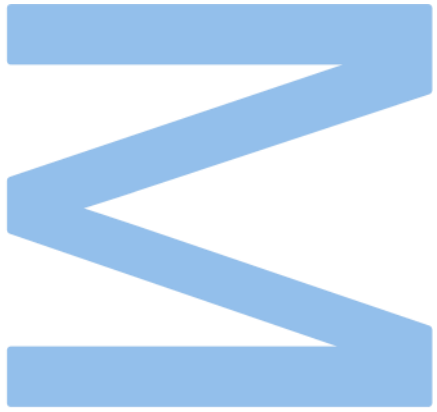
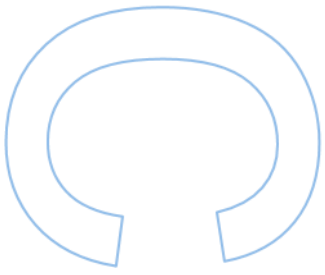
Design of a Flexible and Extensible Fault Injector for Testing Concurrent and Distributed Applications



Pedro Fernando Moreira da Silva Antunes
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2022

Supervisor
João Paulo da Conceição Soares, Assistant Professor, Faculdade de Ciências da Universidade do Porto

Co-supervisor
Rolando da Silva Martins, Assistant Professor, Faculdade de Ciências da Universidade do Porto




U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Sworn Statement

I, Pedro Fernando Moreira da Silva Antunes, enrolled in the Master Degree Information Security at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this dissertation reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this dissertation, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This dissertation does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Pedro Antunes

September 30, 2022

Abstract

The applications developed today are increasingly distributed, which contributes to an increase in their complexity. The imminence of failures in these applications is natural and ensuring the resilience of this software requires the development of fault tolerance techniques. However, there are not many tools capable of assessing the fault tolerance properties of these applications. The existing tools are designed for specific applications.

In this document we will present a latest version of Zermia, a fault injector that can be used by any application developed with any programming language. It is based on the work developed by the previous injectors Hermes, Proteus and Zermia, all specialized in the fault tolerance protocol for distributed systems BFT-SMaRT. Our version generalizes and extends the usability of the tool to any fault tolerance protocols or any type of application.

For fault injection we use an aspect-oriented programming approach that decorates a given application function with another function that contains the faulty code. This mechanism makes it possible to assess an application without changing its source code. However, each language has its own mechanism for aspect-oriented programming, and in some cases, there are minimal changes that must be made to the application's source code.

The faults communicate with a client-server architecture that monitors and coordinates the injection of the faults into the application. Faults can have multiple trigger conditions, either application dependent or application independent, as well as being dependent on the injection of other faults. The user can create custom faults without having to deal with the complexity that comes from our tool.

Maximizing application performance was the focus of our design. The fault triggering has the flexibility to be independent and can trigger faults without interactions with our architecture, which maximizes application performance. However, we designed the tool to decide when it needs to perform communications between the architecture components, which minimizes the number of communications to decide on the injection of a fault.

Throughout this document we discuss the architecture concepts, explain the implementation and usage details, and present injection experiments on two real applications from different programming languages. The first application was developed in Python and is a single-process application. The second application was developed in Java and is an implementation of a RAFT

fault tolerance protocol that can be used in the implementation of distributed systems.

Resumo

As aplicações desenvolvidas nos dias de hoje são cada vez mais distribuídas, o que contribuí para um aumento de complexidade das mesmas. A iminência de falhas nestas aplicações é natural e garantir a resiliência deste software requer o desenvolvimento de técnicas de tolerância a falhas. Contudo, não existem muitas ferramentas capazes de testar as propriedades de tolerância a falhas destas aplicações. As ferramentas existentes foram desenhadas para aplicações específicas.

Neste documento vamos apresentar um nova versão do Zermia, um injetor de falhas que pode ser utilizado por qualquer aplicação desenvolvida em qualquer linguagem. Baseia-se no trabalho desenvolvido dos injetores precedentes Hermes, Proteus e Zermia, todos especializados no protocolo de tolerância a falhas em sistemas distribuídos BFT-SMaRT. A nossa versão generaliza e estende a utilização da ferramenta a várias protocolos de tolerância de falhas ou qualquer tipo de aplicação.

Para a injeção das falhas utilizamos uma abordagem de programação orientada a aspetos que decora uma determinada função aplicacional com outra função que contém o código faltoso. Este mecanismo torna possível testar uma aplicação sem alterar o seu código fonte. No entanto, cada linguagem tem o seu próprio mecanismo de programação orientada a aspetos, e em alguns casos existem alterações mínimas que têm de ser feitas ao código fonte da aplicação.

