs.create(TEST_PATH);
10         // write a half block
11         AppendTestUtil.write(stm, 0, BLOCK_SIZE / 2); stm.hflush();
12         // Look into the block manager on the active node for the block under construction.
13         NameNode nn0 = cluster.getNameNode(0);
14         ExtendedBlock blk = DFSTestUtil.getFirstBlock(fs, TEST_PATH);
15         DatanodeDescriptor expectedPrimary = DFSTestUtil.getExpectedPrimaryNode(nn0, blk);
16         // Find the corresponding DN daemon, and spy on its connection to the active.
17         DataNode primaryDN = cluster.getDataNode(expectedPrimary.getIpcPort());
18         DatanodeProtocolClientSideTranslatorPB nnSpy =
19             InternalDataNodeTestUtils.spyOnBposToNN(primaryDN, nn0);
20         // Delay the commitBlockSynchronization call
21         DelayAnswer delayer = new DelayAnswer(LOG);
22         Mockito.doAnswer(delayer).when(nnSpy).commitBlockSynchronization(
23             Mockito.eq(blk), Mockito.anyLong(), Mockito.anyLong(),
24             Mockito.eq(true), Mockito.eq(false), Mockito.any(), Mockito.any());
25
26         DistributedFileSystem fsOtherUser = createFsAsOtherUser(cluster, conf);
27         assertFalse(fsOtherUser.recoverLease(TEST_PATH));
28         LOG.info("Waiting for commitBlockSynchronization call from primary");
29         delayer.waitForCall();
30         LOG.info("Failing over to NN 1");
31         cluster.transitionToStandby(0);
32         cluster.transitionToActive(1);
33         // Let the commitBlockSynchronization call go through, and check that
34         // it failed with the correct exception.
35         delayer.proceed();
36         delayer.waitForResult();
37         Throwable t = delayer.getThrown();
38         if (t == null) {
39         fail("commitBlockSynchronization call did not fail on standby");
40         }
41         GenericTestUtils.assertExceptionContains("Operation category WRITE is not supported", t);
42         // Now, if we try again to recover the block, it should succeed on the new active.
43         loopRecoverLease(fsOtherUser, TEST_PATH);
44
45         AppendTestUtil.check(fs, TEST_PATH, BLOCK_SIZE/2);
46     } finally {
47         IOUtils.closeStream(stm);
48         cluster.shutdown();
49     }
50 }
```

Listing 6.2: Original lease recovery test case

which takes the number of DNs as a parameter and returns a cluster object. The test case then waits for the cluster to be active and transitions the first NN to be the active NN. Then, a client connection to the cluster is established, a file is opened and half a block is written.

At this point, the test case tries to force a lease recovery for the test file by creating another user and client and calling the *leaseRecovery* method on the client. However, before that, it determines the expected DN to be the primary DN for the recovery process and replaces its protobuf NN client implementation with a mocked object created by Mockito. Because the whole cluster is being deployed in one JVM process, mini cluster has access to the internal states of classes for NNs and DNs and that is how it can apply these kind of changes. This mocked object, when receives a call to *commitBlockSynchronization* with specific parameters, blocks the running thread until it is unblocked by the test case. Using *delayer.waitForCall()*, the test case execution will be blocked until the primary DN reaches to the commit method. When this happens, the test case transitions the first NN to the standby state and the second NN to the active state, and releases the DN blocked thread afterwards by calling *delayer.proceed()*. It is expected that the unblocked thread should throw an exception because the receiving NN is not active anymore. The thrown exception is caught by the mocked object and checked to be the right exception by the test case. Finally, the lease recovery will be repeated until it is successful and the written block will be checked to have the right size.

6.2.3 Re-writing Lease Recovery Test Case Using Failify

To re-write the test case using Failify, we first created a *FailifyHelper* class which for the purpose of this test case replaces *MiniDFSCluster* utility. This class takes the number of NNs and the number of NNs as parameters and creates an initial working deployment definition using Failify. It has helper methods to add instrumentable paths, starting and stopping the

deployment, waiting for the cluster to be active, creating an HDFS client to connect to the cluster, and transitioning NNs from active to standby and vice versa. Also, it is possible to change the deployment definition and add events to the nodes and a run sequence to the definition to reproduce a specific scenario. The source code for this class is available at [8].

It is necessary to pay attention that, when using Failify, we don't have access to the internal states of the nodes unless exposed through an external API. This means that using Mockito is not option. We replace it by defining a set of events and combining them using a run sequence to reproduce the same scenario. However, because Failify's run sequence should be defined before starting the cluster and cannot be adjusted afterwards, we have to change the blocking points. Instead of blocking the protobuf NN client implementation, we block the NN server implementation which is in NN and doesn't require us to know about the expected primary DN to perform the lease recovery. This also helps us with the fact that, this information about a block is not exposed through an external API.

We define event **e1** as *o.a.h.h.s.namenode.NameNodeRpcServer.commitBlockSynchronization* which is the method in NN that handles the block synchronization request from the primary DN. This event lets us know when to do the failover. We also need a test case event **t1** to integrate the test case with the run sequence. This event depends on **e1** and helps us to know when to transition the second NN to the active state and the first NN to the standby state in the test case.

