TÉCNICO LISBOA

FaultSee: Reproducible fault injection in distributed systems

Miguel Antão Pereira Amaral

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Miguel Ângelo Marques de Matos
Prof. Miguel Filipe Leitão Pardal

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. João Carlos Antunes Leitão

December 2019

Acknowledgments

This work is partially supported by Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa and by Fundação para a Ciência e Tecnologia (FCT) through projects with reference UID/CEC/50021/2013 and LISBOA-01-0145-FEDER-031456.

Resumo

Os sistemas informáticos distribuídos são cada vez mais importantes na sociedade moderna operando, muitas vezes, a uma escala global e com requisitos de disponibilidade muito perto dos 100%. Para alcançar um nível tão elevado de disponibilidade é necessário ter processos de desenvolvimento focados na qualidade e com testes rigorosos e exaustivos. Os sistemas distribuídos, por serem formados por vários componentes, são bastante difíceis de avaliar e testar de uma forma sistemática e reproduzível. Esta dificuldade pode constatar-se ao analisar artigos científicos ou outros estudos sobre um sistema distribuído em operação, em que é frequente ver afirmações sobre a injeção de faltas em nós do sistema, que não são explicadas nem contextualizadas de forma a permitir a reprodução num ambiente de teste alternativo. Dado que o comportamento do sistema pode variar substancialmente consoante o tipo de "falta" injetada, torna-se praticamente impossível a um investigador reproduzir o comportamento observado nesse teste. Se não existir reproducibilidade, então não se consegue fazer a adequada comparação de alternativas, e o progresso técnico torna-se mais lento e dispendioso.

Neste trabalho propomos a criação da plataforma *FaultSee* que permite avaliar sistemas reais de uma forma mais sistemática e reproduzível do que o estado da arte. Propomos também uma linguagem, a *FDSL*, usada pelo *FaultSee*, de especificação de sistemas distribuídos e injeção de faltas que capturam precisamente variáveis como o ambiente de teste, a carga de trabalho e o tipo de faltas. Estas funcionalidades são demonstradas em cenários realistas usando a base de dados *Apache Cassandra* e o sistema *BFT-Smart* como casos de estudo.

Palavras-chave: Sistemas distribuídos, Avaliação de sistemas, Reprodutibilidade, Tolerância a faltas.

Abstract

Distributed systems are getting more important in modern society, often operating on a global scale with availability requirements close to 100%. Achieving high levels of availability requires quality-focused development processes with rigorous and thorough testing. Distributed systems, due to having several components, are quite difficult to evaluate and test in a systematic and reproducible manner. When analyzing a study or paper of a distributed system in operation, often there are statements about fault injection in system nodes, which are neither explained nor contextualized to allow reproduction in an alternate test environment. Since system behavior can vary substantially depending on the injected fault, it is virtually impossible for a researcher or engineer to reproduce the behavior observed in a test. Without reproducibility, correct comparison of alternatives is unobtainable, and technical progress becomes slower and more expensive. In this thesis, we propose the *FaultSee* platform that allows to evaluate real systems in a more systematic and reproducible way than the state of the art. We also propose a language, used by FaultSee, for distributed system specification and fault injection that accurately capture variables such as the test environment, workload and fault type. These features are demonstrated in realistic scenarios using the *Apache Cassandra* database and the *BFT-Smart* system as case studies.

Keywords: Distributed Systems, Fault Injection, Reproducibility, Chaos Engineering.

Contents

Acknowledgments . . . iii
Resumo . . . v
Abstract . . . vii
List of Tables . . . xi
List of Figures . . . xiii

1 Introduction **1**
1.1 Requirements . . . 3
1.2 Contributions . . . 4
1.3 Thesis Outline . . . 4

2 Background and Related Work **5**
2.1 Definitions . . . 5
2.2 Fault Injection . . . 7
2.3 Monitoring/Tracing . . . 9
2.4 Deployment . . . 10
2.5 Usability . . . 11
2.6 Overview . . . 12

3 FaultSee **13**
3.1 Approach . . . 13
3.1.1 Requirements . . . 13
3.1.2 Experiment Lifecycle . . . 15
3.1.3 Experiment Configuration . . . 16
3.2 Design . . . 16
3.2.1 FaultSee Architecture . . . 17
3.2.2 FaultSee Domain System Language . . . 21
3.3 Implementation . . . 24

3.4 Overview . . . 27

4 Evaluation **29**

4.1 Features . . . 29

4.1.1 Test Target Functionality . . . 29

4.1.2 Test Order . . . 29

4.1.3 Fail on Docker Pull error . . . 30

4.1.4 Parse . . . 30

4.1.5 Start and Stop Containers . . . 30

4.1.6 CPU Exhaustion . . . 31

4.1.7 Custom faults . . . 31

4.2 Macro Benchmarks . . . 31

4.2.1 Cassandra . . . 32

4.2.2 BFT-Smart . . . 35

4.3 Discussion . . . 39

5 Conclusion **41**

5.1 Achievements/Contributions . . . 41

5.2 Future Work . . . 42

Bibliography **43**

A Apache Cassandra experiments configuration files **47**

A.1 Docker-Compose file . . . 47

A.2 FDSL file for the Kill a node scenario . . . 50

A.3 FDSL file for the CPU exhaustion scenario . . . 53

B BFT-Smart experiments configuration files **57**

B.1 Docker-Compose file . . . 57

B.2 FDSL file for the faultless scenario . . . 61

B.3 FDSL file for the CPU exhaustion scenario . . . 63

B.4 FDSL file for the Kill a node scenario . . . 67

List of Tables

2.1 Related Work Summary 12

3.1 FaultSee features 27

List of Figures

3.1 Overview of the FaultSee experiment lifecycle. 15
3.2 Overview of the FaultSee system. Dotted arrows represent produced logs or metrics, while normal arrows represent control messages. 17
3.3 Service instances throughout time plot 20
3.4 Experiment logs displayed in Dashboard 20
3.5 Dashboard's Filter System 21
3.6 Example of number of outgoing packets during an experiment 21

4.1 Percentage of CPU usage in the test 31
4.2 Number of containers running throughout the Cassandra experiment, in the scenario of killing a node 33
4.3 Average number of operations performed by YCSB-Run Clients throughout the Cassandra experiment, in the scenario of killing a node 34
4.4 Number of containers running throughout the Cassandra experiment, in the CPU exhaustion scenario 35
4.5 Average number of operations performed by YCSB-Run Clients throughout the Cassandra experiment, in the CPU exhaustion scenario 35
4.6 Number of containers running throughout the BFT-Smart experiment, in the faultless scenario 36
4.7 Average number of operations per second performed by the YCSB Client throughout the BFT-Smart experiment, in the faultless scenario 37
4.8 Outgoing network usage for every host, in the faultless scenario 37
4.9 Number of containers running throughout the BFT-Smart experiment, in the CPU exhaustion scenario 38
4.10 Average number of operations per second performed by YCSB Clients throughout the BFT-Smart experiment, in the CPU exhaustion scenario 38
4.11 CPU percentage usage for every host, in the CPU exhaustion scenario 38

4.12 Number of containers running throughout the BFT-Smart experiment, in the kill a node scenario 39
4.13 Average number of operations per second performed by YCSB Clients throughout the BFT-Smart experiment, in the kill a node scenario 39

Chapter 1

Introduction

Technology is taking a central role in modern society. In developed countries, everywhere we look we can see the presence of software. An examples is the software that you are using to read this document or that was used to print it. Technology is also used to support enterprise systems, to process transactions, or in even more critical applications, like the software that helps emergency responders control all active operations and receive new distress calls. Often, applications are running in a global scale with availability requirements close to 100%, running in large data centers, usually referred to as the "Cloud". To handle the global scale, distributed systems are increasingly in the total number of components. Achieving high levels of availability requires quality-focused development processes with rigorous and thorough testing. Distributed systems, due to having several components, are quite difficult to evaluate and test in a systematic and reproducible manner. Furthermore, software is faulty because it is developed by humans, and because the environment is unpredictable. Faults can lead to catastrophes, especially when dealing with critical applications. For example, last July, the cloud provider Cloudflare had an outage that impacted millions worldwide, due to a bad software deployment that was not properly tested [1].

To test software, developers must be able to introduce bad inputs to check how the program will react. In a perfect world, developers will have considered, and tested, all possible inputs, so that all the logic the programmer has represented into his code can be tested, leading them to assert, with confidence that their software is bug-free. However, in most cases, the input space is very large and it is not possible to achieve sufficient test coverage.

Throughout the years we have seen a bigger investment from the community to build tools with the objective of helping developers create software faster and with fewer bugs, for example JUnit, a Java framework to automate tests. There are many areas where this has been happening, such as Integrated Development Environment (IDE), where nowadays multiple IDE exist for

the most popular languages. Continuous Integration (CI) and Continuous Development (CD) allows developers to decrease the time elapsed between writing the code and its deployment to test and production environments, automating the validation steps in the test phase. However, as the problems presented to developers grow, so does the complexity of the respective solution, and the existing tools may no longer be sufficient. Unfortunately, when dealing with multiple components a new type of errors appears: *faults in components*, such as a disk failing or power outage. If not accounted for, a single machine crash can lead to catastrophic results. As the number of possible faults increases, so does the strain on developers to be able to test all possible combinations that could lead to an error. To be able to effectively test components failures developers need better tools, which represents a good opportunity for new contributions, such as our work.

When the code is properly tested the number of error decreases. Nagappan et al. [2] studied how Test Driven Development (TDD) improved code quality in four industrial teams. They showed that the number of errors present in the code decreased between 40% and 90%. George and Williams [3] show that TDD created code that passed 18% more functional black-box tests.

Components failure can be so severe that they result in long downtime or even lead to data loss. Therefore, it is important to validate how systems will react in the presence of failures, to avoid, or at least mitigate, the consequences of components failure. The academic community is actively working to build more resilient systems and has developed several fault-tolerant techniques over the years [4]. However, the community has yet to build the necessary tools to properly test their systems, using these techniques. There are two main challenges to be able to inject faults into systems: deploy the system into a running configuration, and inject faults when required. Docker-Swarm and Kubernetes are two tools that automate the deployment of complex distributed architectures into several physical hosts. Additionally, these tools reduce the time required to deploy new versions of software, thus they help developers to iterate versions of their software faster. Nevertheless, they lack the ability to inject faults. In this work, we introduce *FaultSee*, a tool that simulates components failures patterns, and helps developers mimic faults that might affect their systems in production.

Reproducibility of results is important. In the Academia, reproducibility allows researchers to validate claims made by other researchers. When someone claims they designed a new algorithm that is one order of magnitude faster than the state of the art other researchers will want to validate said claims. Docker-Swarm and Kubernetes enable researchers to create configuration files that allow other researchers to easily deploy the same system, however, as these tools lack fault injection capabilities, researchers need to inject faults manually, which hinders repro-

ducibility. Additionally, it removes the possibility for the creation of automated benchmarks.

Reproducibility is also very relevant to the industry, whenever there is an error that developers need to correct, first they must identify what is the cause for the error, and then correct it. This process is faster if developers are able to reproduce the causes of the error. Furthermore, when the error is fixed, developers can then reproduce the causes of errors to validate this no longer affects their systems.

Depending on the environment, an error may or may not lead to a failure. For example, different implementations of the same software library in different operating systems can have distinct results. Docker containers precisely describe the environment in which the software will run, enabling the developer to control the environment. Combined with the ability to inject faults in specific nodes at precise moments this enables developers to create reproducible results that are easier to analyse.

As shown in this section, there is a need for a tool that enables fault injection automation, in order to study the behaviour of systems when a failure occurs. This opens a good opportunity to combine the deployment flexibility of tools such as Docker Swarm, with the need to have a judicious evaluation platform that is able to subject distributed application to faults.

1.1 Requirements

We identified the opportunity to create a new tool, that helps developers and researchers and these are its main requirements:

- Reproducible experiments - The core feature of the tool is its ability to reproduce previous experiments in the same testing environment, and also in different environments from different users;
  - Automatically deploy the System Under Test (SUT) – the tool must automate the deployment phase, otherwise the manual requirements from its users will hinder the reproducibility;
  - Run experiments – given some configuration files the tool must be able to run the experiment automatically;
  - Inject Faults – we want to test scenarios in which we simulate components failure, as such the tool must be able to inject faults;
  - Gather Metrics – in order to test the SUT performance the system must be able to collect resource usage metrics;

– Produce Plots – plots enable to user to quickly analyse the performance of his system, without parsing through raw text.

1.2 Contributions

The work described in this document enabled us to contribute to advance the state of the art in the following ways:

- FaultSee - A tool that is able to create scenarios that test applications limits and ensure it continues to function correctly under faults or to create benchmark scenarios, to test the performance of various systems under the same faulty scenario;
- FDSL - A language to describe fault injection scenarios, used by FaultSee;
- Scientific publication - "FaultSee: Avaliação Reproduzível de Sistemas Distribuídos Sujeitos a Faltas" in the INFORUM 2019 conference, held in the 5th and 6th of September in Guimarães, Portugal. The paper authors are Miguel Amaral, Miguel L. Pardal and Miguel Matos.

1.3 Thesis Outline

There are five chapters in this document. In Chapter 2 we provide some background required to understand this work and then we describe what is the current state of the art, and what the community has already developed. In Chapter 3, we describe the *FaultSee* design. In Chapter 4, we explain the evaluation conducted, and how we validated the correct behaviour of FaultSee. Finally, in Chapter 5 we conclude the document, draw conclusions from the work done, summarize the main contributions and propose future work to further improve the tool.

Chapter 2

Background and Related Work

In this chapter we provide some background information to the reader, to enable him to comprehend the work described in this document.

