FaultSee: Reproducible Fault Injection in Distributed Systems

Regular Paper

Miguel Amaral
INESC-ID, IST, U. Lisboa
Portugal

Miguel L. Pardal
INESC-ID, IST, U. Lisboa
Portugal

Hugues Mercier
U. of Neuchâtel
Switzerland

Miguel Matos
INESC-ID, IST, U. Lisboa
Portugal

***Index Terms*—dependability, fault-injection, systems evaluation, reproducibility, distributed systems**

***Abstract*—Distributed systems are increasingly important in modern society, often operating on a global scale with stringent dependability requirements. Despite the vast amount of research and the development of techniques to build dependable systems, faults are inevitable as one can witness from regular failures of major providers of IT services. It is therefore fundamental to evaluate distributed systems under different fault patterns and adversarial conditions to assess their high-level behaviour and minimize the occurrence of failures. However, succinctly capturing the system configuration, environment, fault patterns and other variables affecting an experiment is very hard, leading to a reproducibility crisis. In this paper we propose the FaultSee toolkit. The two components of FaultSee are (1) the simple and descriptive FDSL language that captures the system, environment, workload and fault pattern characteristics; and (2) an easy-to-use platform to deploy and run the experiments described by the language. FaultSee allows to precisely describe and reproduce experiments and leads to a better assessment the impact of faults in distributed systems. We showcase the key features of FaultSee by studying the impact of faults with real deployments of Apache Cassandra and BFT-Smart.**

I. Introduction

Modern society is increasingly reliant on distributed systems to support critical infrastructure such as banking and finance, healthcare, e-commerce and social media among many others. It is therefore fundamental to design and build distributed systems with strong dependability requirements. Despite the decades-long effort from both the industry and academia to design and build dependable distributed systems, failures do still happen frequently as can be witnessed by recent outages in services from global IT players [1], [2], [3]. While the occurrence of faults in any real system is inevitable, there is a vast literature on fault tolerant designs to prevent such faults from becoming errors and failures.

Unfortunately, the complexity of modern distributed systems, often composed of thousands of interacting components [4], makes it hard to reason about the impact and consequences of those faults on the system as a whole. In the field of software engineering, test-driven approaches that cover a wide range of scenarios have been shown to reduce the number of bugs (faults) and significantly increase the software quality [5], [6]. While the same systematic approach to testing and evaluation can improve the dependability of distributed systems, this is a challenging endeavor. Distributed systems are inherently complex, have multiple interacting components, are run on heterogeneous and varying environments, and thus subject to a wide range of fault patterns that must be measured and benchmarked.

The underlying reason for this difficulty stems from the fact that the evaluation results are affected by many variables, some known and controlled, others known but not controlled, and finally many completely unknown [7]. As an example, it has been shown that the size of the environment variables can have a significant impact on system performance [8]. Similarly, the profiler configuration can affect the results of well-known benchmarks such as Java Dacapo [9], [10]. The presence of such sources of variability impairs experimental reproducibility precluding researchers to assess each other's work [11], [12]. As a matter of fact, the ability to inject a wide range of faults is fundamental to assess the dependability of distributed systems [13]. To properly evaluate the impact of faults on a system, and hence assess its dependability, we need tools to systematically reproduce experiments that can subject a system to a wide range of fault scenarios.

Technologies such as virtual machines, containers, and orchestration platforms such as Docker Swarm [14] or Kubernetes [15] partially mitigate the *reproducibility* problem by capturing, in part, the system, its configuration and the environment. Specialized tools such as ReproZip [16], [17] or Guix [18] allow to take a snapshot of the system state (binaries and configurations) and reproduce this configuration at a later point in time. However, no existing tool can capture the injected workload and the dynamic properties of the system, such as faults and churn, *i.e.* the continued addition and removal of nodes to the system. Moreover, such details are usually omitted in a research paper due to space constraints and language limitations. As an example, generic sentences like "In this experiment, we failed three replicas, and measure the impact on system throughput", common in systems papers, fail to capture key aspects of the experiment. Which replicas failed? How were the faults injected? By stopping processes? By killing them? By shutting down the network interface? Depending on the answers to questions such as these, the experimental results and conclusions of experiments can be strikingly different [7]. Furthermore, without properly capturing the workload, which entails a subsystem with its own software, lifecycle and configuration, no meaningful conclusions about the system behavior and performance can be drawn.

