FaultSee: Reproducible fault injection in distributed systems

Miguel Antão Pereira Amaral
Instituto Superior Técnico, Universidade de Lisboa

***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.**

I. Introduction

Technology is taking a central role in modern society. In developed countries, everywhere we look we can see the presence of software. An example is 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%. 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, as software is developed by humans, and because the environment is unpredictable, it can be faulty. Faults can lead to catastrophes, especially when dealing with critical applications. Last July, 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 inputted into his code can be tested, leading them to assert, with confidence that their software is bug-free.

Throughout the years we have seen a bigger investment from the community to build tools with the objective to help developers creating 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 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.

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. *FaultSee* is

a tool we created that simulates components failures patterns, which helps developers mimic faults that might affect their systems in production.

Reproducibility of results is important. In 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. Reproducibility is also very relevant to academia, as it 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 reproducibility. Additionally, it removes the possibility for the creation of automated benchmarks.

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.

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. This document presents

II. Related Work

In this section we summarily discuss the related work, focusing on fault injection and deployment.

Cords [5] is a framework developed with the purpose of injecting faults in file systems in order to test distributed applications. Ganesan et al. [5] 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 life-cycle 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.

To motivate its engineers to build more fault tolerant tools, Netflix created the SimianArmy [6] 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;
- Chaos Gorilla - Tests problems related with an availability zone. It supports two distinct modes, terminating all instances in data center or creating a network partition, in which the instances inside that data center are unable to communicate or be reached by services hosted outside;
- Chaos Kong - Enables developers with terminating all services in all datacenters in an entire region;
- Latency Monkey -Simulates partly healthy instances, which increase latency in requests.

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 [7] 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.

When compared to FaultSee, these two tool do not enable to user to create scripts to describe fault scenarios, which could be used to reproduce experiments. Additionally, FaultSee enables its users to gather resource usage metrics, that are available at the end of the experiment in a dashboard.

FEX [8] 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.

Dfuntest [9] 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 centralized host to orchestrate tests, therefore cannot scale indefinitely. Nevertheless, it allows a user to interact with the system while it is being tested.

FaultSee is a tool that is built with the objective to help its users to improve their applications, enabling them to extend FaultSee and create custom faults for their application, that easily integrated into FaultSee. Additionally, FaultSee enables its users to reproduce their experiments, as the fault scenarios are described in files, as well as the deploy configuration,

which leverage Docker capabilities. This way it is possible to recreate the same experiment environments. Finally, as FaultSee injects its faults from each node in the cluster, instead of using a central node, it has fewer problems to scale to bigger experiments, reduces the network usage during experiements and the faults are injected with more accuracy, as the network latency is removed.

III. FaultSee

In this section we describe FaultSee design and specify the FDSL language. We conclude with a discussion highlighting FaultSee main features and how they address the requirements of this work.

A. FaultSee Domain System Language

FaultSee Domain System Language (FDSL) is the language used to describe fault scenarios. Its key concepts is that it describes the experiment properties and the experiment events throughout time.

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 1.

```
environment:
    seed: seed_value
    ntp_server: URL
events:
    - event_1
    - event_2
    - ...
    - event_N
```

Code Example 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 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.

```
- moment:
    time: number_of_seconds
    mark: custom_message
    services:
      service_name_1:
          - service_event_1
          - ...
          - service_event_N

      service_name_N:
        - service_event_1
        - ...
        - service_event_N
```

Code Example 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 allows to inject abnormal behaviour into the System Under Test (SUT). All **faults** have a common structure, depicted in Example 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.

```
- fault:
  target:
     # amount, percentage and specific
     #   are mutually exclusive
     amount: amount_number
     percentage: percentage_number
     specific:
       - ID_1
       - ...
       - ID_N
  fault_type
```

Code Example 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 4 we can see the structure, which includes six fields. *kills_containers* informs FaultSee if this fault kills the

[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 synchronizes 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.

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 scrits arguments, if omited 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*.

```
custom:
  kills_container: yes / no
  fault_file_name: fault_file_name
  fault_file_folder: fault_file_folder
      # default - /usr/lib/faultsee/
  fault_script_arguments:
      # default - empty array
      - arg_1
      - ...
      - arg_N
  executable: executable
      # default - /bin/sh
  executable_arguments:
      # default - empty array
      - arg_1
      - ...
      - arg_N
```

Code Example 4: **Custom Fault** structure

B. FaultSee

FaultSee has four main components. These are the Master Controller (Section III-B2), Dashboard (Section III-B4), Infrastructure (Section III-B1), Local Controller (Section III-B3).

In Figure 1 we can see the whole system and the interactions each component will have when an experiment is being executed.

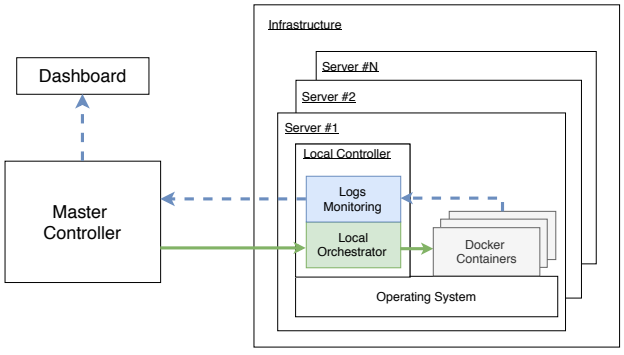

Fig. 1: Overview of the FaultSee system. Dotted arrows represent produced logs or metrics, while normal arrows represent control messages.

1) 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.

2) 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, described in detail in Section III-B3, 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.

3) Local Controller: As depicted in Figure 1, 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 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 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.