In this chapter we provided some background and we present the related work, organised into four different areas: Fault Injection is detailed in Section 2.2, Monitoring/Tracing is described in Section 2.3, Deployment is detailed in Section 2.4 and Interface in Section 2.5.

2.1 Definitions

Computers run software on them, in particular the operating system. From early on, there was a need to create copies of a computer, and its software, for easier management. The copies are called **Virtual Machines**, and they allow the creation of an image that can copied and distributed, always presenting the same behaviour. A computer that runs a virtual machine is called a **host** and many virtual machines can run on the same computer. Virtual machines enable the emulation of the behaviour of a different system, sharing no software with the host, include a complete operating system and as such provide a great degree of isolation. Due to the need to emulate everything, when compared to having no virtualisation, the performance of virtual machines has significant overheads, and, as such, it is not optimal.

Docker [5] is a tool that enables developers to create containers, which are lightweight environments. Containers are similar to virtual machines, however, they do not require the creation of an entire operating system, instead, they share the Linux kernel with the host in which containers are running, thus reducing its size and increasing performance, at the cost of lower isolation between containers and the host itself. A container packs an application, however, it only includes the required components to run the application, such as the libraries and other dependencies. This allows developers to be confident their application will run independently

of the host. Moreover, this enables developers to simulate the production environment in their local system, independently of the operating system they use. **Docker images** are the packages that contain all the information required to create the containers and they can be stored online in public or private registries.

A **Distributed System** is a set of computers that are connected among themselves through a network and work towards a common goal. Every computer on this system is denominated as a **node** of the system. A **Cluster** is a set of computers, generally with the same specifications and geographically close to each other.

Distributed Systems can be deployed in Docker Containers. However, **Docker Containers** technology is only focused on running a single unit on a host. In order to deploy more than one container in multiple hosts an **Orchestrator** can be used. An **Orchestrator** is responsible for managing running, distributing, scaling, and healing a mesh of **Containers** across a collection of **nodes**. Two examples of **orchestrators** are *Docker-Swarm* and *Kubernetes*. A set of **Docker containers** running with the same configurations is a **Docker Service**.

Verissimo and Rodrigues [6] define **failure** as the event of delivering an incorrect service. To deliver an incorrect service at least one of the system's external state deviates from the correct service state. An **error** is the partial state of the system that deviates from the correct state of the system, and may lead to its subsequent service **failure**. A **fault** is the adjudged or hypothesised cause of an **error**.

While analysing the performance of **Distributed Systems**, the resource usage of the system is a relevant information. **Psutils** [7] is a cross-platform library that is used to gather resources usage of the system and details of the running processes.

Reproducibility is a property that characterises the ability to recreate results using the same methodology described by someone else. **Distributed Systems** performance claims must be **reproducible**, otherwise they pose no value, as no one can validate them.

When developing **Distributed Systems**, there are two important properties to take into account: **safety** and **liveness**. **Safety** states that no wrong output will ever come from the system. **Liveness** states that eventually the system will output a correct response. However, it is important to note that no system can simultaneously fulfil both properties to the maximum. If high **safety** is required, then the **liveness** property must be relaxed. On the other hand, if high **liveness** is required, then the **safety** properties must be lowered.

2.2 Fault Injection

Fault injection is a technique used to test code paths that are rarely followed because they correspond to rare occurrences, such as handling component failures. Thus, by using fault injection, developers can create tests for rare, but often critical, paths of the developed software. As an example, in distributed systems, it is crucial to be able to test how the system will react when one node crashes, so that when it happens in a live system, developers can assert that the system will behave as expected and fulfil its safety and liveness properties.

We start by introducing some key concepts and then we discuss the state of the art in this area.

Verissimo and Rodrigues [6] define **failure** as the event of delivering an incorrect service, and are caused by **errors**. An **error** is the partial state of the system that deviates from the correct state. A **fault** is the adjudged or hypothesised cause of an **error**.

Natella et al. [4] identify three fault injection approaches: Injection of Data Errors, through the corruption of memory or registers, Error Injection Interface, that is, invalid input and or output, and Injection of code changes, this technique injects code that mimics the most common bugs.

Gunawi et al. [8] introduce a new fault type, **fail-slow**, which corresponds to components running with degraded performance. The author claims that current systems are robust at logging and recovering from **fail-stop** faults, i.e. nodes crashing or network partitions, however, systems are not yet prepared to deal with fail-slow faults, urging the community to built better systems prepared for this kind of faults.

A **fault model** represents the faults a distributed system designer needs to take into account when designing the system. From the model, the designer can predict the consequences each particular fault and design the system to be resilient to it.

To motivate its engineers to build more fault tolerant tools, Netflix created the SimianArmy [9] set of tools. These tools can inject faults, with a given probability, into production systems on a business day. The motivation behind this tool is that systems will eventually fail, developers just do not know when. By ensuring engineers that components will fail on a daily basis, this tool provides them with extra motivation to build systems that are more robust to faults. Moreover, with the iterative lifecycle of software development, some faults might be introduced in a system by human error, laying undetected until the worse possible moment. SimianArmy ensures the faults will become failures when engineers are most prepared to deal with any issue that may arise. SimianArmy comprises a set of tools, built with fault tolerance as its primary focus:

- Chaos Monkey - The original tool, responsible for randomly terminating virtual instances

(fail-stop fault);

- Chaos Gorilla - Tests problems related with an availability zone[1]. It supports two distinct modes, terminating all instances in an availability zone or creating a network partition, in which the instances inside that zone are unable to communicate or be reached by services hosted outside that availability zone (fail-stop fault);

- Chaos Kong - Enables developers with terminating all services in an entire region[1] (fail-stop fault);

- Latency Monkey -Simulates partly healthy instances, which increase latency in requests (fail-slow fault).

The SimianArmy represents a step towards fault tolerance testing, however, it is intended to ensure that a production system is resilient against faults, rather than helping developers at a debugging phase.

Pumba [10] is another tool developed to inject faults into running systems. After a system is deployed, the user is able to inject faults into Docker containers by running the desired command in the console line. This tool supports both Docker-Swarm and Kubernetes, however it does not support running scripted experiments, the user has to manually run its experiences.

Cords [11] is a framework developed with the purpose of injecting faults in file systems in order to test distributed applications. Ganesan et al. [11] show that redundancy, in fact, does not imply fault tolerance by uncovering a series of bugs in eight popular distributed file system. This shows the need for incorporating better mechanisms into the development lifecycle to test distributed applications, as even applications with large communities have dormant bugs. The team identifies two problems related with file systems, blocks being inaccessible and corrupted data.

LSDSuite [12] is another framework built to help developers test distributed systems. It focuses on automating the deployment of Docker containers on several hosts. LSDSuite allows developers to schedule fault injection into the SUT. However, LSDSuite only allows developers to kill or gracefully shutdown nodes in the system, instead of supporting a wide range of different faults. Moreover, a central node requests faults to be injected as the experiment "goes down" Moreover, the central node sends a network request instructing to inject a fault, which introduces network latency, as such some faults may be injected with significant delay. Both

[1]Geographically AWS has its infrastructure divided into separated locations. These locations are composed of regions and availability zones. Regions are geographically wide distant locations. In every given region AWS has several, isolated, availability zones.

these issues present an opportunity to improve this framework. FaultSee extends this framework to implement new functionalities.

2.3 Monitoring/Tracing

A **tracer** [13] is a component that intercepts application code to record timestamped events. This analysis can lead to the identification of causal paths, that is, which request triggered other requests. Patterns emerge by studying the relation between requests, as the same functions are executed more than once.

Dapper [14] is a system used by Google to monitor its production systems. It was developed with scalability, low overhead and application-level transparency as requirements. The Dapper team showed that through sampling of requests, it is still possible to find patterns in the application, even uncommon ones. One possible improvement the author refers is the ability to sample with fine-grain control over the percentage of logs collected over time. This does not pose a problem with services that have a steady usage. However, when services experience low traffic, if Dapper keeps processing the same volume of data, it is possible to gather a higher percentage of produced logs without impacting performance. A solution would be to provide users with the ability to choose the amount of data to be gathered over a time window, such as ten seconds or one minute, instead of a static percentage. The system enables developers to access information in a small window of time, however, the tool still requires 10 minutes for data to be ready to be processed.

Sherlock [15] is a system that assists IT administrators in detecting systems that are performing poorly. When a system is having issues, it identifies the set of components most probable to be failing. The system analyses packets sent by hosts, routers and links to gather information. Using a multi-level probabilistic model it then automatically infers dependencies between services. Sherlock is able to infer dependencies in a few hours on a business day. Software often has many redundancies, as such, it is common to have multiple instances running the same code. Sherlock leverages this fact to infer dependencies even when information is missing. By aggregating network packets from multiple instances and analysing them as a whole, the system is then able to eliminate false positives, i.e., cases when Sherlock reports a faulty server when it is not.

Aguilera et al. [16] also identify *causal paths*, to identify the source of latency in distributed systems. However, the developed tool is not intended for real-time debugging, as the tool still needs to process data before presenting it to its users. The authors used two techniques to gather the traces, packet sniffing and port mirroring, i.e., copy all traffic to another host.

WAP5 [17] also gathers network traces, to infer causal paths. The system uses checksums to detect duplicate messages. The system identifies patterns even when different hosts are executing different roles in a request, as they have the same code. Additionally, an interpolation library was used, but it requires applying a wrapper around some functions. It does, however, have better efficiency than packet sniffing, messages are logged in the same order applications see them and messages are attributed to processes rather than hosts.

Nagios[18] is an open-source tool that monitors infrastructure, networks and systems. Nagios ensures the infrastructure is in a correct, pre-determined, state, executing corrective actions if configured. Nagios can also alert the system maintainers as the system deviates from its intended state.

These tools were built so that their could monitor their applications, either to detect malfunctions or detect the origin of problems. FaultSee is a tool that is built to help users ensure theirs systems are resilient, therefore it monitors the systems state so the user can analyse them.

2.4 Deployment

When debugging distributed systems, the creation and destruction of hosts in mass quickly poses a challenge, if not automated efficiently, as it is a cumbersome repetitive task.

FEX [19] is a framework that aims at running benchmarks, taking care of the whole life-cycle: deploy, run and plot results. The system leverages Docker to deploy similar nodes of a system in a host. Additionally, Docker ensures better reproducibility of statements made by users. This is achieved because other researchers can replicate original docker images, therefore they can repeat the experiment in the same conditions.

The core objective of the SPLAY [20] system is to enable users to deploy a distributed system in a testbed with the same ease as in a personal computer. It enables the user to deploy in a mixed type of hosts, personal computer, workstations and testbeds, providing resource isolation. Apart from the database, which is a single node, the framework is scalable as its controller can run in one or more nodes. However all code must be written in Lua[2], and all logs must be written directly in the code.

Dfuntest [21] is a framework developed with the intent to automate experiments with distributed systems. It allows the execution to be done in a single host or in a testbed. It makes use of a centralised host to orchestrate tests, therefore cannot scale indefinitely. Nevertheless, it allows a user to interact with the system while it is being tested.

As stated before, in Section 2.2 LSDSuite [12] focuses on automating the deployment of

[2]lightweight, multi-paradigm programming language designed primarily for embedded use in applications

Docker containers on several hosts. Docker containers enable developers to use any programming language, thus it is flexible. This framework allows developers to add nodes to the SUT according to a scripted schedule. Combined with fault injection capabilities this enables developers to create experiments in order to test how their system reacts when a node crashes. However, this framework can still be improved. Monitoring is only available to the developer at the end of the experiment, and the only logs available to the users is the standard output created by all the nodes during the experience. Moreover, this framework does not support any visualization tool, which allows the user to create a better mental image of what is happening in the SUT throughout the experiment.

FaultSee runs experiments to simulate faults in the system. In order to be able to inject the faults into a running system, first the system must be deployed, FaultSee extends LSDSuite and leverages Docker-Swarm and Docker containers to deploy the SUT.

2.5 Usability

One factor that increases any tool adoption is how easy users can use the tool to produce results. Therefore it is crucial for any system that aims to test software to have a good, intuitive user-interface.

Dashboards enable IT teams to detect errors when a plot depicting the throughput of an application is made available. Additionally, a plot showing outbound packets of a virtual machine can alert IT teams to errors in the system when a sudden spike in traffic occurs. On the other hand, when IT teams only have access to text logs they lose the ability to detect anomalies with a simple glimpse of the dashboard, or the ability to easily compare the variance of a given metric over time, such as outbound packets.

Perfume [22] is a tool that transforms raw text logs into graphs. The graphs represent all possible paths users experience in the application. In a study led by the authors, they show that when using Perfume, instead of parsing raw logs, developers answered questions 15 % quicker and 8.3 % more correctly. This result supports the claim that when using a graphical user interface developers can achieve better results.

Tseitlin [9] also shows that monitoring what is happening is important for testing. Faults in a production system impact customers, therefore the team responsible for the system must be able to know when a system is having technical difficulties in order to avoid further weakening an already unhealthy system with automated fault injection.

The Sherlock system presented in Section 2.3, is able to only alert IT managers when a failure that affects end-user exists, rather than a simple fault. The authors refer that other tools, such

	Fault Injection	Monitoring/Tracing	Deployment	Usability
SimianArmy	✓			
Pumba	✓			
Cords	✓			
Dapper		✓		
Sherlock		✓		✓
WAP5		✓		
Nagios		✓		
FEX			✓	✓
SPLAY			✓	
Dfuntest			✓	
Perfume				✓
LSDSuite	✓		✓	
FaultSee	✓	✓	✓	✓

Table 2.1: Related Work Summary

as NAGIOS [18], send too many false alerts to the IT team, thus creating an unnecessary burden on the team responsible for the system.