A. *Our contributions*

In this work we propose the FaultSee toolkit encompassing (1) the simple and descriptive FDSL language that captures the system, environment, workload and fault pattern characteristics; and (2) an easy-to-use platform to deploy and run the experiments described by the language. The FaultSee FDSL language can describe the configuration and lifecycle of the different aspects of the experiment, whereas the platform provides a generic fault injection framework that can be extended to support arbitrary fault patterns. FaultSee builds on top of both Docker Swarm and can, by design, easily evaluate and reproduce unmodified containerized application experiments under a wide range of fault patterns. We illustrate the capabilities and usefulness of FaultSee using two real world applications: the NoSQL Apache Cassandra database [19], and BFT-Smart, a Byzantine Fault-Tolerant State Machine Replication system. Internally, we have used FaultSee sucessfully to conduct system research and also in the experimental work of students.

The rest of this paper is organized as follows. We describe the FaultSee language and present the FaultSee platform in Sections II and III. In Section IV, we illustrate the functionalities of FaultSee in a realistic scenario. Finally, we discuss related work in Section V and conclude in Section VI.

II. FDSL: the FaultSee Domain System Language

In this section, we describe FDSL, the FaultSee Domain System Language. FDSL allows researchers and practitioners to succintly and precisely express the system binaries, configuration, workload, lifecycle and dynamics such as churn and faults. We chose YAML [20] for the FDSL syntax as it is compact and both human and machine readable.

We now describe in detail how an experiment can be expressed with FDSL. To illustrate the main concepts we use the running example depicted in Listing 1. It describes an experiment with the NoSQL Apache Cassandra database [19] evaluated with the YCSB benchmark [21] while subject to faults. The objective is to assess the impact of faults on the system throughput. More details, and results, of this experiment are presented in Section IV.

The intuition behind FDSL is to describe the experiment in two parts. The first part contains the overall environment and the initial state of the system (binaries and configurations), expressed by the `environment` section and `beginning` event. The second part is the sequence of events, such as adding, failing or stopping replicas, that modify the state over time. These are expressed by a sequence of `moment` that correspond to changes in the experiment state.

A. *Detailed FDSL features*

The `environment` captures system wide properties that impact the experiment (Listing 1, lines 2–4). FDSL supports

```
1  #overall environment
2  environment:
3   seed: 568
4   ntp_server: europe.pool.ntp.org
5  events:
6  #setup initial instances
7   - beginning:
8     cassandra: 0
9     setup-service: 0
10    ycsbarun: 0
11    ycsbaload: 0
12 #start one cassandra replica
13  - moment:
14    time: 10
15    services:
16      cassandra:
17      - start:
18        amount: 1
19 #start another cassandra replica
20  - moment:
21    time: 200
22    services:
23      cassandra:
24       - start:
25         amount: 1
26 #load the data in the database
27  - moment:
28    time: 900
29    services:
30      ycsbload:
31       - start:
32         amount: 1
33 #run the benchmark with two clients
34  - moment:
35    time: 1400
36    services:
37      ycsbarun:
38        - start:
39          amount: 2
40 #kill a specific replica
41  - moment:
42    time: 2000
43    services:
44      cassandra:
45        - fault:
46          target:
47            specific: [1]
48            kill:
49 #end the experiment
50  - end: 4000
```

Listing 1: FDSL showing the main language features. The example captures the Cassandra experiment described in Section IV.

the use of a random seed for experiments with random decisions such as killing a random replica. This is crucial for successfully reproducing experiments. If no seed is provided the system uses a seed based on the local time and outputs its value to the log at the end of the experiment (further detailed in Section III.) This allows running several instances of the same experiment under different conditions, while allowing to reproduce relevant experiments by rerunning them with the appropriate seeds. We also support the definition of the NTP [22] for clock synchronization at the environment level. This is relevant when running the experiments as further detailed in Section III.

The `events` describe the experiment timeline (Listing 1, lines 5–50). FDSL supports three types of events: `beginning`, `end` and `moment`. The `beginning` event describes the initial state of the experiment. This includes the system and benchmark binaries, configuration and number of instances. To capture the binaries and configurations we rely on containers as they already specify, in a concrete language (e.g. through a Dockerfile in the case of Docker containers [23], [24]), how the binaries are built, and what are the dependencies and version. Therefore, the `beginning` event describes the name of the container images to instantiate (lines 1–4), the number of container instances to start at the beginning of the experiment, and additional parameters that might be required (not shown in Listing 1).

The `end` event indicates when the experiment ends. At this point all the instances are terminated, and the system and application metrics are aggregated for posterior analysis as discussed in Section III.

