TÉCNICO LISBOA

Reproducible Fault Injection in Distributed Systems

Daniel David Marques de Castro

Dissertation to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Miguel Ângelo Marques de Matos
Prof. Shady Alaaeldin Mohamed Abdelkader Rabie Issa

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. João Tiago Medeiros Paulo

November 2021

Acknowledgments

First, I would like to thank Professor Miguel Matos for his guidance in the correct path in challenging adversities, for his scientific vision, and for the pertinent suggestions that improved this work and my way of critical thinking. I would also like to thank Professor Shady Issa for his auxiliary help.

I also acknowledge the role of Instituto Superior Técnico in supporting my research. This work was partially funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia within the scope of the UT Austin Portugal Program in project ACT-PM (UTA-EXPL/CA/0080/2019) and Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa and by Fundação para a Ciência e Tecnologia (FCT) through project Angainor (LISBOA-01-0145-FEDER-031456).

Finally, I would like to dedicate this thesis:

to Tiago Castro, my brother, David Castro, my father, and Gisela Castro, my mother, for their dedication, confidence, continuous respect, persistence, and support. Pillars of my whole life and knowledge.

to Daniela Alberto, for the unconditional love, for our marvellous affinity and for giving me the strength to accomplish virtually anything together. My ethereal muse;

to all my friends, my gratitude for being by my side and for all the simple moments that create unforgettable memories.

To them, I owe the strength to approach this thesis with truth and dedication.

Resumo

Os Sistemas Distribuídos são a base para aplicações modernas usadas por milhões de utilizadores diariamente. Apesar de estes sistemas serem desenhados e construídos por cima de protocolos de tolerância a faltas, mais frequentemente do que pensamos, estas aplicações continuam a falhar devido a faltas inesperadas e imprevisíveis no sistema. Estas falhas causam enormes prejuízos tanto para o utilizador final como para a empresa que fornece o serviço. Por isso, é fundamental ter um método sistemático e reproduzível para avaliar a resiliência e robustez destes sistemas distribuídos.

O método de análise de sistemas distribuídos é desafiador e difícil devido à grande complexidade e natureza não-determinista dos mesmos. Metodologias de injeção de faltas, tal como *Chaos Engineering*, são usadas pelas maiores empresas de IT para avaliar o comportamento dos seus sistemas após injetarem faltas aleatoriamente. No entanto, devido às faltas não serem baseadas no estado do sistema, a mesma falta pode causar comportamentos diferentes em execuções diferentes. Engenheiros devem conseguir reproduzir faltas com facilidade para perceberem melhor o problema e corrigi-lo.

Nesta dissertação, nós propomos o ReFI (*Reproducible Fault Injection*), uma ferramenta de injeção de faltas reproduzíveis que deve eficientemente obter o estado do sistema distribuído e injetar faltas baseadas no mesmo para reproduzi-las em diferentes execuções. Tratamos o sistema como uma caixa-negra, obtendo informação sobre as chamadas de sistema (*system calls*), para podermos utilizar o ReFI em diferentes sistemas. O ReFI utiliza a tecnologia eBPF, que permite a inserção de código dinamicamente no kernel do Linux. ReFI reproduziu três bugs críticos em dois sistemas distribuídos, MongoDB e Redis, utilizando um ficheiro de configuração para especificar os sistemas e as faltas.

Palavras-chave: Sistemas Distribuídos, Reprodutibilidade, Injeção de Faltas, Robustez, eBPF

Abstract

Distributed systems are at the core of modern applications that are used daily by millions of users. Although these systems are designed and built on top of fault-tolerant techniques to avoid failures, most often than we think, these applications still fail due to unexpected and unpredictable faults in the system. These failures cause severe harm to both the user and the company that provides the service. Therefore, it is fundamental to have a systematic and reproducible way to evaluate these distributed systems' resilience and robustness.

This assessment method is challenging and difficult because of the high complexity and non-deterministic nature of distributed systems. Fault injection approaches, such as Chaos Engineering, are used by major IT companies to evaluate their system's behavior after randomly injecting faults. However, the same fault can cause different system behaviors due to faults not being based on the system state. Engineers should be able to reproduce faults with ease to better understand the problem and correct it.

In this dissertation, we propose ReFI, a Reproducible Fault Injection tool that traces the distributed system state efficiently, in a black-box manner, and injects faults based on the state to reproduce these faults across different executions. ReFI utilizes eBPF technology that enables the insertion of dynamic code in the Linux kernel. ReFI reproduced three critical bugs in two distinguished distributed systems, MongoDB and Redis, using a configuration file to specify the systems and the faults.

Keywords: Distributed Systems, Reproducibility, Fault Injection, Robustness, eBPF

Contents

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

1 Introduction **1**
1.1 Topic Overview . . . 2
1.2 Objectives . . . 3
1.3 Thesis Outline . . . 5

2 Background **7**
2.1 Fault, Error, Failure . . . 7
2.2 eBPF . . . 8

3 Related Work **11**
3.1 Fault Injection Tools . . . 11
3.1.1 Chaos Engineering . . . 12
3.1.2 FCatch . . . 14
3.1.3 Bounded Black-box Crash Testing - B^3 . . . 16
3.1.4 PACE . . . 20
3.1.5 Lineage-driven Fault Injection - MOLLY . . . 22
3.1.6 Faultsee . . . 25
3.1.7 Elle . . . 27
3.2 Tracing Technologies . . . 28
3.2.1 CAT . . . 28
3.3 Bug Analysis . . . 30
3.3.1 Jepsen . . . 31

3.3.2 Partial Network Partitioning . . . 31
3.4 Discussion . . . 33

4 ReFI **35**
4.1 Architecture . . . 37
4.2 Tracing Module . . . 39
4.3 Fault Injection Module . . . 43
4.4 Orchestrator . . . 46
4.4.1 Configuration submodule . . . 47
4.4.2 Submodules Management . . . 49
4.4.3 System State . . . 51
4.4.4 Summary . . . 52

5 Evaluation **53**
5.1 Methodology . . . 55
5.1.1 MongoDB Background . . . 57
5.1.2 Redis Background . . . 58
5.2 Overhead . . . 59
5.2.1 ReFI . . . 59
5.2.2 TM . . . 60
5.3 Experiments . . . 64
5.3.1 Unavailability - MongoDB . . . 65
5.3.2 Data Loss - MongoDB . . . 67
5.3.3 Split-brain - Redis . . . 72
5.4 Discussion . . . 76

6 Conclusions **79**
6.1 Future Work . . . 80

Bibliography **83**

List of Tables

3.1 Comparison between state-of-the-art fault injection approaches and ReFI. 34

5.1 YCSB workloads summary 56

5.2 Time overhead of ReFI using the YCSB workload A for environment without ReFI, with ReFI and with ReFI and verbose mode (ReFI -v). The [R] and [W] in the latency column stand for Read and Write latency. 60

5.3 Time overhead of ReFI using the YCSB workload F for environment without ReFI, with ReFI and with ReFI and verbose mode (ReFI -v). The [R] and [W] in the latency column stand for Read and Write latency. 61

5.4 Task 1: Raw run time overhead of the TM module. In this table, we present the results of two syscalls, ss_sendmsg (security socket send message) and ss_recvmsg (security socket receive message), that compose the network submodule. 63

5.5 Task 1: Raw run time overhead of the TM module. In this table, we present the results of three syscalls, vfs_read, vfs_write, and vfs_open, that are part of the VFS submodule. 64

5.6 Task 2: Average, minimum and maximum values obtained from only the wanted syscalls from TM module. 64

List of Figures

2.1 A fault can activate an error. The error can propagate to a system failure. 7

2.2 The architecture of an attachment of an eBPF program using the BCC framework [26] 8

3.1 The principles to design Chaos Engineering experiments [15]. 12

3.2 TOF bug example on a Hadoop-MapReduce execution [15]. 15

3.3 FCatch flow [15]. 15

3.4 An existing crash-consistency bug that existed in the kernel since 2014. 17

3.5 B^3 approach to test file-system crash-consistency bugs [49] 17

3.6 The 3 phases of CrashMonkey, Record IO, Replay IO, and Auto Checker [49]. . . 18

3.7 The 4 phases of ACE [49] 19

3.8 PACE execution methodology [51]. 21

3.9 Overview of LDFI architecture [55]. 22

3.10 Lineage graph of an execution of a write in a distributed database system [55]. . 22

3.11 FDSL example for a Cassandra experiment, showing the environment attributes and several events specifications [37]. 25

3.12 Fautlsee platform architecture, with Master Controller, Local Controller and a Dashboard [37]. 26

3.13 Context-based (a) vs Content-aware (b) tracing example. [38] 29

3.14 Example of a partial network partition where groups 1 and 2 are disconnected but group 3 can communicate with group 1 and 2. [39] 32

4.1 ReFI architecture (in green) incorporated with a black-box application. The application is issuing syscalls that are handled by the Linux kernel. The TM (in blue), intercept these syscalls to trace the state. The Orchestrator synchronizes the state between the TM and the FIM (in red) to enable reproducible fault injection experiments. 37

4.2 ReFI TM is composed by several tracing eBPF programs that have a corresponding event handler in the Orchestrator. The events are filtered in the eBPF program, leaving no overhead for the Orchestrator, when the event is not useful. 40
4.3 ReFI TM communicates via an BPF ringbuf with events and via eBPF maps. . . 41
4.4 ReFI Orchestrator is composed by the configuration submodule, the tracing frontend, the fault trigger, and the state manager. 47
4.5 Representation of the experiment that the configuration file template describes. The green downwards arrow means that the network is working and the red cross means it is not working. ((P)-Primary, (S)-Secondary, (A)-Arbiter) 49

5.1 The two different approaches for testing ReFI with YCSB workloads. ReFI evaluation utilizes approach (a). 57
5.2 Diagram showing an overview of the MongoDB unavailability bug. We have 3 nodes, a Primary (P), a Secondary (S), and an Arbiter (A). After a partial network partition there is an infinite swap of the new primary. 66
5.3 Diagram showing an overview of the MongoDB data loss bug. We have 5 nodes, a Primary (P), and four Secondaries (S). After a complete network partition, there are two available primaries to write. After healing the fault, the data is lost from one of the primaries. The primary that wins depends on the state chosen in the configuration file. 68
5.4 Diagram showing an overview of the Redis split-brain bug. We have 3 nodes, a Master/Primary (P), and two Secondaries/Slaves (S). After a complete network partition, there are two available primaries, creating a split-brain bug. The two primaries have a different version of the data since both can write, in parallel, without communicating with each other. 73

Chapter 1

Introduction

Distributed systems are the dominant backbone infrastructure for modern applications that users rely on regularly for various purposes, such as e-commerce, social media, entertainment, banking, finance, healthcare, and many others. Thus, the more we rely on distributed systems, the more important it becomes to assess the robustness and reliability of these systems, where faults of different types frequently happen.

Faults lead to application failures that cause loss of productivity and, consequently, the loss of millions in revenue for both the organizations [1] that provides the service and the final user. Hence, it is crucial to test the dependability of these distributed applications to prevent system outages in the presence of faults.

System outages occur more often than we think due to faults caused by natural phenomenon's, such as lightning strikes [2], and even human error [3], but also by unexpected and unpredictable bugs in the software of IT companies [4, 5, 6].

Despite the numerous fault-tolerant designs at the core of distributed systems to prevent several faults from propagating and causing a system failure, faults still occur frequently. Faults are becoming the norm rather than the exception due to the high complexity of distributed applications, often composed of thousands of services and micro-services [7], increasing the chance of having system failures. It also brings tremendous difficulty in analyzing the system to understand which components failed, why they failed, and which components were affected, directly or indirectly.

Computer engineers and programmers who encounter these types of bugs need a precise methodology to reproduce these events to evaluate and analyze their impact on the system. After understanding how the bugs were produced and which parts of the system were affected, the engineers can rectify the bugs caused by these unexpected faults with more ease. In the process of reproducing the bugs, engineers are creating systems that are more robust and resilient

to faults in a more reliable way since they can test similar behaviours on their or other systems.

At present, it is extremely difficult and challenging to reproduce faults in distributed systems because of their non-deterministic nature and their complexity. In addition, the previous tracing tools and mechanisms used by the state-of-the-art approaches are not efficient enough due to their architecture or need to instrument all the code, which is unfeasible in the majority of distributed applications. These tools need to analyze millions of system calls to trace the distributed system and need to be exceptionally efficient for each call.

1.1 Topic Overview

Distributed applications are built on top of fault-tolerant mechanisms, model checking [8], and test-driven [9, 10] approaches. Unfortunately, debugging and testing the fault-tolerance guarantees of distributed systems is a notoriously complex task because distributed systems are highly complex and have a non-deterministic behaviour due to, for example, the latency of network communication between nodes or due to the complex algorithms used in distributed systems.

Nevertheless, these mechanisms and approaches have been shown to reduce the number of faults and improve the resilience and dependability of distributed systems. However, these techniques need to understand the system's model in detail to prevent faults, which can be tedious, time-consuming, and not extendable to other applications. On top of that, these techniques are used at the core of building such applications and do not test the system once it is finished. Thus, if we want to test the robustness of an already built system, these mainly do not apply to them.

Therefore, how can we assess the robustness and reliability of a distributed system that has already been built on top of these strategies without having to rebuild the whole distributed system? The dominant top-down approach in the software engineering and dependability communities is fault injection [11].

Fault injection [12, 13, 14, 15, 16, 17, 18, 19] mechanism tests the system dependability requirements with minimal programmer investment and can quickly identify bugs. Amazon has recently created a cloud service [20] that enables the use of Chaos Engineering fault injection approach in their instances. **Chaos Engineering** is used by leading IT companies, and it started to gain renown after the success of Netflix [21]. Then, Google [22], Microsoft [23], Facebook [24], and Amazon [22] started using this approach too. Chaos engineering creates experiments that randomly inject faults into the system and evaluate several key system metrics.

Since, in Chaos Engineering, these faults are injected randomly and hence do not depend on the system state, the same fault can reflect different behaviours. For example, it can lead to a

system failure, or it can lead to a correct system recovery, depending on the system state. These behaviours happen due to the non-deterministic nature of distributed systems, where the same fault may not always hamper their correctness or performance. This non-deterministic behaviour is the origin of a complicated and challenging problem in distributed systems that engineers and programmers are continuously facing when trying to understand and fix these types of bugs.

Accordingly, to ensure rigorous testing, we need to deterministically reproduce faults in a given system. This dissertation aims to explore the idea of using a system's internal state in a black-box manner (via system calls, I/O, logs, and more) to coordinate fault injection with the state, enabling engineers to correctly reproduce - and fix - incorrect system behaviour. Other state-of-the-art fault injection approaches either are not able to reproduce bugs created by their faults, such as chaos engineering, or are specific to a system type, such as file-systems [17], storage systems [18], or they intend to catch a specific type of bugs, such as TOF bugs [15], or crash consistency bugs [18]. Our approach, ReFI, a Reproducible Fault Injection tool, intends to improve the state-of-the-art in these aspects.

For ReFI to accomplish these goals, great challenges arise. Throughout this dissertation, we will discuss four goals and four key challenges. The goals of this dissertation are to create ReFI, a **reproducible**, **black-box**, **efficient**, **extendable and flexible** prototype. The four key challenges are the **high complexity** of distributed systems, their **non-deterministic** nature, the difficulty to **reproduce experiments** and, correlated to the latter, the problems with **obtaining a system state**.

This document will describe how we approached our goals, how we discussed these complex challenges, how we hypothesized different solutions to them, and why we chose a specific solution over others.

1.2 Objectives

This dissertation has the following four core goals:

Enable injection of faults to be **reproducible**, analyze the system behaviour by treating it as a **black-box** application, accomplishing this in an **efficient** manner, and be **extendable** for different fault patterns and system states as well as **flexible** to different types of distributed systems (database systems, storing systems, file-systems, and more).

The core idea to enable reproducible fault injection is to coordinate fault injection with the system state, as opposed to state-of-the-art time-based or random-based fault injection approaches, such as Chaos Engineering.

In this dissertation, we divided the system state into two categories, uni-state and multi-state.

As the name indicates, a state can be uni-state when it is derived from the analysis of a single source of application output (system calls, logs, I/O, and more), or multi-state, derived from the analysis of two or more sources of application output.

To obtain the system state, we treat the application as a **black-box**. A black-box application means that we do not analyze the inside of the application. We only analyze the distributed application's outputted information to the exterior, such as system calls, logs, I/O, and more. For example, we do not need to know what complex algorithms each system uses nor understand the code each system implements. There are state-of-the-art approaches [25] that utilize the idea of understanding the source code and its algorithms. Still, it comes with the disadvantage that developers need long periods to understand a single system due to the high complexity of each distributed system and the majority of the knowledge acquired is not tangible to other systems.

When we talk about **efficiency**, we are mainly talking about the latency, throughput, and overhead that the ReFI prototype adds on top of the application. These metrics should be as efficient as possible to enable engineers and programmers to easily and rapidly utilize our tool. The bottleneck of the fault injection experiment should not be the tool that is testing the robustness and resilience of the system. Also, the overhead, in this particular case, should be as small as possible because we are dealing with thousands, or even millions, of syscalls per second. The slightest overhead can be transformed into an enormous handicap/inefficiency of the tool.

The fourth and final goal is to create a tool that not only can be **extendable** to different distributed systems and also **flexible** to various outputs of the applications, such as different system calls, and different logs. This extendability and flexibility come from the way that ReFI was designed. By treating the application as a black-box and using eBPF as a technology to accomplish this, we can change the distributed system being tested with ease. On top of that, we created a mechanism to specify the types of faults and the system nodes, by simply specifying them in a configuration file, which largely improves its easiness to be extendable to other distributed systems and fault patterns.

To achieve these goals, we propose ReFI, a **Reproducible Fault Injection** tool that could be used in different distributed systems. ReFI should be able to inject faults in a (1) reproducible manner, using a (2) black-box approach, (3) in an efficient way, and (4) it should enable the creation of different experiments described in a configuration file. To accomplish these goals, we intend to use eBPF [26], a Linux kernel technology that enables the dynamic insertion of code in the Linux kernel that can be used for various purposes such as tracing, profiling, monitoring, and controlling the logic within Linux itself. With eBPF, we can collect an internal state of the system by only analyzing its system calls, meaning that it is a black-box approach since we

do not need to understand the application model and its algorithms. The state will be used to precisely inject faults in specific system states, thus being reproducible. eBPF gives us a negligible overhead because all the infrastructure of this technology is already built in the Linux kernel, and we are only injecting a few lines of code into the already called system call. We tested our ReFI tool with MongoDB and Redis, which are modern distributed applications used by millions of users.

With the ReFI tool, we made the first critical step to prove that fault injection can be reproducible by testing it on two different bugs in MongoDB, a database for modern applications used by millions of users and companies, and one critical bug in Redis, an in-memory data structure store, used as a database or cache for thousands of applications. This ability to reproduce fault injection experiments is a novelty on the state-of-the-art that will help developers and engineers to attack these complex and difficult bugs more efficiently and improve the research area of reproducible fault injection.

As main results, we highlight three key accomplishments. ReFI, a working prototype that was used to reproduce bugs in two real systems, MongoDB and Redis, as well as different types of faults. A presentation of our work at the Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED) 2021 workshop [27] in collaboration with ACM Symposium on Principles of Distributed Computing conference (ACM PODC) [28]. Finally, a collaboration with a colleague working on Kollaps [29], a network emulator, that is also using eBPF to surpass network problems.

1.3 Thesis Outline

The remainder of this document is organized as follows. At the beginning of each chapter, there is a small summary, and at the end, there is a quick recap of the most important solutions and arguments. In Chapter 2, we define basic concepts that are essential to understand our work, such as faults, errors and failures, and describe the eBPF tool, which dynamically inserts code on the Linux kernel. In Chapter 3, we discuss the state-of-the-art approaches on fault injection, their frameworks and prototypes. We also discuss state-of-the-art kernel tracing tools and studies on current bugs in popular systems. In Chapter 4, we detail the architecture of ReFI, a Reproducible Fault Injection tool, as well as different solutions that we analyzed to accomplish our goals (reproducible, black-box, efficient, flexible and extendable). We describe the three main modules of ReFI, Tracing Module, Fault Injection Module, and the Orchestrator and their implementation. In Chapter 5, we present the methodologies that we used to demonstrate the efficiency and the utility of ReFI, show the results of the evaluation, and the fundamental

experiments of ReFI in distributed systems. Finally, in Chapter 6, we conclude the document and discuss how we accomplished the main goals, as well as the future work.

Chapter 2

Background

In this section, we provide a background on basic concepts relevant to our work. Our tool, ReFI, utilizes the injection of faults in distributed systems to catch bugs. The concepts of fault, error, and failure are thus explained to avoid ambiguity and to understand the basic flow of why injecting faults creates system failures. In Section 2.2, we briefly describe eBPF, which is a technology that we will use in our approach.

2.1 Fault, Error, Failure

To understand why fault injection can improve the robustness of an application, it is crucial to understand how a fault can generate a failure.

A **fault** is the adjudged or hypothesized cause of an error.

An **error** is the part of the system's total state that may lead to its subsequent failure.

A **failure** is an event that occurs when the delivered service deviates from the correct service.

For example, a developer can accidentally delete a code line, producing a fault in the system. This code, when executed, can trigger a fault that activates an error in a system component. This error propagates to a system failure where the system crashes because it does not handle that particular line missing. This is a simple example of the flow of a fault, error, and failure.

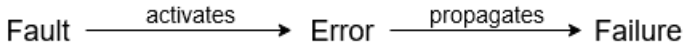


Figure 2.1: A fault can activate an error. The error can propagate to a system failure.

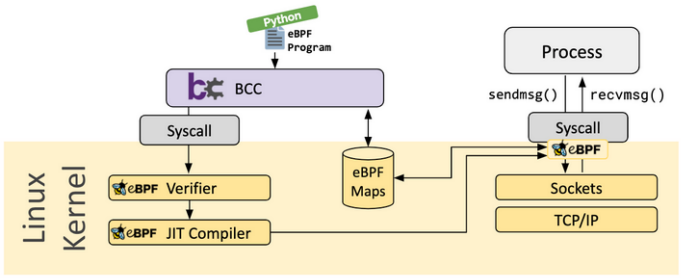


Figure 2.2: The architecture of an attachment of an eBPF program using the BCC framework [26]

2.2 eBPF

eBPF is a relatively new Linux kernel technology that enables the dynamic insertion of a Linux kernel program that can be used for monitoring, security visibility, and control logic within Linux itself. eBPF programs are event-driven, and they execute when a hook point is passed. Typically, these hooks are system calls (syscalls) but can also be network events, kernel tracepoints, kernel probes, among others. While not being executed, the eBPF programs have a zero cost overhead [30]. This shows a better performance, while not in execution than tools such as PIN [31], that injects code into the application, and even if the code is not executed, it still is checked, resulting in unnecessary overhead.