In order to increase the value of the produced results, FaultSee includes a dashboard, so the user can automatically have plots with its system performance. Additionally, FaultSee leverages Docker-Swarm, a tool widely used by the community, which decreases the users learning curve.

2.6 Overview

In Table 2.1 we present a summary of the tools discussed in this chapter. Each line represents a tool and each column a feature. No tool exists that combines all features discussed in this section, therefore in Chapter 3 we demonstrate the key features of FaultSee, a system that addresses all these issues.

The research community has been actively working in providing tools that help programmers build better and more dependable software, however, there is still room for improvement. Docker Swarm and Kubernetes are tools that enable developers to easily deploy complex architectures. Dapper, Sherlock and WAP5 focus on monitoring a system. Perfume provides a graphical user interface to help developers understand the root cause of any issue that may be the bottleneck of the system. The SimianArmy and Pumba are two tools designed with fault injection at its core, they enable developers to test their software against components failure.

Chapter 3

FaultSee

In this chapter we describe the design and implementation of *FaultSee*. In Section 3.1 we present an overview of the requirements, and explain experiments lifecycle. In Section 3.2, we describe the FaultSee architecture and specify the FDSL language. In Section 3.3, we present the reasons behind the decisions taken, and then, in Section 3.4, we discuss the main features of FaultSee and how they address the requirements of this work.

3.1 Approach

One of the goals of our work is to improve the development and evolution of distributed systems. With this in mind, the goal of FaultSee is to enable the user to test the System under Test (SUT) under workloads with faults, described in configuration files. A FaultSee's user is able to launch a complex distributed system automatically, inject faults, and analyse the experimental results. FaultSee takes care of deployment, injecting faults, monitoring and producing plots with the results.

The other goal is to facilitate the creation of reproducible experiments. As we want to achieve reproducibility, we decided that the SUT nodes have to use Docker containers. Docker containers allow software to be packaged together with all its dependencies, and also to create different versions of the same application. All versions of an image can be published to public repositories, as such two users can easily run the same version of a software and replicate results.

3.1.1 Requirements

FaultSee has the following requirements: deployment, injecting faults, monitor and plot results.

There is a central component of the system that is responsible for deploying the whole SUT.

To inject faults we decided it is crucial to have a local component running on each host of

the cluster, so that FaultSee can inject faults without any network latency. Network latency introduces delay between the user specification and FaultSee execution. Furthermore, when injecting a fault in multiple nodes, the delay is different for every node in the cluster, which increases the error.

Additionally, to inject faults simultaneously in several nodes, clock synchronisation is required, which is simpler to achieve with a local component in each host. Experiments can be long, which can cause the clock to skew. As such, clocks are synchronised before every experiment, in order to mitigate clock desynchronisation errors.

During the experiment, the component running in each host will monitor the hosts' resource usage. The information monitored by FaultSee could either be transfered throughout the experiment or at the end. Having the resource usage in real time is interesting, as it enables the user to interact with an experiment in real time. However, it also introduces overhead in the network, therefore tainting the results. As such, we decided this information is gathered and saved locally during the experiment, and only assembled together in the end of the experiment.

We decided to enable the user to have some random options in the experiments, for example: kill a random node of the SUT. In order to be able to always kill the same node, so that experiments are reproducible, every experiment has a numeric seed that is fed to the random number generator used to decide the random values in the experiment, and we provide the user with the ability to control this seed.

Another FaultSee key feature is the ability to inject any custom fault into the system, that way the user can use FaultSee to inject any faults beyond the ones supported by default, enriching the experiments possible with this system.

In the end of the experiments the results are merged from all the hosts, processed and then they can be displayed in a dashboard.

The most important requirements for FaultSee are:

- Synchronise clocks of all components of the system;
- Be able to parse configuration files (architecture and experiment events);
- Deploy the SUT;
- Inject faults into the SUT's Docker containers;
- produce telemetry regarding all Docker containers;
- Track the experiment progress;
- Send all logs at the end of an experiment to the Master Controller;

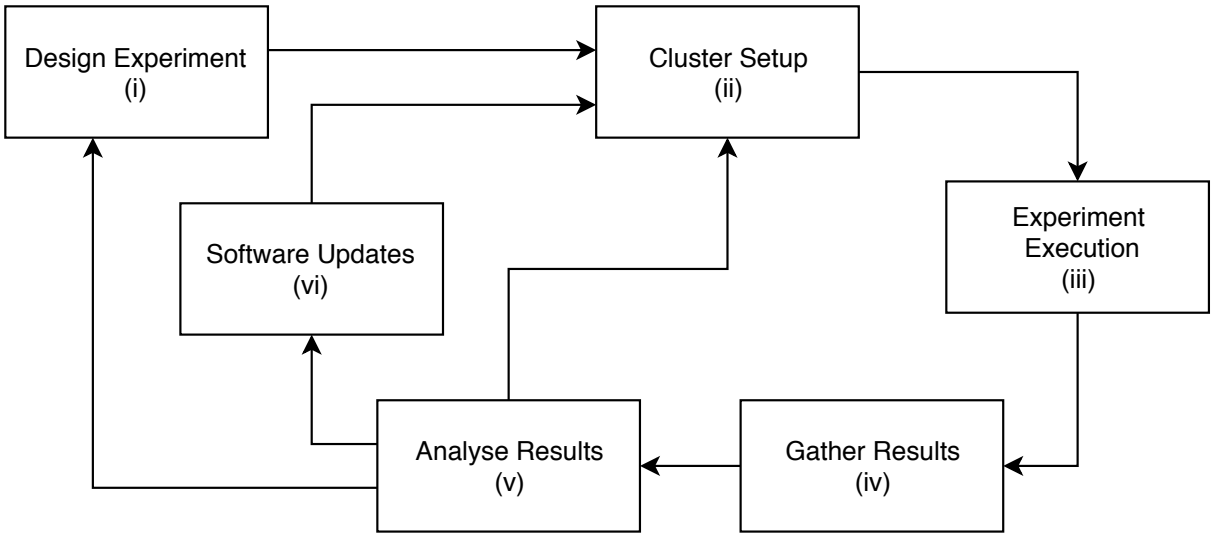

Figure 3.1: Overview of the FaultSee experiment lifecycle.

- Process logs at the end of the experiment;
- Display experiment logs in Dashboard;
- Plot resource usage in Dashboard.

3.1.2 Experiment Lifecycle

In this section, we detail an example of how FaultSee is used and the several stages that comprise an experiment.

Prior to executing any experiment, the user realises that he has the need to understand how the system he is developing behaves when a failure occurs. To study his system behaviour, he decides to use FaultSee. In Figure 3.1, we can see an experiment lifecycle, that has the following stages:

(i) **Design Experiment** - Then, the user needs to design an experiment, and create the FDSL files that describe the experiment he has in mind.
Additionally, the user can also share his configuration files with another user, and the second one is able to reproduce the same experience.

(ii) **Cluster Setup** - Before starting the experiment, the users needs to setup the cluster that FaultSee will use to run the experiment, by detailing in a configuration file the nodes IP and authentication configurations.

(iii) **Experiment Execution** - As soon as FaultSee has access to the cluster and the experiment configuration files, the user can start the experiment. FaultSee runs the experiment autonomously, while outputting its progress for the user to see.

(iv) **Gather Results** - After the experiment is complete, FaultSee gathers the logs from all the nodes in the system and merges them into a single file, so that the user can analyse them raw. FaultSee also informs the user if the experiment ran with success or if there was an error.

(v) **Analyse Results** - Once FaultSee has completed the previous stage, the user can check the dashboard to study the experimental results, that is, observe the resource usage plots and the experiment logs, filtered by service or container.
As reproducibility is one of FaultSee's core requirements, the user is able to replay all the information at the dashboard on a later date, to study the experimental results in detail.
If the user detects any problem with the experiment, he can always iterate the experiment or design a new one.

(vi) **Software Updates** - Let us suppose that, during the analysis, the user finds some relevant information that leads to the discovery of a module with performance issues. After applying a patch to the software, the user can now run the same experience, with the new patch. Then the user can compare the experimental results before and after the software patch, and calculate performance gains.

3.1.3 Experiment Configuration

We can divide the information required to run an experiment in two main categories: the experiment information (that is, the events in the experiment and experiment properties) and the characteristics of the SUT (that is, the nodes information, how to deploy them, any network restrictions, network topology and resources constraints).

The experiment language is denominated FDSL and is described in detail in Section 3.2.2. Its key concepts is that it describes the experiment properties and the experiment events throughout time.

The SUT characteristics are described in a Docker Compose language. Its key concepts is that it describes Docker services, that is a set of containers running the same configuration, configurations, that describe the containers environment.

3.2 Design

In this section, we describe the two components designed, the platform: FaultSee, which architecture is described in Section 3.2.1, and the experiment language: FDSL, described in Section 3.2.2.

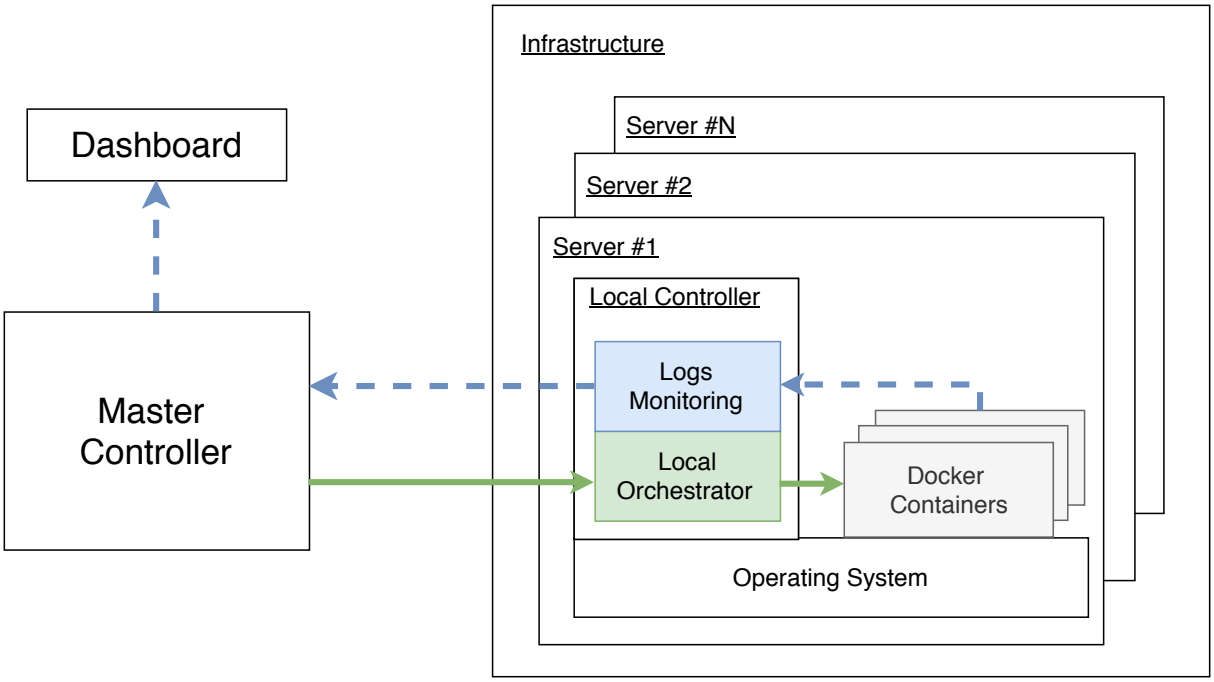

Figure 3.2: Overview of the FaultSee system. Dotted arrows represent produced logs or metrics, while normal arrows represent control messages.

3.2.1 FaultSee Architecture

FaultSee has four main components. These are the Master Controller, Dashboard, Infrastructure and Local Controller.

The LSDSuite framework, described in Chapter 2, already tackles the deployment of an experiment and enables a user to schedule churn models. As such we decided to extend this framework and create FaultSee.

In Figure 3.2 we can see the whole system and the interactions each component will have when an experiment is being executed.

Infrastructure

To run an experiment there must exist an infrastructure. The user must have access to a cluster. The cluster access is described in a configuration file, where the user details the hosts authentication information for remote access. FaultSee supports two authentication mechanisms: password authentication and public key authentication.

Master Controller

This is the main component of the system and it is responsible to orchestrate the whole experiment. To run experiments, the user only needs to interact with this component.

While preparing an experiment, this module will transmit the scripts to all the hosts in the cluster, in order to be possible to trigger faults in the same moment throughout the components of the SUT without compromising scalability. Due to Docker-Swarm constraints, this is the only node with the ability to launch new nodes of the SUT. Therefore, the Master Controller is responsible for deploying the SUT nodes in the available infrastructure and preparing the network. Before the experiment starts, the initial nodes of the SUT are deployed.

However, to avoid injecting the same fault in two distinct moments, clocks need to be synchronised. As such, before starting any experiment, this module synchronises all clocks using NTP. Finally, it coordinates the starting moment with all Local Controllers, so that all hosts start the experiment at the same time.

As the Master Controller is the component which the user uses to interact with the system, this component also gives a visual progress of the current status of the experiment.

During the experiment, log messages are created with a timestamp, which is fetched from each host clock. At the end of the experiment, FaultSee merges and sorts chronologically all log messages.

Local Controller

As depicted in Figure 3.2, the Local Controller runs in every host of the cluster.

The Local Controller has many responsibilities. At the beginning of the experiment, it receives the experiment script, parses it thoroughly, and sets all the internal control variables so that it can inject the faults in the correct moments.