The `moment` events describe changes in the state of the experiment, which can be `start` and `stop` to start and gracefully stop a replica, respectively, and `fault` to inject faults. These events apply to the specified `services` which correspond to a set of containers with the same functionality and configuration, following the Docker terminology. For instance lines 13–18 specify to start one new Cassandra replica 10 seconds after starting the experiment.

The `fault` event allows to inject a fault in the service replicas. Faults can be applied to one or more containers and different faults are supported, including custom ones. The specification of each `fault` follows a common structure, detailed in Listing 2. This structure includes the `target` and the `fault_type`. The `target` specifies the replicas where the faults should be injected. The `target` options are (1) `amount`, indicating the exact number of replicas affected, selected uniformly at random; (2) `percentage`, indicating the percentage of replicas affected when the fault is injected; and (3) `specific`, indicating the exact set of replicas that should be affected. Note that, for a given `target`, only one of `amount`, `percentage` or `specific` can be defined, however different faults can select different options for the same experiment. The `fault_type` indicates the type of fault to be injected. Currently, FDSL supports three fault types: `kill`, `cpu` and `custom`. `kill` specifies that the container should be killed. `cpu` specifies a CPU exhaustion fault by running a compute intensive task for a given period of time, hence modeling resource starvation or interference from other containers. `custom` allows to define custom faults.

```
- fault:
  target:
  # specify in how many/which
  # targets to inject the fault
  # the options below are
  # mutually exclusive
    amount: <natural number>
    percentage: <natural number>
    specific:
      - <replica id>
      - ...
      - <replica id>
  fault_type
```

Listing 2: Detailed structure of a fault event in FDSL.

Custom faults allow the researcher or practitioner to run an arbitrary script inside the containers, and inject application-dependent faults, for instance corrupting the data or log files of a database system. Listing 3 shows the structure of a custom fault. The parameter `kills_containers` indicates whether this fault results in a total failure of the container (i.e. a crash) or not. This information is used for two different purposes. First it allows checking if the application is resilient to this fault or not. If the fault is not supposed to crash the container (i.e. `kills_containers` is false) and the container ends up failing, then the user is warned about this and can proceed accordingly. The second reason is to allow for the correct computation of the `percentage` of containers that should be affected by a fault (Listing 2) as this should only affect containers that are alive at the time the fault is injected. The remaining parameters allow defining the location of the script, the execution environment and their parameters.

With an experiment described in FDSL, the next step is to run the experiment with the FaultSee platform which we describe in the next section.

B. Towards improved experiment reproducibility

Despite it conciseness, FDSL allows to capture the characteristics of an experiment. Our long-term objective is to allow researchers and practitioners to easily distribute, along with their code and documentation, the FDSLs used in their experiments, hence contributing to experiments that are easier to reproduce. In this spirit, the FDSLs that were used in the experiments of Section IV, together with FaultSee itself, are made available to the community[1].

III. The FaultSee Platform

In this section we describe the FaultSee platform that operationalizes experiments from a FDSL listing in a working

[1] Available at https://angainor.science/faultsee

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

Listing 3: Structure of a custom fault in FDSL.

infrastructure, providing access to a cluster of virtual or physical machines. The only requirment is for those machines to be acessible through SSH, either via password-based authentication or public key authentication.

The general architecture of FaultSee is depicted in Figure 1. FaultSee has three major components, namely the Master Controller (Section III-A), the Local Controller (Section III-B) and the Dashboard (Section III-C). The FaultSee platform is implemented in Python and Go.

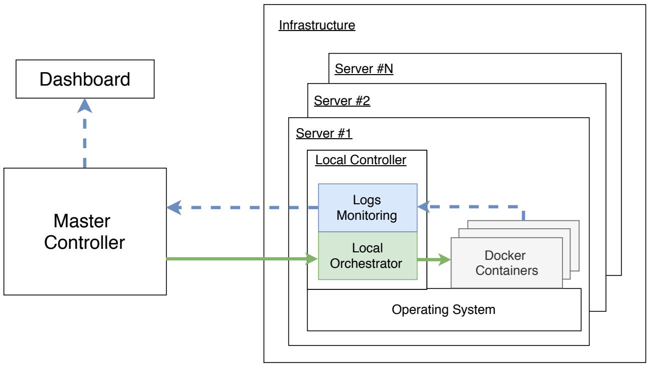


Figure 1: FaultSee architecture. Regular arrows represent control messages whereas dashed arrows represent telemetry information, namely logs and metrics.