Since eBPF programs run inside the Linux kernel, they must follow specific security policies, such as not enabling dynamic loops. These eBPF programs can be updated and injected to the Linux kernel without any changes to the application code or configuration. The eBPF programs are written in the form of bytecode but fortunately, there are abstractions on top of eBPF bytecode that facilitate the writing of an eBPF program such as *BCC* [32], *BPFtrace* [33], *Cilium* [34], and *libbpf* [35]. We used libbpf-bootstrap [36] as an abstraction, to create ReFI using C in the frontend and a pseudo-C in the attached eBPF program to the Linux kernel. The pseudo-C code is then compiled to bytecode. An important difference between C and this pseudo-C is that pseudo-C does not allow loops, to avoid infinite loops that could cause the kernel to crash.

A key component of eBPF is sharing information between different eBPF programs through a built-in data structure known as **maps**. These maps can be arrays, stacks, or a key-value data structure (similar to maps in other programming languages). These maps can store state data and are stored in a different stack than the eBPF program itself. Since the eBPF program is loaded to the kernel, huge data structures would take all the small available space, but because the maps are in a different stack, this is not a problem.

In Fig. 2.2 we can see the architecture of an eBPF program that runs on top of the BCC

abstraction. The program is attached to a specific network system calls (syscall), and it is executed before the syscall return to the application, storing data in the eBPF Maps. Before the eBPF program is attached, it must be verified to avoid kernel crashes and translated to machine code by the JIT compiler. It is possible to have multiple eBPF programs monitoring different syscalls and sharing the state through Maps.

eBPF offers a robust, efficient, and secure mechanism to attach functionality to the Linux kernel to monitor applications and inject faults.

Chapter 3

Related Work

In Section 3.1, we discuss the state-of-the-art fault injection approaches as well as present their advantages, disadvantages, key solutions and ideas and what can be improved. In Section 3.2, we consider the main tracing tools that are used in the Linux kernel environment in state-of-the-art approaches and describe their applications in specific types of applications. In Section 3.3, we analyze current bugs of real and modern distributed systems that are documented. In Section 3.4 we summarize, in a table, the main aspects of the state-of-the-art fault injection approaches and a comparison to what ReFI improves on each of them. We also summarize and discuss essential conclusions outside of fault injection approaches such as configuration language details from Faultsee, why we used eBPF instead of Strace from CAT conclusions and specific types of bugs that ReFI enables us to test.

3.1 Fault Injection Tools

Fault injection is a mature topic in distributed systems [12, 13] that has been growing quite rapidly in the last decade. Since distributed systems have critically failed and caused significant damage to applications multiple times due to faults [2, 3, 4, 5, 6], approaches to make these systems more reliable have appeared. Fault injection mechanisms are the main approach to test the resilience, dependability and understand the impact of system component's failures.

In this section, we will discuss the state-of-the-art approaches to fault injection. Some of these approaches only try to find specific failures/bugs in distributed systems, such as FCatch [15] and B^3 [17]. Others are less specific and try to find a variety of bugs, such as LDFI [11] and Chaos Engineering [14]. Both have their advantages and disadvantages and we are going to explain and discuss them.

Besides state-of-the-art fault injection approaches, we will discuss important key mechanisms

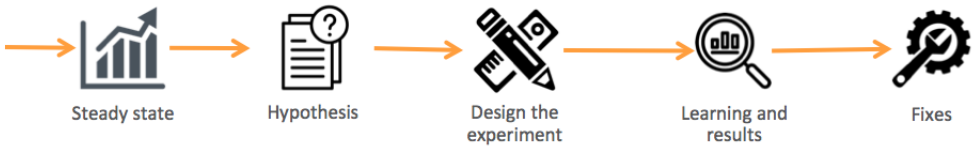


Figure 3.1: The principles to design Chaos Engineering experiments [15].

that the ReFI framework infrastructure utilizes. We need to have a language that is simple and describes these fault injection experiments in a concise way and to reproduce them like Faultsee [37]. From CAT [38], we can understand eBPF advantages, in a similar concept, over other technologies, such as Strace. Finally, but not least, analyze distributed system bugs [25, 39] of, partial and complete, network partitions issues that were presented in different distributed systems, including MongoDB, and test ReFI with these complex bugs.

3.1.1 Chaos Engineering

Chaos Engineering *"is the discipline of experimenting on a distributed system to build confidence in its capability to withstand turbulent conditions in production"* [14], i.e., by testing a distributed system with faults injection and observing how it reacts, we can build trust on the system's robustness. The distributed system is tested by injecting simulated real-world faults such as temporary network partitions, malformed data inputs, drop of incoming requests, a large number of requests in a short time to congest the network traffic, among others.

In Chaos Engineering, engineers tend to look at the user response and related metrics rather than if the system is behaving like the specification. In other words, after injecting a fault, if the system is not too slow and the client receives a correct response, the system is considered correct although the specification may not be fully met. The definition of *correct* depends on each system.

The base of Chaos Engineering is to test the resilience of the system by running thousands of experiments. Then, with the help of metrics, analyze the results and improve the robustness by correcting bugs. The four principles to design Chaos Engineering experiments are: i) build hypotheses around a steady-state behaviour, ii) vary real-world events by designing experiments, iii) run these experiments in production and analyze the results, and iv) automate experiments to run continuously. In Fig. 3.1, we have a simplified scheme that shows the principles in practice.

A steady-state behaviour depends on the system itself but in general, could be if the end-user can access what the application has to offer. For example, a metric for Netflix would be if the

user can watch streams. For an e-commerce application, such as Amazon, would be if the user can buy products. To build a hypothesis on this, we need to understand what metric should we analyze. For example, a metric could be streams started per second in Netflix or completed purchases per second in the e-commerce application.

After finding a metric that can measure the normal behaviour of the system, we start to inject faults that may happen at the production level. Principle ii) and iii) depend on each other since we can only run experiments in production after varying the real-world events. Examples are terminating virtual machines, dropping messages, among others. These faults are randomly applied to the system and do not follow a model to inject faults. This is one of the major downsides of this approach and the main reason that thousands of faults need to be injected at different times to detect bugs. Most of the time, only a small amount of bugs are found, giving the wrong impression that the system is robust and even bug-free. Bugs can be time and state sensitive and/or depend on coordinated behaviours.

Finally, automation of this approach is needed, not only by varying events and the number of faults to inject but also by the timings to when the faults start and their regularity. The goal is to plan an experiment and apply it, e.g. crash a node for 10 minutes, gather the results of the metrics to analyze later, recover that node after 1 minute, and repeat for each experiment.

The first dominant company using this approach was Netflix [21] but many large tech organizations such as Amazon [22], Google [22], Microsoft [23], and Facebook [24] started using Chaos Engineering to test the resilience of their distributed systems.

Over the last decade, Netflix has created the Simian Army [40] that has been developing various tools that are based on Chaos Engineering. The most famous one is ChaosMonkey 2.0 [21]. Other tools are Chaos Kong [41] and ChAP [42]. FIT [43] is an auxiliary framework for these tools that perform fault injection.

Chaos Monkey [44] is a tool that randomly terminates virtual machine instances and containers in a production environment. The goal is to expose engineers to the failures that are *hidden* in the complexity of their systems and to fix them before it harms the clients or become unmanageable. The name of the tool itself, Chaos Monkey, reassembles something like putting a monkey in a system that tries to destroy it uncoordinated and causes chaos.

Netflix also uses the tool **Chaos Kong** that performs exercises that simulate the failure of an entire Amazon EC2 region. **Failure Injection Testing (FIT)** simulates the loss of requests between Netflix services and verify that the system degrades gracefully.

This approach is simple to understand and easy to implement. Randomly killing processes and crashing virtual machines is not complex to do compared with other approaches using models

for injecting faults. Nevertheless, it has demonstrated practical robustness in distributed systems in the production environment, which allowed organizations to comfortably and confidently use this approach for their benefit.

However, random injection of faults may show a false sense of robustness because the majority of the faults injected do not produce bugs. Since the experiments are random and uncoordinated, most of the faults may be redundant and test similar parts of the system which makes it inefficient. Another important disadvantage is that faults are not reproducible. This means that because they are randomly injected and do not depend on the state of the system, they can not always recreate the same results. If we do the same steps, it likely will not cause the bug to appear again because the production system is in another state.

If by crashing a virtual machine, the system shows below the standards measures, developers and engineers still need to understand the root cause behind the failure, which can be a tedious and time-consuming task. There is almost no control over the outcome of the faults, so engineers must be careful when planning the experiments to avoid this kind of situation.

Lastly, bugs that consist of a combination of more than one fault can not be detected with the simple Chaos Engineering approach. Modifications of the approach or even a completely new approach is needed for this to happen.

3.1.2 FCatch

FCatch is a prototype developed by Liu et al. [15], in 2018, that aims at automatically detecting bugs, specifically, **Time-of-Fault (TOF)** bugs, in cloud systems.

DCatch, the previous work of Liu et al [16], aims at automatically catching concurrency bugs using identical principles. TOF bugs are a new type of concurrency bugs that are more harmful to cloud systems.

TOF bugs are critical faults that occur at a particular moment and system states. They are critical because they can cause a whole system to fail by breaking fault-tolerance schemes. TOF bugs mainly manifest when a node crashes or a when message is dropped at a specific moment.

Fig. 3.2 shows an example of a TOF bug in a Hadoop-MapReduce execution. The `Application Manager (AM @Node1)` receives a `CanCommit` message from @Node2 and updates the `T.commit` variable so no other node can start a commit while this one is happening. After that @Node1 is waiting for a `DoneCommit` from @Node2 but @Node2 crashes before sending it. This crash triggers a retry event on @Node3 that sends a `CanCommit` message. Despite the fault-tolerant method implemented by the developers, @Node3 is not able to commit since the `T.commit` variable is waiting on @Node2. If @Node2 crashed before `CanCommit`, the fault-tolerant process

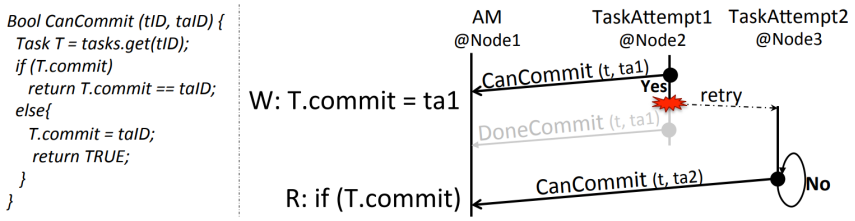


Figure 3.2: TOF bug example on a Hadoop-MapReduce execution [15].

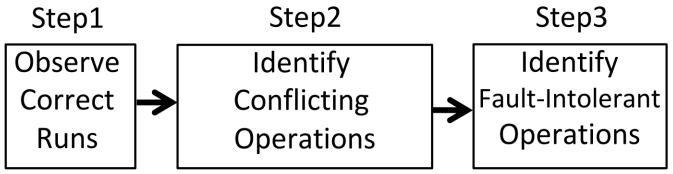


Figure 3.3: FCatch flow [15].

would have made the system correct. If @Node2 crashed after `DoneCommit`, the system would be correct because the execution was finished. Since the TOF bugs have an extremely small window to occur, faults need to be coordinated. A brute force approach, such as Chaos Engineering, is unlikely to catch TOF bugs because they are only produced when a set of faults are time precise and synchronized. Also, it would be infeasible because of the large number of potential sets of faults that can be injected into the system.

FCatch uses program analysis to *automatically predict* TOF bugs instead of a manually or randomly guided fault injection. The model has two key properties: **Triggering conditions**, specially timed faults, and **root causes**, such as a node crash leaving a shared resource in a state that cannot be changed.

In Fig. 3.3 we can see the three steps that FCatch follows. Step 1, *Observe Correct Runs*, monitors executions of the distributed system that are fault-free and record the resource-accesses operations, happens-before operations, and time-out operations. These correct runs are not random, and it is one of the main challenges of this work. Sometimes multiple runs of the same execution are needed to monitor and compare. Step 2, *Identify Conflicting Operations*, analyzes the traces collected in the previous runs to identify conflicting operations such as writes and reads the same resource from different nodes. Step 3, *Identify Fault-Intolerant Operations*, test these conflicting operations to see if there is actually a TOF bug and there are no fault-tolerant mechanisms to protect from those TOF bugs. At the end of the flow, FCatch creates a report with info about the bugs found to help the developers.

Conflicting operations are **Reads** and **Writes** that access resources that are in different nodes. In particular, Fig. 3.2 shows an example of conflicting operations (Write from @Node2 and Read from @Node3).

In step 1, FCatch is able to trace any correct run that has conflicting operations but there are special TOF bugs, such as crash-recovery, that need more than one correct run to be analyzed, which can difficult the analysis. Nevertheless, in general, their solution is to have two runs, fault-free and faulty run.

Another challenge comes when trying to compare these two different runs. Object hashes that are from the same object may differ from run to run, creating a non-deterministic execution and difficult analyses of the distributed systems. To this, they simply use checkpoints created before the fault injection given by a virtual machine or a container environment. They get a checkpoint right before they intend to inject the fault. From the checkpoint, the system resumes in two ways. One run without faults and the other injects a fault immediately after the checkpoint resumes. This mechanism completely eliminates the non-deterministic execution problem. In the end, both runs share the same initial state given by the checkpoint. Thus completely eliminating the non-deterministic execution problem.

This approach does not need to identify the system model. It analyzes only the inter-node, inter-thread, and intra-thread operations to find conflicting W-R pairs of operations. This is better than approaches that collect and analyze the system model because extracting it is difficult and sometimes infeasible. However, FCatch is still expensive in terms of performance. FCatch imposes a 5.6x - 15.2x slowdown, mainly because they use Javassist, a dynamic instrumentation tool, to inject tracing into the code causing the overhead. It is also slow due to their mechanism of creating checkpoints because these can be extremely expensive and access the storage and memory several times.

The authors claim that FCatch is more effective at finding TOF bugs than a random injection of faults since 400 random fault injections did not reveal any TOF bugs but only 1 round of FCatch can.

3.1.3 Bounded Black-box Crash Testing - B^3

The state-of-the-art crash-consistency tests in the kernel for Linux file-systems (ext4 [45], xfs [46], btrfs [47], and F2FS [48]) are dependent on around 400 tests and only 5% are for testing crash-consistency bugs. Testing before deployment of the application is fundamental but it is not enough to prevent crash-consistency bugs. Therefore, there is a need for fault injection mechanisms to test the resilience and the correctness of these file-systems.

The Bounded Black-Box (B^3) Crash Testing [17] approach has the goal of testing the correctness of file-systems in the presence of crashes. They evaluate the correctness of the file-system by injecting crash faults into it and analyze if the state after the crash is consistent.

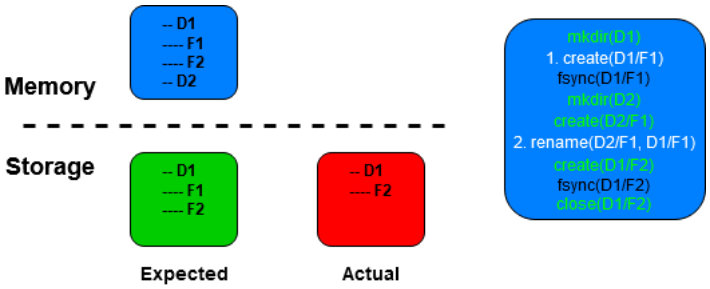


Figure 3.4: An existing crash-consistency bug that existed in the kernel since 2014.

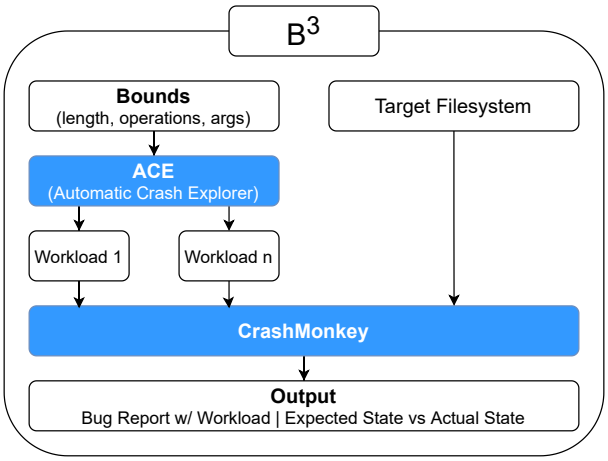


Figure 3.5: B^3 approach to test file-system crash-consistency bugs [49]

A file-system is **crash-consistent** if it always recovers to a correct state after a crash fault. For instance, a crash fault can be a system power loss or even natural phenomenons, such as lightning or earthquakes. In a more general idea, a system is crash-consistent if, after a crash fault, the system presents no failures. The file-system state is **correct** if its internal data structures are consistent, and files that were persisted before the crash are not lost or corrupted.

An example of a crash-consistency bug is described in Fig 3.4. A file F1 is created in the directory D1. A `fsync()` call is done to persist this file. F1 is created in D2 and rename/moved from D2-F1 to D1-F1. F2 is created in D1. A `fsync()` call is done to persist these files. A crash happens and we expected that after recovering, the files D1-F1 and D1-F2 should be persisted but only D1-F2 is persisted. It is claimed in the article that this crash-consistency bug of data loss exists in the kernel since 2014.

The two main challenges that the B^3 approach tries to resolve are **(1)** lack of an automated infrastructure to systematically test the file-systems and **(2)** the infinite number of workloads that need to be tested. A workload is a sequence of file-system operations such as `open()`, `close()`, `fsync()`, `fdatasync()`, `rename()`, `sync()`, and others.

To be able to accomplish this, B^3 uses two tools, one for each challenge.

CrashMonkey [50] (not to confuse with ChaosMonkey from the Chaos Engineering ap-

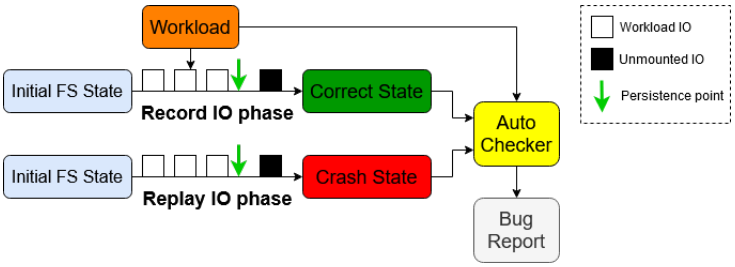


Figure 3.6: The 3 phases of CrashMonkey, Record IO, Replay IO, and Auto Checker [49].

proach) automatically simulates faults at different points of a given workload and tests the correctness of the file-system after each fault injection. Normally, these faults are crash faults, that simulate unexpected system failures such as power loss. To enable an efficient snapshot of the file-system, CrashMonkey uses a copy-on-write RAM block device.

CrashMonkey follows three phases. First, the **(1) Record IO** phase starts from an *initial file-system state* and collects a correct state of the file-system composed by operations and IO requests until a persistent point, all the IO after that are safely unmounted. This phase is similar to the correct run approach discussed in Section 3.1.2, since it collects a correct state from a fault-free execution of a workload.

Secondly, the **(2) Replay IO** phase executes the workload until a persistent point is met. A persistent point is when a sync operation, such as `fsync(), fdatasync(), and sync()`, is executed. Then, a fault is injected, normally a crash fault. After being given some time to recover, a crash state is collected.

Finally, the **(3) Auto Checker** phase is where it tests for the correctness of the crash state. It compares the crash state from the Replay IO phase with the correct state from the Record IO phase. If it is not correct, a crash-consistency bug has been found and a bug report is generated. It is possible to have false positives in this phase that can only be verified by a developer at the end of the execution of CrashMonkey.

This is a black-box approach since it does not need to analyze the model nor the code of the file-system being tested. This is extremely helpful since it can be used in existing and different file-systems with minimal modification such as changing arguments. Also, it is not costly since the process should be the same and no internal knowledge of the file system is needed to be able to test it with B^3 approach.

The solution to the challenge (2) not only finds a way to handle the infinite space of workloads but also how to create these workloads. **Automatic Crash Explorer** (**ACE**) exhaustively generates workloads that satisfy the bounds specified by the test developer. These bounds are used to narrow the space of workloads by arguments such as length of the workloads, number of

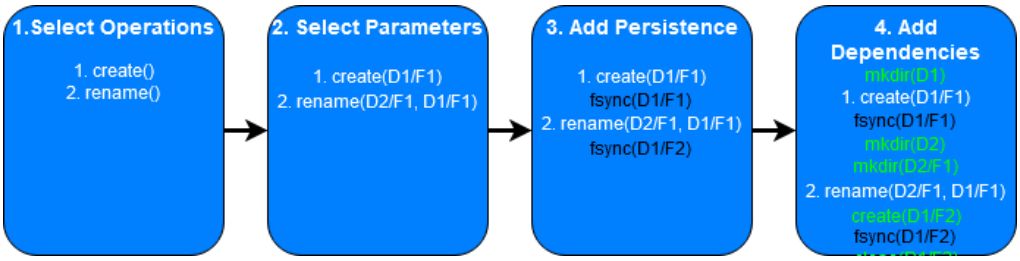


Figure 3.7: The 4 phases of ACE [49]

operations, and the arguments to the system calls.

The challenges of ACE are i) infinite length of workloads, ii) large set of file-system operations, iii) infinite parameter options (file/directory names, depth), iv) infinite options for initial file-system state, and v) when in the workload to simulate a crash?

The solutions presented to these challenges are based on a study of the currently documented crash-consistency bugs in file-systems. For challenges i) to iv), the solution is to bound their size. They bound the number of file-system operations in the workload to a maximum of 3 because the study showed that two to three core operations were sufficient to catch almost all the crash-consistency bugs. They also bound the parameters of these operations and set the initial state before the beginning of each workload generation. The solution to challenge v) was discussed earlier, and they simulate/inject a crash after a `sync()` call.

ACE has 4 phases: i) select operations, ii) select parameters iii) add persistence, and iv) add dependencies.

Fig. 3.7 is an example of the process of the ACE following each phase. First, it selects the operations that we want to create workloads for, and then it generates all the possible sequences of 2 of these operations (or 3 depending on the arguments given to ACE). If we pick `create()`, `rename()` and `write()`, it will generate 3^3 sets of 2 operations (`create()` and `rename()` set is exemplified in the Figure). Second, it selects the parameter for these functions, such as the name of the file to create. Third, it adds a persistence call after each operation, an `fsync()` for example. Finally, add dependencies for each function chosen, such as create a new directory before creating a file, and the workload is ready to be fed to the CrashMonkey framework.