As falhas comunicam com uma arquitetura cliente servidor que monitoriza e coordena a injeção das falhas na aplicação. As falhas podem ter várias condições de acionamento, dependentes ou independentes da aplicação, como também podem depender da injeção de outras falhas. O utilizador pode criar falhas personalizadas sem ter que lidar com a complexidade proveniente da nossa ferramenta.

Maximizar o desempenho aplicacional foi o principal foco do nosso projeto. O acionamento das falhas tem flexibilidade para ser independente e poder acionar falhas sem interações com a nossa arquitetura, o que maximiza o desempenho aplicacional. No entanto, a ferramenta foi desenvolvida para decidir quando é que precisa de efetuar comunicações entre os componentes da arquitetura, o que minimiza o número de comunicações para decidir a injeção de uma falha.

Ao longo deste documento discutimos as ideias da arquitetura, explicamos os detalhes de implementação e de utilização e apresentamos experiências de injeção em duas aplicações reais e de diferentes linguagens de programação. A primeira aplicação foi desenvolvida em Python

e é uma aplicação de um processo único. A segunda aplicação foi desenvolvida em Java e é uma implementação de um protocolo de tolerância a falhas RAFT que pode ser utilizado na implementação de sistemas distribuídos.

Acknowledgments

I want to express my sincere gratitude to who never stopped supporting me during the development of my dissertation.

I thank the thesis supervisor Professor João Soares for having supported me, for helping me solve the problems that emerged and for sharing his knowledge with me.

I also thank all my friends for having motivated me during this journey and for helping me to overcome all the difficulties.

Finally, and the most important thanks, goes to my parents and other family members. My parents were the biggest pillars for everything to be possible, and to them I want to dedicate all my academic path and all the success that will follow, for believing in me, for the strength and love. To the other family members for always wishing the best for me and for always being able to take the best from me.

"Remember to look up at the stars and not down at your feet"

Stephen Hawking

Contents

Abstract i

Resumo iii

Acknowledgments v

Contents x

List of Tables xi

List of Figures xiv

Listings xv

Acronyms xvii

1 Introduction 1

1.1 Motivation 1

1.2 Proposed Solution 2

1.3 Contribution 2

1.4 Organization 3

2 Background and Related Work 5

2.1 The Two Generals Problem 5

2.2 The Byzantine Generals Problem 6

2.3 Distributed Systems Models 7

2.3.1 Network behavior model . . . 8
2.3.2 Node behavior model . . . 8
2.3.3 Timing behavior model . . . 9
2.4 Fault Tolerance . . . 9
2.4.1 Replication . . . 10
2.4.2 Consensus . . . 11
2.4.3 Crash Fault Tolerance . . . 12
2.4.4 Byzantine Fault Tolerance . . . 15
2.5 Fault Injection . . . 19
2.5.1 Xception . . . 20
2.5.2 FERRARI . . . 21
2.5.3 FIAT . . . 21
2.5.4 Ftape . . . 21
2.5.5 DOCTOR . . . 22
2.5.6 PROPANE . . . 22
2.5.7 Loki . . . 23
2.5.8 J-SWFIT . . . 23
2.5.9 LLFI . . . 24
2.5.10 Discussion . . . 24

3 Prior Work **25**

3.1 Aspect Oriented Programming . . . 25
3.2 gRPC . . . 27
3.3 Hermes . . . 28
3.4 Zermia . . . 29
3.5 Proteus . . . 31

4 Design **33**

4.1 Zermia Design . . . 33

4.1.1 Server . . . 34
4.1.2 Client . . . 35
4.2 Faults . . . 35
4.3 Schedulers . . . 36
4.4 Triggers . . . 37
4.4.1 Coordinator-based Triggering Model . . . 38
4.4.2 Agent-based Triggering Model . . . 38
4.4.3 Hybrid Triggering Model . . . 38
4.4.4 Independent Triggering Model . . . 39
4.5 Determinism . . . 39
4.6 Discussion . . . 39

5 Implementation details **41**
5.1 Coordinator . . . 41
5.1.1 gRPC service . . . 45
5.2 Agent . . . 47
5.2.1 gRPC service . . . 48
5.3 Hook . . . 48
5.3.1 Faults . . . 50
5.3.2 Predefined Faults and Extensibility . . . 51
5.3.3 Java implementation . . . 51
5.3.4 Python implementation . . . 54
5.4 How to assess the behavior of Fault Tolerance protocols . . . 56
5.5 Script setup for compiling and running . . . 57
5.5.1 Configuration file . . . 58

6 Experiments **63**
6.1 Simple Python application testing . . . 63
6.1.1 Experimental setup . . . 63

6.1.2 Baseline . . . 64
6.1.3 Delay Fault . . . 64
6.1.4 Range Modification Fault . . . 65
6.1.5 Delay and Range Modification Faults . . . 65
6.2 RAFT testing . . . 66
6.2.1 Experimental Configuration . . . 66
6.2.2 Baseline results . . . 66
6.2.3 Crash Fault . . . 67
6.2.4 Delay Fault . . . 69
6.2.5 Delaying the Leader Election . . . 70
6.2.6 Delaying Heartbeats . . . 70