A. Master Controller

The Master Controller is the main component of the platform: it is responsible for orchestrating the whole experiment. Users can interact with the Master Controller with scripted experiments via the command line, and with the web-based Dashboard. After receiving an FDSL file, the Master Controller performs three validation steps: (1) validation of the syntax of the experiment; (2) validation of the semantics of the experiment; and (3) validation of the cluster state. Step (2) ensures that the specified container images are available and that the described faults patterns are satisfiable. An example of an unsatisfiable pattern would be, for a cluster of 10 nodes, having one fault that kills 6 nodes followed by a fault that kills 5 nodes). Step (3) ensures that the cluster is operational and reachable. If all the validations are successful, the FDSL is converted into an *experiment plan* which is an internal representation describing the experiment. All random events are sampled and the replicas where faults need to be injected are specified. The result is a sequence of `(time, event, replica)` tuples which identify when, which, and where events need to be injected. The Master Controller then starts the Local Controllers on each of the cluster machines and distributes the *experiment plan* to each of them.

Pre-generating the experimental plan has several advantages. First, it simplifies the logic of the Local Controllers which simply have to go over the plan and implement it without any further processing logic. Second, it allows the Local Controllers to work in a fully independent manner without any sort of coordination with the Master Controller or other Local Controllers. This is essential for running large-scale experiments and to minimize the impact of FaultSee on network traffic by limiting the number of control messages. Finally, this allows the experimental plan to remain the same, regardless of the number of Local Controllers and placement decisions of the container orchestrator (such as Docker Swarm or Kubernetes) that might choose to place containers in different machines across different runs.

The Master Controller also synchronizes the machine clocks through NTP before starting the experiment. Since Local Controllers work independently during the experiment and follow the experiment plan by executing events at specified times, this synchronization step is important to bound the experimental error [25]. Once this step is complete, the Master Controller instructs all Local Controllers to start running the experiment.

B. Local Controller

As depicted in Figure 1, each machine in the cluster runs a Local Controller whose main responsibility is to apply the *experiment plan* that globally describes the experiment to the containers running locally. The Local Controller also gathers resource metrics from the local machine and each local container. These include: CPU usage, memory usage, network usage, and disk I/O. These metrics, along with the log events produced by the application containers, are kept locally during the experiment to reduce the impact of FaultSee on the network.

At the end of the experiment, the Local Controller sends the logs and telemetry information to the Master Controller, which then merges and chronologically sorts all the events. This information allows the researcher or practitioner to assess the experiment either by analysing the log files or by studying this information in the Dashboard.

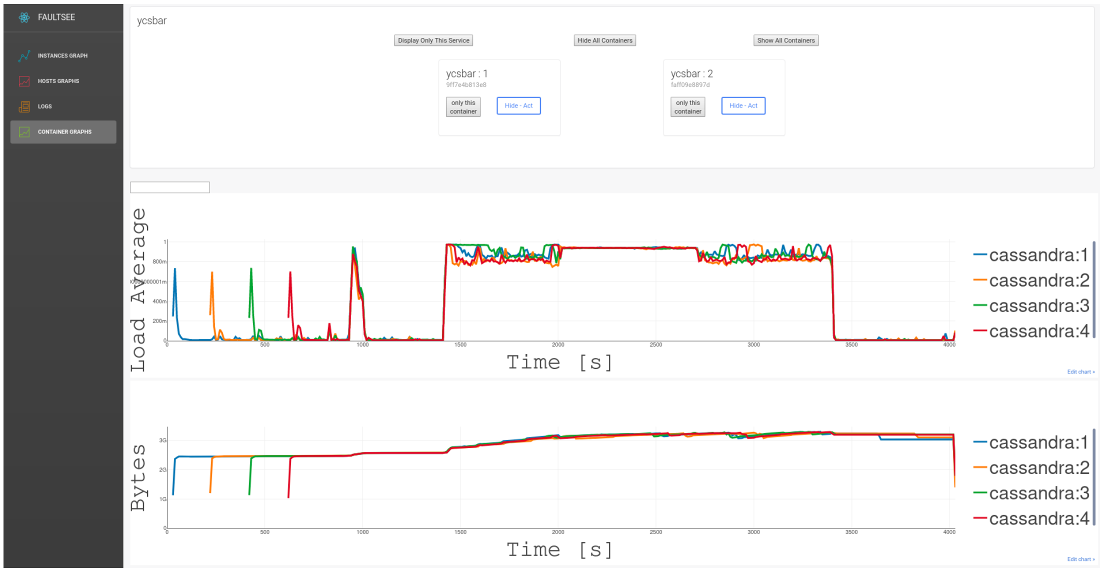


Figure 2: Dashboard example showing the average load and network usage for the Cassandra experiment.