It is essential to note that these challenges are not specific to this approach. Most of them are generic to the fault injection approaches studied before and they will need to be addressed in our work.

B3 is a black-box approach that allows testing various systems without knowing their contents. Also, B3 can be used in an already existing system. It has been tested in file-system and showed that the crash-consistency bugs can be caught with its tools.

But the benefits of the B^3 approach came with a few limitations. By analyzing the Crash-

Monkey phases, it can be seen that it only catches bugs that can appear after a `sync()` call. Crash-consistency bugs in the middle or not exactly after a `sync()` call cannot be catch with this approach. B^3 only tests file-systems, such as ext4, btrfs, and F2FS. They catch a specific class of bugs and not even all of them, thus not having completeness. As all black-box approaches, it cannot say where in the code the bug is. Although this is a limitation, the benefit of the black-box approach is far more useful. It is not scalable, since, with larger workloads, the cost of computing the workloads is exponential because, for the core operations, B^3 has to calculate all the possible skeletons that can be created with those operations. Finally, they rely on the deterministic nature of the system, so the correct run state must always be deterministic, which may not always be the case in distributed systems.

3.1.4 PACE

The core mechanism of reliability used in distributed file-systems is *replication* of data in different nodes. What happens if all the nodes that have this data crash? Distributed file-systems have built fault-tolerant mechanisms to endure single machine crashes. Are the mechanisms to recover from a correlated crash reliable? Multi-node or total system crashes happen more often than we think due to natural phenomenon's, such as lightning strikes [2] and even human error [3].

A **correlated crash** is when all the nodes that are correlated crash, i.e., all the nodes that possess the crash data. After a correlated crash, the system recovers using recovery protocols. But do they recover correctly after a correlated crash in a distributed system? When distributed file-systems recover from correlated crashes, it is expected that persisted data is not lost. Often, this is not the case and it represents a user-level guarantee violation.

These user-level violations are a vulnerability of the distributed system due to a correlated crash thus being called **correlated crash vulnerabilities**.

Alagappan et al. [18] created PACE, Protocol-Aware Crash Explorer, a generic correlated crash exploration framework. PACE explores correlated crash vulnerabilities in distributed file-systems by systematically generating persistent states that exist in the execution in the presence of correlated crashes.

The PACE framework was created with the goal of answering the following two critical questions related to correlated crashes: (1) Do distributed systems violate user-level expectations in the presence of correlated crashes? and (2) How to check for correlated crash vulnerabilities?

To understand how PACE works, first, we should understand how to catch persistent states that occur during a correlated crash. These states are called **globally reachable states** and we can find them by analyzing the operations made in the distributed system execution. Only

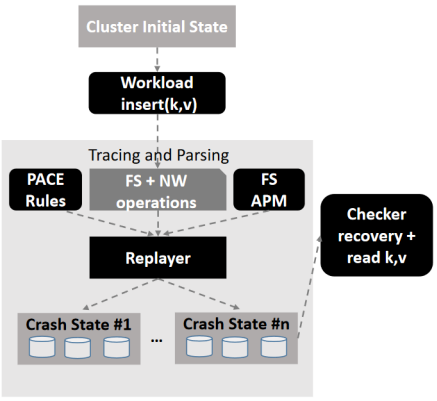


Figure 3.8: PACE execution methodology [51].

a few operations, such as `write()`, change the globally reachable state of the distributed file-system to the next one. PACE assumes that send and receive messages operations (`send() & recv()`), do not change the globally reachable state.

Distributed file-systems are complex and so there are operations done differently in the background. **Relaxations** such as reordering of writes and atomicity of operations are applied to the execution for the improvement of the system performance. For example, two sequential writes, w_1 and w_2, could be reordered by a relaxation thus when a correlated crash happens, the recovery protocol could get old data for w_1 but w_2 would be new and correct. Another example is the atomicity of the write, it could happen, that a write is not atomic and when the recovery protocol executes, only a part of the write is consistent.

When these relaxations are applied to a combination of nodes, it affects the recovery protocols and makes the generation of persistent states explode in size. PACE uses generic rules to prune this huge state space. For example, one of the rules generates states for relaxations applied to the leader of the distributed system.

PACE starts by executing a workload in the initial state of the distributed system to analyze, trace, and parse the operations. Operations can be of file-system type, such as `write()`, or of network type, such as `send() & recv()`. Then, PACE prunes the possible states by using PACE generic rules and replays those generated states. While the workloads are running, correlated crash faults are injected and the results are collected. Finally, a report is generated to show correlated crash vulnerabilities in the system. PACE goal is to examine and explore the global crash recovery protocols in the presence of correlated crashes.

This framework was tested on systems, such as Redis [52], MongoDB [53], and etcd [54]. Instance, for Redis, with 11x fewer states than the brute force approach, PACE found the same 3 vulnerabilities.

In conclusion, PACE injects correlated crash faults to test the recovery protocols of the

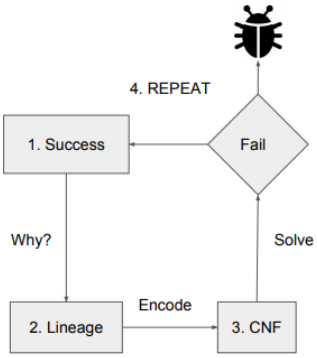


Figure 3.9: Overview of LDFI architecture [55].

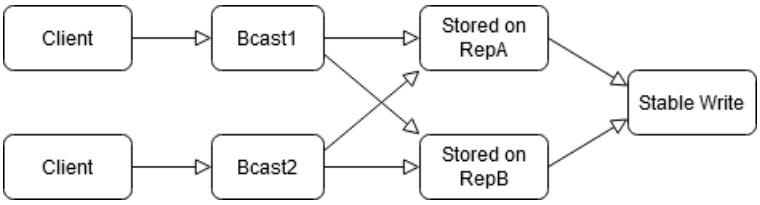


Figure 3.10: Lineage graph of an execution of a write in a distributed database system [55].

distributed systems and prunes the space with generic rules. Yet again, it uses a similar approach of a correct run at the beginning of the methodology. It proved to have a good performance but it is not a black-box approach. Also, PACE only focuses on very specific correlated crash vulnerabilities, which makes it not versatile. Nevertheless, it is a well developed state-of-the-art framework that overcomes challenges like the prune of the huge state space and automatizes the process of fault injection and testing the distributed system.

3.1.5 Lineage-driven Fault Injection - MOLLY

Peter Alvaro et al. [11, 55] developed an approach to fault injection called Lineage-driven Fault Injection. It is based on the concept of data lineage, used in database literature [56, 57, 58, 59, 60, 61]. This lineage can be seen as the *model* of a particular execution of a process.

LDFI follows a top-down strategy to discover bugs in distributed systems using fault injection. In Fig. 3.9 we can see the architecture of the LDFI approach and its specific steps such as (1) Success, (2) Lineage (3) Conjunctive Normal Form (CNF), and (4) Repeat.

For a particular configuration and input, the LDFI approach does a forwarding step by executing a **(1) success** run and obtaining the correct result. The LDFI then works backwards from the correct results of this forwarding step, asking *why* this execution was successful. To answer this, a **(2) lineage** is extracted from the execution, i.e., a lineage graph is created.

For example, Fig. 3.10 shows a lineage graph from a hypothetical execution of a write in a distributed database system where *BcastX* means that a broadcast event named X occurred and

Stored on RepX means that some data was stored in a replica X.

It is extremely difficult to obtain a lineage graph in a production distributed system where there are many messages with various nodes and with the high complexity of each execution run.

This lineage graph is *encoded* into a **(3) Conjunctive Normal Form (CNF)** formula.

To *solve* the CNF formula, first we need to specify the events that can fail in this execution. For Fig. 3.10, the events are:

$$E \equiv \{RepA, RepB, Bcast1, Bcast2\}$$

Being E the processes in the lineage graph, there are 2^E combinations of faults that can be injected in a lineage graph and specifically, we can inject $2^4 = 16$ faults in this example. This creates an *exponential space* of faults to explore. However, by carefully analyzing the lineage graph, it is possible to observe a couple of **redundancies**. It can be seen that injecting a fault in *BcastA* and *RepA* would still make the system and the execution correct by using the alternative and redundant path, BcastB and RepB. With this type of reasoning through the lineage graph, they can prune the exponential space of possible faulty executions by avoiding redundant paths.

After solving the CNF formula, a set of faults that correspond to the solution is created. The system can output two outcomes for each fault in the solution set:

1. The system fails with an incorrect outcome and a bug report is created. The lineage graph is presented to help correct the bug and the execution terminates;

2. The system succeeds with a correct outcome and the process is repeated.

MOLLY is a prototype that implements the LDFI approach. MOLLY guarantees a **completeness** property, i.e., if the analysis completes without finding bugs, they guarantee that no bugs exist for that configuration, even if the space of possible executions is enormous. This completeness property is extremely useful for engineers and developers since it gives confidence in the resilience of the system being tested.

However, this property comes with a downside which is that it only stands because of a major simplification made in the model. They assume that ***delivered messages are received in a deterministic order*** thus, they convert an asynchronous system into a synchronous one. If we could assume this deterministic order in distributed systems, lots of complex problems, such as Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system" [62] and others, would become much easier to solve. This simplification is not feasible in the majority of

distributed systems where messages are commonly being delivered out of order by each node on the system, because of network issues, thus not ensuring the completeness property.

Peter Alvaro et al. [55] took a step further and tested the LDFI approach (not the prototype MOLLY) in a production-level distributed system, Netflix. To be able to accomplish this task, they utilized the Netflix infrastructure "building blocks" that is composed of heterogeneous systems that provide different services, such as isolation of failures, communication with remote services, IPC, databases, caches, etc. As expected, each distributed system brings a huge complexity with it and so, they had to make lots of adjustments specific to the Netflix system.

One of the adjustments was on the obtaining of the lineage graph. As said before, obtaining the lineage graph is an extremely complex task and not feasible in the Netflix system. So they used one of the tracing services to get a call graph. A **call graph** is similar to a lineage graph in the sense that it shows what processes were executed during an execution. However, it does not represent the alternative paths that the execution could have taken. Because of that, the redundancy that a lineage graph has is not obtainable and they cannot apply their strategy to prune the exponential space of faults to inject.

To address this, they grouped the execution runs into classes of runs, i.e., different runs but with similar outputs are grouped as one class. Then they ran the same experiments and mapped them into the respective classes thus obtaining redundancy on the same class of executions for each call graph. This reduction resolves one of the most difficult problems in fault injection in distributed systems today, **exponential space of possible executions**. Chaos Engineering, explained in Section 3.1.1, does not resolve this issue and Chaos Engineering is one of the main fault injection approaches used by the dominant IT companies such as Amazon, Netflix, and Microsoft.

Replayability was another problem solved by this solution of mapping executions to a finite number of classes. These classes of requests were considered equal if they caused the same back-end interactions as if they were replays of a single canonical request.

To conclude, the LDFI completeness property, which gives trust to the developers, comes with a simplification that most of the distributed system can not have, transforming an asynchronous into a synchronous system. We also acknowledge that lineage graphs solve the problem of exponential space to fault injections but it is very difficult to have a graph like this in a complex distributed system, although with some adjustments it is possible. In other words, it is very difficult to obtain a system model that is needed to apply the LDFI approach.

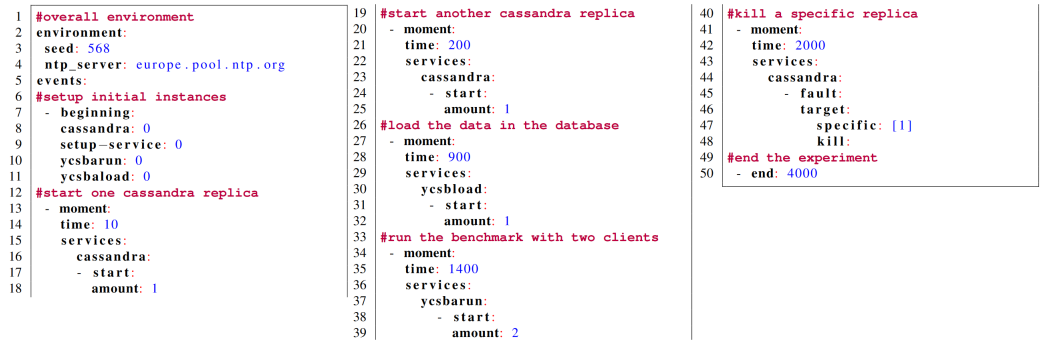


Figure 3.11: FDSL example for a Cassandra experiment, showing the environment attributes and several events specifications [37].

3.1.6 Faultsee

It is critical to have a well-defined, precise, and concise description of fault injection experiments to be able to evaluate different experiments with the same objectivity. As an example, different research works using the same system can come to different conclusions depending on the way faults were injected. A solution to this standard problem is Faultsee.

Faultsee is a toolkit composed by (1) **Faultsee Domain System Language** (FDSL), a configuration language based on YAML to concisely describe the experiment, and (2) a **Faultsee platform** to deploy and automatically execute the experiments specified in FDSL.

To describe the FDSL language, let's use Fig. 3.11 as a concrete example of an experiment written in FDSL. There are two main sections. The *environment* section and the *events* section.

The **environment** section specifies the system's initial state, such as a seed to be able to reproduce the experiment, the NTP clock synchronization of the nodes, and other specifications such as the number of replicas.

The **events** section support *beginning*, *moment* and *end* events that can begin, add, fail and stop nodes. In lines 7-11, the beginning event describes the artifacts that the experiment will use and the 0 indicates that no artifacts are initialized at the beginning. This event is considered to be part of the environment phase since it also specifies the initial state. The artifacts, such as Cassandra in line 8, are described using Docker images and Dockerfiles. At lines 40-48 we can observe an event that provokes a fault in the system. The fault is injected at time 2000 in a Cassandra node, specifically in node 1 and it is a kill fault.

This language is decisive to obtain a well-defined system that can be reproduced with ease and also allowing to be precise on *what* fault we want to inject and *when*.

Faultsee platform runs the experiments detailed by the FDSL and has the architecture depicted in Fig.3.12.

The **Master Controller** is responsible for converting the FDSL listing document into a

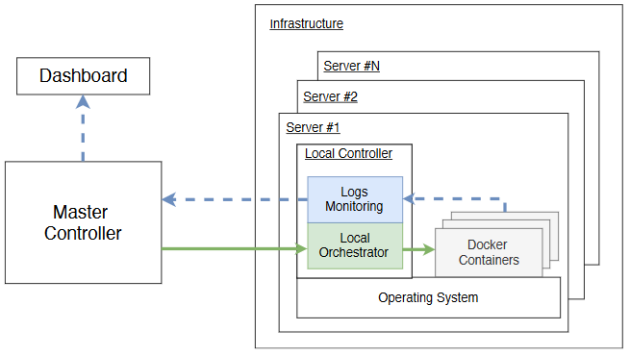


Figure 3.12: Fautlsee platform architecture, with Master Controller, Local Controller and a Dashboard [37].

concrete execution plan that is going to be received by the Local Controller. This has the advantage that no extra coordination or communication is needed during the experiment between the Master Controller and the Local Orchestrators and thus having better scalability. These Local Orchestrators are the processes that run the experiment and are managed by the Local Controller.

The **Local Controller** after receiving the concrete execution from Master Controller, starts to execute the moments described by the FDSL such as starting instances and injecting faults. The System metrics are gathered by the Local Controller and then sent to the Dashboard.

The **Dashboard** is a web application using a visual interface that shows the metrics and logs in a more presentable way and additionally can control the experiments.

We can observe how easy and simple it is to specify an experiment with the FDSL, with just a few dozen lines, and observe with transparency what the experiment does and importantly, reproduce it. The results gathered from the experiment can be checked in the dashboard after it finishes.

Certain faults, such as injecting faults in the network, are not available in Faultsee but the system is extendable and designed to support custom faults. The main limitation is that it only supports injecting faults base on time, and, as we have been discussing, there are bugs that only manifest after complex and precise events (state) occur.

The goal of Faultsee is not to explain the behaviour of the system given the faults that are injected, but rather to give us a powerful, yet easy and simple to use, toolkit to specify and run experiments in a standard way. The results from the experiments can be visualized in a dashboard to hypothesize about them and make conclusions.

3.1.7 Elle

Elle [19] is a tool that detects anomalies in distributed database applications using transaction histories and directed graphs to find them.

This checker tool was created by the author of Jepsen and of LDFI, and so, Elle combines the Jepsen testing framework with transaction histories (similar to the lineage concept/idea from LDFI).

A **transaction** is a set of operations that should be performed as one and can be committed (C), when all operations succeed, otherwise the transaction is aborted (A) if any operation fails. Transactions have a unique identifier to avoid ambiguity.

The transaction histories are based on **Adya history** [63, 64] and are composed of a set of transactions on objects in a database. These transactions have an **event order** that indicates the order of the operations in the transactions and show a **version order** $\ll$ for each object. The version order $\ll$ for an object x can be represented by a list that appends each write on x. Since only appends are made to this list, the version order is *traceable*. For example, if x has the version history of [1, 2], we can trace the version order as [ ], [1] and [1, 2]. Also, the tests append unique numbers to the list of object x thus, the exact write can be deduced and its *recoverable*. The extraction of these Adya histories is done by observation of the outcomes of the experiments done to the application.

Elle plots a Direct Serialization Graph (DSG) to visualize the dependencies between transactions. A **read dependency** is when a read comes after a write and so it shows an event order of writes that happens before the read (w $<$ r). A **write dependency** is the opposite of a read dependency, when a write comes after a read (r $<$ w).

To be able to find anomalies in the database systems, Elle identifies cycles in the DSG. These cycles in the graph are found by applying Tarjan's algorithm to get the strongly connected components (SCC). Then a breadth-first search (BFS) is applied to each SCC to identify short cycles.

Elle is capable of identifying several types of anomalies through the analysis of cycles and dependencies on the DSG's it. For example, it can identify data loss and future writes and reads, i.e., operations that return data that should not be available given the isolation properties of each transaction history.

These are the type of anomalies that Elle automatically finds in the tests after injecting faults like partition failure and message dropping. Besides cycles, there are other anomalies that Elle also catches such as *duplicated writes*, *garbage reads* and *internal inconsistencies*.

Using the DSG, Elle can automatically check if there are contradictions in the specifications

of the application, in the presence of fault injection and generate visualizations and human-verifiable explanations. Also, they show that the performance is better than a previous similar approach. Since thousands of experiments are made and the injection faults are not precise, it is extremely difficult to reproduce a test that produces a failure.

As a black-box approach, it needs to create the Adya histories of the database through observations of the application behaviour. These Adya histories can be seen as the state of the distributed application.

3.2 Tracing Technologies

From the previous section, where we discussed state-of-the-art fault injection frameworks and techniques, we can conclude that *Reproducible Fault Injection* in distributed systems is a complex and challenging task. Since the principal premise of this dissertation is to obtain reproducibility in fault injection experiments by collecting a state, we must research the area of tracing, what tracing tools exist and their performance.

3.2.1 CAT

Tracing tools are necessary to test the correctness and dependability of distributed systems.

CAT [38] is a non-intrusive Content-Aware Tracing and analysis framework. CAT improves the state-of-the-art in the concept of tracing the content and the context, instead of the usual only context-driven tracing. They collect system I/O requests, following a non-intrusive (black-box) approach. These I/O messages can come from the network, storage and more.

However, we are focusing on their study on the tracing tools to help us decide the best tracing tool in terms of performance but also in terms of easiness to utilize. CAT, on top of their main goal, tracing the content, considers what we believe are the fundamental aspects of tracing, in a black-box manner. These fundamental aspects are the different Linux kernel tracing tools, their advantages and disadvantages, their overall performance, what they can and cannot do, and accomplishing this by only tracing into the system calls and I/O messages (black-box approach).

In Fig. 3.13, we can see a context-based tracing approach (a) and the benefit of having a content-aware tracing approach (b). On (a), node 1 sent 12 bytes and received 12 bytes, so it is assumed that the application was acting as expected. However, as we can see on (b), the 12 bytes message received on node 1 are incorrect, and the application was, in fact, acting incorrectly on node 2.

With this type of analysis, there are bugs that CAT captures that, otherwise, we were unable to. This comes with this great benefit but also with more overhead because we not only have no

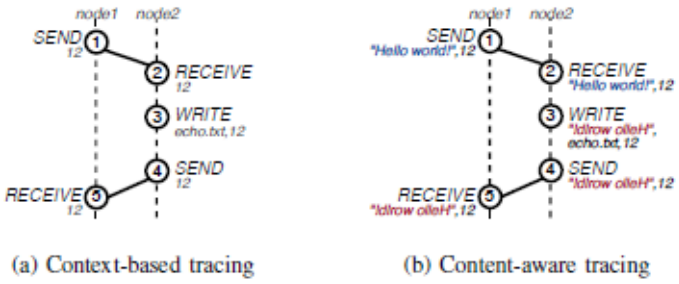


Figure 3.13: Context-based (a) vs Content-aware (b) tracing example. [38]

analyze the context, which is normally a few arguments (file descriptor arguments, IP, port, and others), but now we have to also analyze the content that can be small or huge, depending on the message size.

Other state-of-the-art tracing approaches either take an intrusive approach, where it requires the source code or binary instrumentation [65, 66, 67], using tools such as Intel PIN [31], or they only consider the requests context [68, 69, 70]. The problem with these approaches is that these systems are extremely complex, and understanding the source code and where to tackle the problem is a notoriously complex task in distributed systems. CAT defends that a black-box approach enables CAT to be application-agnostic, thus being more flexible to different developers and engineers.

On top of being a black-box approach and also analyzing the content instead of only the context, CAT also innovates by creating an algorithm that evaluates if the content is correct and, if not, how much is incorrect. This is a crucial aspect to improve tracing tools in the research community. However, it is not fundamental to explore in detail in this particular thesis document because we do not consider it directly relevant to the theme, thus being quickly mentioned. Instead, we focus on the black-box approach (to be application-agnostic), the Linux kernel tracing tools, and how they used them to enable this black-box property, and their respective results.

How can CAT capture/trace the system's events in a non-intrusive way? There is a panoply of Linux tracing tools such as Ftrace, LTTng, eBPF, SystemTap, Strace, and more to trace the system in a non-intrusive way. Due to the capability of tracing system calls, Linux kernel and userspace function calls, CAT focused on eBPF (CatBpf) and Strace (CatStrace). They further tested these two tools with two Big Data systems, Tensorflow [71] and Apache Hadoop [72].

The eBPF Linux kernel tool enables the dynamic insertion of code to the Linux Kernel without having to change the kernel or load the modules. eBPF is further explained in Section 2.2.