During the experiment, the Local Controller is responsible to inject any faults described in the fault configurations. Additionally, this module is also responsible for collecting logs. During the experiment, FaultSee produces generic metrics, not only per user container but also per infrastructure host. These will include CPU usage, Memory usage, disk usage and Network usage.

At the end of the experiment, the Local Controller will then send all SUT logs to the Master Controller, so they can be consulted by the user. This is not performed during the experiment due to performance reasons, as it would clog the network and negatively impact the results of the experiment.

Below are the faults natively supported by the Local Controller:

- CPU-intensive tasks - Exhaust CPU of the container;
- Kill nodes - Kill a container;

- Send Signals - Send a signal to the container;
- Custom Faults - Run any script inside the container.

These faults were chosen as a proof of concept, by supporting custom faults FaultSee can be used to inject any fault the user may require, as it can run any script the user develops. The other faults were developed as they are generic faults that can be used in every experiment. The faults implementation details are described in the Section 3.3.

Dashboard

The Dashboard displays visual information to the user. After the Local Controller processes the logs of the experiment, smaller files are created with the logs. This way, when the Dashboard is loading telemetry, it only needs to load smaller files, and therefore being more performant. Nevertheless, longer experiments will produce bigger files, leading to longer times for the dashboard to complete.

There are five files:

- **Container Information** - This file holds identification information about the containers that ran in an experiment, such as its full id, the service the container belonged to and the slot inside the service.
- **Container Events** - This file includes the information about the lifecycle of containers, such as when they started and stopped.
- **Container Logs** - This file holds all the applicational logs produced by the containers of the SUT
- **Container Stats** - This file has the metrics produced during the experiment about the resource usage of each container.
- **Hosts Stats** - Like the previous file, this file holds metrics of the experiment resource usage, however this metrics are stored by host in the experiment.

The *Container Information* file allows the Dashboard to provide additional information about the containers information stored in the other files.

The *Containers Events* enables the Dashboard to display a plot that represents the number of containers in each service throughout the experiment, one example is provided in Figure 3.3

The *Containers Logs* file enables the Dashboard to display all logs produced by the containers during the experiment. We can see a Dashboard screenshot that shows how this information is

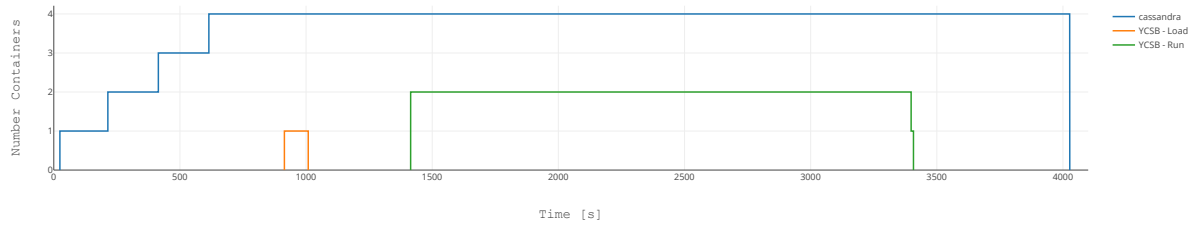

Figure 3.3: Service instances throughout time plot

displayed to the user in Figure 3.4. Additionally, the Dashboard enables the user to filter the logs by service and container, as seen in Figure 3.5. In the Figure, we can see that containers of service cassandra number 1 and 4 are shown, while the containers 2 and 3 are disabled. Additionally, the user can activate or disable all the containers in the service by clicking in one of the upper buttons.

946.221626116	8b1b91063fcb	32 sec: 39928 operations; 65 current ops/sec; [READ AverageLatency(us)=1533.85] [UPDATE AverageLatency(us)=355221.16]
947.222100116	8b1b91063fcb	33 sec: 41985 operations; 2050.85 current ops/sec; [READ AverageLatency(us)=1821.1] [UPDATE AverageLatency(us)=6068.78]
948.222738116	8b1b91063fcb	34 sec: 44541 operations; 2553.45 current ops/sec; [READ AverageLatency(us)=1603.77] [UPDATE AverageLatency(us)=4582.25]
949.223503116	8b1b91063fcb	35 sec: 44842 operations; 300.7 current ops/sec; [READ AverageLatency(us)=1614.65] [UPDATE AverageLatency(us)=4238.67]
950.223910116	8b1b91063fcb	36 sec: 45713 operations; 871 current ops/sec; [READ AverageLatency(us)=1647.33] [UPDATE AverageLatency(us)=31334.25]
951.224328116	8b1b91063fcb	37 sec: 48130 operations; 2414.59 current ops/sec; [READ AverageLatency(us)=1764.46] [UPDATE AverageLatency(us)=5254.74]
952.224952116	8b1b91063fcb	38 sec: 49961 operations; 1831 current ops/sec; [READ AverageLatency(us)=1646.56] [UPDATE AverageLatency(us)=4570.87]

Figure 3.4: Experiment logs displayed in Dashboard

The files *Container Stats* and *Hosts Stats* enable the Dashboard to display plots with the resource usage throughout the experiment for each container and host. The plotted metrics are CPU, memory, and network usage.

In Figure 3.6 we can see an example of a plot of the number of outgoing packets during the experiment, for each container.

In order to extend the Dashboard capabilities, we decided to develop it with the possibility to easily extend it, the users can develop plugins to parse the logs of the Docker Services to produce custom application metrics, such as request throughput and latency throughout time.

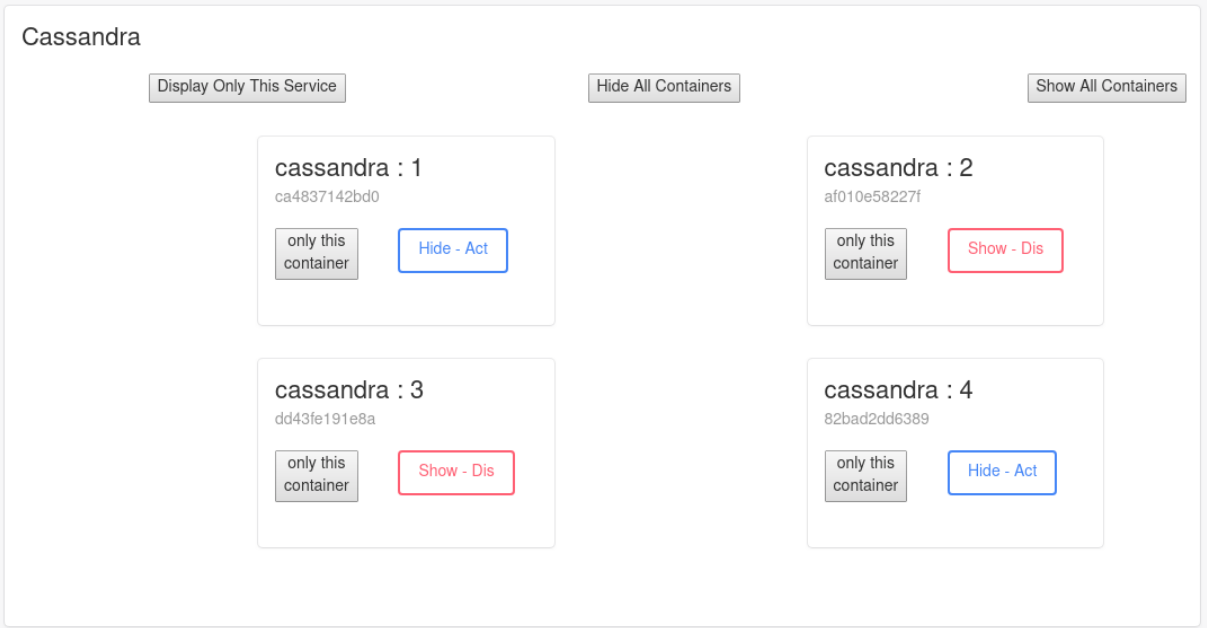

Figure 3.5: Dashboard's Filter System

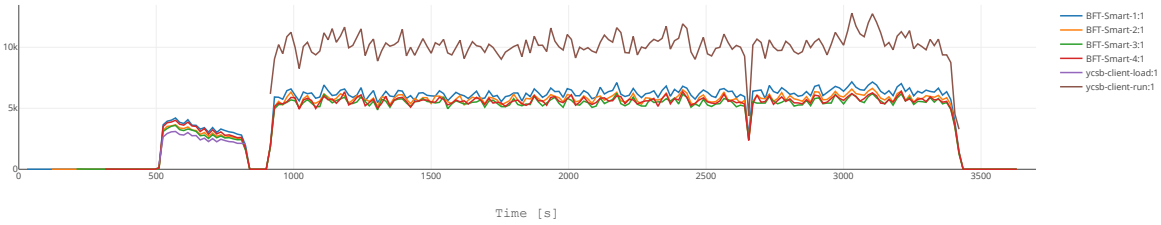

Figure 3.6: Example of number of outgoing packets during an experiment

3.2.2 FaultSee Domain System Language

The FaultSee Domain System Language (FDSL) contains two main sections, **environment** and **events**. **Environment** defines the resources required for the experiment and its configuration options that are specific to each experiment and **Events** describes the experiment throughout time, as show in Example 3.1.

```
1 environment:
2       seed: seed_value
3       ntp_server: URL
4 events:
5       - event_1
6       - ...
```

```
7       - event_N
```

Code Example 3.1: FDSL main structure

Currently, there are two properties that FaultSee recognises, the experiment seed and the NTP[1] server to be used when synchronising clocks. However, if the need arises, FDSL can easily be extended to include other properties.

Events describes the experiment timeline. There are three types of events: **beginning**, **end** and **moment**. The **beginning** event describes the initial state of the experiment. For each service of the experiment, it defines the number of initial containers. The **end** event indicates when the experiment ends. **Moments** describe a point in time in which something happens in the experiment, in Example 3.2 we can see its structure, which has three components, time, mark and the services. Time represents the moment in seconds, mark is the message to appear in logs when this moment is processed and services holds the **Service Events** for every service in the experiment. Services are the Docker Services running in the experiment, that is, the set of docker containers running the same configurations.

```
1  - moment:
2      time: number_of_seconds
3      mark: custom_message
4      services:
5        service_name_1:
6            - service_event_1
7            - ...
8            - service_event_N
9
10       service_name_N:
11           - service_event_1
12           - ...
13           - service_event_N
```

Code Example 3.2: **Moment** structure

During an experiment there can be three types of **Service Events**: **start**, **stop** and **fault**. **Start** adds nodes to a service and **Stop** gracefully removes nodes. **Fault** is a service event that

[1]RFC 1059 - Network Time Protocol (NTP) provides the mechanisms to synchronise time and coordinate time distribution in a large, diverse internet operating at rates from mundane to lightwave. It uses a returnable-time design in which a distributed subnet of time servers operating in a self-organizing, hierarchical master-slave configuration synchronises logical clocks within the subnet and to national time standards via wire or radio. The servers can also redistribute reference time via local routing algorithms and time daemons.

allows to inject abnormal behaviour into the SUT. All **faults** have a common structure, depicted in Example 3.3. This structure includes the *target* and the *fault_type*. The target can be set to amount, percentage or specific, these options are mutually exclusive. The first is an absolute quantity, picked randomly, the *percentage* also acts upon an absolute quantity randomly, this value is calculated as the round up percentage of the expected healthy containers in the service, while *specific* is the indication of the exact containers that are affected.

```
1  - fault:
2      target:
3          # amount, percentage and specific
4          #   are mutually exclusive
5          amount: amount_number
6          percentage: percentage_number
7          specific:
8              - ID_1
9              - ...
10             - ID_N
11     fault_type
```

Code Example 3.3: **Fault** structure

Currently there are three types of **faults**, covering the fundamental *fault types* that are relevant to distributed systems in operations: **CPU exhaustion**, **KILL containers**, and **Send a Signal** to the container. Additionally, there is the **Custom Fault** that is extensible with more specific behaviours. It enables the user to run a script inside the containers, using any executable available inside the container, with any arguments. This allows the user to do everything he may need. In Example 3.4 we can see the structure, which includes six fields. *kills_containers* informs FaultSee if this fault kills the container or not. *fault_file_name* is the script filename, while *fault_file_folder* is the path inside the container where the script is located, FaultSee, by default mounts, faults scripts in */usr/lib/faultsee/*, therefore, if this field is omitted, this is the default value. Nevertheless, the user may create a container with a fault in a different path. *fault_script_arguments* stores the scripts arguments, if omitted this field is empty. By default FaultSee injects the fault by executing the script with */bin/bash* without any argument, however, the user may specify a different *executable* or set its arguments in *executable_arguments*.

```
1  custom:
2          kills_container: yes / no
3          fault_file_name: fault_file_name
```

```
4         fault_file_folder: fault_file_folder
5           # default - /usr/lib/faultsee/
6         fault_script_arguments:
7           # default - empty array
8             - arg_1
9             - ...
10            - arg_N
11        executable: executable
12          # default - /bin/sh
13        executable_arguments:
14          # default - empty array
15            - arg_1
16            - ...
17            - arg_N
```

Code Example 3.4: **Custom Fault** structure

FDSL examples, that we used in the use cases can be found in Appendix A and Appendix B.

3.3 Implementation

Significant work has already been developed in order to easily deploy complex systems. Docker-Swarm and Kubernetes are two technologies that easily manage complex Docker deployments across multiple hosts and locations from one central location. Furthermore, these technologies have high adoption among the community, which contributes to increase ease of adoption of FaultSee. Therefore, instead of creating our own deploy system we decided to use Docker-Swarm. As Docker-Swarm has already defined their own configuration file, we decided to use it instead of creating another one. This creates the additional benefit that previous configurations created for deployment of systems can be directed imported into our system without any additional effort. We chose Docker-Swarm as it has a lower learning curve when compared to Kubernetes. As we decided to use Docker-Swarm, the deployment requirement is fulfilled and no further development is required in this area. Docker Application Programming Interface (API) also has some monitoring capabilities that detail the level of resource usage by each individual container, that we take advantage of. Additionally, we also want to monitor the resource usage of every host in the cluster, therefore we needed another framework, so we used the psutil framework.