7 Conclusion **73**

7.1 Future Work . . . 74

Bibliography **75**

List of Tables

6.1 Baseline results for first leader elected after x milliseconds. 70

6.2 Average results obtained after injected delays. In each "Injected delay", six tests were performed, and the result of the replicas are in milliseconds. 71

6.3 The total amount of candidates that were established leaders after injecting delays. Each column represents a replica and the number of times it got elected leader. . 71

6.4 The average results obtained from Table 6.3 that also represent the number of candidates that became leaders. 72

List of Figures

2.1 The two generals' problem[1] . . . 5

2.2 Scenarios of communications lost . . . 6

2.3 The Byzantine Generals Problem[1] . . . 6

2.4 Scenarios of malicious intentions . . . 7

2.5 Paxos agreement mechanism using three nodes with all roles and supporting one node crash . . . 13

2.6 Raft server states[2] . . . 14

2.7 Pratical Byzantine Fault Tolerance (PBFT) normal case operation using four replicas and one faulty node[3] . . . 16

2.8 PBFT view change using four replicas node[4] . . . 17

2.9 BFT-SMART normal phase message pattern[5] . . . 18

2.10 BFT-SMART synchronization phase message pattern[6] . . . 19

3.1 gRPC overview[7] . . . 27

3.2 Hermes's architecture[8] . . . 29

3.3 Zermia architecture[9] . . . 30

3.4 Fault schedule example through command line [9] . . . 30

3.5 Proteus architecture [10] . . . 31

4.1 Overview of the latest Zermia architecture . . . 34

4.2 Zermia's fault triggering models . . . 37

5.1 Zermia framework hooked in application . . . 49

5.2 Fault abstraction 50

6.1 Simple Python application experiments 64

6.2 Performance obtained by 1 million requests 66

6.3 Performance obtained by 1 million requests by injecting a crash at specific points (Cluster of four nodes (C4) configuration) 67

6.4 Performance obtained by 1 million requests by injecting a crash at specific points (Cluster of six nodes (C6) configuration) 68

6.5 Results obtained through C4 69

Listings

3.1 Simple example of an aspect implementation using AspectJ 27
3.2 Service declaration from proto files[7] 28
5.1 Running the Coordinator on port 9090, using 4 agents and the settings for the fault schedules in the agents_config.json file 41
5.2 An Agent Fault Schedule Configuration File Example 42
5.3 Widespread configurations example 44
5.4 Coordinator service interface (gRPC methods) 45
5.5 Protocol Buffers messages for AgentConnectionReply 45
5.6 Protocol Buffers messages for dependencies validation 46
5.7 Protocol Buffers messages for fault execution notifications 46
5.8 Running Agent on port 9000, defining the IP and port to listen on by the Coordinator 47
5.9 Running Agent on port 9000, with fault schedule settings in the agents_config.json file and assuming an Agent-based Triggering Model 47
5.10 Agent configuration file example agents_config.json 47
5.11 Agent service interface (gRPC methods) 48
5.12 Crash Fault Java implementing following the example shown in Listing 5.10 . . . 52
5.13 Bootstrap Fault Java example 53
5.14 Agent configuration file example agents_config.json 54
5.15 Zermia framework compile command for Java 54
5.16 Crash Fault Python implementing following the example shown in Listing 5.10 . 54
5.17 Associating a Crash Fault to a target method in Python 55
5.18 Bootstrap Fault Python example 56
5.19 Associating the Boostrap Fault to the main method in Python 56
5.20 Running experiences from setup Zermia script 57
5.21 Script configuration file for Python applications 60
5.22 Script configuration file for Java applications 61

Acronyms

BFT	Byzantine Fault Tolerance
CFT	Crash Fault Tolerance
PBFT	Pratical Byzantine Fault Tolerance
RPC	Remote Procedure Call
AOP	Aspect Oriented Programming
TLS	Transport Layer Security
FLP	Fischer, Lynch and Paterson theorem
FI	Fault Injection tools
SFI	Software Fault Injection
CPU	Central Process Unit
EGM	Experiment Generation Module
ECM	Experiment Control Module
FIA	Fault Injection Agent
DCM	Data Collection Module
PSC	PROPANE Setup Creater
PCD	PROPANE Campaign Driver
PL	PROPANE Library
PDE	PROPANE Data Extractor
GUI	Graphical User Interface
LLVM	Low Level Virtual Machine
EDC	Egregous Data Corruptions
DSL	Domain Specific Language
JAR	Java ARchive
JVM	Java Virtual Machine
DDoS	Distributed Denial of Service
JSON	JavaScript Object Notation
OOP	Object Oriented Programming
JDK	Java Development Kit
RAM	Random Access Memory
YCSB	Yahoo! Cloud Serving Benchmark
C4	Cluster of four nodes
C6	Cluster of six nodes
SSH	Secure Shell
IP	Internet Protocol
UML	Unified Modeling Language
NFS	Network File System