C. Dashboard

The Dashboard provides a web-based interface to analyze the results of the experiments. It displays the following information:

- **Container Information:** information about each container that ran in an experiment, such as its full identifier and the service it belonged to. This allows to easily analyze specific containers after the experiment.
- **Container Events:** information about the lifecycle of containers, such as when they started and stopped.
- **Container Stats:** resource usage metrics of each container.
- **Hosts Stats** resource usage metrics of cluster machines.

Figure 2 depicts the average load and network usage metrics for the Cassandra experiment described in Listing 1.

IV. Evaluation

In this section we showcase the capabilities of FAULTSEE . We evaluate the behaviour of two systems: the Cassandra database [19] (Section IV-A), and BFT-Smart [26], a Byzantine Fault-Tolerant State Machine Replication system (Section IV-B). The Cassandra experiments were run in Google Computing Cloud with six *n1-standard-1* servers, each with 1 vCPU and 3.75 GB memory, running Ubuntu 16.04.6 LTS and Docker 19.03.4. The BFT-Smart experiments were run in Amazon Web Services with six *t2.medium instances* each with 2 vCPUs and 4 GB memory, running Ubuntu 16.04.6 LTS and Docker 19.03.4. We chose different cloud providers to highlight the fact that FAULTSEE abstracts the underlying infrastructure, allowing one to easily deploy the same experiment in different cluster.

A. Cassandra

As a starting point, we first consider the running example of Apache Cassandra introduced in Listing 1. The experiment consists of starting a Cassandra cluster with four nodes, subject the cluster to a workload, inject a fault during this period and assess the impact on throughput. The workload was injected by YCSB, a general benchmarking tool for NoSQL databases [21]. We used two YCSB clients, each with 10 threads and default configurations[2]. As the workload, we selected YCSB workload A, which corresponds to a mix of 50% read and 50% write operations. Note that our goal in this evaluation is not to stress test Cassandra but rather to illustrate the type of experiments enabled by FAULTSEE . We use Cassandra version 3.11.4 configured with a replication factor of 3, and YCSB version 0.14.

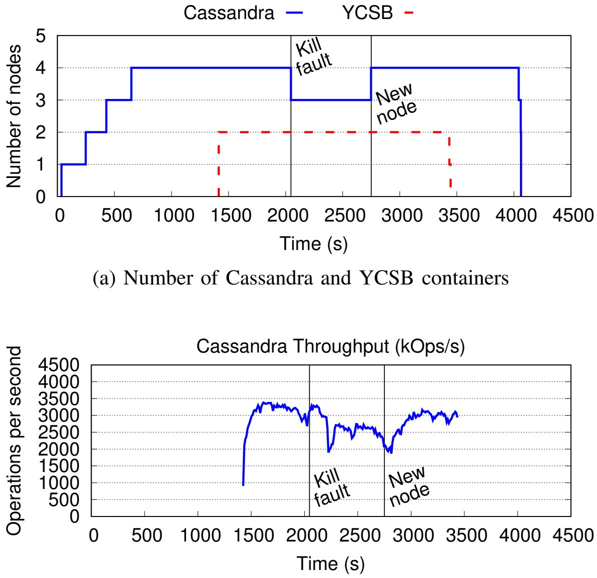
We evaluate Cassandra under two fault scenarios. The first scenario is the one described in Listing 1 where a Cassandra replica is killed approximately 33 minutes (2000 seconds) after starting the experiment, and a new replica is added 12 minutes later (2700 seconds)[3]. In the second experiment, we run the same scenario but instead of killing a replica, we inject a CPU exhaustion fault on a single replica. Expressing this in FDSL simply requires changing lines 45–48 of Listing 1 to specify a CPU exhaustion fault rather than a kill fault.