Docker-Swarm and Kubernetes could have been chosen to implement the deployment solu-

tion. The system was designed in a way to support both technologies in the future. In order to support Kubernetes as well, we only need to develop a new module that implements the Kubernetes logic, without having to fiddle with the FaultSee core logic.

In order to programmatically integrate with the Docker API, so that we could access its resources, such as deploying nodes or gather resource usage metrics, we were limited to either use its API, producing HTTP requests for every action, or using one of their available SDKs. The SDKs provided by Docker are written in Python and GO. We decided to use Docker SDK as they are actively maintained by Docker, as such both the Local and Master Controller were limited to be implemented in GO or Python. We decided that the Master Controller would be developed in Python, as it is a higher level language, that is easier to program, furthermore there are community built frameworks that enable to process large quantities of data with few lines of code. This decision comes at the expense of some performance, that we considered to be an acceptable trade-off, as this component main workload does not directly impact the experiments. On the other hand, the Local Controller runs on every host of the experiment, as such we want it to be as performant as possible, therefore this component is implemented in GO, which is a language that has better performance, at the cost of extra development effort.

In order to access the various hosts in the experiment `ssh` must be used, either through user-password authentication or by using `RSA` keys. These are the protocols chosen as they are widely available in most operating systems.

Before launching an experiment the Master Controller needs to make sure all the nodes in the cluster have the necessaries images for the experiment, as such all hosts receive a Docker Pull command for every image present in the system. This also makes sure that if any host in the cluster had a previous version of the Docker Container Image then it is updated. This command is sent in parallel to all hosts, so that the time to prepare an experiment does not increase by incrementing the number of hosts in the experiment.

Every host has a local controller running locally, and every local controller knows the whole experiment, which is implemented using a sorted list, that contains the time at which the fault needs to be injected, the target container identification, which is its service name and slot number[2], and the fault itself. The local controllers then iterate this list, sleeping the required time between the current and previous fault, then launching a thread in the background that is responsible for injecting the fault. This way if there are many faults to be injected at the same instant in the same host, the time elapsed between the first and last fault will be significantly inferior.

[2]Container Number inside the Docker Service

However, before injecting the faults, the local controllers need to decide whether or not they are responsible for injecting the current fault. Every fault has a target container, and if the container is running in the host, then the local controller must inject the fault. In order to quickly make a decision the controller must have an updated structure containing the containers running in the host. In order to instantaneously access the information, this structure is implemented using a map of maps. The main map has the service name as a key, and a map as its value, this second map then has the slot number as its key and a boolean that means whether the container is running on the host. By default, this structure starts empty, and the absence of information means the container is not running. This structure can be accessed concurrently, therefore it is protected by read-write locks. To populate this structure the local controller listens to a local Docker API that spawns Docker Events, in particular to the events that register the containers that are started and the ones that are stopped.

Conceptually, faults can either be a signal sent to main container process or a script that runs inside the container. The Docker API supports both things and FaultSee leverages that to inject the faults. Nevertheless, in order for the scripts to be able to be executed they first need to be available inside the container. To achieve this, all faults are copied to every host in the cluster, and then the local folder is mounted into all containers that are started in the context of the experiment. Additionally, the user can also execute scripts that are already in the container, upon the image creation.

The CPU intensive is achieved by creating the hash of a simple string in an infinite loop. After the required seconds the script stops itself. The script runs inside the container, instead of running on the host, in order to simulate the behaviour of a container exhausting the CPU. If the user configures limits to the resources available to each container, then this fault does not affect other containers running in the same host, therefore the user can test how his system will behave when one container starts consuming all the available CPU.

The KILL nodes fault is implemented with sending a SIGKILL signal to the container, and shares the implementation with the Send Signal fault. Both these faults leverage the Docker API that supports sending signals to containers.

The dashboard is a web component, as such we decided to use React JS, as there are many dashboard templates and many resources online that can help with developing it. The user can create plugins in order to create custom plots from the experiment. These plugins are a simple class that must implement a define interface, that only has one function: Process. It will process every line of log produced by the services of the experiment. This function receives as input a line of log, the time it was produced and the service that produced it. As output it is expected

Feature	Implemented	Future Work
Deploy system under test	x	
Support Docker images with credentials		x
Add containers to the experiment	x	
Stop containers in the experiment	x	
Kill containers in the experiment	x	
Exhaust CPU in the experiment	x	
Support injection of custom faults	x	
Pick randomly an amount of containers for fault injection	x	
Pick randomly a percentage of containers for fault injection	x	
Support targeting specific containers for fault injection	x	
Display injected faults logs		x
Monitor containers resource usage	x	
Monitor hosts resource usage	x	
Quantify network communication between nodes		x
Plot resource usage	x	
Plot resource usage in real time		x

Table 3.1: FaultSee features

to produce a list of points. Each point has an X value, Y value and the series name, different series names enables the user to have multiple lines in the same plot. Additionally, for every plugin the user installs, a new plot is displayed.

FDSL is written in YAML. We decided to choose this data format as Docker also uses YAML to describe the containers, therefore the users are already familiar with it. YAML is an extensible format, so we can easily add new elements to the language. While specifying the language we designed it in a way that in every moment the user can inject as many faults as he wants into the available containers.

3.4 Overview

In this chapter we presented FaultSee, a tool that leverages Docker functionalities to enable its users to create reproducible scenarios that emulate components failures. The experimental results are then displayed in a dashboard that creates plots automatically. FaultSee enable users to use custom faults in their experiments and custom plugins for the dashboard. Additionally, we specified FDSL, a language that describes fault scenarios, that can be shared among users, thus making experiments reproducible. In Table 3.1 we can FaultSee features, and whether or not they are implement

Chapter 4

Evaluation

In this chapter, we describe the evaluation of FaultSee. First, in Section 4.1, we show how we validated the FaultSee features implementation. Then, in Section 4.2 we detail experiments we created to demonstrate how we could use FaultSee. Then, in Section 4.3 we discuss the achievements of FaultSee.

4.1 Features

In order to validate that the software works as intended we created a set of simple tests so that we could individually test some features. We then run this tests manually one by one, and observe the behaviour of the system, to ensure the features are implemented. All the described tests can be reproduced.

4.1.1 Test Target Functionality

We support three ways to target containers: pick N random containers, pick a percentage of containers or pick specific containers. In order to test that we decided to create a test in which all the three targets are present, that way we can check if they are implemented correctly. We used the KILL fault and checked exactly what containers were killed. While running the experiment we can see that FaultSee does implement the three targets functionality correctly.

4.1.2 Test Order

The events of an experiment all have a time associated, so that they can then be triggered at the intended moment, however the order in which they are specified in the file is irrelevant. In order to ensure that all events are applied in the correct order we created a simple test, with several unsorted moments, and verified that all the moments were triggered in the correct moment.

4.1.3 Fail on Docker Pull error

All the containers in docker have an associated image, which is stored in a repository. In order to use one image in the experiment the user must specify the image name, which includes its repository location. However, the user may misspell the image name or the cluster's hosts might not have authorisation to access the repository. In these cases the experiment will inevitably fail, as the docker images will not be available for use. As such we created a scenario in which an image does not exist, so that we could ensure the experiment fails fast whenever there is an error downloading a docker image. We verified that FaultSee aborts an experiment when it fails to download one of docker images

4.1.4 Parse

To run a experiment, it first must be specified on the FDSL language, as such it is also important to test that the systems aborts the experiment whenever there are errors. As such we created four set of FDSL experiments that should fail. If the user specified an empty moment, it probably is a typo, as such it is one of the cases we test for. Another typo is if the user details an event after the time the experiment is considered to have ended, or if the user forgot to specify the time of the event. Finally, we also test that if the user is trying to inject a fault into more containers of a service than the ones expected to be alive at that moment the system aborts to start the experiment. We validated that in all four scenarios FaultSee aborts to start the experiments, while warning the user about the error.

4.1.5 Start and Stop Containers

Starting and stopping containers is one of the core features of FaultSee, as such we must test if FaultSee behaves as expected when trying to launch and stop containers.

To test starting containers, we created a simple experiment with several docker services, in which we added containers in two distinct moments, first in one service and then in a different one, and validated the behaviour. This way we asserted that FaultSee launches the correct docker containers.

The stop behaviour is slightly more complex, as the stopped container has ten seconds to stop gracefully, otherwise its process is terminated. In order to be able to test this behaviour we created two distinct docker images. The first is a very simple image that keeps on running until its process is terminated externally and the second listens for the SIGKILL signal so that it can gracefully terminate its own process. This way we validated that the correct container is stopped at the correct moment, and if the containers fails to terminate itself on request, FaultSee

does terminate it abruptly.

4.1.6 CPU Exhaustion

One of the faults FaultSee supports is the CPU exhaustion. In order to test this fault we created an experiment that uses containers without any workload. We then inject the CPU exhaustion fault in three distinct moments with different durations. Then we check the containers resource usage in the dashboard, to ensure the CPU was in fact exhausted. In Figure 4.1 we can observe the results of the test, there were three durations, the first was 20 seconds, followed by 50 seconds and then 200 seconds.

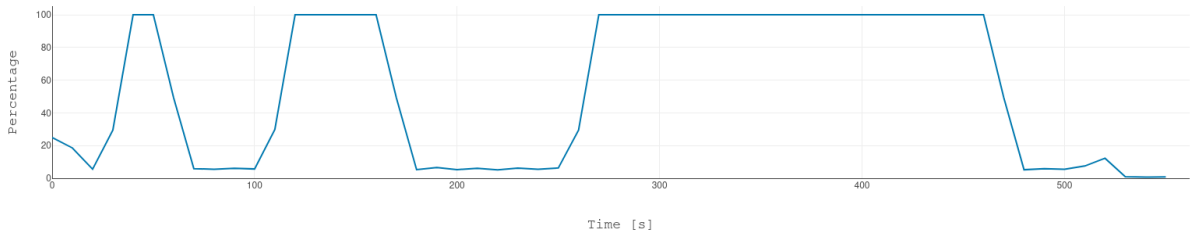

Figure 4.1: Percentage of CPU usage in the test

4.1.7 Custom faults

In order to make sure custom faults were injected, we created a script that writes the current timestamp into a file. Then, we ran an experiment with a single container, and injected this script. After the experiment, we checked the value, in the file, and asserted that FaultSee had injected the custom fault we created.

Additionally, in order to test if the faults are all injected simultaneously, we repeated the same test for three containers in four hosts and validated the values were all identical.

4.2 Macro Benchmarks

The evaluation has two main objectives, show that simple configuration files can model complex behaviours in the SUT and show that after creating an experiment, the user can easily repeat the experiments.

We created two sets of scenarios. The first set uses the database *Apache Cassandra* [23], that is heavily used by the industry, while the second set uses the academic library *BFT-Smart* [24]. Both systems are set up in a cluster of four nodes and then subject to the Yahoo! Cloud Serving Benchmark (YCSB) [25], which is a database benchmark used for NoSQL databases. While the

benchmark runs we inject different faults to see how it affects the system. In order to run the benchmark we first had to load the data, so that we can then run the benchmark. In all the experiments we used the YCSB (version 0.14) and run the *Workload A* (Operations: 50% Read and 50% Write). The experiments were performed on *Ubuntu 16.04.6 LTS*, with *Docker 19.03.4*. Additionally, all plots displayed in this chapter are downloaded from the FaultSee dashboard.

4.2.1 Cassandra

We ran this experiment in a Google Cloud Platform (GCP) cluster, with six *n1-standard-1* servers. The servers have 1 vCPU and 3.75 GB memory. We used *Cassandra* version 3.11.4 and the YCSB ran 5 750 000 Operations. The *Cassandra* cluster has 4 nodes, each running in a separate host. One of the remaining hosts controls the experiment and the other runs the YCSB benchmark, both the Load and Run clients. The deployment files used to run this set of experiments is present in Appendix A.1.

Inject Fault

The first scenario we decided to study is how *Cassandra* reacts when a fault that kills one of the nodes is injected. Then, after a brief period we then launch a new container, so that we can also study the impact of adding a new node to a running system. The FDSL file used in this scenario is present in Appendix A.2.

The experiment can be summarized in the following points:

- Start 4 *Cassandra* nodes;
- Load the necessary data to run the benchmark, with data being replicated into 3 nodes;
- Start the YCSB benchmark;
- Kill one of the nodes in the cluster 600 seconds later;
- Start a node 700 seconds later.

In Figure 4.2 we can see the variation in number of nodes in the cluster of *Cassandra* servers throughout the time of the experiment, blue line, and of the service YCSB, the load stage is the yellow line and the run phase is the green line. In the plot we can see that only a node at a time is started until the second 600. At the second 900 the YCSB Load client is started and lasts around 100 seconds. At the second 1 400, two YCSB - Run clients are started, each one with 10 threads, this service runs for approximately 2 000 seconds, and each client performs 5 750 000 operations, half of the operations are updates of existing records, while the other half is reading

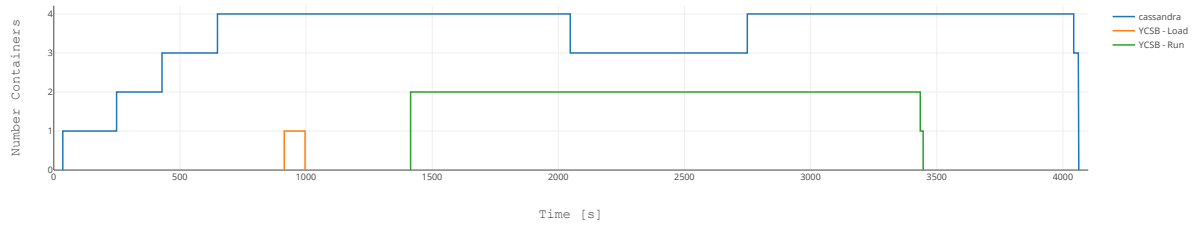

Figure 4.2: Number of containers running throughout the Cassandra experiment, in the scenario of killing a node

the stored information. At 2 050 seconds, one of the containers of the service *Cassandra* is killed, this container is replaced 700 seconds later, when another node is started.