Strace is a Unix command-line tool that captures the system calls issued by a process. Strace stops the target process when entering or exiting a system call, allowing Strace to capture the program information.

By analyzing the results, they concluded that CatBpf imposed significant lower performance and storage overhead but lost a considerable amount of events, being better than CatStrace on a production setup. However, CatStrace had a better performance when collecting the events because it got them all. For our ReFI particular case, since we do not gain much from analyzing the content, eBPF was a trivial choice. eBPF also enables the modification of logic in the kernel to implement fault injection and not only tracing. The Strace tool would be an excellent fit for more critical distributed system applications since they are more confident in capturing all the events.

In conclusion, CAT is highly efficient (negligible overhead) at tracing the context and the content of I/O messages in a black-box approach. The Linux kernel tracing tools used were eBPF and Strace, where both showed a negligible overhead. eBPF surpassed Strace because it showed better performance (time and storage), and it had a better functionality, even tho Strace had no events lost, and eBPF had. eBPF can inject code into the kernel, userspace and system calls, where Strace can only show the context and content of each call and not modify the logic. This makes eBPF good to inject faults into the system too, and not only trace it. CAT also analyzes the content due to a complex algorithm developed by the authors to help understand the problems that may surge.

3.3 Bug Analysis

State-of-the-art fault injection approaches usually follow two strategies when it comes to bug analysis. They either try to replicate bugs that are already documented, such as B^3 [17] or they try to find specific types of bugs, such as FCatch with TOF bugs [15].

The primary goal of this dissertation is to create a **reproducible** fault injection framework and not to find new bugs. Thus, we followed the first approach, replicating already documented bugs.

In this section, we will explore two research studies on the analysis of distributed system bugs. We will start with Jepsen [25], a project and a framework that finds and documents new bugs for production distributed systems, such as MongoDB [53], Redis [52], PostgreSQL [73], and many more. Then we will look at Alfatafta et al. [39] study, which shows us that partial network partition bugs in distributed systems are more frequent than we may think. Astonishingly, they are easy to replicate by only needing 3 to 5 nodes and a few operations.

3.3.1 Jepsen

Jepsen [25] is a framework that utilizes fault injection mechanisms to verify the properties and specifications claimed by distributed applications such as MongoDB, Redis, PostgreSQL, Cassandra, and many others.

Jepsen first replicates the environment of the distributed application by creating a set of distributed nodes. To set up these nodes, Jepsen offers several options to set up these nodes, such as Docker [74] and AWS [75]. Then, after the setup is ready and the system starts, a control node is used to coordinate the tests. The tests are generated and each client node executes the operations in the correct order coordinated by a clock synchronization from the control node. All operations are saved to generate a history that is analyzed at the end of each testing process. While these tests are running, a special process injects faults into the system that were scheduled at the generation of the tests. These faults can be, for example, partitioning of the network that connects the nodes, crashing nodes, etc.

When the tests are completed, Jepsen generates reports and graphs of availability and performance to help the developers analyze the history for correctness. This verification process is not automated since developers need to check and understand the results, making this process slow.

A complete analysis of the distributed system, like MongoDB distributed database, can take months to be completed due to its difficulty and the necessity of having a specialized understanding of the system. Engineers need to understand the system model, find ways to break it, using the Jepsen framework, and then analyze if it caused a bug. This is an extense and long process for engineers and developers.

Nevertheless, they publicly show the results gathered by testing different distributed systems such as MongoDB, PostgreSQL, etcd, and Redis, which helps production distributed systems correct major bugs. On top of this, it enables fault injection frameworks, like ReFI, to focus on testing specific distributed systems bugs instead of having to find complex and difficult bugs throughout the exponential space of possible bugs.

3.3.2 Partial Network Partitioning

Alfatafta et al. [39], conducted a comprehensive study on *partial network partitions*. They found 51 failures in 12 popular distributed systems, where 39 were catastrophic failures (e.g., data loss and system unavailability) that easily manifest. All the mentioned failures are caused only by partially partitioning a single node.

A partial network partition is a specific type of network partition where a group of nodes can

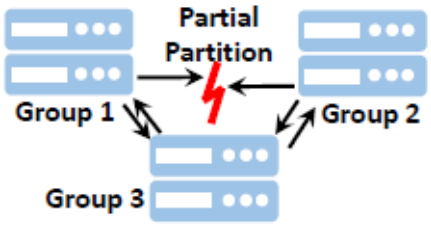


Figure 3.14: Example of a partial network partition where groups 1 and 2 are disconnected but group 3 can communicate with group 1 and 2. [39]

not communicate between another group of nodes but can communicate with other groups. In Fig. 3.14, we can see a simple example of a partial network partition on a three-group system.

This study has three key goals. The first goal is to find and study failures mentioned in the issue websites of the twelve popular systems (Elasticsearch [76], MongoDB [53], RabbitMQ [77], MapReduce [72], HBase [78], Mesos [79], HDFS [72], Ceph [80], MooseFS [81], Kafka [82], ActiveMQ [83], and DKron [84]) to improve systems' resiliency to this specific type of fault. The second goal is to dissect fault tolerance techniques in these popular systems and identify their disadvantages. The third goal is to design a generic tolerance technique to improve the system for partial network partitioning.

In this section, we will only focus on the first goal because it is the one that is essential for this dissertation. We only need to understand the impact of these faults in distributed systems, how can we replicate them, and their characteristics. The other two goals focus on improving the systems via a generic tolerance technique, called Nifty, which is out of the scope of this work since we are focusing on fault injection with ReFI.

The impact of more than 74% (39 in 51) of the failures is **catastrophic**. Once more, this shows how vital these types of studies are and the importance of creating fault injection frameworks to try to catch these critical failures.

On top of being catastrophic, these failures are **easy to manifest** because the majority of them require less than four events (e.g., reads and writes) to occur and less than five nodes. Furthermore, all the failures studied are triggered by partially partitioning a single node.

Although the authors mention these 51 failures, they do not present nor document all of them in their article. However, they provide the link for the bug report and we can access them (e.g., 3 MongoDB and 7 ElasticSearch [76] failures). Given this, we could replicate some of them in ReFI, showed in the results section.

Alfatafta study enables us to focus on the more essential aspects of this work by efficiently documenting the catastrophic bugs they found, how they can be replicated, and their impact on the systems. These documented bugs are a critical part of testing the tool's reliability. If the

tool can quickly reproduce these bugs, we can confidently affirm that the tool is working for that specific scenario.

3.4 Discussion

Table 3.1 places our framework, ReFI, in comparison with state-of-the-art fault injection tools, frameworks, prototypes, and their ideas. These approaches use the mechanism of injecting faults in distributed systems to test their robustness and discover bugs that cause catastrophic failures, such as irreversible data loss or corruption and system unavailability.

To inject faults into various systems in a scalable way (automatic and fast), a tool should have a minimal learning curve and reduce modifications for each different system. A black-box approach does not depend on the system. Therefore, it is amenable to achieve these objectives as the programmer only needs minimal knowledge to understand the system and its model and does not need to modify depending on what system is testing.

A framework is application agnostic when it does not depend on a specific application type. FCatch, B^3, and PACE are not application agnostic because they are specially made for file-system applications. These frameworks are also made to catch specific bugs such as TOF bugs, crash-consistency bugs, and correlated crash-consistency bugs, respectively. ReFI follows the idea of Faultsee where it uses a configuration file to specify the experiment that transverse to different distributed systems, thus accomplishing the goal of being application agnostic.

The reproducibility of an experiment is fundamental when we want to have a standard way of evaluating different systems and comparing these evaluations. One of the main goals of our system ReFI is to accomplish this task utilizing a black-box approach that aims to correlate the internal state of a system with an observable external state allowing for systematic reproducibility of an experiment. B^3, although it is a black-box approach and can reproduce the tests through different file-systems, it misses the scalability factor and ability to test systems other than file-systems. Reproducibility is the major downside of Chaos Engineering since developers need to understand the system model in depth to identify the bug. Reproducibility is the major improvement that ReFI accomplishes from the state-of-the-art frameworks such as Chaos Engineering, FCatch and PACE.

Elle uses tests based on specific workloads. However, they are not reproducible since we can input these workloads and obtain different behaviours due to the non-deterministic nature of distributed systems. However, these workloads are public and are reusable to try to reproduce these bugs. By doing this in a transparent way, the community of distributed systems can only improve the reliability of their systems.

To have reproducible experiments, we decided to trace and collect a distributed system state. For this, we analyzed CAT's work where they used two tools, eBPF and Strace, to trace and collect the content and context of system I/O requests. From their results, eBPF was better suitable to our needs because it yielded better overall performance (time overhead, memory and storage) than Strace. Also, the advantage of Strace over eBPF was that it lost fewer events and performed better when collecting the content. Since tracing the context is enough to obtain a usable system state, the system calls content, and the loss of events was not critical to our solution.

eBPF had a more general technology where we could trace the system calls and change the logic of those methods without changing the Linux kernel. This is particularly useful for ReFI because we intend to inject faults and collect a usable state.

After understanding the state-of-the-art Linux kernel tracing tools, it was essential to analyze known system bugs. For this, we studied Jepsen, a framework with several documented bugs for different distributed systems, where we concluded that they had two great experiments. One for MongoDB and another for Redis that are described in the evaluation Section 5.3. We also studied partial network partitions issues from the research of Alfatafta et al. [39]. We understood how to replicate specific bugs from these two works and, with ReFI, we could make them 100% reproducible.

In the next chapter, we present the implementation of ReFI. ReFI is a Reproducible Fault Injection framework that improves state-of-the-art fault injection approaches shown in Table 3.1. We also discuss the key ideas to address the above challenges based on the related work and further research. Finally, how we attacked these difficult and complex challenges and why we choose some solutions over others.

	Fault-free Run	Black-Box Approach	Application Agnostic	Different Bugs	Reproducible
Chaos Engineering	✗	✓	✓	✓	✗
FCatch	✓	✗	✗	✗	✗
B^3	✓	✓	✗	✗	✓
PACE	✓	✗	✗	✗	✗
LDFI	✓	✗	✓	✓	✓
Faultsee	✗	✓	✓	✗	✓
Elle	✗	✓	✓	✓	✗
ReFI	✓	✓	✓	✓	✓

Table 3.1: Comparison between state-of-the-art fault injection approaches and ReFI.

Chapter 4

ReFI

In this chapter, we will briefly recap the objectives of this dissertation and how they correlate to the implementation of ReFI. We will describe ReFI's architecture, its components and how they correlated with each other and ReFI's goals in a high-level top-down fashion. We will discuss how we divided the problem into modules and critical decisions on each of these modules. Furthermore, we will tackle how we obtained a usable state of the system to base our fault injection. Finally, we will explain how we create experiments for ReFI using a configuration language.

As indicated earlier, the objective of this dissertation is to surpass the state-of-the-art in the following four essential aspects.

1. **Reproducibility** of fault injection
2. Treat the application as a **black-box**
3. **Efficiently** perform tracing and fault injection
4. **Extendability** to various system calls and **Flexibility** for different distributed systems

*How does ReFI enable **reproducible** fault injection?*

ReFI reproducibility is built on the concept that if we base a fault on the system state, instead of based on time or random injection (as state-of-the-art fault injection approaches discussed in Section 3 do), ReFI can reproduce these faults. Thus, to have reproducible faults, we need to collect a system state.

Collecting a system state is a challenging task in distributed systems due to their non-deterministic nature and complex architecture often composed of several different services. In some cases, it is even impossible for the state to always be coherent between all nodes. So, we had to make a few adjustments discussed in subsequent sections of this implementation chapter.

*How does ReFI treat the application as a **black-box**?*

ReFI makes use of the Linux kernel tracing tools. In particular, ReFI uses eBPF to trace, collect, analyze, and even change the logic of the system calls that the application does to the kernel. ReFI accomplishes this without having to either instrumentate the code of the distributed system or understand the algorithms of the system.

*How does ReFI **efficiently** trace and inject faults into the application?*

ReFI utilizes the eBPF technology that adds minimal overhead to the application. eBPF can trace almost all system calls, such as storage calls, network calls, and much more. However, why is eBPF so efficient? One of the reasons is that eBPF programs have a zero-overhead when not in use [30] and minimal (nanoseconds) overhead for each program's execution because they run on top of the already built kernel infrastructure.

*How does ReFI **extend** to various system calls and I/O requests?*

ReFI was built in a way that we can add an eBPF program for each new system call and I/O request. Thus, enabling the addition of more submodules to trace other requests of the distributed system. We explain in detail the specific details of the extendability.

*How does ReFI can be **flexible** to support different distributed systems?*

ReFI is guided by a configuration file. The configuration file specifies the distributed system nodes and the faults to be injected, depending on the state. Since ReFI treats the application as a black-box, we can apply the same structure and reasoning to other applications. The meaning of the state may vary, and engineers need to understand how they can take a viable state from the tested distributed system.

The eBPF technology helps us accomplish the goals that we have. The complex part is the interconnection of all these goals into an efficient and coherent tool. For that, we have a particular module, called Orchestrator that coordinates all the modules of the tool.

It is essential to highlight that some of these objectives are interconnected. By having a tool that treats distributed applications as a black-box, we can accomplish the objective of being flexible to different distributed systems with more ease since we do not need to comprehend details of these distributed systems with the granularity of the engineers that developed it. Also, we can extend to various system calls because we treat it as a black-box.

The following sections will describe the architecture of ReFI and how we divided and conquered this complex problem into three modules. A Tracing Module (TM) that handles the tracing of the state of the application. A Fault Injection Module (FIM) that handles the injection of faults using the state collected by the TM. An Orchestrator that facilitates all the communication between the TM and the FIM and manages the observed system state, and

the initiation of the tool, including the TM and FIM. Finally, we will explain how to create experiments with ReFI using the configuration file.

4.1 Architecture

ReFI intends to accomplish its objectives by dividing the architecture into simpler modules that compose ReFI. As depicted in Fig. 4.1, we can delimitate three core components. The Tracing Module (TM), the Fault Injection Module (FIM) and the Orchestrator. Each of these modules has a particular goal.

The architecture of ReFI can also be divided by user space and Linux kernel space. This division facilitates the understanding of the user and the implementation.

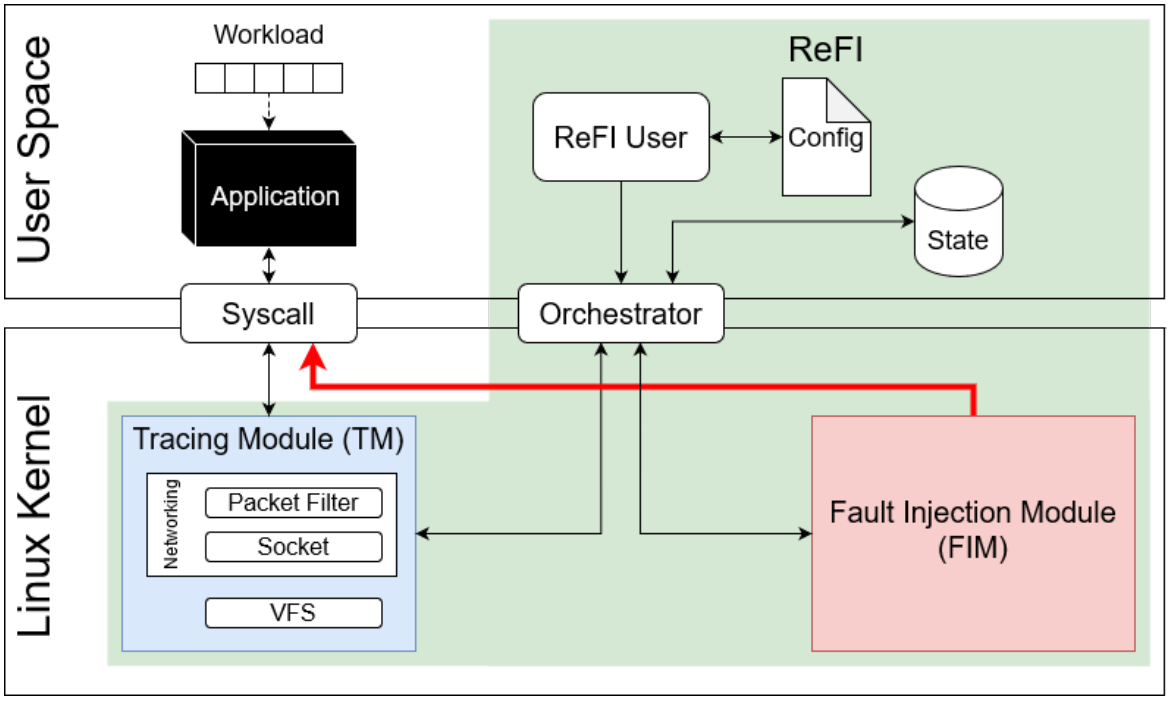


Figure 4.1: ReFI architecture (in green) incorporated with a black-box application. The application is issuing syscalls that are handled by the Linux kernel. The TM (in blue), intercept these syscalls to trace the state. The Orchestrator synchronizes the state between the TM and the FIM (in red) to enable reproducible fault injection experiments.

The **Tracing Module** (TM) has the goal of deriving information of the application by tracing and profiling its behaviour. TM is composed of several eBPF programs that intercept and inject code into specific syscalls. TM collects information about networking, Virtual File-System, and general information deduced by system calls and I/O. In particular, the TM collects TCP and UDP socket details (ipv4 or ipv6 protocols), uses XDP (eXpress Data Path), a low-level eBPF framework, to monitor and filter packets, and inspects the VFS (Virtual File System) most used syscalls (e.g., write and read). This critical information is passed through the Orchestrator which

uses it to create a usable state. In the Tracing Module subsection, we will discuss in more depth the collected state.

The **Fault Injection Module** (FIM) has the goal of injecting faults into the application. The faults are predefined before the experiment begins via the configuration file. At the beginning of the experiment, we already know when and what type of faults will be injected into the application in a specific state.

The **Orchestrator** collects the information from the TM and creates a state that is used to precisely notify the FIM that a fault is to be injected. It also parses the configuration file that possesses the information about the distributed application and the faults that will be injected. The Orchestrator manages the state by modifying it and, when needed, informing the other two modules, TM and FIM, of these changes when necessary. The user of the ReFI tool receives information about the experiment via the Orchestrator.

The configuration file has a crucial role in the design and architecture of ReFI. It enables ReFI to be flexible in a different distributed system by specifying the nodes that compose the distributed system and specifying different fault patterns to test the robustness and resilience of those systems.

Throughout the development and implementation of ReFI, we found challenging problems, such as coordinating different information of the TM to create a state. eBPF intrinsically came with several challenges. eBPF is a relatively recent technology that is constantly changing, creating unstable documentation. This caused compatibility problems and developing changes due to recent advancements throughout the implementation of ReFI.

Another challenge is with having several eBPF programs on the TM and on the FIM. Coordination and communication between these programs needed to be efficient. This was one of the main reasons we created the Orchestrator, to facilitate this coordination between modules, state, and configuration. Even when we have an excellent performance in the eBPF programs, the response time of each program might add a significant impact on the overhead of the framework.

By making ReFI a modular framework and dividing it into modules and submodules, we gained indispensable advantages. In the TM, having several eBPF programs, allows each program to have a specific function in the overall system. We can extend our TM submodules to have more tracing syscalls making ReFI an extendable framework. In the FIM, we can add different fault patterns, e.g., adding rules for packet filtering.

The following sections will explain how we overcame these challenges, how these decisions improved ReFI, and a more detailed explanation on the implementation of the Core Modules (TM, FIM, and Orchestrator) and the tools around it (configuration file).

4.2 Tracing Module

This section will describe the objective of the Tracing Module (TM) and how it correlates with the objectives of ReFI. We will explain what information is crucial to trace and profile, why it is crucial and how we obtain it. We discuss how the TM is extendable to different syscalls and flexible to other distributed systems. We will understand the division in submodules, where each traces a specific syscall and collect information from it in a black-box manner using eBPF. Finally, we will see how the TM coordinates and communicates with the Orchestrator via events and eBPF maps. We will continuously explain how TM helps accomplish the four ReFI objectives (reproducibility, black-box, efficiency, flexible and extendable).

The main objective of TM is to derive information about the distributed application by tracing and profiling its behaviour in a black-box manner.

What information do we need to create a usable state? (1) To answer this question, we first need to answer another fundamental question.

What type of faults do we want to inject? (2) In fact, our objective is for ReFI to inject all different fault types and patterns. Unfortunately, this is an impracticable objective due to the exponential space of faults to inject. Nevertheless, we implemented the TM in an extendable way to enable all types of faults in the future and flexible to be used in different systems. Throughout the development of ReFI, we studied at least 25 different issues in different distributed systems. In Section 3.3, we describe the leading research where we discussed Jepsen [25], and Alfatafta et al. [39] article on generic fault tolerance. We decided to focus on network and storage bugs because they were the most frequent issues in these reviews. Later, we created specific experiments in these areas, described in more detail in Section 5.3. To answer question (2), we wanted to inject all types of faults, such as faults that perturb the storage system and the file system. However, since this is a challenging and impracticable objective, we focused on a more narrow space of networking and storage faults but always enabling the extension to other types of faults. An example of a network fault is a network partition, where we, partially or completely, isolate a group of nodes from another group by disabling their communication via the network.

Now, we can answer question *(1)*. Since we want to inject networking and storage faults, we need to trace network and storage information outputted from the distributed system. To accomplish this in an extendable way, we subdivided the TM into submodules.

Fig. 4.2 shows the general implementation of a submodule of TM. Each submodule is an eBPF program for each important syscalls. We can see three submodules. Packet filter, socket and VFS. We can also notice that the packet filter and the socket submodule are in a group of networking. The networking group means that the submodules collect information about the

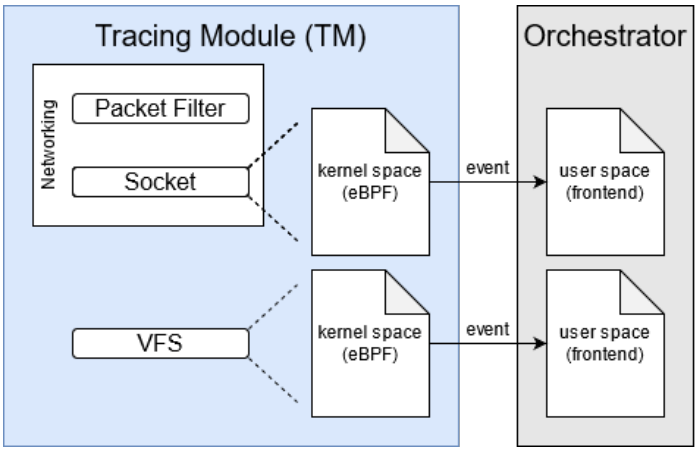


Figure 4.2: ReFI TM is composed by several tracing eBPF programs that have a corresponding event handler in the Orchestrator. The events are filtered in the eBPF program, leaving no overhead for the Orchestrator, when the event is not useful.

network.