The results for the first scenario, where we kill a Cassandra replica, are shown in Figure 3. Figure 3a shows the number of Cassandra and YCSB replicas over time. Note that the number of Cassandra replicas is the one reported by Cassandra itself, hence when a replica is killed at second 2000, this is only reported by Cassandra 47 seconds later. The first vertical bar in the plot indicates when Cassandra considered the replica as failed and the second vertical bar when Cassandra recognized the new replica that was added to the system. We also show

[2] The default configuration can be seen at https://github.com/brianfrankcooper/YCSB/wiki/Core-Properties

[3] Note that for brevity the addition of a new replica after the fault is injected is not shown in Listing 1. This consists of a copy of the lines 19–25 but with the time set to 2700. The full experiment requires 78 lines of FDSL.

the lifecycle of YCSB replicas that inject the workload in the database. The resulting throughput is shown in Figure 3b. After the YCSB clients start, and the initial ramp-up period, throughput stabilizes at ≈ 3250 operations per second. After the fault, the throughput remains briefly unaffected and then drops dramatically before recovering to levels smaller than prior to the fault, due to the smaller capacity of the system which only contains three replicas rather than four. This is explained by the impact of the recovery protocol which needs to readjust the replication factor of the data that was in the failed replica. Interestingly, after the new replica joins the system the throughput also drops, due to the need to perform state transfer to the new replica, until it recovers to steady state levels before the fault.



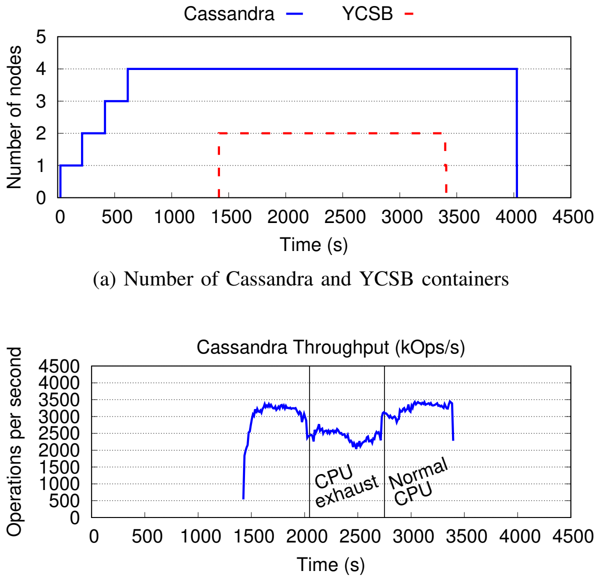
(a) Number of Cassandra and YCSB containers

(b) Throughput

Figure 3: Number of nodes and throughput, when a Cassandra replica is subject to a KILL fault.

Figure 4 shows the results for the experiment of injecting a CPU exhaustion fault in one replica. Figure 4a depicts the number of Cassandra replicas and YCSB instances, which in this scenario are not affected by crash faults. The throughput results are shown in Figure 4b where the vertical lines bound the time interval during which the CPU exhaustion fault occurred. We observe there is no sharp drop in throughput as in the previous scenario, although the overall throughput is smaller because one of the replicas has an overloaded CPU. As soon as the fault is removed the throughput stabilizes to its original level.

Interestingly, even though there is a larger variation in throughput in the KILL scenario than in the CPU exhaustion scenario, the average throughput of the former is $\approx 2655\ op/s$ while the average throughput of the latter is $\approx 2453\ op/s$. While we expect this relation to be reversed had the faulty period been shorter, this illustrates the impact of stranglers (i.e. slow replicas) in the overall system performance [27].



(a) Number of Cassandra and YCSB containers

(b) Throughput

Figure 4: Number of nodes and throughput, when a Cassandra replica is subject to a CPU exhaustion fault.

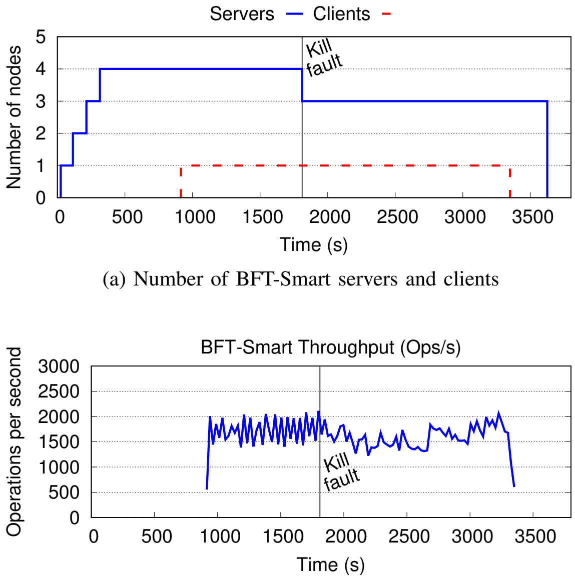
B. *BFT-Smart*

As we did for Cassandra, we also show two fault scenarios for BFT-Smart: a kill fault and a CPU exhaustion fault. The results for the kill scenario are depicted in Figure 5. We first start the cluster, then start the clients, and finally kill one of the replicas after ≈ 30 minutes, as can clearly be observed in Figure 5a. The throughput results are shown in Figure 5b. We observe that the fault initially negatively impacts the throughput, but eventually the system recovers. This is because BFT-Smart has full data replication and the remaining replicas are able to accommodate the offered load. The FDSL of this experiment is similar to the one presented in Listing 1 and the full experiment can be described in just 58 lines (not shown).