In Figure 4.3 we can see the average of operations per second the YCSB clients perform. After a few seconds the system stabilises at around 3 250 operations per second. At second 2 150 we can observe that *Cassandra* cluster performance plunges. This is a direct result of the cluster restoring the replication factor to 3. The reason this dip in the performance only happens at 100 seconds after the node crashes is due to how *Cassandra* operates: when a node crashes it is not yet considered failed, as it may be a simple power outage, and then the node would recover and receive a small batch of the updates it missed. Instead, 100 seconds after the node crashes, the cluster receives the information that the dead node will not recover, as such, the cluster uses the available 3 nodes to achieve a replication factor of 3. This information is received through the injection of a custom fault: a script that runs a command that instructs the cluster to consider the dead node as failed.

After the restoration of the replication factor is concluded, the cluster stabilises at a new value of 2 750, a substantially lower value than the previous 3 250 operations per second. This can be explained by the fact that now there are only 3 live nodes to answer the clients' requests. At 2 700 seconds, a new node is started, to replace the dead one. As a consequence of this operation, *Cassandra* performance once again plunges. This a direct result of the cluster transferring a part of the data stored into the new node, this way utilising resources that could be used to answer clients' requests. Approximately 100 seconds later, this operation ends and performance improves once again, stabilising around 3 250 operations per second.

At the end of the plot we can see a sudden spike in throughput, this is explained to one of the two containers ends its 5 750 000 operations before the other. As a result, there are only half of the concurrent requests, therefore more resources are available to the remaining client requests, significantly improving its throughput.

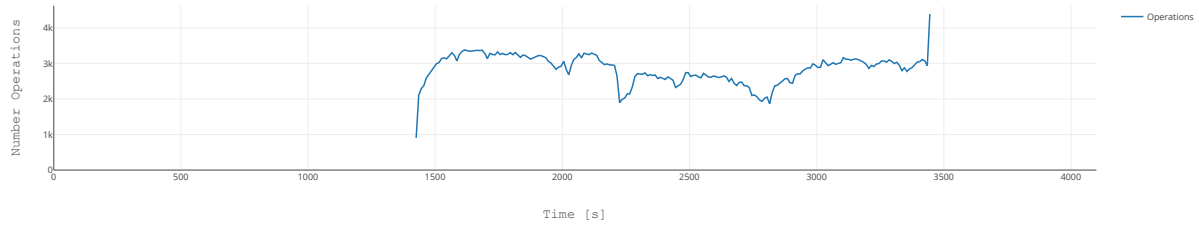

Figure 4.3: Average number of operations performed by YCSB-Run Clients throughout the Cassandra experiment, in the scenario of killing a node

Resources Exhaustion

In the second scenario we intend to show how the exhaustion of resources affects the performance of a *Cassandra* cluster. During this scenario, we run the same YCSB benchmark and during 700 seconds we exhaust the CPU of all the containers in the *Cassandra*'s cluster. The FDSL file used in this scenario is present in Appendix A.3.

The experiment can be summarized in the following points:

- Start 4 *Cassandra* nodes;
- Load the necessary data to run the benchmark, with data being replicated into 3 nodes;
- Start the YCSB benchmark;
- Exhaust CPU for 700 seconds.

In Figure 4.4 we can see the variation in number of nodes in the cluster of *Cassandra* servers throughout the time of the experiment, blue line, and of the service YCSB, the load stage is the yellow line and the run phase is the green line. This figure is very similar to Figure 4.2, however, in this scenario the number of nodes in service *Cassandra* remains constant.

In Figure 4.5 we can see the average number of operations per second of the *YCSB* clients, similar to Figure 4.3. After a few seconds the system stabilises at around 3 250 operations per second. At second 2 000 a fault is injected that starts consuming CPU, and it lasts for 700 seconds. We can see that the performance plunges to approximately 2 750 operations per second. As soon as the CPU stops being exhausted, at second 2 700, the performance improves rapidly, stabilising around 3 250 again. Similarly to the previous scenario, at the end of the plot we can see a sudden spike in throughput, as one of the YCSB clients ends its 5 750 000 earlier than the other.

Cassandra Discussion

Analysing the results in both experiments we can conclude that injecting faults has a real impact in the performance of *Apache Cassandra*. Comparing both scenarios, the CPU exhaustion had a bigger impact than removing one of the nodes of the cluster. Even though both scenarios resulted in reducing performance from 3 250 to 2 750 operations per second, *Apache Cassandra* was able to recover from one of the nodes crashing, whilst it was unable to recover from the CPU exhaustion until it terminated. We can see this by comparing Figure 4.5 with Figure 4.3.

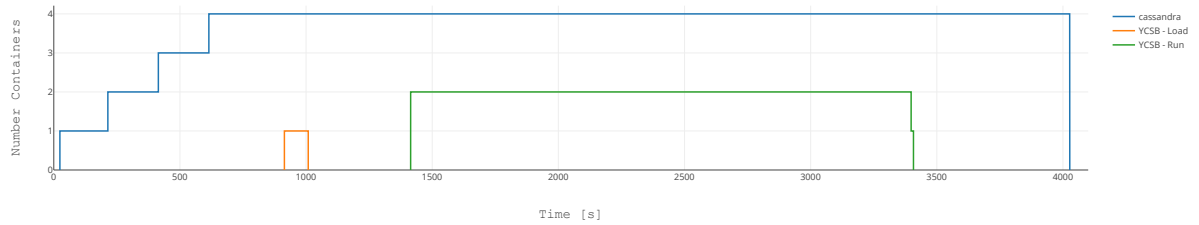

Figure 4.4: Number of containers running throughout the Cassandra experiment, in the CPU exhaustion scenario

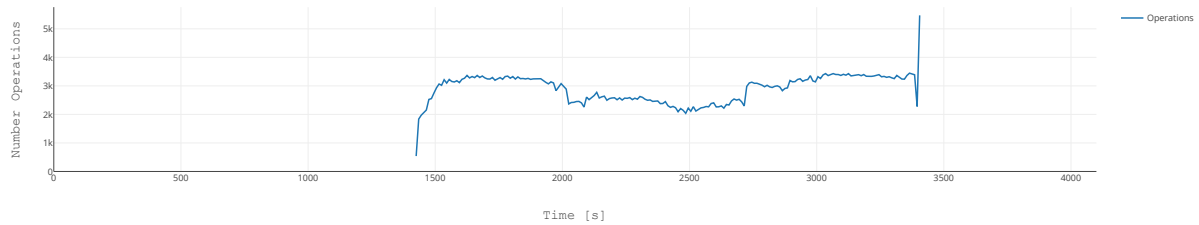

Figure 4.5: Average number of operations performed by YCSB-Run Clients throughout the Cassandra experiment, in the CPU exhaustion scenario

4.2.2 BFT-Smart

We ran this set of experiments in an Amazon Web Services (AWS) cluster, with six *t2.medium* instances, which are virtual machines running in shared servers, with 2 vCPUs and 4 GB memory. The YCSB benchmark ran 4 000 000 Operations in this set of experiments.

We decided to create three sets of experiments, the first we run the benchmark without interference, in the second we inject a CPU exhaustion fault and in the third scenario we kill one of the nodes. The *BFT-Smart* cluster has 4 nodes, each running in a separate host. One of the remaining hosts controls the experiment and the other runs the YCSB benchmark, both the Load and Run clients. The deployment files used to run this set of experiments is present

in Appendix B.1.

Faultless scenario

In the first scenario, we intended to see what was the system performance against the YCSB benchmark, without injecting any faults. The FDSL file used in this scenario is present in Appendix B.2.

The experiment can be summarized in the following points:

- Start 4 *BFT-Smart* nodes;
- Load the necessary data to run the benchmark;
- Start the YCSB benchmark.

In Figure 4.6 we can see the number of containers of each service throughout the experiment. The blue line represents the BFT-Smart nodes, the yellow line represents the YCSB Load service, that loads all the necessary data, so that the YCSB benchmark can run. As we can see in the green line, the YCSB benchmark service has 1 container, that runs with 10 threads.

In Figure 4.7 we can see the number of operations per second throughout the experiment. We can observe the cluster processes 1 600 operations per second throughout the experiment, with an exception at 2 700 seconds, when the performance has a negative spike due to some requests timing out. In Figure 4.8 we can see that the network activity also has a negative spike, as until the previous requests complete, the client does not create other requests.

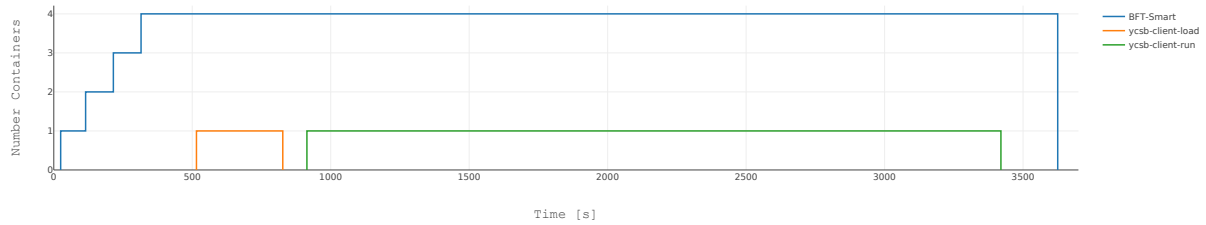

Figure 4.6: Number of containers running throughout the BFT-Smart experiment, in the faultless scenario

Resources Exhaustion

In the second scenario we intend to study how the exhaustion of CPU affects the performance of the *BFT-Smart*'s cluster. This scenario is similar to the faultless scenario, however, during the YCSB benchmark we exhaust the CPU for 700 seconds of all *BFT-Smart* containers. The FDSL file used in this scenario is present in Appendix B.3.

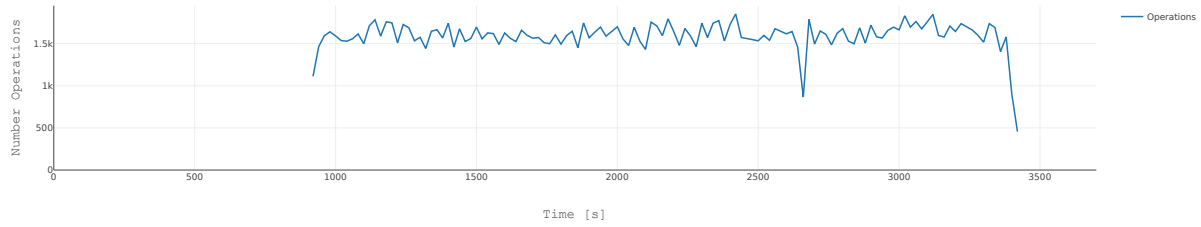

Figure 4.7: Average number of operations per second performed by the YCSB Client throughout the BFT-Smart experiment, in the faultless scenario

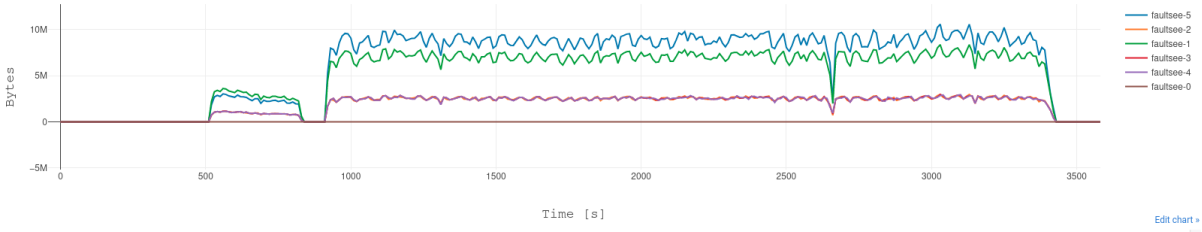

Figure 4.8: Outgoing network usage for every host, in the faultless scenario

The experiment can be summarized in the following points:

- Start 4 *BFT-Smart* nodes;
- Load the necessary data to run the benchmark;
- Start the YCSB benchmark;
- Exhaust CPU for 700 seconds.

The Figure 4.9 is very similar to Figure 4.6 as the containers have the same churn model.

In Figure 4.11 we can see the percentage of CPU of all the hosts in the cluster. Faultsee-0, yellow line, only runs the Master Controller, Faultsee-{1..4} run the BFT-Smart containers and FaultSee-5, the green line, runs the benchmark loader and executioner. The first 4 spikes represent the individual boot of the BFT-Smart's containers, then we can see the toll of loading the data for the benchmark from second 500 until 800, and finally, from second 900 until 3 400 the benchmark is running. The CPU fault is clearly visible from second 1 800 until 2 500 as the lines representing the four hosts that have the BFT-Smart's containers are maxed out at 100%.