Chapter 1

Introduction

Facing the new industrial revolution that accompanies the constant innovations and evolution of technology, computing power plays a key role in the services provided by organizations today. Large-scale applications use the decentralization of resources and services to increase computational power. However, this contributes to an increase in their complexity.

Distributed systems are distinguished by their better dependability. These systems need to have resilience to different communication problems, for example latency and message corruption. This requires applications to be assessed and validated in realistic scenarios, which include faults. Even if one node fails or one communication fails, then there is the possibility that the remaining computers will be able to perform the work intended for the failing node. Thus, it allows the system to behave correctly and would still be operational even when some node is not cooperating. That said, a distributed system is a system that is known to be fault tolerant.

1.1 Motivation

Achieving the fault tolerance property is one of the great challenges of distributed systems. How can system perform operations in such a way that it can tolerate faults? Over the years, the academic community has been implementing fault tolerance algorithms for distributed systems, for example Byzantine Fault Tolerance (BFT)[3, 5, 11, 12] and Crash Fault Tolerance (CFT)[2, 13]. However, these algorithms have complex implementations that makes it difficult to rely on the security properties to adopt them in real application scenarios. To assess the security properties of these protocols, it is used fault injection tools that can emulate specific practical attack scenarios, and thus evaluate whether the system behaves correctly as expected or, on the other hand, whether the security properties can be violated and consequently compromise the system.

However, we consider that the existing fault injection tools are quite dependent on the application context, because too often their design is specific for test one application. They end up missing the necessary features to successfully assess the properties that a distributed system demands, or they are exclusively designed to assess a protocol and/or are specific to a

programming language. Therefore, we consider that no extensible tools are available that can be adapted in a generic way to fault tolerance protocols.

Our research work is based on the implementation of a fault injection tool that is extensible to any fault tolerance protocol, to any language and that allows the user to have freedom to customize the tests they want to run.

1.2 Proposed Solution

In this dissertation, we present a latest version of Zermia, a fault injection tool that improves on previously presented solutions (Hermes[8] and Zermia[9]) with a focus on flexibility and extensibility.

This latest version of Zermia uses gRPC technology to make Remote Procedure Call (RPC) between the several components of the tool and Aspect Oriented Programming (AOP) paradigm, used in the previous versions, as a fault injection mechanism through aspects. However, the previous versions are specific for the BFT-SMaRt protocol, which makes it quite difficult to adapt the implementation to other types of fault tolerance protocols.

The improvements made in our research involve changing the tool design to make it more easily portable for testing other fault tolerance protocols and to make it usable in more programming languages, by making this portability happen with the implementation of as little code as possible. We also introduced deterministic mechanisms to implement fault dependencies within a single node or across multiple nodes of the target system, which allow the creation of a synchronization phase between a set of faults to achieve a coordinated attack that comes from different nodes and different causes. To make this possible, we create abstraction mechanisms regarding the faults implementation that need to be developed on the user's side, allowing the customization of faults which is independent of Zermia's structure, so that the user does not have to worry about the faults synchronization when developing the faulty code.

Also, our solution can leverage the work previously developed in the Proteus[10] injector to automatically generate the code that the user would have to develop.

1.3 Contribution

The contributions made by this research were as follows:

- Identifying the Zermia older versions limitations.
- Modifying the design to increase the flexibility and extensibility of the fault injection tool.
- Development of deterministic mechanisms to synchronize and coordinate fault injection.

- Development of fault abstraction mechanisms that allow faults customization by the user.
- Building an independent client-side component specifically for fault injection.
- Independent client-side component implementation in Java and Python languages.
- Building a script for running test experiments through configuration files.

1.4 Organization

The following structure is used in this document. Chapter 2 presents and discusses the state of the art and related work. It describes models and problems in implementing distributed systems, explores replication and consensus mechanisms that are used in the implementation of fault tolerance protocols, and finally discusses some existing fault injection tools. Chapter 3 presents the prior research work that serves as a background to our research, as well as the technologies they use to implement fault injection mechanisms. Chapter 4 we presented our design for a latest version of Zermia, the responsibilities of each component of our tool, and a trade-offs discussion that led to the final decision to change Zermia's architecture. Chapter 5, discusses the implementation details of the tool, presents fault development in Java and Python languages, and finally presents a script for automatic local execution experiment in Java applications and Python applications. Chapter 6 we evaluate the effectiveness of the tool by testing different applications. We demonstrate the use of the tool by evaluating a single-process application developed in Python. Finally, we adapted some tests done in past Zermia investigations and analyzed the efficiency and robustness of the Raft protocol in a Java implementation. We follow this in Chapter 7, analyzing the results and describing the future work that can be done.