In the second experiment, we run a similar scenario but rather than killing a replica, we inject a CPU exhaustion fault on a single replica after ≈ 30 minutes. Results are shown in Figure 6. The number of replicas remains constant during the time the client is injecting the load, as can be observed in Figure 6a. Throughput results for this experiment are shown in Figure 6b. As expected, overloading a replica with a CPU exhaustion fault negatively affects throughput. The throughput recovers to the original level as soon as the fault is removed from the system.

C. *Discussion*

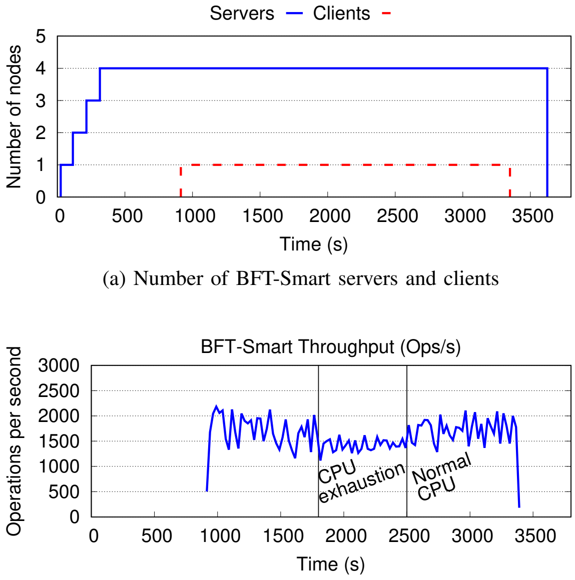
Despite their simplicity, these experiments show that it is possible to succinctly and precisely describe complex scenarios with several machines and fault patterns, thus allowing to assess the behavior of a system under a wide range of scenarios. Besides, by having a minimal set of requirements regarding the target cluster where the experiment is run, it is easy to deploy experiments in cloud environments, such



(a) Number of BFT-Smart servers and clients

(b) Throughput

Figure 5: Number of nodes and throughput, when a BFT-Smart replica is subject to a KILL fault.



(a) Number of BFT-Smart servers and clients

(b) Throughput

Figure 6: Number of nodes and throughput, when a BFT-Smart server is subject to a CPU exhaustion fault.

as we did for Amazon Web Services and Google Cloud, as well as on local clusters. Finally, because FDSL fully captures the experiment dynamics and workload, the process of reproducing experiments is greatly simplified.

V. Related Work

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

Cords [28] is a framework developed with the purpose of injecting faults in file systems, to test distributed applications. Ganesan et al. [28] show that redundancy, in fact, does not imply fault tolerance by uncovering a series of bugs in eight popular distributed file systems. This shows the need for incorporating better mechanisms into the development lifecycle and the need to have better tools to subject distributed systems to faulty scenarios in a systematic way.

To motivate its engineers to build more fault tolerant tools, Netflix created the SimianArmy [29] set of tools, of which the ChaosMonkey is the most popular. 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 these faults will likely become failures when engineers are most prepared to deal with any issue that may arise.

Pumba [30] is a tool 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 tools do not provide a succint and systematic way to describe fault scenarios hence hampering experimental reproducibility. Additionally, FaultSee enables its users to gather resource usage metrics, that are available at the end of the experiment in a dashboard.

FEX [31] 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. However, FEX does not encompasses fault injection nor the life-cycle of the system and its clients

Dfuntest [32] 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.

VI. Conclusion

In this work we presented FaultSee , a platform that enables the simple, automated, and reproducible execution of fault scenarios in distributed systems, and the FSDL language for succintly describing those experiments. FaultSee solves an important problem, as distributed systems are operating at an increasing scale and with an increasing number of complex and heterogeneous components. This complexity, in turn, makes it harder for researchers, practictioners and engineers to reason about the system and its behavior under different fault conditions. By automating and systematizing part of this process, we hope that FAULTSEE is a step towards more reproducible experiments, thus more dependable distributed systems. Our internal usage of FAULTSEE both to conduct systems research and as a tool for the experimental work of students shows that the systematization of the experiments greatly reduces the overall experimentation time and smooths the learning curve of newer students and researchers who start to work on an existing projects modeled with FAULTSEE .