In Figure 4.10 we can see the *BFT-Smart* performance throughout this experiment. Similarly to the faultless scenario, the number of operations per second processed stabilises around 1 750. Then, when the CPU fault is injected, the performance decreases to 1 400 operations per second. When the fault end, at second 2 500, *BFT-Smart* performance goes back to 1 750.

The dip in performance is explained by the *BFT-Smart* cluster having limited CPU resources in that moment, as such it could not handle the same amount of requests simultaneously.

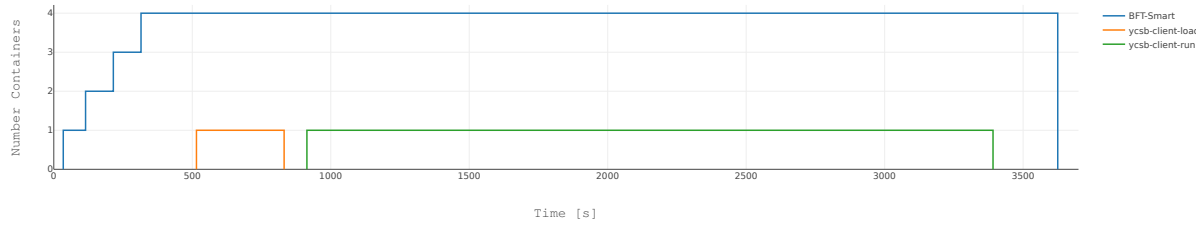

Figure 4.9: Number of containers running throughout the BFT-Smart experiment, in the CPU exhaustion scenario

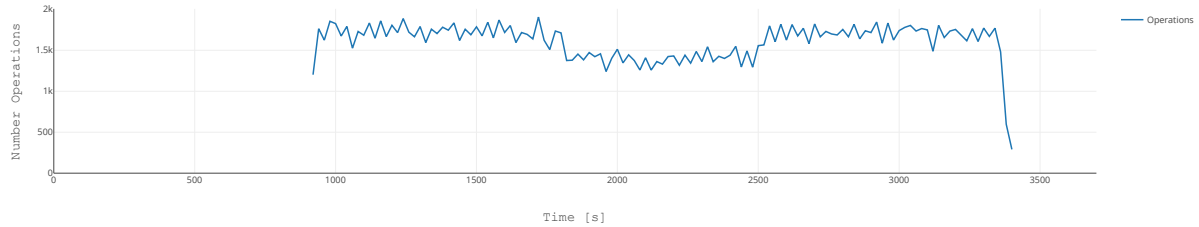

Figure 4.10: Average number of operations per second performed by YCSB Clients throughout the BFT-Smart experiment, in the CPU exhaustion scenario

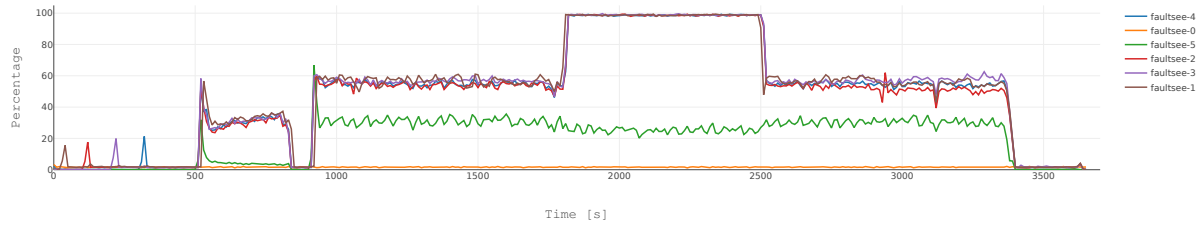

Figure 4.11: CPU percentage usage for every host, in the CPU exhaustion scenario

Inject Fault

In the third scenario we intended to study the impact killing one of the nodes had on the cluster performance. The FDSL file used in this scenario is present in Appendix B.4.

The experiment can be summarized in the following points:

- Start 4 *BFT-Smart* nodes;
- Load the necessary data to run the benchmark;

- Start the YCSB benchmark;
- Kill one of the nodes in the cluster 600 seconds into the benchmark.

In Figure 4.12 we can see the number of containers of each service throughout the experiment. The blue line represents the BFT-Smart nodes, the yellow line represents the YCSB Load service and the green line shows the YCSB benchmark. We can see that at second 1 700 the number of BFT-Smart containers decreases, this is due one of the containers being killed in this scenario.

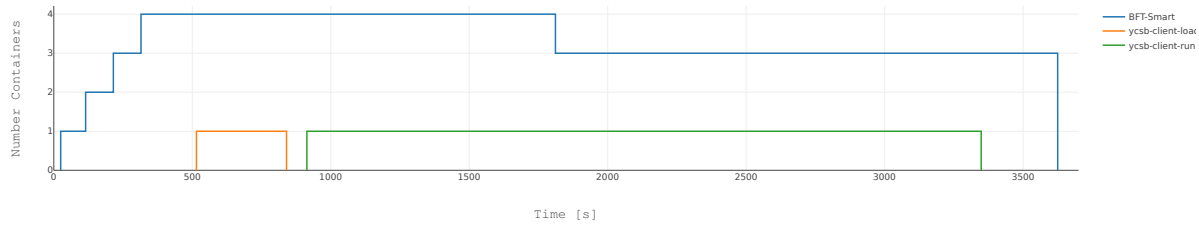

Figure 4.12: Number of containers running throughout the BFT-Smart experiment, in the kill a node scenario

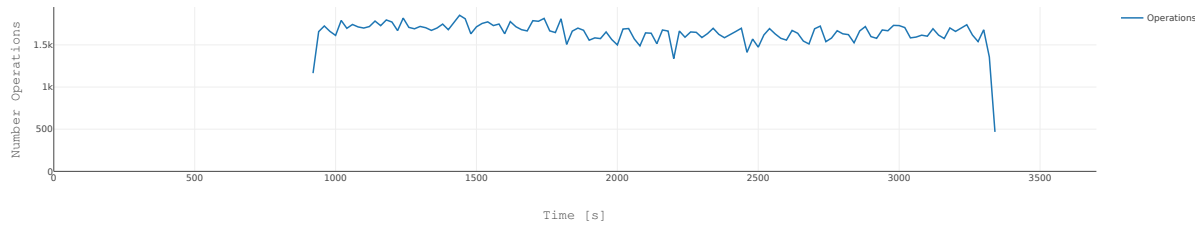

Figure 4.13: Average number of operations per second performed by YCSB Clients throughout the BFT-Smart experiment, in the kill a node scenario

In Figure 4.13 we can see the *BFT-Smart* performance throughout this experiment. Similarly to the faultless scenario, the performance remains constant throughout the experiment, at around 1 750 operations per second. From the figure we can also conclude that killing one of the nodes had no impact on *BFT-Smart* performance.

4.3 Discussion

We created two sets of experiments, each with more than one scenario, to study the impact that faults had on the performance of two distributed systems.

We were able to reuse much of the configuration files in experiments. In the same set of experiments we could reuse the same deployment configuration files and with small addendums

we were able to modify the faults to inject. Even from one set of experiments to the other we could reuse much of the events configuration files, as even though the experiments incided over different systems, the experiment flow was the same.

In the first set of experiments, we could verify that the injection of both faults affected the *Cassandra* performance. While on the second set of experiments, we verified that the *BFT-Smart* cluster was only affected in the CPU exhaustion scenario.

With those experiments we were able to demonstrate that with a few lines of configuration files we can model complex behaviours. These complex behaviours can then easily be repeated, even by other people, without much human effort. Additionally, we were able to demonstrate that FaultSee supports both AWS and GCP, without the need for any additional configurations.

Chapter 5

Conclusion

In this chapter we conclude the document, draw the main conclusions of the work and discuss future work.

5.1 Achievements/Contributions

In this dissertation we designed and implemented *FaultSee*, a tool that enables the execution of fault scenarios in distributed systems, which represents an increasing need, as distributed systems are operating at an increasing bigger scale and with a bigger number of components. The automation of the execution of fault scenarios enables the improvement of the lifecycle of software development, as defects can be detected sooner. FaultSee supports custom faults, which enables the user to create new fault scenarios. FaultSee also comes with a dashboard so that its users can see plots of resource usage generated automatically, or even extend this dashboard with custom plugins.

Moreover, this system improves the area of research in distributed systems, as it allows scenarios to be reproductible by independent investigators, which will only require the configuration files used in the original experiment and access to servers with equivalent computational power. For example, different consistency schemes and configurations for a set of servers can be compared.

In this dissertation we also showed that FaultSee supports different cloud providers that rent virtual machines. Specifically we ran the experiments in both AWS and GCP. We also showed that experiments created in FaultSee are easily reproduced by other users by sharing the configuration files. Additionally, we showed that with simple configuration files it was possible to emulate complex behaviours. Moreover, reusing most of the fault scenario configuration file, changing only the names of the services and the deployment configurations, we were able to

create a benchmark of a CPU exhaustion scenario for two different systems, *Apache Cassandra* and *BFT-Smart*.

This work also resulted in a published paper in the INFORUM 2019 conference: *FaultSee: Avaliação Reproduzível de Sistemas Distribuídos Sujeitos a Faltas*. The paper authors are Miguel Amaral, Miguel L. Pardal and Miguel Matos.

5.2 Future Work

This work represents a leap forward in the state of the art, as FaultSee enables independent researchers to reproduce experiments created by other researchers. However, the work in this area is far from complete. FaultSee itself could be improved in several ways. As future work we suggest increasing the number of available faults to the users, such as emulating network failures. Another useful feature would be the support for different experiment flows, currently FaultSee only supports temporal events, however, triggering events based on the application state could enrich the tool. In the current model, faults are injected based on how many seconds have elapsed since the beginning of the experiments, when replaying the same experiment, faults can then be injected in different application states, which can hinder the experiment reproducibility. Due to this fact we can only atest to the validity of results by repeating the same experiment several times, to be able to confidently take conclusions. An improvement would be to be able record the application state on the first run of the experiment, then state conditions for fault injection, thus enabling faults to be injected in the same application state. This would increase the reproducibility of the experiments.

The dashboard was not the core focus of the work. As future work the dashboard could be improved in order to be able to interact with the cluster to run experiments and have an experiment editor. Additionally, the dashboard could include features to accompany the experiment in real time, such as the running containers in the cluster or event the resources being used by every container or host, by displaying information sampled every period of time, such as 10 seconds. Finally, the dashboard should also enable the user to compare results from two different experiments side by side, in order to take conclusions faster.

Bibliography

[1] John Graham-Cumming. Details of the Cloudflare outage on July 2, 2019, 2019. URL https://web.archive.org/web/20190712160002/https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[2] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, jun 2008. ISSN 13823256. doi: 10.1007/s10664-008-9062-z. URL http://link.springer.com/10.1007/s10664-008-9062-z.

[3] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5 SPEC. ISS.):337–342, 2004. ISSN 09505849. doi: 10.1016/j.infsof.2003.09.011. URL www.elsevier.com/locate/infsof.

[4] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing Dependability with Software Fault Injection. *ACM Computing Surveys*, 48(3):1–55, 2016. ISSN 03600300. doi: 10.1145/2841425. URL http://dl.acm.org/citation.cfm?doid=2856149.2841425.

[5] Docker Inc. Docker Documentation, 2018. URL https://docs.docker.com/.

[6] Paulo Verissimo and Luis Rodrigues. *Distributed systems for system architects*, volume 1. Springer Science & Business Media, 2012.

[7] Giampaolo Rodola. Psutil package: a cross-platform library for retrieving information on running processes and system utilization, 2016.

[8] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):23, 2018.

[9] Ariel Tseitlin. The antifragile organization. *Communications of the ACM*, 56(8):40, aug 2013. ISSN 00010782. doi: 10.1145/2492007.2492022. URL http://dl.acm.org/citation.cfm?doid=2492007.2492022.

[10] Pumba: Chaos testing tool for Docker, 2016. URL https://github.com/alexei-led/pumba.

[11] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance. *ACM Transactions on Storage*, 13(3):1–33, 2017. ISSN 15533077. doi: 10.1145/3125497. URL http://dl.acm.org/citation.cfm?doid=3141876.3125497.

[12] Ismaïl Senhaji. Lsdsuite: Evaluation framework for large-scale distributed systems. Master thesis, Université de Fribourg, 2018.

[13] M.a Roza, M.a Schroders, and H.b Van De Wetering. A high performance visual profiler for games. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games, Sandbox '09*, pages 103–110, New York, New York, USA, 2009. ACM Press. ISBN 9781605585147. doi: 10.1145/1581073.1581090. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-70450233363{&}partnerID=40{&}md5=7b55258d7b1f679c009148162e510a2c.

[14] Benjamin H Sigelman, Luiz Andr, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. *Google Technical Report dapper-2010-1*, page 14, 2010. ISSN <null>. doi: dapper-2010-1. URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf.

[15] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13, 2007. ISSN 01464833. doi: 10.1145/1282427.1282383. URL http://portal.acm.org/citation.cfm?doid=1282427.1282383.

[16] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, 37(5):74, 2003. ISSN 01635980. doi: 10.1145/945445.945454. URL http://portal.acm.org/citation.cfm?doid=945445.945454.

[17] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. Wap5. *Proceedings of the 15th international conference on World Wide Web - WWW '06*, page 347, 2006. doi: 10.1145/1135777.1135830. URL http://portal.acm.org/citation.cfm?doid=1135777.1135830.

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

[19] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, and Christof Fetzer. Fex: A Software Systems Evaluator. In *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*, pages 543–550. IEEE, jun 2017. ISBN 9781538605417. doi: 10.1109/DSN.2017.25. URL http://ieeexplore.ieee.org/document/8023152/.