Event **e2** is defined as *o.a.h.h.s.namenode.FSNamesystem.commitBlockSynchronization* which is the actual point that the request from primary DN should be blocked and wait for the failover to happen. As such this event depends depends on **t1**.

Finally, to handle the exception assertion part of the test case, as we don't have any access to the internal state of the DN, we make sure that the right method in NN is called. This method is only called if the NN is in standby mode and it throws exception when it is called

```
1  public void testFailoverRightBeforeCommitSynchronization()
2          throws IOException, RuntimeEngineException, InterruptedException, TimeoutException {
3      FSDataOutputStream stm = null;
4
5      FailifyHelper failifyHelper = new FailifyHelper(3, NUM_OF_DNS);
6      failifyHelper.addInstrumentablePath("/share/hadoop/hdfs/hadoop-hdfs-3.1.2.jar");
7      failifyHelper.getDeploymentBuiler().node("nn1")
8          .stackTrace("e1",
9      "org.apache.hadoop.hdfs.server.namenode.NameNodeRpcServer.commitBlockSynchronization")
10         .stackTrace("e2",
               "org.apache.hadoop.hdfs.server.namenode.FSNamesystem.commitBlockSynchronization")
11         .stackTrace("e3",
12             "org.apache.hadoop.hdfs.server.namenode.FSNamesystem.commitBlockSynchronization,"
13                 + "org.apache.hadoop.hdfs.server.namenode.FSNamesystem.checkOperation,"
14                 + "org.apache.hadoop.hdfs.server.namenode.ha.StandbyState.checkOperation")
15         .and().testCaseEvents("t1").runSeq("e1 * t1 * e2 * e3");
16
17     FailifyRunner runner = failifyHelper.start();
18     failifyHelper.waitActive();
19
20     try {
21         failifyHelper.transitionToActive(1);
22         FileSystem fs = failifyHelper.getFileSystem();
23         stm = fs.create(TEST_PATH);
24         // write a half block
25         AppendTestUtil.write(stm, 0, BLOCK_SIZE / 2);
26         stm.hflush();
27
28         DistributedFileSystem fsOtherUser = createFsAsOtherUser(failifyHelper);
29         assertFalse(fsOtherUser.recoverLease(TEST_PATH));
30
31         failifyHelper.runner().runtime().enforceOrder("t1", () -> {
32             failifyHelper.transitionToStandby(1);
33             failifyHelper.transitionToActive(2);
34         });
35
36         failifyHelper.runner().runtime().waitForRunSequenceCompletion(30);
37         loopRecoverLease(fsOtherUser, TEST_PATH);
38         AppendTestUtil.check(fs, TEST_PATH, BLOCK_SIZE/2);
39     } finally {
40         IOUtils.closeStream(stm);
41         runner.stop();
42     }
43 }
```

Listing 6.3: Lease recovery test case re-written in Failify

as part of the lease recovery process. To this end, we define **e3**. The stack trace used to define this event is shown in Listing 6.3. This event depends on **e2** as the exception happens after the unblocking.

The run sequence to glue all of these events together is **"e1 * t1 * e2 * e3"**. Also, the order of event **t1** is enforced in the test case by calling the *enforceOrder* method which waits for the event's dependencies to be satisfied, performs the failover, and marks the event as satisfied. The rest of the test case remains almost the same as the original one.

6.2.4 Discussion

Although we were able to re-write the HDFS test case using Failify, it is necessary to discuss how things could go wrong and how failify can be improved to accommodate for that.

In the HDFS original test case, Mockito is being used in a way that the spy object only answers to a specific set of input parameters. Right now, this can't be done using Failify. We didn't need this in the Failify test case, as we had only one block in the system and we knew that the commit method would be only called with that block as the first parameter. Defining a parameter selection API for Failify can be tricky as it should be able to work with different programming languages. This is definitely a feature that should be considered in the future work and the next versions of Failify.

One of the other improvements can be adding test case events as unblocking conditions for blocking events. This means that the test case event has the blocking event as its dependency, but the blocking event won't be unblocked or marked as satisfied until the test case event is satisfied. In the HDFS test case, it could help us with removing **e2** and add **t1** as the unblocking dependency of **e1**. It is important to note that this is not just about reducing the number of events. There are situations where it is not possible to define an extra event to

reproduce the scenario of "when reached to this stack trace, block the thread, inject failure X, and unblock the thread".

In general, it is important that distributed systems provide more visibility into the internal state of the system through external APIs. Also, relying on internal exceptions for assertion in an end-to-end test should be avoided. An alternative can be, for example, an external API that shows that a specific background task has been failed.

6.3 Performance Considerations

As Failify is not purposed for performance testing of distributed systems, we skip the performance analysis of the tool. However, if internal events are defined and used in the run sequence, Failify does a minimal amount of instrumentation on the affected nodes. As also discussed in [38], this has a very small footprint. The real impact comes from the fact that in order to reproduce a time-sensitive scenario, some nodes may be blocked for a small amount of time, so the others could catch up, but this should be very minimal in race condition scenarios. Further Docker on Mac and Windows is slow with a handful amount of bind mounts. This is a known problem in Docker because the CPU will be busy synching the files in Linux VM running the Docker engine with the host Mac or Windows machine.