Chapter 2

Background and Related Work

In this section we describe two faulty experiences of classical thoughts, the most relevant approaches to troubleshoot and ensure the operation and security of a distributed system, present certain fault-tolerant protocols, and discuss some fault injection tools that allow emulation of communication and component faults to assess the resilience of systems security assumptions.

2.1 The Two Generals Problem

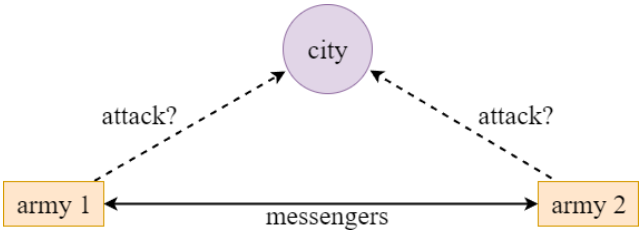
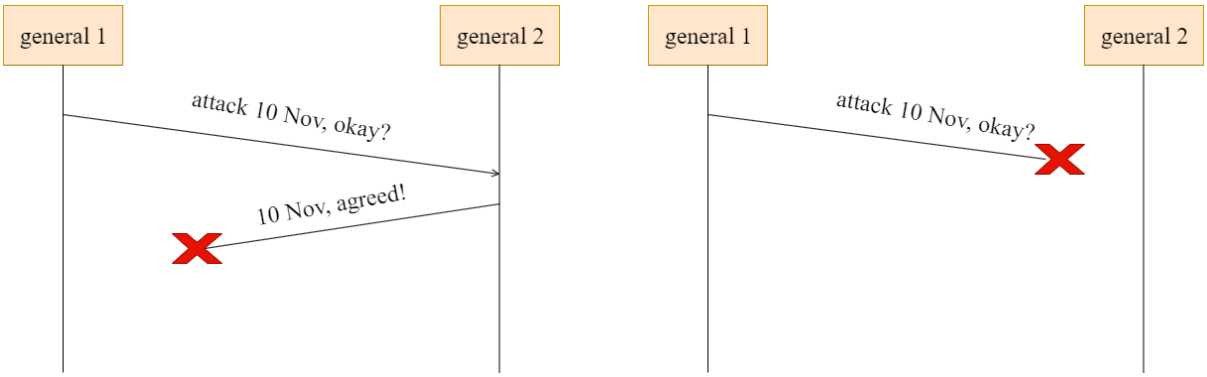


Figure 2.1: The two generals' problem[1]

The faulty experience of the two generals problem presents in Figure 2.1, where we have two armies and each controlled by a general, the goal is to attack and capture a city. If the armies cannot do a combined attack, that is, if the armies attack one at a time, then the city's defenses will defeat them. Therefore, if both intend to attack, it is important that they attack at the same time, so they know they will win. To coordinate attacks, the generals exchange messages but when a message is sent from one general to another, that message may or may not reach its destination.

As illustrated in Figure 2.2a, General 1 decides to attack on November 10 and communicates his intention to General 2 asking him if he agrees. General 2 receives the message, agrees, and responds affirmatively to General 1's plan. Unfortunately, General 1 will not receive a reply.

In Figure 2.2b we see another scenario of what can happen, General 1 decides to attack on November 10, but the initial message is lost and will never reach to General 2. Now, general 2



(a) Missing response on a date plan[1]

(b) Date plan message lost[1]

Figure 2.2: Scenarios of communications lost

will not receive any message and therefore will not send any response.

In both cases, what General 1 observes is that there is no response to his attack intention. That said, this problem lies in what General 1 should do. However much the odds increase by sending countless messages so that some get to their destination, it is in fact impossible to solve this problem. In distributed systems, this problem is called no common knowledge[14]. However arbitrary chain confirmation messages may be constructed, the result will always be the uncertainty that we will never know if the messages arrived correctly at the destination.

2.2 The Byzantine Generals Problem

There is also the faulty experience of The Byzantine Generals Problem[15], which we can see in Figure 2.3. Like The Two Generals Problem, we also have armies, generals who command armies and who want to attack a city simultaneously and who communicate through messages for this purpose. However, in this problem we have three armies or more.