The concept of syscall, throughout this document, is defined broadly. When we talk about syscalls, we mean all the following kernel and user space probes: kprobes, kretprobes, tracepoints, uprobes, uretprobes, USDT probes, raw tracepoints, kfuncs, kretfuncs, lsm probes, bpf iterators, and more. This simplification was made to facilitate the reading and explanation of ReFI at a higher level.

To build these submodules, we analyzed the already built-in eBPF tools that can be found on their open-source project, IOvisor [85]. From this, we based our infrastructure on two tools, where only the syscall names and the skeleton were used.

In more detail, we first developed these tools using BCC, an eBPF project that uses Python as the frontend and facilitates the development of eBPF tools by hiding infrastructure implementation. Soon, we realized that this was not efficient enough and started searching for more efficient options. We discovered libbpf-bootstrap [86], used by Facebook's kernel engineers and changed our stack to it. This change enabled us to divide the eBPF program into two components: kernel and user space. It also allowed us from using C instead of Python, which is much more efficient.

Each eBPF process comprises an eBPF program (kernel space program) and the frontend (user space program) that is stored in the Orchestrator.

The eBPF program contains the code that is running in the kernel, and it has unique security requirements, e.g., not extending beyond a specific size and disabling all loops to avoid infinite loops that could crash the kernel. Because of these requirements, it follows a specific *open, load, and attach* protocol.

The frontend sets up the eBPF program as well as create a direct communication channel

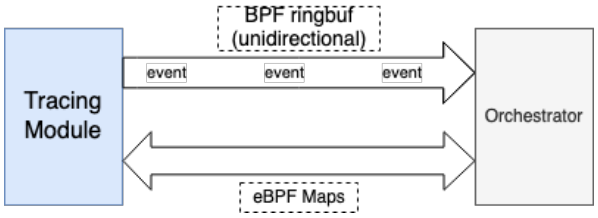


Figure 4.3: ReFI TM communicates via an BPF ringbuf with events and via eBPF maps.

for the events and access to the eBPF maps.

In the kernel space (TM), eBPF program implements the following aspects:

1. Initializes the eBPF maps
2. For each syscall/event:
   (a) Filters the syscalls by PID (Process Identifier)
   (b) Creates the event data structure with the information from the syscall
   (c) Sends the event to the frontend, located in the Orchestrator

In the user space (Orchestrator), the frontend implements the following aspects:

1. Opens the eBPF program
2. Loads the eBPF program by checking with the compiler for errors
3. Attaches the eBPF program in the kernel space and checks for correctness
4. Waits, without blocking, for events
5. Accesses the eBPF maps for information

In Fig. 4.3 we can see that the eBPF programs have two types of data structures to receive information and data. These types are events and eBPF maps. eBPF is an event-driven technology, and each specific syscall represents an event caught in the eBPF program. After catching a syscall, the eBPF program creates an event data structure sent to the front end via a direct communication channel.

There are two types of channels, BPF ringbuf and BPF perfbuf [87]. We decided to use BPF ringbuf because it is more recent and modern as well as has the most crucial features of BPF perfbuf, namely enabling variable-length data records, and efficiently reading data from user space. It also improves several BPF perfbuf issues such as better performance, better applicability, and less memory overhead.

However, ringbuf and perbuf are both unidirectional communication channels. Thus, we need them for an efficient way to treat millions of syscalls, but we also need a way to bidirectionally communicate between the frontend and eBPF program. eBPF maps are the solution. They are key-value data structures, similar to the general concept of maps that can be accessed on both sides.

TM collects information about networking, Virtual File-System, and general information deduced by system calls and I/O. In particular, the TM collects TCP and UDP socket details, using a security socket syscall, and inspects the VFS (Virtual File System) most used syscalls (write, read, open, fsync, and create). This critical information is passed through the Orchestrator that uses it to create a usable state. In Section 4.4, we will discuss in more detail the usable state.

In the TM, we have a special submodule that is not divided into two spaces, the Packet Filter submodule. The Packet Filter submodule is based on a technology called XDP (eXpress Data Path). XDP is a low-level eBPF framework, to monitor and filter packets, that enables eBPF applications to perform high-speed packet processing. XDP enables faster response to network operations because it runs a BPF program immediately as a packet is received by the network interface. This program is attached to the virtual network device thus, not needing the kernel space program. However, the Orchestrator opens, loads and attaches it to the virtual device.

The infrastructure of the XDP program is more complex than the other two submodules, and it requires more setup and libraries. However, this comes with the great advantage of being extremely efficient since it runs the program at a lower level immediately as the packet is received. With the power of filtering packets and collecting their information, we can also use the XDP framework to precisely inject faults into the system, based on the source IP and destination IP of the packets, as we will discuss in more detail in Section 4.3.

An example of data that the TM sends via events is, in the networking submodule, the IP address, the family (ipv4 or ipv6), the protocol (UDP or TCP), the ports, and more. We use eBPF maps for the rules of the XDP submodule because the eBPF program needs to access them, and they are changed via the frontend in the Orchestrator.

It is important to note that an extremely small and not significant amount of events can be lost. However, the faults injected have enough flexibility to this event loss and it does not affect the tool and the bugs caught.

In conclusion, we saw how the TM could be extendable for different syscalls, using the submodules, and flexible to different systems by using a general, black-box manner to extract information. This information is transmitted to the Orchestrator via events and eBPF maps. We also discussed why division into two programs is crucial for the submodules. We tackled the

improvements in the efficiency of the TM. In the following Section 4.3, we will discuss how can we inject faults using the traced information from TM.

4.3 Fault Injection Module

This section will describe the objective of the Fault Injection Module (FIM) and how it satisfies ReFI's objectives. We will recapitulate what types of faults ReFI intends to inject. We will explain why we affirm that these faults are reproducible. We will discuss how we implemented and improved the performance of injecting faults compared to state-of-the-art fault injection mechanisms.

FIM's objective is to inject reproducible faults into the distributed system. The FIM injects a fault after receiving the order from the Orchestrator, which is the module that manages the state. These faults are reproducible because they are injected based on the system state as we show later in the experimental evaluation. Throughout this section, we will abstract the implementation of the Orchestrator and how it manages the state because, in the next Section 4.4, we will explain this topic in more detail. With this abstraction in mind, we will focus on explaining the FIM.

As discussed in Section 4.2, although the goal of ReFI is to inject various types of faults, due to its impracticability, we narrowed these to two types: **network and storage faults**. However, ReFI can be extended to several types of faults, in the future, because of its implementation in submodules.

How does the FIM implement fault injection? In the FIM, we implemented two ways of injecting faults into distributed systems:

1. XDP (packet filter)

2. override the return of the syscalls

XDP

How did we utilize XDP to inject faults into the distributed system?

We used a concept of rules, similar to the idea of firewall rules, where each rule enable or disable unidirectional filtering of packets between two nodes. For example, we can have a system with three nodes (A, B and C) and a network that connects each node. We can add a unidirectional rule to filter out the packets from A to B and another for B to A, creating a partial network partition between nodes A and B.

```
1  ...
2
3  #pragma clang loop unroll(full)         // eBPF flag to enable loops by unrolling them
4  for (__u32 i = 0; i < MAX_RULES; i++) { // at most MAX_RULES iterations
5      __u32 key = i;
6
7      value = bpf_map_lookup_elem(&rules_map, &key);
8      if (value) {
9          if (value->src_ip == 0) {       // to avoid looping an empty array, since the
       loop must be static and not dynamic
10             break;
11         }
12         if (src_ip == value->src_ip && dest_ip == value->dest_ip) {
13             return XDP_DROP;
14         }
15     }
16 }
17
18 return XDP_PASS;
```

Listing 4.1: Loop through the XDP rules

The XDP rules are implemented by attaching the eBPF program to a Virtual Ethernet Device, also known as `veth`. Each node of the distributed system has its own `veth` that handles the network communication. By adding a rule to a specific `veth`, we can enable or disable the communication between the node with that `veth` and other nodes. We use this to create partial and complete network partitions.

How can we specify these rules in the kernel side from the user side (frontend)?

We use the bidirectional way of communicating between kernel and user side, eBPF maps. The Orchestrator creates an eBPF map at the start of the execution of ReFI. Both the kernel and user sides possess a reference to this map, but only the user side attaches rules. The kernel side only has to loop through the rules to filter specific packets. However, this is a challenging task for eBPF since it does not enable dynamic loops because it may cause security vulnerabilities such as infinite loops that crash the kernel.

How did ReFI surpass the challenge of eBPF not enabling dynamic loops?

We used static loops with a specific compiler flag for eBPF. In Listing 4.1, we can observe a few critical changes. First, in line 3, we see the flag enabling the compiler to unroll the loop fully. This flag not only allows the usage of static loops in eBPF but also improves its performance by skipping a few *JUMP* instructions. This performance gain is critical since we are filtering a considerable amount of packets each second. Line 4 indicates that we can attach at most *MAX_RULES*, making it a static loop. Then we have another performance gain, in lines 9 and 10, where we break when the rules are empty. To conclude, lines 13 and 18 show how simple it is to drop or pass packets, respectively, using XDP. These are called XDP actions [88] and can be XDP_PASS, XDP_DROP, XDP_TX and XDP_ABORTED.

Override return

The **override** approach uses the method `bpf_override_return` [89], a specific method from the eBPF debugging methods described in eBPF BCC Reference guide [90].

When used in an eBPF program attached to a syscall, it causes the syscall execution to be skipped, immediately returning an error value instead, which is passed as an argument to the method. Immediately returning means that the usual logic of the syscall is not executed, thus being extremely useful and can be used for fault injection. For example, if we are profiling the syscall `__x64_sys_write`, which is typically used to write in the VFS, we can change the return value to an error value (e.g., -1) and simulate a fault in this syscall.

`bpf_override_return`, by default, only works to a certain subset of syscalls. In the official Linux man-page [91], we can find the subset list of syscalls. However, the probed function can be whitelisted to allow error injections. Whitelisting entails tagging a function with the tag `ALLOW_ERROR_INJECTION` in the kernel source tree. By changing the Linux kernel to add this tag, we would be losing the advantage of using eBPF. With eBPF, we can add logic to the Linux kernel syscalls without changing the kernel itself. When adding this tag, we would lose this great advantage, and every time a final user wanted to enable a new type of fault injection, the kernel had to be changed. On top of this, some syscalls, even with the tag, do not enable this for security reasons. Adding this tag to every probe is not feasible because it would be quite demanding and lose the objective of ReFI being a flexible tool.

In Listing 4.2, we can see a method used in the FIM, where we first filter the syscalls by PID, equal to what we do in the TM. Then, with a certain probability, we override the syscall return in lines 15 and 23 hence simulating a fault.

The XDP approach is more specific than the override of the return approach because it only works for packet filtering. In contrast, the override approach applies to the syscalls, thus being more general. Throughout the development of these two approaches, we inject faults into the sockets syscalls to create network partitions. The XDP approach, in this specific case of network faults, yields better performance results, although the implementation was more complex and challenging due to all the dependencies and learning curve that XDP has.

The state-of-the-art approaches to fault injection, such as Jepsen, used the *IPTABLES* command to inject network faults into distributed systems. The XDP surpasses the `IPTABLES` in performance. The learning curve of XDP is higher than the `IPTABLES`, i.e., XDP is harder to implement at the beginning, but in the long run, it is easier to adjust the rules compared with `IPTABLES`.

In conclusion, we saw how the FIM implements the fault injection using XDP and the override

```
static __always_inline
int fault_injection(struct pt_regs *ctx)
{
  int err;

  if(!filter_pids()) {
    return 0;
  }

  if(probability == 100) {
    err = send_event();
    if(err) {
      return -1;
    }
      bpf_override_return(ctx, -1);
  }
  else {
    if(bpf_get_prandom_u32() < MAX_INT/100*probability) {
      err = send_event();
      if(err) {
        return -1;
      }
      bpf_override_return(ctx, -1);
    }
  }

    return 0;
}
```

Listing 4.2: Usage of the bpf_override_return

of the return. XDP has a better efficiency in the exchange of a steep learning curve. We accomplish reproducible faults by receiving an event from the Orchestrator to inject a fault. We inject faults without understanding the code of the distributed system application, making it a black-box manner to inject faults. We also discussed performance gain over the state-of-the-art approach, `IPTABLES`. In the next section, we will discuss how can we manage the state and coordinate the TM with the FIM.

4.4 Orchestrator

This section will describe the objective of the Orchestrator and how it satisfies ReFI's goals. We will follow the flow of the execution of the Orchestrator. First, we will discuss the initialization of the configuration variables by describing the configuration file template and the configuration submodule. Second, we will discuss how the Orchestrator manages all the frontend (user side) of the eBPF programs. We also present, to the final user, the logs from the TM and how it gets notified of the attached and detach faults from the FIM. We will then discuss how it creates and manages the state. Third, we will see how it coordinates the state to send an event to the FIM to inject reproducible faults.

The **Orchestrator goal** is to initiate, coordinate and manage the flow of the ReFI tool and its main modules, the Tracing Module and Fault Injection Module. Fig. 4.4 shows the design of the Orchestrator module and its components. These components are the configuration submodule,

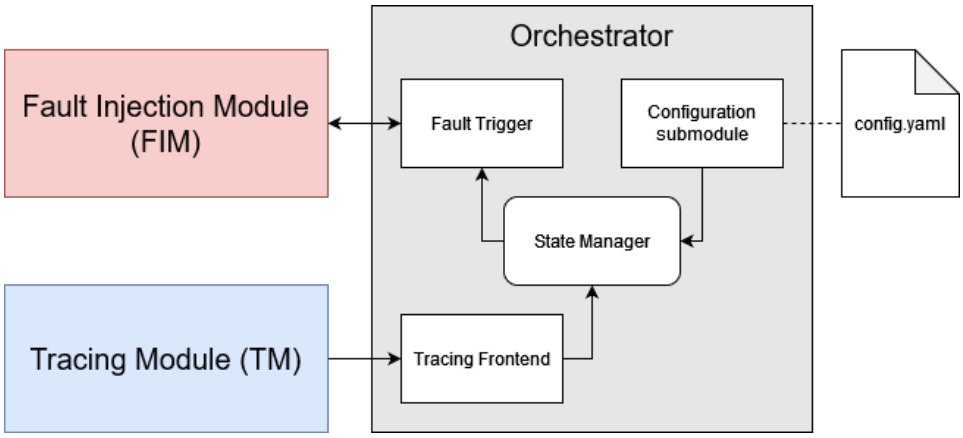


Figure 4.4: ReFI Orchestrator is composed by the configuration submodule, the tracing frontend, the fault trigger, and the state manager.

```
# configs/template.yaml

system : mongo
docker_amount : 3
docker_name : mongo0
docker_name : mongo1
docker_name : mongo2
[primary_index : 0]

fault :
  - begin_state : 10
    end_state : 50
    rules :
      - src_ip : mongo0_IP
        dest_ip : mongo1_IP
      - src_ip : mongo1_IP
        dest_ip : mongo0_IP
      [- isolate : primary]
```

Listing 4.3: The template of the configuration file. The lines with `[]` are optional lines.

the submodules management (tracing frontend and fault trigger), and the state manager. We can also see that the Orchestrator communicates with the TM in a unidirectional channel and with the FIM in a bidirectional channel.

4.4.1 Configuration submodule

The configuration file template describes the basic information of the distributed system configuration. The configuration file is described using the YAML language because it showed to be simple to write, read and understand its content. In our experiments, we used Docker [74], more specifically Docker Compose [92], because it would facilitate the deployment of several distributed systems with different properties in a testing environment. Since the injection of faults can critically fail distributed systems with no comeback, Docker also enabled us to quickly destroy and rebuild the distributed system into a similar initial state. With this, we had extreme flexibility to quickly test different environments by simply modifying a Dockerfile and using

Docker Compose.

The scenario described in the Listing 4.3 is only used for explaining purposes. Briefly, it describes a MongoDB distributed system with three nodes and it injects one fault. In Chapter 5, we demonstrate specific examples of configuration files that were used in the experiments made with ReFI.

In Listing 4.3, we define several fundamental constants used in the flow of ReFI. First, the system type change how we collect the state for each system. Secondly, the amount and the names of the containers. Thirdly, the faults, where we specify the initial and final state and what are the fault rules. As a quick recap from Section 4.3, these rules are unidirectional rules similar to firewall rules, where we specify the source and destination IP we want to disable communications.

The `docker_amount` is a configuration variable that stores the number of nodes that compose the distributed system. This constant is useful because we are using C as our main language for the development and implementation of ReFI. In C, although there are workarounds for this problem, such as dynamically allocating memory to the vectors, this simple configuration variable facilitates the implementation of several aspects of the ReFI, such as the creation of vectors and specify the number of iterations in loops.

The `docker_name` are a sequence of variables that are stored to obtain variables from it. For example, from the name of the container, we can deduct their IP and their `veth` name (used for XDP attachment). These are crucial variables that we indirectly deduct from the configuration file.

The *fault* is composed by a `begin_state` and an `end_state`, and a set of *rules* to enable a partial network partition. Looking into the specific Listing 4.3, we will inject a network partition fault in the state 10 and heal the partition in the state 50, and the network partition partially isolates the `mongo0` from `mongo1`, and vice-versa. In Fig. 4.5, we can see a sketch of how the fault would work based on this configuration file template.

There are two optional properties that can be added to the YAML configuration file. The first is the `primary_index` property and the second is the `isolate` property.

The `primary_index` property enables the Orchestrator to know, beforehand, which of the nodes is the leader. Of course, this is only useful in distributed systems that utilize leaders to manage the system. For the specific case of MongoDB, we created a Python script that automatically discovered the leader node.

The other optional property is in the fault section called *isolate*. This fault property is, again, specific for distributed systems that have a primary/leader. The *isolate* specifies a specific group

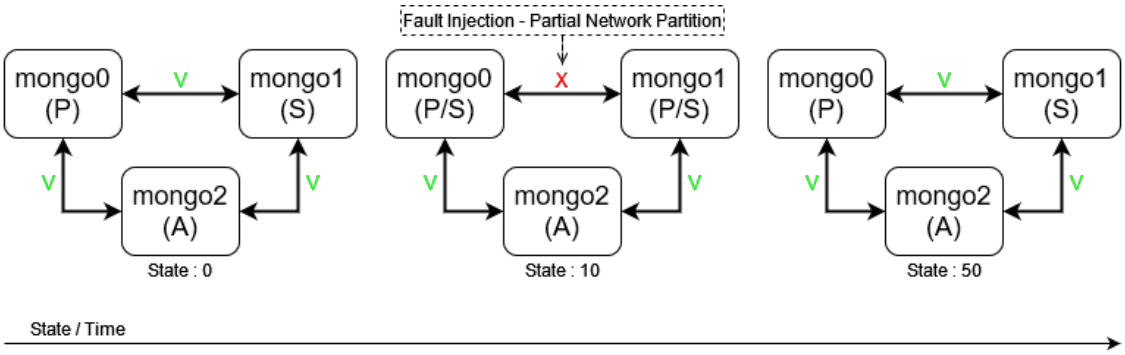


Figure 4.5: Representation of the experiment that the configuration file template describes. The green downwards arrow means that the network is working and the red cross means it is not working. ((P)-Primary, (S)-Secondary, (A)-Arbiter)

of rules and facilitates the reading and writing of the configuration file. In Fig. 4.5, if we wanted to isolate the primary from all the nodes, we would have to make two rules per node, excluding the primary, i.e., four rules. With this optional property, we simply use one rule to specify that we want to isolate the primary. This facilitates fault specification for large systems.

Note that the state shown in the configuration file is not time. We will explain this in detail in Section 4.4.3.

Note that some of the information described in the configuration file must be known before the experiments by the user of the ReFI tool, independently of how they fix these values, such as the names of the containers and their IP's. For example, in our experiments, we specified the names and IP's of the containers in the Dockerfile.

4.4.2 Submodules Management

The Orchestrator is responsible for the initialization of the eBPF code responsible for tracing and injecting the faults. To be more specific, it is responsible for the following eBPF programs: a program that traces the network sockets, a program that traces the VFS syscalls and the fault injection program that overrides the return of the syscalls. The first three programs are from the tracing frontend and the last program is from the fault trigger.

To initialize these programs, we use three core methods that come from eBPF helper functions: `open`, `load` and `attach`. In Listing 4.4 we can see the frontend (user side) of the VFS tracing submodule. The open follows the same flow as when opening a file in the Unix system. Then we can make operations, in these cases, specify the PID of a node of the system or the probability of the fault to be successful and other variables for the eBPF program. Then, we load the program into the kernel. When the program is compiled, it is checked for security issues and other issues. Finally, if it passes all checks, it gets attached to the kernel and now it is running

```c
static struct vfsstat_bpf* vfsstat_open_load_attach()
{
  int err;
  /* Load and verify BPF application */
  vfsstat_skel = vfsstat_bpf__open();
  if (!vfsstat_skel) {
    fprintf(stderr, "Failed to open and load BPF skeleton\n");
    return NULL;
  }

  /* Parameterize BPF code with filter PID parameter */
  vfsstat_skel->rodata->filter_pid = env.pid;
  err = bpf_map__reuse_fd(vfsstat_skel->maps.rb, bpf_map__fd(tcptop_skel->maps.rb));
  if (err) {
    fprintf(stderr, "Failed to load and verify BPF skeleton\n");
    goto cleanup;
  }

  /* Load & verify BPF programs */
  err = vfsstat_bpf__load(vfsstat_skel);
  if (err) {
    fprintf(stderr, "Failed to load and verify BPF skeleton\n");
    goto cleanup;
  }

  /* Attach tracepoints */
  err = vfsstat_bpf__attach(vfsstat_skel);
  if (err) {
    fprintf(stderr, "Failed to attach BPF skeleton\n");
    goto cleanup;
  }

  return vfsstat_skel;

cleanup:
  /* Clean up */
  vfsstat_bpf__destroy(vfsstat_skel);

  return NULL;
}
```

Listing 4.4: The template of the configuration file

and intercepting the syscalls.

The frontend of the eBPF programs is also composed of the non-blocking waiting for events that send information from the TM to the Orchestrator. The Orchestrator creates a ring buffer that is constantly polling, without blocking the execution of ReFI. The polling is done every ten milliseconds, but it can be adjusted. The smaller the interval, the more CPU overhead it would take. Thus, ten milliseconds was a middle term between CPU usage and losing events, and hence precision.

As explained before, the events are not the only way to communicate between the TM and FIM with the Orchestrator. eBPF maps allow for bidirectional communication. Both sides of the communication can use them to add, delete and update the key-value data structure.