4) 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 contaiener.
- **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.

From the files, the Dashboard component can create plots, in Figure 2 we can see an example of a plot of the number of outgoing packets during the experiment, for each containers.

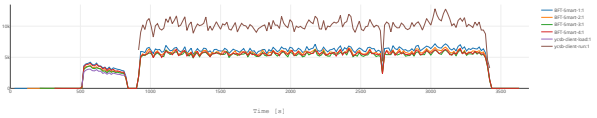

Fig. 2: Example of number of outgoing packets during an experiment

C. Overview

In this section we presented FaultSee, a tool that leverages Docker functionalities to enable its users to create reproducible scnenarios that emulate components failures. The experimental results are then displayed in a dashboard that creates plots automatically. Additionally, we spectified FDSL, a language that describes fault scenarios, that can be shared among users, thus making experiments reproducible.

IV. Evaluation

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* [10], that is heavily used by the industry, while the second set uses the academic library *BFT-Smart* [11]. Both systems are set up in a cluster of four nodes and then subject to the Yahoo! Cloud Serving Benchmark (YCSB) [12], 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* (Operarions: 50% Read and 50% Write). The experiements were performed on *Ubuntu 16.04.6 LTS*, with *Docker 19.03.4*. Additionally, all plots displayed in this section are downloaded from the FaultSee dashboard.

A. 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 Operarions. 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.

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 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 3 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 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 we can see the average of operations per second the YCSB clients perform. After a few seconds the system stabilizes 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

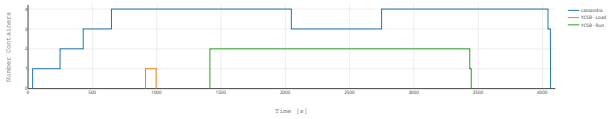

Fig. 3: Number of containers running throughout the Cassandra experiment, in the scenario of killing a node

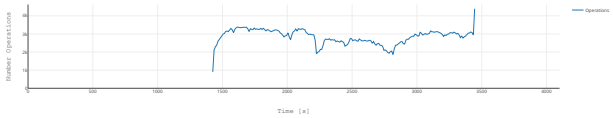

Fig. 4: Average number of operations performed by YCSB-Run Clients throughout the Cassandra experiment, in the scenario of killing a node

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 stabilizes at a new value of 2 750, a substantialy 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 transfering a part of the data stored into the new node, this way utilizing resources that could be used to answer clients' requests. Approximately 100 seconds later, this operation ends and performance improves once again, stabilizing 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 troughput.

2) 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 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 5 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 similiar to Figure 3, however, in this scenario the number of nodes in service *Cassandra* remains constant.

In Figure 6 we can see the average number of operations per second of the *YCSB* clients, similar to Figure 4. After a few seconds the system stabilizes 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, stabilizing around 3 250 again. Similarly to the previous scenario, at the end of the plot we can see a sudden spike in troughput, as one of the YCSB clients ends its 5 750 000 earlier than the other.

3) 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 scnearios 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 6 with Figure 4.

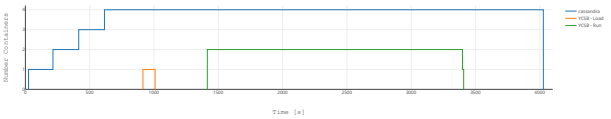

Fig. 5: Number of containers running throughout the Cassandra experiment, in the CPU exhaustion scenario

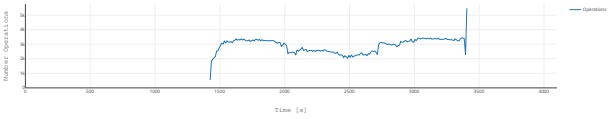

Fig. 6: Average number of operations performed by YCSB-Run Clients throughout the Cassandra experiment, in the CPU exhaustion scenario

With this experiment we were able to demonstrate that with a few lines of configuration files we can modulate 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.

V. Conclusion

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

In this dissertation we created FaultSee, a tool that enables the execution of fault scenarios in distributed systems, which represents an increasing necessity, 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 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 investigation in the science of 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.

In this dissertation we showed FaultSee supports all cloud providers that rent virtual machines, we ran the experiments in both Amazon Web Services (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.

A. 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 faults could be inject in the same application state, this would increase the reproducibility of the experiments.

The dashboard was not the core focus of the work, as such it can be improved. 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.

References

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

[2] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: Results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, no. 3, pp. 289–302, jun 2008. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9062-z

[3] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5 SPEC. ISS., pp. 337–342, 2004. [Online]. Available: www.elsevier.com/locate/infsof

[4] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–55, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2856149.2841425

[5] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy Does Not Imply Fault Tolerance," *ACM Transactions on Storage*, vol. 13, no. 3, pp. 1–33, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3141876.3125497

[6] A. Tseitlin, "The antifragile organization," *Communications of the ACM*, vol. 56, no. 8, p. 40, aug 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2492007.2492022

[7] "Pumba: Chaos testing tool for Docker," 2016. [Online]. Available: https://github.com/alexei-led/pumba

[8] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer, "Fex: A Software Systems Evaluator," in *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*. IEEE, jun 2017, pp. 543–550. [Online]. Available: http://ieeexplore.ieee.org/document/8023152/

[9] G. Milka and K. 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)*. Springer, Springer Nature, mar 2018, vol. 10777 LNCS, pp. 395–405. [Online]. Available: http://arxiv.org/abs/1803.04442%0Ahttp://dx.doi.org/10.1007/978-3-319-78024-5_35

[10] A. Cassandra, "Apache cassandra," *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, p. 13, 2014.

[11] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.