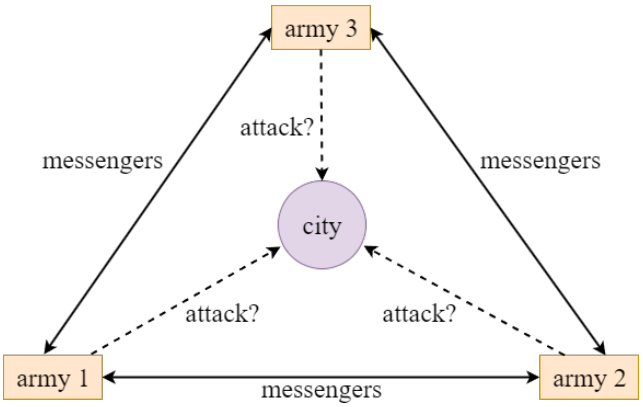
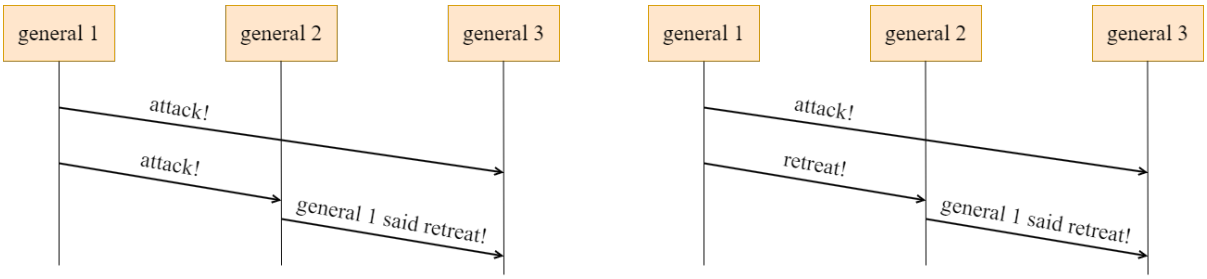


Figure 2.3: The Byzantine Generals Problem[1]

TThis problem is easier on one hand because it is assumed that the communications are dependable. Therefore, any message sent will be received. However, the problem becomes more

difficult when we assume that generals can be traitors. The messages can be corrupted or generals can send different messages to each of them, which could compromise the honest generals actions. Still, what is intended is that the honest generals come to an agreement on the city attack.

In Figure 2.4a example, trivially, we note that General 2's behavior is malicious because he is claiming that General 1 sent him a message to retreat when in fact, General 1 sent a message to attack. Unfortunately, from General 3's perspective, it is not trivial to identify what is happening. In Figure 2.4b, we have another example of different messages exchanged by General 1 and General 2, and the General 3 perspective is exactly the same as in Figure 2.4a.



(a) General 2 acting with malicious intentions[1] (b) General 1 acting with malicious intentions[1]

Figure 2.4: Scenarios of malicious intentions

Leslie Lamport et al[15] argue that up to f malicious generals in a total of n generals, where the honest generals don't know which ones are the malicious generals and the malicious generals can work in coordination to fool the honest generals, it takes $n \geq 3f + 1$ generals to tolerate f malicious generals and to make sure that the honest generals can agree on a plan. In other words, the number of malicious generals must be less than $\frac{1}{3}$). To tolerate a system with one malicious general, we need at least four generals.

2.3 Distributed Systems Models

Through the experiments above, we have been able to obtain a perception of some problem types that can emerge when we talk about distributed systems. In the problem of the two generals, we can observe the network behavior model and in the problem of the Byzantine generals we observe the nodes behavior model. To design secure distributed systems means designing a system where we assume that both network and nodes can do things wrong in any diverse ways[1, 16]. Any distributed system algorithm needs to have assumptions of system behavior properties, such as:

- Network behavior
- Node behavior
- Timing behavior

2.3.1 Network behavior model

Typically, these systems have a bidirectional peer-to-peer communication. That is, any node in a link between two nodes, can play the role of sender or receiver. Thus, we must create assumptions to determine how dependable these links are.

The simplest model to develop is when we assume that are perfectly reliable links. That is, a link does not lose messages during communications. Realistically, this is a strong assumption, since in practice connections are not dependable to this extent.

Another assumption we can make is to assume fair-loss links. That is, messages can get lost, can be reordered, or even duplicated. However, if there are an infinite number of sending attempts the message will eventually be delivered to its destination, but it is impossible to determine how long it takes to be delivered.

The weakest assumption we can make is to assume arbitrary links where we assume the presence of an adversary in the link. The actions that the malicious actor can perform are intercept, modify, drop, spoof, or replay messages.

With awareness of these three network behavior models, it is possible to convert one model into another. Assuming a fair-loss link, it can be converted into a reliable link if we assume that we will constantly try to send the message until the receiver receives it. It is also feasible to convert an arbitrary link into a fair-loss link if we use a cryptographic protocol in the communication, such as Transport Layer Security (TLS). By implementing this protocol, we would have a tamper-resistant communication. After the TLS established communication links, all traffic is encrypted, which results in resilience against interference from external actors.