The information collected from the events and eBPF maps of the TM is then converted into a usable state. The final user can also turn on the logs for the ReFI, which is useful for seeing the information being sent from the Orchestrator to the TM, in the beginning, and from the TM to the Orchestrator, throughout the execution of the program. The logs from the TM consist of

events, and from the FIM consist of when a fault is injected and healed.

4.4.3 System State

The state of the system can be **uni-state** or **multi-state**. A state is described as a **uni-state** when it is based on only one type of data that comes from the TM. When the state is based on a group of data from the TM, it is called a **multi-state** state.

In ReFI, we collect information about the network through socket communication and information of the system storage, through VFS syscalls, such as writes, reads, opens, fsyncs and creates. The usable state depends on this information, but it is different depending on the system being tested.

Throughout all different experiments made with ReFI, we used different approaches. In the first approach, we used the communication via sockets to obtain information about the system state. This turned out to be excellent when evaluating the system state via the client. However, in a later analysis, we decided to base our state on the nodes of the system to enable more fault patterns, and there was a lot of communication that did not contribute to the state when analyzing the nodes communication, such as heartbeats from each server to the leader. To filter out this useless information, we tried a solution using a multi-state such as the following. When the system called a socket syscall of type receiving (*recv*), we would look into the eBPF maps that contain that information and compare it with the eBPF maps that contain the information from the VFS. If the VFS map changed too, it would mean that it was a socket syscall that changed something essential in the system that had to be stored in the VFS. This strategy was challenging to have low false positives, and it became complex, so we took another approach. Instead of analyzing both and creating a multi-state, we followed the *principle of parsimony*, where we used simple strategies to accomplish our goals. In this matter, if the state depends on each system, we created a state that fitted our experiments. In our particular case, the state would increment when we received a message with a specific size, thus filtering out the unnecessary heartbeat messages that difficult the collection of a usable state.

This state was stored in the execution of the Orchestrator. There are two core methods - update the state and check the state. When we received an event from the TM, it would pass through the state updater that would increase the state depending on the information. If it modified the state, it would then check if the fault injection condition described in the configuration file was met. If yes, it sends an event for the FIM, and a fault is injected.

In conclusion, we saw how the Orchestrator initializes ReFI using a YAML configuration file. We saw how the Orchestrator manages the frontend by initializing them with the correct

arguments. We also described the usable state in more detail, such as uni-state and multi-state state, and which one we used and when. We presented how we used this state to communicate to FIM that a fault needs to be injected as well as the logs information.

4.4.4 Summary

The ReFI design takes into consideration ReFI goals. For ReFI to be reproducible, we built the Orchestrator module that receives data from the TM, creates a usable state, and use it to trigger the FIM. This enables the injection of faults based on the system state.

The TM collects data in a black-box manner because it only traces syscalls. The implementation of the TM in submodules that have a specific syscall to trace, enable the extendability for new submodules to trace new syscalls. The flexibility to use ReFI for different distributed systems is obtained due to the black-box approach of collecting the data.

Finally, we use eBPF, and its technology, such as XDP, to efficiently trace data and trigger faults.

Chapter 5

Evaluation

This chapter describes how we evaluated the ReFI prototype and how this evaluation correlates with its goals. In a detailed manner, we describe the methodology used to evaluate ReFI and its modules and how the experiments were created at the beginning of each section. In Section 5.1, describes the methodology used in our evaluation and a brief background of MongoDB and Redis. In Section 5.2, we present the results of ReFI's efficiency. We used three quantitative metrics, time overhead, throughput, and latency. We also present one qualitative metric, the severity of the bugs detected, to evaluate the experiments described in Section 5.3. Section 5.3 describes real experiments that utilize the ReFI prototype and exemplify how ReFI can improve the resilience and robustness in real modern distributed systems. We perform, from basic to advanced, three different experiments: Unavailability MongoDB, Data Loss MongoDB and Split-brain Redis. These three experiments show real use cases and demonstrate how ReFI accomplishes its main goals. Finally, in Section 5.4 we quickly recap the important aspects, as well as respond to the three key questions that this chapter intends to fulfil.

The success of ReFI depends on the performance of accomplishing its four key goals. ReFI's prototype (1) should be able to inject faults in a reproducible manner, (2) by treating the application as a black-box, (3) in an efficient way, and (4) it should be flexible to different distributed systems as well as extendable to different syscalls.

Given these goals, our evaluation intends to answer three key questions:

1. Is ReFI able to reproduce faults?
2. Is ReFI efficient?
3. Is ReFI able to test different distributed systems?

The goal of treating the application as a black-box is already proven in Section 4 since we explain how the TM module collects information about the application without understanding

its source code and algorithms. However, we once more architect our evaluation and experiments to restate that ReFI satisfies its goals.

Question 1 allows us to access how ReFI accomplish the goal of **reproducible** injection of faults. To evaluate if ReFI can reproduce the injected fault, we will use a standard approach on experimental evaluation, which is the repetition of the same experiment several times and check if the results are identical. Repeating the same experiment several times will accomplish two essential results. Firstly, it will decrease errors in measuring, a standard procedure in experimental evaluation. Secondly, it will prove ReFI can reproduce the injected faults based on the state because the system shows similar behaviour for each execution. The experiments that inject faults can either produce a bug or not. Regardless of the final result, the experiments will be repeated for the two reasons explained before. It is important to note that if the injected fault produces a failure, it is crucial to our work since we intend to discover bugs in applications to improve their resilience and robustness. Nevertheless, if the fault does not produce a failure, it could prove that the application is resilient and robust. Thus, fault-free and faulty executions need to be evaluated several times.

Question 2 intends to prove that ReFI accomplishes its goals in an **efficient** way. To evaluate if ReFI is efficient, we measure the overhead of the application with and without ReFI tracing enabled, with a specific workload, and then compare them. It is expected that the time without ReFI should always be shorter than the time with ReFI. Nevertheless, this directly shows the low overhead, thus, the high efficiency of ReFI.

We will evaluate the efficiency of the ReFI framework as a whole but also the efficiency of the Tracing Module. We predict that the TM will be the component with the highest overhead as it will be running all the time and filtering millions of syscalls.

Since ReFI's primary technology is eBPF, programs can be attached and detached to enable and disable specific tracing tools. As opposed to instrumentation tools, such as PIN [31], eBPF adds zero-overhead [30] to the application, when not in use, due to its event-based architecture.

The throughput metric will consist of the number of operations executed during an experiment. Since ReFI is a black-box approach, we will measure the throughput through the output of operations. For example, in a distributed database such as MongoDB [53], the number of writes and reads that we can execute per minute.

Question 3 demonstrates how ReFI can test different distributed systems. Our experiments carefully test different systems with different requirements to demonstrate the application-agnostic property that ReFI enables. For this, we developed two experiments in MongoDB and one in Redis. For ReFI to work with different distributed systems, we only had to create a new

configuration file for the specific system and change how we managed the state.

The bugs found in the experiments have different severity depending on the impact that they cause on the distributed system. The severity of a bug can be `Minor`, when the bug manifests unexpected or undesired behaviour, but not enough to disrupt system function. `Major`, when the bug can collapse large parts of the system. `Critical`, when the bug can trigger complete system shutdown.

Since ReFI is a novelty in fault injection approaches because it was created to be reproducible, it is challenging to analyze and compare ReFI with other baselines. However, one of the state-of-the-art prototypes that test their overhead is FCatch [15]. FCatch imposes a 5.6x - 15.2x overhead slowdown on the system depending on the workload. Most of this slowdown is caused by the inefficient technology utilized, Javassist [93]. In our case, we used eBPF, which has been tested by the community [26, 30, 94] and recognise it as an efficient tool.

5.1 Methodology

The methodology of the ReFI's overhead evaluation largely depends on the workload of the application. For this reason, we had to decide which system would be used to test ReFI overhead. The best candidate we had was MongoDB [53] because of three concrete reasons. We already had experiments using this particular distributed application. Researchers and the community largely use Yahoo Cloud Serving Benchmark [95] workloads to test their applications. YCSB already has several fundamental workloads that we can test different behaviours.

After deciding on MongoDB as our testing distributed system, we have another critical decision which is to decide what workload to test ReFI. Since we already planned to use YCSB, there are six default workloads that came with the benchmark. These workloads are named starting on A and ending in F. Most of the workloads are focused on querying/reading the data.

Workload A, *update heavy workload*, has a mix of 50% read and 50% writes.

Workload B, *Read mostly workload*, is a read-mostly workload that has 95% reads and 5% of writes.

Workload C, *Read only*, is a read-only workload, so it has 100% read operations.

Workload D, *Read latest workload*, inserts new records and the most recently inserted records are the most popular, i.e., the records with the most read frequency.

Workload E, *Short ranges*, queries short ranges of records, instead of individual records. Workload E is more used for applications where bulk reads are important.

Workload F, *Read-modify-write*, the client will read a record, modify it, and write back the changes. YCSB uses a random delta for writing rather than some value derived from the current

YCSB Workloads	Read (%)	Write (%)	Behavior
A	50	50	Default
B	95	5	Default
C	100	0	Default
D	95	5	Mostly read latest writes
E	95	5	Short bulk reads
F	50	50	Read, modify, write back

Table 5.1: Summary of the YCSB workloads. In our evaluation we used workload A - Update heavy workload - and workload F - Read-modify-write.

record (say incrementing a counter).

Table 5.1 summarizes the percentage of read and write operations, as well as the general behaviour of the YCSB workloads.

For our purposes, we chose workload A, which has a mix of 50% read and 50% writes, and workload F, which simulates a client reading a record to modify it and write the changes back. We decided to use these two specific workloads because they were the ones that had a significant amount of writes and this enables us to test the read and write syscalls. The other workloads, B to E, are mainly composed by reads and would not test the write syscalls enough. These two systems are used in the experiment section, thus being a critical factor to choose these workloads.

To run these workloads, we have two key commands, *load* and *run*. The load command loads the database and the run command runs the workload specified into the loaded database. However, none of these commands deletes the database at the end and so, we created a script to delete the database at the end of each run. Each run, consisting of loading, running and resetting. Depending on the arguments given to load and run commands, they could perform different behaviours, such as being synchronous or asynchronous, and more.

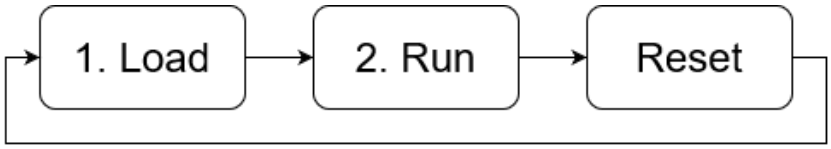
The evaluation followed the three steps represented on Fig. 5.1 (a):

1. We would load the database.

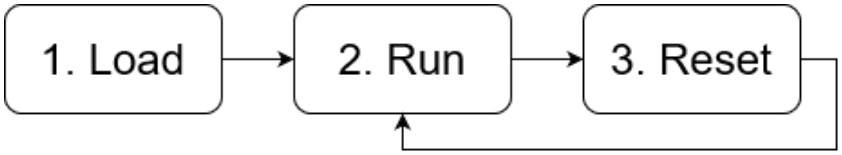
2. Then we would run the experiment specified in the workload A and F.

3. Finally, we connect to the distributed database application and reset it, by deleting the YCSB testing database (created on step 1) to perform the same test.

However, there was another method that we considered, approach (b) showed in Fig. 5.1. Although approach (b) is considerably faster than approach (a), we discarded this method because we wanted to have isolation between tests, where a test does not depend on other tests. The approach (b) has a dependency from the first load and thus can be biased to the first load, where the approach (a) shows a more realistic test because it is not deterministic, similar to distributed systems.



(a) Approach



(b) Approach

Figure 5.1: The two different approaches for testing ReFI with YCSB workloads. ReFI evaluation utilizes approach (a).

We repeated these three steps, five times for each environment we wanted to evaluate. We evaluated three environments. The first was the workload without ReFI. The second was the workload with ReFI. The third and final was the workload with ReFI and verbose mode enabled. Verbose mode is when ReFI prints to the console/terminal the logs of the collected information from each module, TM, FIM, and Orchestrator. The results obtained are presented in Table 5.2. These three environments enable us to present ReFI's overhead and how the logging system affects ReFI.

We briefly give a background on the distributed applications used in Section 5.3, MongoDB and Redis.

For all the experiments we used a VM with Ubuntu 20.10, 64 bits, installed with 4GB of RAM, 4 processor cores, and 110 hard disk capacity. For this VM setup, we used VMWare Workstation 16 player. The main reason for the usage of Ubuntu 20.10 was that it already comes with the default installation of some eBPF prerequisites.

5.1.1 MongoDB Background

MongoDB is a cross-platform document-oriented database program. MongoDB uses JSON-like documents that do not need a schema but you can optionally define one if it makes sense. MongoDB provides the most common data structures and enables atomic operations. It supports transactions, ad-hoc queries, indexing, and asynchronous replication.

MongoDB can be accessed via Cloud using the MongoDB Atlas product but in this dissertation, we are only using the server based product, MongoDB Enterprise Advanced. We made this decision because the Enterprise version gives us the flexibility to create our own replica set with specific rules. A MongoDB replica set is a distributed system where each node has a replica of the data. MongoDB uses a PSA hierarchy where it stands for the *Primary*, as a leader, *Secondary*

and *Arbiter* nodes.

The primary server is the leader and it is elected by the secondary and arbiters in a leader election. The primary is the only one that accepts writes and reads to the database application. Although it is possible for secondaries to accept reads, MongoDB states that they only query stale reads, which means that their might already be accepted new writes that are being replicated and do not show in these secondary reads.

The secondary server has to goal to store replicated data and contribute to the leader election voting.

The arbiter server has the goal to only contribute to the leader election voting but do not store replicated data.

MongoDB has already built in the failover system for when it starts a new leader election, thus it does not need any auxiliary node to manage this behaviour.

MongoDB also supports sharding, i.e., there can be a cluster, not a replica set, that contains nodes that only have specific data and this is all managed by the internal software of MongoDB. In the experiments, we did not use this feature of MongoDB.

5.1.2 Redis Background

Redis is an in-memory data structure store, used as a database, cache, and message broker. It differs from the other databases as it is an in-memory data storage system, thus being normally used as a faster cache system but can also persist data by periodically dumping the dataset to disk or by appending each command to a disk-based log. Redis provides the most common data structures and enables atomic operations like increment a value among others. Redis has built-in replication, asynchronous replication, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

Before continuing, it is important to notice that **Redis and MongoDB** have a similar approach where it comes to the replication of the data. Although some of the names are different, the idea behind them is similar in both storage systems. For example, a Redis cluster is the same as a MongoDB replica set, a Redis master and slaves are the same as a MongoDB primary and secondaries.

Nevertheless, Redis introduces new concepts to the storage distributed systems, such as **Redis Sentinel**. Redis Sentinel has the goal of monitoring, notification, automatic failover, and configuring the provider, all of this being an isolated component from the Redis instances (master and slaves). They provide high availability to the distributed system by checking if the nodes are alive. They are responsible for *monitoring* the master and the replicas to check if they

are working as expected. They *notify* the system administrator when something unexpected happened with the Redis instances. Most important, they are the mechanism that *automatically starts a failover* process, such as leader election.

With this background of Redis, we will now discuss the experiment.

5.2 Overhead

This section presents the throughput and latency of ReFI, the methods used to evaluate TM independently from the other ReFI modules and shows the results collected. TM's overhead is evaluated independently because it has the capability of processing millions of syscalls requests and each request needs to be efficient.

5.2.1 ReFI

From Table 5.2, we can observe that for 100000 operations, where 50% are reads and 50% are writes, it took, on average, 22 seconds. With ReFI turned on, it suffer an increase of 30% (1.311357396) on the overall run time compared with the default, passing from 22 to 27 seconds.

In regards to the throughput, we had a decrease of 24% (0.7625766585) in workload A. In regards to the latency, ReFI adds around 38% (1.383724836) read and 26% (1.26847078) write latency.

From Table 5.3, we can observe that for 100000 operations, where 50% are reads and 50% are writes, where the behaviour is read, modify and write back, it took, on average, 41 seconds. With ReFI turned on, it suffer an increase of 23% (1.230200496) on the overall run time compared with the default, passing from 41 to 51 seconds.

In regards to the throughput, we had a decrease of 19% (0.8133174104), i.e., in terms of operations, ReFI slows the throughput around 500 operations per second in the workload F. In regards to the latency, ReFI adds around 25% (1.25367864) read and 20% (1.205141055) write latency, i.e., ReFI adds around 50 microseconds to reads and 70 microseconds to writes.

In regards to the ReFI with verbose mode enabled, in both workloads, they increase the overhead more than without the verbose mode enabled. This was expected and it is a trade-off that engineers need to do when understanding the system via ReFI. Approximately, the verbose mode adds twice the overhead when compared to the non-verbose mode.

1 000 000 Operations				
	Run Time(ms)	Throughput(ops/sec)	[R] avg. Latency(us)	[W] avg. Latency(us)
wo/ ReFI				
	22917	4363.572893	188.5043584	260.4294423
	21165	4724.781479	169.1793846	245.0113554
	21505	4650.081376	171.1940353	248.5934046
	22741	4397.344004	184.3561961	259.8886288
	22384	4467.476769	183.4059152	254.4316614
avg.	22142.4	4520.651304	179.3279779	253.6708985
Baseline	1	1	1	1
ReFI				
	29422	3398.817212	252.6373249	324.3983046
	30441	3285.043198	262.5314917	335.1478458
	27774	3600.489667	234.927099	310.2084342
	29182	3426.769927	248.3570339	324.7921866
	28364	3525.595826	242.2499347	314.3238452
avg.	29036.6	3447.343166	248.1405768	321.7741233
Overhead	1.311357396	0.7625766585	1.383724836	1.268470783
ReFI -v				
	58655	1704.884494	470.9349874	675.241313
	59262	1687.421957	477.2578528	681.1301531
	58200	1718.213058	473.1592996	664.1231584
	57811	1729.77461	465.2649108	662.9462809
	58067	1722.148553	472.9311754	663.7909049
avg.	58399	1712.488534	471.9096452	669.4463621
Overall	2.637428644	0.3788145599	2.631545009	2.639034931

Table 5.2: Time overhead of ReFI using the YCSB workload A for environment without ReFI, with ReFI and with ReFI and verbose mode (ReFI -v). The [R] and [W] in the latency column stand for Read and Write latency.

5.2.2 TM

methodology

To evaluate the TM independently, we subdivided this task into two parts. The first part, we used the methodology available by the bpftool [96], a helper tool for eBPF developers. The bpftool enables us to turn on the flag `kernel.bpf_stats_enabled sysctl` that turns on stats collection for all eBPF programs. When this flag is enabled, all eBPF programs collect their status, it does not give the option to choose which one to collect stats. This flag exposes two variables, `run_time_ns` and `run_cnt`, that show the total run time, in nanoseconds, for a specific

	1 000 000 Operations			
	Run Time(ms)	Throughput(ops/sec)	[R] avg. Latency(us)	[W] avg. Latency(us)
wo/ ReFI				
	41168	2429.071123	234.17338	335.3317545
	42895	2331.274041	244.90414	349.4563969
	41737	2395.955627	239.78182	336.1715639
	40992	2439.50039	236.0936	330.7062237
	41092	2433.563711	234.47033	334.6617348
avg.	41576.8	2405.872978	237.884654	337.2655348
Baseline	1	1	1	1
ReFI				
	48828	2048.005243	286.71355	383.474496
	50133	1994.694114	292.60207	398.7653049
	51657	1935.846062	300.28097	410.6435521
	52486	1905.269977	305.63403	417.6562138
	52635	1899.876508	305.92393	421.7231449
avg.	51147.8	1956.738381	298.23091	406.4525423
Overhead	1.230200496	0.8133174104	1.253678642	1.205141055
ReFI -v				
	74864	1335.755503	423.39517	614.8937484
	78216	1278.510791	442.6566	643.4304953
	76886	1300.626902	435.55735	633.3025883
	77842	1284.653529	442.59977	639.0090146
	78182	1279.066793	442.04749	643.5863571
avg.	77198	1295.722704	437.251276	634.8444408
Overall	1.856756653	0.5385665474	1.838081056	1.882328241

Table 5.3: Time overhead of ReFI using the YCSB workload F for environment without ReFI, with ReFI and with ReFI and verbose mode (ReFI -v). The [R] and [W] in the latency column stand for Read and Write latency.

eBPF program and the number of times that the eBPF program was called, respectively. With these two variables, we can check the performance of the attached eBPF programs from the TM.

We used the workload A with approach (a), in the same manner that we used in the ReFI overhead evaluation, but instead of measuring the overhead, we enabled the `kernel.bpf_stats_enabled` `sysctl` flag and measured the `run_time_ns` and `run_cnt` variables from the TM eBPF programs attached. These eBPF programs are `security_socket ss_sendmsg` and `security_socket ss_recvmsg` from the network submodule of TM, and `vfs_read`, `vfs_write`, and `vfs_open` from the VFS submodule of TM. The `vfs_fsync` and `vfs_create` are not considered because they do not make sense in this particular case, i.e., they are not utilized. These would make sense in a file-system,

```
SEC("kprobe/vfs_read")
int BPF_KPROBE(vfs_read, struct file *file, void *buf, size_t size)
{
    u64 start = bpf_ktime_get_ns();     // start schedule clock

    u32 pid = bpf_get_current_pid_tgid() >> 32;

  if (filter_pid && pid != filter_pid) {
    return 0;
  }

    // ... code to get information from syscall

    bpf_printk("READ %d\n", bpf_ktime_get_ns() - start); // log of end of schedule clock

  return inc_stats(S_READ);
}
```

Listing 5.1: Code example of task 2 evaluation.

for example.

However, eBPF collects information from syscalls that come from every process. eBPF filters unwanted syscalls by checking the PID of the process, and filtering out unwanted processes. Thus, a huge size of the syscalls is discarded without almost no computation. Since these unwanted/filtered calls appear in more quantity than the wanted/not filtered calls, this task shows an average of the overall syscalls, which is essential to analyze the system but also we need to evaluate the wanted calls with more precision.

For this reason, we created a second task, where we could evaluate specifically the calls that were not filtered and thus, they executed all the code that the eBPF program attached to the Linux kernel. The second task has the goal to only evaluate these wanted syscalls. We schedule a clock at the beginning of each syscall that only terminates if the call goes until the end, thus it only register the wanted calls. Then we collect this clock via logs from the eBPF program.