6.4 Limitations and Threats to Validity

Although Failify is a general purpose testing infrastructure that can be used for a wide variety of distributed systems in multiple languages, it may not be applicable to the systems that require special hardware, such as specific types of storage. Further, libfaketime library is dependent on cglib and may not work on Alpine Linux, and also, it can sometimes make

programs crash. In this case, it is possible to disable the clock drift capability in Failify at the service level. Also, Failify, at the time of writing this paper, does not support filesystem failure injections. This feature can be supported by using libraries like libfuse [14] and can be added in the future.

For the evaluation part, although we did our best to choose a wide variety of distributed systems written in different programming languages and we developed test cases to show case the effectiveness of Failify, we cannot ensure that Failify is generalizable to all systems, deployment architectures, and bug-reproduction scenarios. We, however, designed Failify to be extensible to allow for future enhancement and extension, should we encounter such limitations.

Chapter 7

Conclusion

In this thesis, we present Failify, a deterministic, programmable, programming-language-agnostic failure testing library for distributed systems, where it is possible to reproduce scenarios with specific timing constraints. We evaluated the compactness of Failify's deployment API with 6 different open-source distributed systems, written in different programming languages and from diverse categories. Our results indicate that, in average, the most reliable deployment architecture for these systems can be defined in less that 17 lines of code. We also re-wrote an existing test case for lease recovery process in HDFS using Failify to demonstrate potential scenarios where Failify's deterministic environmental manipulation and failure injection API can be effective.

Our demonstration with HDFS showed that adding features like test case events as an unblocking condition for blocking events and a parameter selection API when defining stack traces can be really useful when trying to reproduce specific scenarios. Also, we think that distributed systems should expose more details about the internal state of the system through external APIs to make the end-to-end testing of these systems more effective.

Future work includes applying the Failify deterministic environmental manipulation API to

more systems to find out new ways where the API can be improved. Implementing automatic consistency violation detection frameworks on top of Failify and adding post-mortem log query and analysis for a better insight over failed test cases are also possible avenues for future work.

Bibliography

[1] Amazon ec2. https://aws.amazon.com/ec2/.

[2] Cassandra. http://cassandra.apache.org.

[3] Ducktape. https://github.com/confluentinc/ducktape.

[4] Failify. http://failify.io.

[5] Failify repository. http://github.com/failify.

[6] Hadoop. https://hadoop.apache.org.

[7] Hbase. https://hbase.apache.org.

[8] Hdfs lease recovery test case using failify. https://github.com/failify/example-hdfs/tree/first-test.

[9] Jepsen. http://jepsen.io.

[10] Junit. https://junit.org.

[11] Kafka. https://kafka.apache.org/.

[12] Kubernetes. https://kubernetes.io.

[13] Libfaketime. https://github.com/wolfcw/libfaketime.

[14] Libfuse. https://github.com/libfuse/libfuse.

[15] Namazu. http://osrg.github.io/namazu/.

[16] Nose. https://nose.readthedocs.io.

[17] Pin. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[18] Rethinkdb. http://github.com/failify.

[19] Scalatest. http://www.scalatest.org/.

[20] Spark. https://spark.apache.org.

[21] Swarm. `https://docs.docker.com/engine/swarm/`.

[22] Tidb. `https://www.pingcap.com/`.

[23] Zopkio. `https://github.com/linkedin/Zopkioe`.

[24] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, Carlsbad, CA, 2018. USENIX Association.

[25] A. Balalaie and J. A. Jones. Towards a library for deterministic failure testing of distributed systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 486, New York, NY, USA, 2019. Association for Computing Machinery.

[26] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[27] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[28] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[29] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 110–121, New York, NY, USA, 2005. Association for Computing Machinery.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[31] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[32] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.

[33] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, New York, NY, USA, 2016. ACM.

[34] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66, June 2016.

[35] P. Joshi, H. S. Gunawi, and K. Sen. Prefail: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

[36] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, Apr. 2007. USENIX Association.

[37] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, Broomfield, CO, 2014. USENIX Association.

[38] K. Li, P. Joshi, A. Gupta, and M. K. Ganai. Reprolite: A lightweight tool to quickly reproduce hard system bugs. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 25:1–25:13, New York, NY, USA, 2014. ACM.

[39] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, and et al. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[40] G. Milka and K. Rzadca. Dfuntest: A testing framework for distributed applications. In R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 395–405, Cham, 2018. Springer International Publishing.

[41] J. Simsa, R. Bryant, and G. Gibson. Dbug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, page 3, USA, 2010. USENIX Association.

[42] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, page 229–244, USA, 2009. USENIX Association.

[43] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[44] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. 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, 2014. USENIX Association.