2.3.2 Node behavior model

Nodes can also fail in diverse ways. The first type of fault we can consider is a crash fault. Abstractly, in a crash-stop we are assuming that a process can crash at any time, and once it crashes it will never execute again.

A second potential assumption we can make is to assume that after the crash, the node can eventually recover execution after some period. Thus, any information kept in ephemeral memory is lost. However, information that is kept in stable memory (such as disk storage) is preserved after the crash.

The last behavior model we can assume for nodes is the Byzantine model. As we saw earlier in the Byzantine general's problem, we assume that a node is faulty if it deviates from the correct behavior defined in its algorithm. Without constraining its behavior, a faulty node can take any kind of action to try to tamper the system state.

2.3.3 Timing behavior model

After we have assumed the node and network behavior models, to be able to detect failures we need to assume time synchronization aspects for both the nodes and the network.

In a synchronous system model, it is assumed that after a message is sent, there is a maximum message delay boundary that will define whether the message has been delivered or has been lost in the network.It is also assumed that the nodes follow the algorithm by executing each instruction with a well-known speed, and thus there is a maximum time bound for them to succeed in the algorithm execution.

Another assumption that can be do, is a partially synchronous model[17], where in some finite time periods the system may experience asynchronous states (i.e., periods where there are no upper time bounds), and in different periods it may experience synchronous states.

Finally, in an asynchronous system model there are no time guarantees. That is, we did not assume time constraints for either the message delays or the processing execution. In both cases, there is the possibility of extensive time delays.

The timing model is the most difficult assumption to define in a distributed system. The asynchronous model is good because we do not have to set time bounds, but there are types of problems that cannot be resolve without assuming these time bounds. And even then, it is dangerous to assume that the time bounds will always be met. Too many vulnerabilities expose the security of distributed systems due to these time constants implemented in each algorithm. The partially synchronous model tries to perform a kind of compromise between the synchronous and asynchronous model to sensitize and smooth out these timing problems.

2.4 Fault Tolerance

In distributed systems, there is a differentiation between failure and fault. When a certain system node is not working as intended, we have a fault[18]. The faults can happen for several reasons and relative to several node components or to the network. When the system as a whole is not working, we have a failure.

That said, if we allow parts of the system to fail momentarily, then paradoxically we increase the system resiliency by avoiding a Single Point of Failure (SPOF), such as a node or a network link whose fault leads to failure and the system becomes unavailable. This is one of the main reasons for using state machine replication mechanisms. Through replication[19], we can obtain system information copies by multiple nodes. One replica (i.e., node that contains a data copy) can be faulty, but we still have the necessary system data accessible and available on the remaining replicas.

Nevertheless, fault tolerance protocols should implement fault detection algorithms[20]. The method for detecting and treating faults is through heartbeat timeouts. Replicas send periodically

messages to each node, and if there is no response from a certain replica during the stipulated time, then the node is faulty labeled. However, this way of detecting faults through timeouts, does not define the type of fault present in the system if we consider partially synchronous or asynchronous systems. It becomes impossible for the sender to know why his message did not have any reply. For example, the timeout response may fail because messages lost or delayed in the network, or because the node simply crashed.

Consensus algorithms focus on trying to detect the types of faults that may occur. Through timing mechanisms, these algorithms use fault detection and total order broadcast messages mechanisms where, with the information from all nodes, they try to reach a general conclusion about what may have happened in the possible presence of a fault.

2.4.1 Replication

Replication is a technique that provides a system dependability by creating data redundancy across multiple nodes[18]. If one node is unavailable, replication provides fault tolerance, and the information is accessible on the remaining nodes.

Another reason for implementing replication is load balancing. If a system carries too many clients, from various places around the world, and all requesting the same information content, it will lead to an overload for just a single node. Having the same information content on multiple nodes allows for efficient load balancing and makes the system able to manage more requests from more clients, which is important in large scale services.

Therefore, replicating static information is trivial. However, replicating information that is constantly having updates to multiple nodes becomes much harder and requires incredibly careful processing to ensure the data integrity. To perform dynamic data updates across all replicas it is necessary to deal with request deduplication and solve inconsistency problems that may arise[21].

The deduplication scenario[22] occurs when you send a message to a node with updated data and for whatever reason the acknowledgment reply does not reach the sender. Without this reply, the sender sends the data update again and so we may be updating already updated data, which can lead to unrealistic data states. To solve these problems timestamp sorting and broadcast algorithms mechanisms can help to manage request deduplication.