To better explain this division, in Listing 5.1, we demonstrate the pseudo-code of the eBPF program attached to the kernel. In this code, we can see a condition, an if statement, that filters out all the syscalls that came from another PID, i.e, that came from unwanted processes. After this condition, the code that intercepts the information that the syscall possess and where the computation takes more performance and time. Since only one in around 1000 syscalls are wanted, with the first part of the evaluation, we were averaging out these important calls. This second task is a solution designed by us to prevent this problem in the evaluation.

results

In Table 5.4 and Table 5.5, we present the results of task 1 measures. We also present some useful results that we can derive from the `run_time_ns` and `run_cnt` variables. From observing these tables, we can see that the `security_socket ss_sendmsg` syscall has a much higher average

overhead than the `vfs_read` syscalls. This can happen because we have much more information to collect from the network submodule, 10 arguments, whereas from the VFS submodule we only collect 3 arguments.

In Table 5.6, we present the results of task 2 measures. We present the average of the wanted calls, as well as the minimum and maximum of these values to show that the values are really sparse, due to being called on the Linux kernel which has a high number of interruptions. We can see, for example, that although we have an average of 8131 nanoseconds for each *VFS READ* syscall, without interruptions we can have 273 nanoseconds call, which is extremely efficient for the goal of collecting information from syscalls.

	1 000 000 Operations		
	Number of Syscalls	Overall Run Time (ns)	Run Time per Call (ns)
ss_sendmsg			
	219631	2483335923	11306.85524
	222558	2487414767	11176.47879
	219732	2355059248	10717.87108
	218573	2395660524	10960.45954
	220635	2560211766	10960.45954
avg.	220225.8	2456336446	11153.0996
ss_recvmsg			
	363573	3752322687	10320.68577
	372472	3428599944	9204.98707
	363693	3637070691	10000.38684
	359427	3450084490	9598.846191
	365985	3720742584	10166.38
avg.	365030	3597764079	9858.257174

Table 5.4: Task 1: Raw run time overhead of the TM module. In this table, we present the results of two syscalls, ss_sendmsg (security socket send message) and ss_recvmsg (security socket receive message), that compose the network submodule.

The method used to collect these results add, on average, 20 nanoseconds per traced syscall because both of them make two calls to *sched-clock()* method. One call to start and the other to end the clock. The *sched-clock()* is a kernel method to obtain the current time, and each call takes approximately 10 nanoseconds.

	1 000 000 Operations		
	Number of Syscalls	Overall Run Time (ns)	Run Time per Call (ns)
vfs_read			
	139237	44997633	323.1729569
	139799	46705091	334.087447
	136789	54248082	396.5821959
	136255	44056278	323.3369638
	138344	53283369	385.1512823
avg.	138084.8	48658090.6	352.4661692
vfs_write			
	124339	262932787	2114.644536
	127128	331148504	2604.843182
	125966	396723115	3149.446001
	125519	335608231	2673.764378
	123479	293422153	2376.291944
avg.	125286.2	323966958	2583.798008
vfs_open			
	19826	7496909	378.1352265
	19319	7518188	389.1603085
	17886	7247458	405.2028402
	17732	6725277	379.2734604
	19186	6727356	350.6387991
avg.	18789.8	7143037.6	380.4821269

Table 5.5: Task 1: Raw run time overhead of the TM module. In this table, we present the results of three syscalls, vfs_read, vfs_write, and vfs_open, that are part of the VFS submodule.

Task 2	Average (ns)	Minimum (ns)	Maximum (ns)
ss_recvmsg	13998	990	314599
ss_sendmsg	18889	2967	281478
vfs_read	8131	273	110403
vfs_write	16936	444	274254
vfs_open	12921	205	95637

Table 5.6: Task 2: Average, minimum and maximum values obtained from only the wanted syscalls from TM module.

5.3 Experiments

In this section, we detail three experiments made with ReFI, two using MongoDB and one with Redis. For each experiment, we describe the overview of the experiment, how we setup the distributed system, how we integrated ReFI to trace and inject faults into the distributed

system, the workload used, how and when we inject the fault, and why this caused a bug. We also present the results and evaluate the severity of the failure/bug encountered.

5.3.1 Unavailability - MongoDB

This experiment was reproduced by ReFI and it has the specific goal to test the FIM, the most critical part of a Fault Injection framework. This experiment demonstrates that ReFI FIM is working correctly and as expected. We used MongoDB [53], a document-based distributed database, as our distributed application. In specific, for this experiment, we used MongoDB 3.2.10 version.

The experiment is based on an issue made in the MongoDB Jira forum [97], found in the research article made by Alfatafta et al. [39] described in the Section 3.3.2.

Overview

The experiment consists of three nodes that have a network connection between each node. These three nodes form a replica set, i.e., they form a group of database servers that will replicate data to each other. They replicate data, following a PSA hierarchy, with one primary, one secondary and one arbiter.

In Fig. 5.2 shows a diagram of the unavailability MongoDB bug. The bug consists of an injection of a partial network partition fault that triggers an infinite loop of leader elections. For a primary node to stay primary, it needs to have a majority of nodes, in this case 2. The secondary cannot communicate with the primary node, so it starts a new leader election. Since it can communicate with itself and the arbiter, the secondary node wins the leader election. The same process happens to the primary since the arbiter tells it that there is a new primary. Node 1 (initially primary) and node 2 (initially secondary) are constantly swapping between primary and secondary roles. While this is happening, the clients cannot write to the storage system because they do not agree on which is the primary node.

Setup

For the setup of MongoDB, we used Docker Compose [92] and the *docker-compose.yaml* file, to automatically initiate the distributed system. For this experiment, we initiated three nodes that formed a replica set, each of them having a static IP address and a static port to facilitate the usage of ReFI.

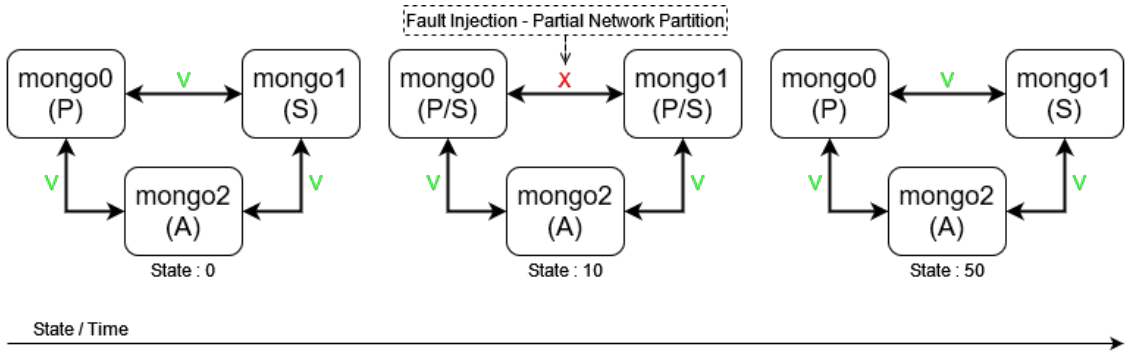


Figure 5.2: Diagram showing an overview of the MongoDB unavailability bug. We have 3 nodes, a Primary (P), a Secondary (S), and an Arbiter (A). After a partial network partition there is an infinite swap of the new primary.

```
# config/public.yaml

system : mongo
docker_amount : 3
docker_name : mongo3-2-10_rs0-0
docker_name : mongo3-2-10_rs0-1
docker_name : mongo3-2-10_rs0-2
primary_index : 0

fault :
  - begin_state : 0
    end_state : 50
    rules :
      - src_ip : 172.19.0.2
        dest_ip : 172.19.0.3
      - src_ip : 172.19.0.3
        dest_ip : 172.19.0.2
```

Listing 5.2: Configuration file for the unavailability MongoDB bug.

ReFI integration

To enable ReFI to use the TM to collect information, create a state, and inject reproducible faults using that state, we specify the configuration file.

In Listing 5.2, we can see that we specify three docker names as well as their IP addresses for the rules of the faults that we want to inject. These IP addresses are static and specified in the Docker Compose configuration file.

We only have to specify the configuration file as an argument for ReFI to parse it and the experiment executes.

Workload

This experiment is the most simple one and has the specific goal to prove that ReFI FIM is working correctly and as expected. So in this particular case, no workload was needed. We analyzed the logs of the distributed system, from the *docker-compose up* command, and can take the necessary conclusions to observe the unavailability of the system.

Injected Faults

As we can see from the configuration file, presented in the Listing 5.2, we inject a fault that consists of two rules. Each rule prevents unidirectional communication with two nodes. The fault begins at state 0, at the beginning of ReFI and ends in state 50. In this specific experiment, the state is the amount of *WRITE* syscalls that are of a certain size.

The logs shown from the *docker-compose up* command precisely show the primary and secondary continuously changing, and this happens until the partial network partition is healed.

Bug explanation

The bug occurs because node 0 and node 1, the nodes that store data, cannot communicate with each other, but there is a third node, node 2, that can communicate with both. This implies that whenever the secondary cannot communicate with the primary and it can communicate with a majority of nodes, 2 in this case, itself and the arbiter, it invokes a leader election that it wins since it is the only viable candidate. Nevertheless, the same happens again to the new secondary and an infinite loop is generated.

This shows a problem on the MongoDB failover system, as well as, that the FIM is working as expected, the faults are successfully being injected, and attacking the system reliability.

Bug severity

The unavailability bug is a major bug since it disables all operations that can be made to the distributed database. It is also an extremely simple yet devastating bug. This is just a great example of how we can improve the modern application distributed system with a small and precise injection of faults and improve their robustness and reliability.

5.3.2 Data Loss - MongoDB

This experiment was reproduced by ReFI and it has the goal to test the ReFI prototype and all its modules, i.e, TM, FIM, and Orchestrator. This experiment demonstrates that ReFI, as a whole, is working correctly and as expected. On top of this, it shows a particular aspect of ReFI, where, depending on the state, the system has different behaviours. This not only shows that ReFI can be reproducible but also shows the great advantage of having fault injection prototypes that are reproducible. Similarly to the Unavailability experiment, described in Section 5.3.1, we use MongoDB as the underlying distributed application to reproduce this critical bug. The experiment is based on an online article [98] from Jepsen analyses [25]. However, due to the

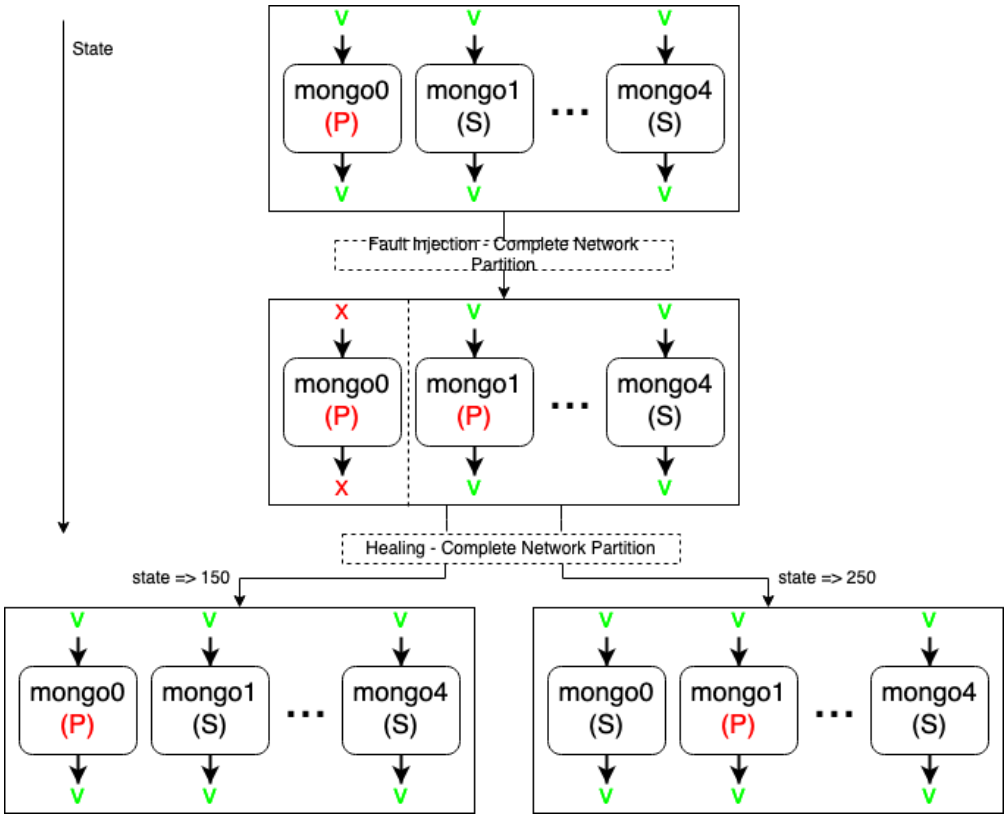


Figure 5.3: Diagram showing an overview of the MongoDB data loss bug. We have 5 nodes, a Primary (P), and four Secondaries (S). After a complete network partition, there are two available primaries to write. After healing the fault, the data is lost from one of the primaries. The primary that wins depends on the state chosen in the configuration file.

docker image repository, we use the MongoDB 2.2.7 version instead of the Jepsen analyses, which uses the MongoDB 2.4.3 version.

Overview

The experiment consists of five nodes that have a network connection between each node. These five nodes form a replica set. In this experiment, there is only one primary as expected, and secondary servers, and no arbiter servers.

Fig. 5.3 shows a diagram of the data loss MongoDB bug. The bug consists of an injection of a complete network partition fault that isolates the primary from the other nodes, creating two groups/partitions. The first group is composed of only the initial primary and the second group is composed of the other four secondary servers.

Before explaining how the bug works, it is crucial to understand that, depending on the state where we heal the complete network partition, we can show different system behaviours. The two behaviours show the initial primary losing data or the second primary losing data. This not only shows that the ReFI tool is working, but it clearly shows that it injects reproducible faults

that affect directly the outcome of a bug and how important ReFI is compared to state-of-the-art random fault injection approaches such as Chaos Engineering.

After a specific state, a complete network partition fault is injected to isolate the primary. Since the initial primary server is still up, the operations that connected clients send to it are successful and do not show any error message. In fact, if we look at the logs, we see that after writing a value and reading it, we can check that the data is, supposedly, replicated and persisted.

Meanwhile, the other partition, composed of four secondaries, start realizing that there is no primary and so they start a new leader election. Since more than a majority of servers are reachable in this partition, they elect a new primary server. In Fig. 5.3, for simplification, we identify the node 2, *mongo1*, as being the new primary but it depends on the leader election and so it could have been any secondary to become primary. This is a perfect example of the non-deterministic nature of distributed systems and the sort of hard challenges we face throughout the development of ReFI and these experiments.

In this moment of the experiment, we have two accepted primaries, an initial and a second primary, but in theory, only the second primary should be able to perform writes and reads. Because of this, both primaries have one client writing and reading from them and we start seeing two different results from reading the same document. Finally, after state 80, the initial primary realizes that he does not have a majority and it stops accepting writes and reads, but this is enough to cause damage and data loss in the application. After state 150 is met, the complete network partition is healed and we lose all the data that the client connected to the initial primary thought it was persisted, because writes were accepted, and did not report any error, and also reads showed what was expected.

Setup

For the setup of MongoDB, we once more used Docker Compose [92] and the *docker-compose.yaml* file, to automatically initiate the distributed system. For this experiment, we initiated five nodes that formed a replica set, each of them having a static IP address and a static port to facilitate the usage of ReFI.

ReFI integration

Listing 5.3, we can see that we specify five docker names as well as the fault rule. Since partitioning the primary from the other secondary servers is extremely useful for forcing a new leader election and identifying bugs in the algorithm, we can see a simplification made in the configuration file where we can specify to isolate the primary instead of describing all (N - 1)x2

```
# config/public.yaml

system : mongo
docker_amount : 5
docker_name : mongo2-4rs0-0
docker_name : mongo2-4rs0-1
docker_name : mongo2-4rs0-2
docker_name : mongo2-4rs0-3
docker_name : mongo2-4rs0-4

fault :
  - begin_state : 10
    end_state : 150
    rules :
      - isolate : primary
```

Listing 5.3: Configuration file for the data loss MongoDB bug.

(in this case eight) rules. This is also why we do not need to specify the primary index property since a script is executed to identify which of the nodes in the primary.

We only have to specify the configuration file as an argument for ReFI to parse it and the experiment executes.

Workload

For the workload, we had to choose between the MongoDB drivers. We choose Python for the following reasons: we had already made a few preliminary testing of ReFI with the Python driver, pymongo, which had more information online than Java driver; Python is also simpler than Java when it comes to create these types of scripts.

The workload is the most complex part of the experiment because we use a parallelism method to simulate several clients writing into the system and to the different nodes that are not primaries. One, unknown secondary node will eventually become a primary because we are going to inject faults into the initial primary and it will create new leader elections and a new primary will arise. For this, we had to surpass a few challenges, where is important to mention how we accomplished it.

We used three key techniques, the first is writing sequence integers into a list, this enables us to track the writes more easily and check for missing writes. Secondly, we start to write from a specific integer for each server, i.e., for node/server 0, we write 00001, 00002, 00003, and so forth, for server 1 we write 10001, 10002, and so forth for the other servers. This enables us to quickly analyze the logs and see which server is the primary. Thirdly, we purposely create clients that write into secondaries that fail, so when they become primary, they will start accepting the new writes, where the old failed writes are just not acknowledged, as we wanted.

With these three techniques, which are based on the work of Jepsen [25], we can easily prove when a bug has occurred. We simply have to look at the logs and visually see the discrepancies.

Injected Faults

From the configuration file, we deduct that we are only going to inject one fault that contains a single rule. However, this single rule is, in fact, a simplification made that contains 8 rules ((N-1)*2). So, to be more precise, we are injecting a unidirectional rule for each communication between node 1 (initial primary) with the outgoing channels.

It is important to note that the state is extremely important and has to be precise for the faults to cause a failure, in this case, a data loss bug. We can also observe that depending on the end state when we heal the complete network partition, we get two different types of bugs. One where the first primary loses data and the other where the second primary loses data. We start the fault after the system had already been going for a few states to enable the system to be fully operable before we inject faults.

Throughout the process of creating this experiment, it was expected that the state interferes with the behaviour of the system and the bug shown. These faults show the potential of this prototype and this approach to fault injection.

Bug explanation

The bug occurs because node 1, the initial primary, does not check, with heartbeats, if it has a majority, and also because the error from the secondary servers is delayed due to the secondary servers not being reachable. In this case, node 1 delays the realization that it has no majority and cannot accept writes or reads.

In the other partition, the new primary has no way to know that the old primary is still alive, and so it proceeds as expected and continues the execution of write and read operations.

This creates a great conflict when we heal the partition and both of the primaries realize that they have a different sequence of writes. In this case, the initial and oldest primary wins, and all the writes accepted from the new primary get rollback. This is a huge problem in security, affecting Mongo's reliability.

On the other hand, if we can delay the state into a specific state where the initial primary has realized that he has no majority and step down from the leader position, the new primary wins the rollback, i.e., the initial primary writes get rollback.

With this experiment, we not only can reproduce these experiments and observe their consequences but also we could better understand the algorithm and where it can be improved. Developers can more easily fix these problems in their distributed systems. Engineers can more easily architect their solutions knowing that these types of errors are common. Finally, more robust and resilient distributed systems are built.

Bug severity

The data loss bug is a critical bug since it affects the main goal of a database, which is persisting data. And the client does not even is notified or understands the problem, since it is connected to a primary but the bug happens on the rest of the distributed system.

5.3.3 Split-brain - Redis

This experiment was reproduced by ReFI and it has the goal to test the ReFI prototype and all its modules, i.e, TM, FIM, and Orchestrator. This experiment demonstrates that ReFI is compliant with its fourth goal, being **flexible** to different distributed systems, as well as, **extendable** to different syscalls.

Similarly to the Data Loss MongoDB experiment, described in Section 5.3.1, this experiment creates a complete network partition that isolates the primary from all the other nodes. The experiment is based on an online article [99] from Jepsen analyses [25]. However, due to the docker image repository and legacy changes, we use the Redis 3.0.7 version instead of the Jepsen analyses, which uses the Redis 2.6.13 version.

Overview

The experiment consists of six nodes that have a network connection between each node. Three nodes form a Redis cluster set (which is equivalent to the MongoDB replica set). On top of these three nodes, each node has a corresponding sentinel node. Note that this sentinel node is independent and isolated from the Redis instances. There is only one master, as expected. The rest are slaves and sentinels.

Fig. 5.4 shows a diagram of the split-brain Redis bug. The bug consists of an injection of a complete network partition fault that isolates the master from the other nodes, creating two groups/partitions. The first group is composed of only the initial master and the second group is composed of the other two slaves. The sentinels are not part of the cluster, so they are not considered to the partition. However, they are the ones that monitor the master and start the failover process when the network partition occurs.

This bug is similar to the MongoDB Data Loss bug described in the Section 5.3.2 but it differs that the two primaries are still up even after the healing of the partition, demonstrating not only that eventually the data of one of the primaries is lost but also the inconsistency of the distributed system with its nodes. It is even possible to write into both of the masters at the same time, making this a perfect demonstration of a split-brain bug, where there are two active brains, in this case, masters, enabling the write and read to the storage distributed system.

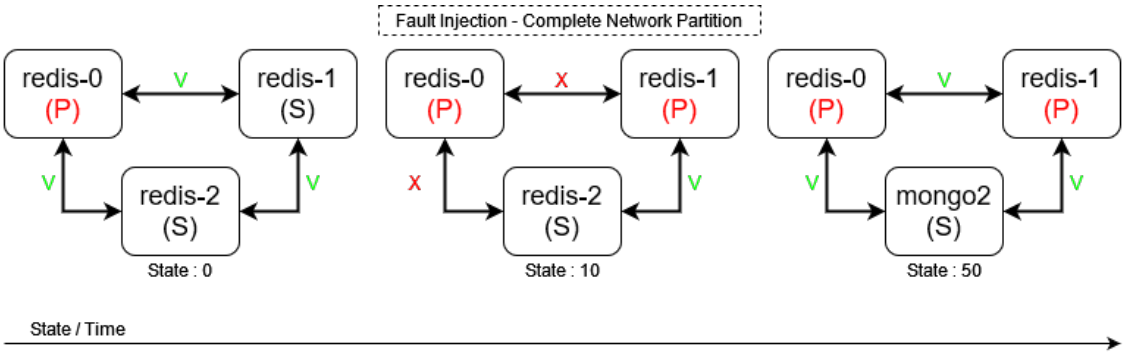


Figure 5.4: Diagram showing an overview of the Redis split-brain bug. We have 3 nodes, a Master/Primary (P), and two Secondaries/Slaves (S). After a complete network partition, there are two available primaries, creating a split-brain bug. The two primaries have a different version of the data since both can write, in parallel, without communicating with each other.