FAULTSEE greatly simplifies the process of reproducing experiments: only the binaries with the configuration files and the FDSL desribing the original experiments are required. FAULTSEE includes a dashboard automating tasks such as the generation of plots of resource usage, and can be extended with custom plugins. It supports a wide range of deployment environemts as we illustrated with experiments in both Amazon Web Services (AWS) and Google Cloud Platform (GCP). We showed the ease of use and versatility of FAULTSEE by creating a benchmark for a CPU exhaustion scenario for two different systems, *Apache Cassandra* and *BFT-Smart*, that only required changing the names of the services and the deployment configurations in the configuration file.

As future work, we plan to extend FAULTSEE to support more types of faults such as network faults. We also want to support different experiment flows. Currently, FAULTSEE only a linear flow since faults are injected based on the time elapsed since the beginning of the experiment. Allowing the language to express experiments with conditional flows and loops allows researchers to express more complex experimental scenarios in a concise way. We plan to add support for expressing experiments based on the observed state, for instance when throughput reaches a given value, or else inject a fault or start a client only after all the server replicas have been started successfully. This would allow a greater expressiveness in FDSL, and hence expand the types of fault scenarios that can be modeled and simulated.

Acknowledgments: We would like to thank the anonymous referees for their valuable comments and suggestions, which improved the presentation of the work and will be useful as we extend the toolkit features. This work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, with reference UIDB/50021/2020 and project Lisboa-01-0145- FEDER- 031456 (Angainor).

REFERENCES

[1] "Google Cloud Networking Incident 18012," https://status.cloud.google.com/incident/cloud-networking/18012?m=1., 2019.

[2] "GitHub incident history," https://www.githubstatus.com/history, 2019.

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

[4] "Microservices deathstar," https://www.deployhub.com/configuration-management-how-to-navigate-the-microservice-deathstar/, 2019.

[5] 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, 2008.

[6] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5 SPEC. ISS., 2004.

[7] S. M. e. a. Blackburn, "The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 4, 2016.

[8] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. ACM, 2009.

[9] ——, "Evaluating the accuracy of java profilers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. ACM, 2010.

[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. ACM, 2006.

[11] J. Vitek and T. Kalibera, "Repeatability, reproducibility and rigor in systems research," in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. IEEE, 2011.

[12] "ACM SIGMOD Reproducibility Award," 2020. [Online]. Available: https://sigmod.org/sigmod-awards/sigmod-most-reproducible-paper-award/

[13] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection," *ACM Computing Surveys*, vol. 48, no. 3, 2016.

[14] "Docker Swarm," https://docs.docker.com/engine/swarm/, 2019.

[15] "Kubernetes," https://kubernetes.io/, 2019.

[16] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "Reprozip: Computational reproducibility with ease," in *International Conference on Management of Data*, vol. 26-June-2016. Association for Computing Machinery, 2016.

[17] Reprozip, "ReproZip! the reproducibility packer," 2019. [Online]. Available: https://www.reprozip.org/

[18] GNU Guix , "Guix," 2019. [Online]. Available: https://guix.gnu.org/

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

[20] "YAML Specification Index," https://yaml.org/spec/, 2020.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM symposium on Cloud computing*. ACM, 2010.

[22] J. Martin, J. Burbank, W. Kasch, and P. D. L. Mills, "Network Time Protocol Version 4: Protocol and Algorithms Specification," RFC 5905, 2010. [Online]. Available: https://rfc-editor.org/rfc/rfc5905.txt

[23] Docker Inc., "Docker Documentation," 2018. [Online]. Available: https://docs.docker.com/

[24] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, 2015.

[25] C. D. Murta, P. R. Torres Jr., and P. Mohapatra, "Qrpp1-4: Characterizing quality of time and topology in a time synchronization network," in *IEEE Globecom 2006*, 2006.

[26] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014.

[27] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *In {USENIX} Symposium on Networked Systems Design and Implementation*, 2013.

[28] 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, 2017.

[29] A. Tseitlin, "The antifragile organization," *Communications of the ACM*, vol. 56, no. 8, 2013.

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

[31] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer, "Fex: A Software Systems Evaluator," in *IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2017.

[32] G. Milka and K. Rzadca, "Dfuntest: A testing framework for distributed applications," in *Lecture Notes in Computer Science*. Springer, Springer Nature, 2018, vol. 10777 LNCS.