[20] Lorenzo Leonini, É Rivière, and Pascal Felber. SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze). *Nsdi*, 9:1–20, 2013. URL http://www.usenix.org/event/nsdi09/tech/full{_}papers/leonini/leonini{_}html/.

[21] Grzegorz Milka and Krzysztof Rzadca. Dfuntest: A testing framework for distributed applications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10777 LNCS, pages 395–405. Springer, Springer Nature, mar 2018. ISBN 9783319780238. doi: 10.1007/978-3-319-78024-5_35. URL http://arxiv.org/abs/1803.04442{%}0Ahttp://dx.doi.org/10.1007/978-3-319-78024-5{_}35.

[22] Tony Ohmann, Ryan Stanley, Ivan Beschastnikh, and Yuriy Brun. Visually reasoning about system and resource behavior. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 601–604, 2016. ISBN 9781450342056. doi: 10.1145/2889160.2889166. URL http://dl.acm.org/citation.cfm?doid=2889160.2889166.

[23] Apache Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra.org/what-is-apache-cassandra*, page 13, 2014.

[24] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears.

Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

Appendix A

Apache Cassandra experiments configuration files

A.1 Docker-Compose file

```
1  version: "3.1"
2  services:
3    cassandra:
4      labels:
5            # gather detailed stats
6        org.lsdsuite.stats: "true"
7      image: docker.io/mapa12/cassandra:server
8      environment:
9        DNS_CASSANDRA_NAME: tasks.cassandra
10       cassandra.consistent.rangemovement: "false"
11       MAX_HEAP_SIZE: 2048M
12     deploy:
13       replicas: 0
14       placement:
15         constraints:
16         - node.hostname != faultsee-5
17         - node.hostname != faultsee-0
18     networks:
19       - "backend"
20
```

```
21    setup-service:
22      deploy:
23        replicas: 0
24      image: docker.io/mapa12/cassandra:setup
25      networks:
26        - "backend"
27
28      ycsbload:
29        image: docker.io/mapa12/ycsb:latest
30        environment:
31          WORKLETTER: a
32          ACTION: load
33          DBTYPE: cassandra-cql
34          DBARGS: -p hosts=tasks.cassandra
35          RECNUM: 100000
36          OPNUM:  5750000
37        deploy:
38          replicas: 0
39          placement:
40            constraints:
41              - node.hostname == faultsee-5
42        networks:
43          - "backend"
44
45      ycsbarun:
46        image: docker.io/mapa12/ycsb:latest
47        environment:
48          WORKLETTER: a
49          ACTION: run
50          DBTYPE: cassandra-cql
51          DBARGS: -p hosts=tasks.cassandra
52          THREADS: 10
53          RECNUM: 100000
54          OPNUM:  5750000
```

```
55              deploy:
56                replicas: 0
57            placement:
58              constraints:
59                - node.hostname == faultsee-5
60          networks:
61            - "backend"
62
63  networks:
64    backend:
65      driver: overlay
66      ipam:
67        config:
68          - subnet: 10.22.0.0/16
```

A.2 FDSL file for the Kill a node scenario

```
1  environment:
2      seed: 568
3      ntp_server: europe.pool.ntp.org
4  events:
5      - beginning:
6          cassandra: 0
7          setup-service: 0
8          ycsbarun: 0
9          ycsbaload: 0
10     # give the cluster time to be good
11     - moment:
12         time: 10
13         services:
14           cassandra:
15             - start:
16                 amount: 1
17     - moment:
18         time: 200
19         services:
20           cassandra:
21             - start:
22                 amount: 1
23     - moment:
24         time: 400
25         services:
26           cassandra:
27             - start:
28                 amount: 1
29     - moment:
30         time: 600
31         services:
32           cassandra:
33             - start:
```

```
34                  amount: 1
35      - moment:
36          time: 800
37          services:
38            setup-service:
39              - start:
40                  amount: 1
41      - moment:
42          time: 900
43          services:
44            ycsbaload:
45              - start:
46                  amount: 1
47      - moment:
48          time: 1400
49          services:
50            ycsbarun:
51              - start:
52                  amount: 2
53      # inject the fault
54      - moment:
55          time: 2000
56          services:
57            cassandra:
58              - fault:
59                  target:
60                    specific: [3]
61                  kill:
62      - moment:
63          time: 2100
64          services:
65            cassandra:
66              - fault:
67                  target:
```

```
68                          specific: [2]
69                  custom:
70                          kills_container: "no"
71                          fault_file_name:
72                            remove_dead_node_from_cluster
73      - moment:
74          time: 2700
75          services:
76            cassandra:
77              - start:
78                  amount: 1
79      - end: 4000
```

A.3 FDSL file for the CPU exhaustion scenario

```
environment:
    seed: 568
    ntp_server: europe.pool.ntp.org
events:
    - beginning:
        cassandra: 0
        setup-service: 0
        ycsbarun: 0
        ycsbaload: 0
    - moment:
        time: 10
        services:
          cassandra:
            - start:
                amount: 1
    - moment:
        time: 200
        services:
          cassandra:
            - start:
                amount: 1
    - moment:
        time: 400
        services:
          cassandra:
            - start:
                amount: 1
    - moment:
        time: 600
        services:
          cassandra:
            - start:
                amount: 1
```

```
34      - moment:
35          time: 800
36          services:
37            setup-service:
38              - start:
39                  amount: 1
40      - moment:
41          time: 900
42          services:
43            ycsbaload:
44              - start:
45                  amount: 1
46      - moment:
47          time: 1400
48          services:
49            ycsbarun:
50              - start:
51                  amount: 2
52      # inject the fault
53      - moment:
54          time: 2000
55          services:
56            cassandra:
57              - fault:
58                  target:
59                    amount: 4
60                  cpu:
61                    duration: 700
62              - fault:
63                  target:
64                    amount: 4
65                  cpu:
66                    duration: 700
67              - fault:
```

```
68                      target:
69                        amount: 4
70                      cpu:
71                        duration: 700
72        - end: 4000
```

Appendix B

BFT-Smart experiments configuration files

B.1 Docker-Compose file

```
1  version: '3.4'
2  services:
3    server-1:
4      labels:
5        # gather detailed stats
6        org.lsdsuite.stats: "true"
7      image: docker.io/mapa12/bft-smart-experiment:latest
8      environment:
9        REPLICA_INDEX: 1
10       MAX_HEAP_SIZE: 2560M
11     deploy:
12       resources:
13         limits:
14           memory: 3G
15       replicas: 0
16       placement:
17         constraints:
18           - node.hostname == faultsee-1
19         networks:
20           - "backend"
```

```
21
22    server-2:
23      labels:
24        # gather detailed stats
25        org.lsdsuite.stats: "true"
26      image: docker.io/mapa12/bft-smart-experiment:latest
27      environment:
28            REPLICA_INDEX: 2
29            MAX_HEAP_SIZE: 2560M
30      deploy:
31        resources:
32          limits:
33            memory: 3G
34      replicas: 0
35        placement:
36          constraints:
37            - node.hostname == faultsee-2
38      networks:
39            - "backend"
40
41    server-3:
42      labels:
43        # gather detailed stats
44        org.lsdsuite.stats: "true"
45      image: docker.io/mapa12/bft-smart-experiment:latest
46      environment:
47        REPLICA_INDEX: 3
48        MAX_HEAP_SIZE: 2560M
49      deploy:
50        resources:
51          limits:
52            memory: 3G
53      replicas: 0
54        placement:
```

```
55          constraints:
56            - node.hostname == faultsee-3
57      networks:
58        - "backend"
59
60    server-4:
61      labels:
62        # gather detailed stats
63        org.lsdsuite.stats: "true"
64      image: docker.io/mapa12/bft-smart-experiment:latest
65      environment:
66        REPLICA_INDEX: 4
67        MAX_HEAP_SIZE: 2560M
68      deploy:
69        resources:
70          limits:
71            memory: 3G
72      replicas: 0
73        placement:
74          constraints:
75            - node.hostname == faultsee-4
76      networks:
77        - "backend"
78
79    ycsb-client-load:
80      labels:
81        # gather detailed stats
82        org.lsdsuite.stats: "true"
83      image: docker.io/mapa12/bft-smart-experiment:latest
84      environment:
85        ACTION: load
86        RECNUM: 100000
87        OPNUM:  125000
88      deploy:
```

```
89          replicas: 0
90          placement:
91            constraints:
92              - node.hostname == faultsee-5
93        networks:
94          - "backend"
95
96    ycsb-client-run:
97        labels:
98          # gather detailed stats
99          org.lsdsuite.stats: "true"
100       image: docker.io/mapa12/bft-smart-experiment:latest
101       environment:
102         ACTION: run
103         THREADS: 10
104         RECNUM: 100000
105         OPNUM:  4000000
106         START_CLIEND_ID: "{{.Task.Slot}}"
107       deploy:
108         replicas: 0
109         placement:
110           constraints:
111             - node.hostname == faultsee-5
112       networks:
113         - "backend"
114 networks:
115   backend:
116     driver: overlay
117     ipam:
118       config:
119         - subnet: 10.22.0.0/16
```

B.2 FDSL file for the faultless scenario

```
1  environment:
2      seed: 568
3      ntp_server: europe.pool.ntp.org
4  events:
5      - beginning:
6          server-1: 0
7          server-2: 0
8          server-3: 0
9          server-4: 0
10         ycsb-client-load: 0
11         ycsb-client-run: 0
12
13     - moment:
14         time: 10
15         services:
16             server-1:
17                 - start:
18                     amount: 1
19     - moment:
20         time: 100
21         services:
22             server-2:
23                 - start:
24                     amount: 1
25     - moment:
26         time: 200
27         services:
28             server-3:
29                 - start:
30                     amount: 1
31     - moment:
32         time: 300
33         services:
```

```
34              server-4:
35                  - start:
36                      amount: 1
37
38      - moment:
39          time: 500
40          services:
41              ycsb-client-load:
42                  - start:
43                      amount: 1
44      #  wait for operation LOAD to end
45      - moment:
46          time: 900
47          services:
48            ycsb-client-run:
49              - start:
50                    amount: 1
51      - end: 3600
```

B.3 FDSL file for the CPU exhaustion scenario

```
1  environment:
2      seed: 568
3      ntp_server: europe.pool.ntp.org
4  events:
5      - beginning:
6          server-1: 0
7          server-2: 0
8          server-3: 0
9          server-4: 0
10         ycsb-client-load: 0
11         ycsb-client-run: 0
12
13     - moment:
14         time: 10
15         services:
16             server-1:
17                 - start:
18                     amount: 1
19     - moment:
20         time: 100
21         services:
22             server-2:
23                 - start:
24                     amount: 1
25     - moment:
26         time: 200
27         services:
28             server-3:
29                 - start:
30                     amount: 1
31     - moment:
32         time: 300
33         services:
```

```
34              server-4:
35                  - start:
36                      amount: 1
37
38      - moment:
39          time: 500
40          services:
41              ycsb-client-load:
42                  - start:
43                      amount: 1
44      #  wait for operation LOAD to end
45      - moment:
46          time: 900
47          services:
48            ycsb-client-run:
49              - start:
50                    amount: 1
51
52      # inject the fault
53      - moment:
54          time: 1800
55          services:
56            # inject 3 times for each server
57            server-1:
58              - fault:
59                  target:
60                    amount: 1
61                  cpu:
62                    duration: 700
63              - fault:
64                  target:
65                    amount: 1
66                  cpu:
67                    duration: 700
```

```
68              - fault:
69                  target:
70                    amount: 1
71                  cpu:
72                    duration: 700
73            # inject 3 times for each server
74            server-2:
75              - fault:
76                  target:
77                    amount: 1
78                  cpu:
79                    duration: 700
80              - fault:
81                  target:
82                    amount: 1
83                  cpu:
84                    duration: 700
85              - fault:
86                  target:
87                    amount: 1
88                  cpu:
89                    duration: 700
90            # inject 3 times for each server
91            server-3:
92              - fault:
93                  target:
94                    amount: 1
95                  cpu:
96                    duration: 700
97              - fault:
98                  target:
99                    amount: 1
100                 cpu:
101                   duration: 700
```

```
102             - fault:
103                 target:
104                   amount: 1
105                 cpu:
106                   duration: 700
107           # inject 3 times for each server
108           server-4:
109             - fault:
110                 target:
111                   amount: 1
112                 cpu:
113                   duration: 700
114             - fault:
115                 target:
116                   amount: 1
117                 cpu:
118                   duration: 700
119             - fault:
120                 target:
121                   amount: 1
122                 cpu:
123                   duration: 700
124     - end: 3600
```

B.4 FDSL file for the Kill a node scenario

```
1  environment:
2      seed: 568
3      ntp_server: europe.pool.ntp.org
4  events:
5      - beginning:
6          server-1: 0
7          server-2: 0
8          server-3: 0
9          server-4: 0
10         ycsb-client-load: 0
11         ycsb-client-run: 0
12
13     - moment:
14         time: 10
15         services:
16             server-1:
17                 - start:
18                     amount: 1
19     - moment:
20         time: 100
21         services:
22             server-2:
23                 - start:
24                     amount: 1
25     - moment:
26         time: 200
27         services:
28             server-3:
29                 - start:
30                     amount: 1
31     - moment:
32         time: 300
33         services:
```

```
34              server-4:
35                  - start:
36                      amount: 1
37
38      - moment:
39          time: 500
40          services:
41              ycsb-client-load:
42                  - start:
43                      amount: 1
44      - moment:
45          time: 900
46          services:
47            ycsb-client-run:
48              - start:
49                  amount: 1
50      - moment:
51          time: 1800
52          services:
53            server-3:
54              - fault:
55                  target:
56                    amount: 1
57                  kill:
58      - end: 3600
```