In a more descriptive and precise manner, we have a counter variable and we wait for the first master to acknowledge a few writes and reads to the counter. Then we inject a complete network partition where we isolate the master from the other four nodes. Then after this state, the sentinels realize that the master is down and they start a new leader election. After the end of the leader election state, we have a new master, that can be specified in the configuration file, by changing the priority property of the slave. Then we create a similar client that is writing to the storage system in parallel to the new/second master. By checking both reads, from the first client and from the second client, we realize two different behaviours. The first client reads are continuously changing the value and do not realize the network partition. The second client starts at a specific value and it diverges from the first. Creating a split-brain bug where there are two viable values for the same variable. Of course, we as an outside spectators, know that the second master which has a majority should be the only node to write to the counter.

Setup

For the setup of Redis, we used Docker [74] and the for each Redis instance (master, slave and sentinels) we created a configuration file, to automatically initiate each node with the appropriate properties. The main properties that we changed were the *slaveof* property for the slave nodes, *priority* property for the respective node that we wanted to be the second master, after the fault was injected, we also added a few properties to enable the cluster creation with replication data, and other minimal properties.

For this experiment, we initiated six nodes where three formed a cluster and the other three were the sentinels. We also created a docker network for them to communicate with each other. Also, the clients that write and read into the master needed to be connected to this network. The clients simply write to a counter that is incremented every 2 seconds.

```
# config/public.yaml

system : redis
docker_amount : 6
docker_name : redis-0
docker_name : redis-1
docker_name : redis-2
docker_name : sentinel-0
docker_name : sentinel-1
docker_name : sentinel-2
primary_index : 1

fault :
  - begin_state : 10
    end_state : 50
    rules :
      - isolate : primary
```

Listing 5.4: Configuration file for the split brain Redis bug.

ReFI integration

Starting from MongoDB's previous experiment, the ReFI integration took less than an hour to be completed. It consisted of two main processes: Changing how we collected the state based on the TM module and creating a configuration file for the Redis system, which was just reusing the already written configuration files for MongoDB.

In Listing 5.4, we can see that we specify six docker names as well as the fault rule. Since partitioning the primary/master from the other secondary servers is extremely useful for forcing a new leader election and identifying bugs in the algorithm, we reused the simplification made in the configuration file where we can specify to isolate the primary instead of describing all ((N-1)*2) (in this case four rules, N is three because we do not need to count the sentinels).

We only have to specify the configuration file as an argument for ReFI to parse it and the experiment executes.

Workload

For the workload, we had several API's to decide from. The main two options were Python and Go. Since we had already used Python for the MongoDB experiments, this seemed a natural choice. However, the complexity that the Python added and the lack of community help was huge limitation. So we decided to use the Go client because it was also easy to implement as well as having a helpful online community.

The workload consisted of a connection from a client to the Redis master. The master name is passed as an argument to the client Go program and then the application connects to the master node. After verifying that the connection was successful by checking for any error, the client proceeds to try to write into the master. In the first attempt, since there are no keys, it fails but the client catches this exception and creates the key with the respective item. In the

consecutive writes, everything works as expected. The client simply increments a counter key by one every two seconds and after successfully writing into the counter key, it reads it and prints it to a log for us to check if everything was correct.

After the fault is injected, a new client must be created, in parallel, to connect to the new master. We just follow the same approach by passing the new master name as an argument. The clients follow the same steps described previously.

Injected Faults

From the configuration file, we deduct that we are only going to inject one fault that contains a single rule. However, this single rule is, in fact, a simplification made that contains four rules ((N-1)*2). So, to be more precise, we are injecting a unidirectional rule for each communication between node 1 (initial master) with the on and outgoing channels.

It is important to note that the state is extremely important and has to be precise for the faults to cause a failure, in this case, a split-brain bug. We can also observe that in the end state 50 when we heal the complete network partition, we get an inconsistent value for the counter variable.

This experiment shows how simple it is for ReFI to be flexible since the faults are injected and implemented in a way that enables the use for different distributed systems.

Bug explanation

The bug occurs due to the way that the sentinel nodes work. In the Redis case, the sentinels are the key working node when it comes to availability. There is one sentinel for each Redis node monitoring their availability.

After a fault is injected at state 10, where we isolate the Redis node 0, the primary, from the other slaves, the sentinels start a failover procedure (leader election). For simplicity purposes, node 1 wins and becomes the new primary. Nevertheless, the old primary is not aware of this voting and is still up.

Since the sentinels are separated nodes, when we heal the partition, they do not check if there is already a primary. They only check if the current primary can reach a majority, which in this case is true for both. Thus, none of the primaries steps down. The real problem in this distributed system is the way that the sentinels work.

Since there are two primaries, they both accept reads and writes and have their own version, creating a split-brain. This is a huge problem in security, affecting Redis's reliability.

With this experiment, we not only can reproduce these experiments and observe their conse-

quences but also we could better understand the algorithm and where it can be improved even without looking into the code. Developers can more easily fix these problems in their distributed systems. Engineers can more easily architect their solutions knowing that these types of errors are common. Finally, more robust and resilient distributed systems are built.

Bug severity

The split-brain is a critical bug since the fault creates inconsistency in the data that the clients are reading. It would be possible that this storage system was storing a bank and a client would have more money, depending on the server that is connected. This of course would be a huge problem, not only for the bank but legally.

The real severity of these simple yet devastating bugs that engineers and developers need to be considerable of is a huge problem in today's modern applications.

5.4 Discussion

In this section we will discuss the evaluation's three key questions:

1. Is ReFI able to reproduce faults?
2. Is ReFI efficient?
3. Is ReFI able to test different distributed systems?

From the experiments done in a realistic distributed system, we can affirm that ReFI is reproducible. All the experiments also show the advantage of the reproducibility that ReFI gives. In particular, the experiment described in Section 5.3.2 shows two distinct behaviours of the distributed system that happen depending on the state that the fault is injected and healed. Chaos Engineering approach fails in this precise topic, where two runs of the fault injection approach can lead to two different behaviours.

The results obtained in the Section 5.2 were auspicious in comparison to other state-of-the-art fault injection approaches such as FCatch. The key aspects were the use of a highly efficient technology with a reliable design architecture. The design of ReFI was, from the start, planned to use eBPF and thus creating an efficient prototype.

We can see from Table 5.2 and Table 5.3, that, approximately, ReFI adds 25% to 30% more overhead to the application. Given what ReFI accomplishes, the tracing of millions of syscalls has minimal overhead. In comparison with FCatch, which imposes a 5.6x - 15.2x overhead slowdown, this is negligible.

On a more precise analysis, ReFI TM also presents a low overhead for each operation. From Table 5.4 and Table 5.5, TM an average of wanted and unwanted syscalls of 350 nanoseconds for reads and writes, 2500 nanoseconds for writes and 11000 and 10000 nanoseconds for the network submodule. In Table 5.6, we observe the results from only the wanted calls, which largely depend on the Linux kernel and their interruptions because we see a huge discrepancy from the minimum and maximum values.

Finally and crucially, The experiment described in Section 5.3.3 shows ReFI's flexibility by using it in different distributed systems. This experiment proves that ReFI can be used in other distributed systems by simply modifying a configuration file, changing how the state is managed, and creating a new workload for that system.

The experiment described in Section 5.3.1 tests the ReFI FIM module and presents a simple yet critical bug that ReFI can catch.

All bugs caught are major or critical bugs. All of the bugs require five or fewer nodes to be reproduced and two clients to write and read at the same time from the system. This represents the importance of ReFI and fault injection prototypes in the modern world and why it is essential to create more resilient and robust distributed systems.

Chapter 6

Conclusions

This chapter restates and summarizes the other chapters most essential aspects, as well as an overall view of this dissertation. It also reiterates the problems and the key solutions designed. It presents our achievements by stating the ReFI goals and how we accomplished them. Finally, we present several future work aspects and areas that can improve the ReFI tool.

The main goal of this thesis was to innovate the state-of-the-art in fault injection approaches by passing from the era of random and uncoordinated faults, used by Chaos Engineering, to a modern and reproducible era where we can test the resilience and robustness of the distributed systems via reproducible fault injection experiments. We accomplished this by creating ReFI a Reproducible Fault Injection prototype that enables developers and engineers to test their distributed applications in a reproducible manner.

ReFI, has four key goals. Create a **reproducible** fault injection framework, that treated the application as a **black-box**, had an **efficient** mechanism, and finally, to be **flexible and extendable**.

Throughout the design and implementation of ReFI, we had in consideration our key goals. To accomplish reproducible fault injection, we decided to create an Orchestrator module that coordinates the collection of data, from the TM, and the fault trigger, from the FIM. The TM collects data of the system, in a black-box manner, by analyzing syscalls, their arguments and logic. The Orchestrator not only coordinates this fault injection, making the faults based on the system state, but it also initialized the system via a configuration file. This configuration permits to specifying different distributed systems as well as different fault patterns. Using eBPF to monitor syscalls and inject faults is a novelty in the state-of-the-art. We also measured the efficiency of ReFI by measuring the overhead, throughput, and latency in two YCSB workloads and showing a negligible overhead when compared with other state-of-the-art fault injection tools.

We showed three essential experiments. Each of them has a specific goal. The first one, the MongoDB unavailability bug, was to test the FIM module since it is the critical module of ReFI. The second one, the MongoDB data loss bug, had the goal to show the reproducibility and the importance of the state that ReFI has. The third and final experiment, the Redis split-brain bug, showed that ReFI could be flexible to other distributed systems as well as extendable to different syscalls. It also showed how easy it is to use ReFI in other systems by only needing to change the configuration file and the state which is specific to each system.

6.1 Future Work

This work explores a new path to improve fault injection systems that can use eBPF technology. eBPF was known to be used for tracing and monitoring systems but with this work, we unlocked a new perspective not only for eBPF but to the community, in regards to reproducible fault injection.

However, there are still improvements to be done and we will enumerate the main ones.

1. Although ReFI is easily adjustable to test different distributed systems, there is a need to test more different distributed systems. Only by doing this, we can see improvement in the distributed systems community on reliability and resilience.

2. ReFI was built to be extendable to different syscalls. There are thousands of syscalls but finding which ones are useful is a challenging and complex task. Trying new syscalls and a combination of two or more syscalls would be a great step for the future.

3. The current workload and setup of the experiments could be even further automated. With this current ReFI version, we automated the setup of the distributed system, we automated the clients' workload by having a more realistic approach of several parallel clients, writing and reading, and we also automated the setup of ReFI using a configuration file. The improvement that could be done is to automate even further by having either a newly created module or by using already built systems, such as Kollaps [29]. In other words, ReFI could be integrated with other systems to facilitate and automate the evaluation process. This is a crucial step to facilitate ReFI's adoption because the more simple it is to use a system, the more easily developers and engineers are willing to utilize it.

4. The configuration file can be further customized by adding new features that help to precisely identify the distributed systems as well as new types of faults.

5. ReFI could test their experiments with the more recent version of the systems. This could mean finding new bugs and enabling not only the correction of these bugs but also a more automatic approach to test new systems.

6. The `isolate` property in the configuration file could be generalized for random nodes instead of a specific one (the leader). This would increase the types of faults that ReFI enables.

Bibliography

[1] "Google lost $1.7M in ad revenue during YouTube outage," last accessed on 01-01-2021. [Online]. Available: https://www.foxbusiness.com/technology/google-lost-ad-revenue-during-youtube-outage-expert

[2] "Lightning disrupts google cloud services," http://www.datacenterknowledge.com/archives/2015/08/19/lightning-strikesgoogle-data-center-disrupts-cloudservices/, last Accessed on 25-11-2020.

[3] "Admin downs entire joyent data center," http://www.theregister.co.uk/2014/05/28/joyent_cloud_down/, last Accessed on 25-11-2020.

[4] "Google Cloud Status Dashboard," last accessed on 01-01-2021. [Online]. Available: https://status.cloud.google.com/incident/zall/20013

[5] "GitHub Status - Incident History," last accessed on 01-01-2021. [Online]. Available: https://www.githubstatus.com/history

[6] "Details of the Cloudflare outage on July 2, 2019," last accessed on 01-01-2021. [Online]. Available: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/

[7] "How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play," last accessed on 01-01-2021. [Online]. Available: https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254

[8] D. Fisman, O. Kupferman, and Y. Lustig, "On verifying fault tolerance of distributed protocols," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 315–331.

[9] 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, 2008.

[10] B. George and L. Williams, "A structured experiment of test-driven development," *Information and software Technology*, vol. 46, no. 5, pp. 337–342, 2004.

[11] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 331–346.

[12] S. Dawson, F. Jahanian, and T. Mitton, "Orchestra: A fault injection environment for distributed systems," Citeseer, Tech. Rep., 1996.

[13] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 248–260, 1995.

[14] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[15] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "Fcatch: Automatically detecting time-of-fault bugs in cloud systems," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 419–431, 2018.

[16] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 677–691, 2017.

[17] J. Mohan and et al., "Finding crash-consistency bugs with bounded black-box crash testing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 33–50.

[18] R. e. a. Alagappan, "Correlated crash vulnerabilities," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 151–167.

[19] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *arXiv preprint arXiv:2003.10554*, 2020.

[20] "AWS Fault Injection Simulator – Fully managed chaos engineering service – Amazon Web Services," last accessed on 01-08-2021. [Online]. Available: https://aws.amazon.com/fis/

[21] C. R. Lorin Hochstein, "Netflix chaos monkey upgraded, netflix tech blog," https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d, Oct. 2016, last accessed on 05-11-2020.

[22] J. Jesse Robbins, K. Krishnan, J. Allspaw, and T. Limoncelli, "Resilience engineering: Learning to embrace failure, acm queue, vol. 10, iss. 9," http://queue.acm.org/detail.cfm?id=2371297, Sept. 2012, last Accessed on 05-11-2020.

[23] H. Nakama, "Inside azure search: Chaos engineering, microsoft azure blog," https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/, July 2015, last Accessed on 05-11-2020.

[24] Y. Sverdlik, "Facebook turned off entire data center to test resiliency, data center knowledge," https://www.datacenterknowledge.com/archives/2014/09/15/facebook-turned-off-entire-data-center-to-test-resiliency, Sept. 2014, last Accessed on 05-11-2020.

[25] "Jepsen website," https://jepsen.io/, last Accessed on 09-12-2020.

[26] "eBPF," last accessed on 21-12-2020. [Online]. Available: https://ebpf.io/

[27] "Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems, Held in conjunction with PODC-2021, a virtual workshop, on July 30, 2021," last accessed on 02-08-2021. [Online]. Available: http://www.cse.chalmers.se/~elad/ApPLIED2021/

[28] "ACM Symposium on Principles of Distributed Computing," last accessed on 02-08-2021. [Online]. Available: https://www.podc.org/

[29] P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: decentralized and dynamic topology emulation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[30] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[31] "Pin - A Dynamic Binary Instrumentation Tool," last accessed on 26-12-2020. [Online]. Available: https://www.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[32] "bcc," last accessed on 26-12-2020. [Online]. Available: https://github.com/iovisor/bcc

[33] "bpftrace," last accessed on 26-12-2020. [Online]. Available: https://github.com/iovisor/bpftrace

[34] "cilium," last accessed on 26-12-2020. [Online]. Available: https://cilium.io/

[35] "libbpf," last accessed on 02-08-2020. [Online]. Available: https://github.com/libbpf/libbpf

[36] "libbpf bootstrap," last accessed on 02-08-2020. [Online]. Available: https://github.com/libbpf/libbpf-bootstrap

[37] M. Amaral, M. L. Pardal, H. Mercier, and M. Matos, "Faultsee: Reproducible fault injection in distributed systems," in *2020 16th European Dependable Computing Conference (EDCC)*. IEEE, 2020, pp. 25–32.

[38] T. Esteves, F. Neves, R. Oliveira, and J. Paulo, "CAT: Content-aware Tracing and Analysis for Distributed Systems."

[39] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany, "Toward a generic fault tolerance technique for partial network partitioning," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 351–368.

[40] A. Tseitlin, "The antifragile organization," *Queue*, vol. 11, no. 6, pp. 20–26, 2013.

[41] A. Basiri, L. Hochstein, A. Thosar, and C. Rosenthal, "Chaos engineering upgraded, netflix tech blog," https://netflixtechblog.com/chaos-engineering-upgraded-878d341f15fa, Sept. 2015, last Accessed on 05-11-2020.

[42] A. Basiri, A. Blohowiak, L. Hochstein, N. Jones, C. Rosenthal, and H. Tucker, "Chap: Chaos automation platform, netflix tech blog," https://netflixtechblog.com/chap-chaos-automation-platform-53e6d528371f, July 2017, last Accessed on 05-11-2020.

[43] B. S. Kolton Andrus, Naresh Gopalani, "Fit: Failure injection testing, netflix tech blog," https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2, Oct. 2014, last Accessed on 05-11-2020.

[44] L. H. et al., "Chaos monkey github," https://github.com/netflix/chaosmonkey, last accessed on 05-11-2020.

[45] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[46] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system." in *USENIX Annual Technical Conference*, vol. 15, 1996.

[47] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[48] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, 2015, pp. 273–286.

[49] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing," https://www.usenix.org/conference/osdi18/presentation/mohan, last accessed on 20-11-2020.

[50] M. et al., "Crash monkey github," https://github.com/utsaslab/crashmonkey, last Accessed on 18-11-2020.

[51] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Correlated crash vulnerabilities," https://www.usenix.org/conference/osdi16/technical-sessions/presentation/alagappan, last Accessed on 25-11-2020.

[52] "Jespen: Redis," last accessed on 02-08-2021. [Online]. Available: https://redis.io/

[53] "MongoDB," last accessed on 26-12-2020. [Online]. Available: https://www.mongodb.com

[54] "etcd, A distributed, reliable key-value store for the most critical data of a distributed system ," last accessed on 02-08-2021. [Online]. Available: https://etcd.io/

[55] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein, "Automating failure testing research at internet scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 17–28.

[56] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *International conference on database theory*. Springer, 2001, pp. 316–330.

[57] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the lineage of view data in a warehousing environment," *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 2, pp. 179–227, 2000.

[58] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2007, pp. 31–40.

[59] G. Karvounarakis, Z. G. Ives, and V. Tannen, "Querying data provenance," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 951–962.

[60] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *Usenix annual technical conference, general track*, 2006, pp. 43–56.

[61] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, "Efficient querying and maintenance of network provenance at internet-scale," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 615–626.

[62] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[63] A. Adya and B. H. Liskov, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and ..., 1999.

[64] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 2000, pp. 67–78.

[65] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 378–393.

[66] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[67] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 595–604.

[68] F. Neves, N. Machado *et al.*, "Falcon: A practical log-based analysis tool for distributed systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 534–541.

[69] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 405–418.

[70] C. H. Suen, R. K. Ko, Y. S. Tan, P. Jagadpramana, and B. S. Lee, "S2logger: End-to-end data tracking mechanism for cloud data provenance," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 594–602.

[71] "Tensorflow," last accessed on 02-08-2021. [Online]. Available: https://www.tensorflow.org/

[72] "Apache Hadoop," last accessed on 02-08-2021. [Online]. Available: https://hadoop.apache.org/

[73] "PostgreSQL," last accessed on 26-12-2020. [Online]. Available: https://www.postgresql.org/

[74] "Empowering App Development for Developers | Docker," last Accessed on 31-12-2020. [Online]. Available: https://www.docker.com/

[75] "Amazon Web Services (AWS) - Cloud Computing Services," last Accessed on 31-12-2020. [Online]. Available: https://aws.amazon.com/

[76] "Elasticsearch," last accessed on 02-08-2021. [Online]. Available: https://www.elastic.co/products/elasticsearch

[77] "Rabbitmq message broker," last accessed on 02-08-2021. [Online]. Available: https://www.rabbitmq.com

[78] "Apache hbase," last accessed on 02-08-2021. [Online]. Available: https://hbase.apache.org/

[79] "Apache mesos," last accessed on 02-08-2021. [Online]. Available: http://mesos.apache.org/

[80] "The ceph object store," last accessed on 02-08-2021. [Online]. Available: https://ceph.io/

[81] "Moosefs: Distributed file system," last accessed on 02-08-2021. [Online]. Available: https://moosefs.com/

[82] "Kafka: A distributed streaming platform," last accessed on 02-08-2021. [Online]. Available: https://kafka.apache.org/

[83] "Activemq: Flexible & powerful open source multiprotocol messaging," last accessed on 02-08-2021. [Online]. Available: http://activemq.apache.org/

[84] "Dkron: A distributed cron service," last accessed on 02-08-2021. [Online]. Available: https://dkron.io/

[85] "IOvisor," last accessed on 02-08-2021. [Online]. Available: https://github.com/iovisor

[86] "Andrii Nakryiko's blog," last accessed on 02-08-2021. [Online]. Available: https://nakryiko.com/

[87] "BPF ring buffer," last accessed on 02-08-2021. [Online]. Available: https://nakryiko.com/posts/bpf-ringbuf/

[88] "XDP actions," last accessed on 02-08-2021. [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html

[89] "Fault Injection method: bpf_override_return," last accessed on 02-08-2021. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#1-bpf_override_return

[90] "BCC Reference Guide," last accessed on 02-08-2021. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md

[91] "Subset of syscalls that enable override of the return." last accessed on 02-08-2021. [Online]. Available: https://man7.org/linux/man-pages/man2/syscalls.2.html

[92] "Define and run multi-container docker applications | docker compose," last Accessed on 02-08-2021. [Online]. Available: https://docs.docker.com/compose/

[93] "Javassist by jboss-javassist," last accessed on 02-08-2021. [Online]. Available: https://www.javassist.org/

[94] "Javassist by jboss-javassist," last accessed on 02-08-2021. [Online]. Available: https://en.pingcap.com/blog/why-we-switched-from-bcc-to-libbpf-for-linux-bpf-performance-analysis

[95] "YCSB," last accessed on 02-08-2021. [Online]. Available: https://github.com/brianfrankcooper/YCSB/

[96] "Ubuntu Manpage: bpftool-prog," last accessed on 26-12-2020. [Online]. Available: https://manpages.ubuntu.com/manpages/focal/en/man8/bpftool-prog.8.html

[97] "Arbiters in pv1 should vote no in elections if they can see a healthy primary of equal or greater priority to the candidate." last accessed on 02-08-2021. [Online]. Available: https://jira.mongodb.org/browse/SERVER-27125

[98] "Jepsen: MongoDB 2.4.3," last accessed on 02-08-2021. [Online]. Available: https://aphyr.com/posts/284-call-me-maybe-mongodb

[99] "Jepsen: MongoDB 2.4.3," last accessed on 02-08-2021. [Online]. Available: https://aphyr.com/posts/283-call-me-maybe-redis