All this makes replication a strong mechanism for security, availability, and data protection. However, as we saw above, it is necessary to tolerate certain faults and to solve concurrent problems[23] that can lead to inconsistent data states in the set of replicas. Several algorithms have been adopted to solve these issues, namely consensus algorithms[24], which are nothing more than algorithms for sending and receiving messages to all nodes belonging to the system (broadcast messages) to implement state updates in all replicas and thus allow the full operation and security of a replicated/distributed system.

2.4.2 Consensus

In a state machine replication system, where replicate servers allow fault tolerance[19], its operation is based on the interaction and decisions made by its set of nodes, it is necessary to create a communication procedure where the system reaches an agreement[25] regarding the system data derived from the actions performed by the clients. This procedure must ensure reliability in the nodes network, the liveness of the system, the system information consistency, and the availability in response to requests from the system's clients. The combination of these aspects is what characterizes a consensus algorithm in distributed systems and is one of the most critical parts of design and implementation of such system. Because it involves too many components, because it deals with conflicting concurrent situations, the coordination of their states increases implementation complexity, and it becomes quite difficult to assess and ensure the system's correct operation.

One consensus implementation problem is to secure message propagation order. In certain conflicting concurrent moments, where a node may receive several messages coming from different nodes, and each message requires a new update on the same information element, if the messages do not follow a well-defined order, some updates may be lost by rewriting updates on the same element state.

To solve this issue, we use total order broadcast algorithms[26] which can either choose symmetric or asymmetric approaches. The symmetrical approach is where any replica node server is allowed to respond to client requests, then the request is propagated to the remaining nodes and they must reach a consensus value so that the receiver node can then reply to the client. The assymetrical approach is where a leader or a set of leaders from all the nodes is allowed to respond to requests. One node is referred as the leader and whenever any node wants to send any message, that message is sent back to the leader and the leader relays it to all system nodes. So when the leader/receiver node propagates the messages, trivially it can be assumed that as it relays the messages it is creating an order since it is not possible to propagate all the messages it receives in the same time frame.

The consensus protocol also is responsible for ensuring data consistency. Because of network problems that can occur and conflicts in concurrent data accesses, these protocols must be able to have mechanisms to ensure data integrity. To solve inconsistencies, these protocols implement atomic state transactions. Through atomic commitment protocols[27], a data update is effective when all nodes or a quorum of nodes have committed the update. Thus, whenever a node is ready to store data, it notifies the others. When a particular node receives notifications from all nodes, or depending on specifications, notifications from a quorum of nodes, it finalizes the state change by notifying system nodes that they can commit their new state. This procedure makes sure that updates do not happen instantaneously on each node but in a coordinated way.

As we saw above, consensus protocols implement mechanisms that require coordination. All nodes must reach a certain conclusion or agreement to ensure safety properties. Furthermore, any kind of coordination must respect time constraints to ensure decision termination and,

consequently, the system's liveness[28]. Finally, data validation and integrity should depend on the system's assumptions on the type of fault tolerance by nodes, network, and timing synchronization. The next section discusses consensus protocols that implement different distributed system models.

2.4.3 Crash Fault Tolerance

Crash Fault Tolerance (CFT) is one resiliency level, where the system can still correctly reach consensus if components fail. Having n nodes in your consensus system CFT capable to withstand up to $\frac{n}{2}$ such crashes.

In this model there is (optimal) quorum of $\frac{n}{2}+1$ which must agree on certain value, which means if $\frac{n}{2}+1$ nodes are available, the system have a majority quorum that will be able to reach agreement. Therefore,, CFT system requires $2f+1$ nodes to tolerate f failures.

Like we said before, it is necessary to base system security on assumptions. CFT model guaranties resilience of crash nodes or network partitioning. However, it cannot guarantee anything in presence of malicious actors.

The Fischer, Lynch and Paterson theorem (FLP) impossibility[29] showed that for asynchronous timing models, which means there is no known bound on message delay and no known bound on processing speed, is impossible to guarantee termination. Because the algorithm cannot distinguish a crash fault from a delayed message. To circumvent this problem, the most mechanism widely used so far is a momentary synchronization periods. This makes CFT based systems assume the partially synchronous timing model.

The implementation of CFT algorithms has evolved over the years and new protocols will most probably emerge. Without going into comparisons of the best algorithm to date because it would be impossible to highlight all of them, we would like to highlight the most popular algorithms that continue to serve as the foundations for the implementation of new ones. These are Paxos and Raft, which we will now explain in more detail.

2.4.3.1 Paxos

Paxos[13] is a consensus algorithms family that define three classes of agents: proposers, acceptors, and learners. The proposers are who propose values to reach consensus and sends it to a set of acceptors. The acceptor is who contribute to reaching the consensus itself, so may accept the received value or reject it. Finally, the learner is who learn the agreed upon value by adopting the value when a large enough set of acceptors have accepted it. In an Paxos implementation, nodes can take multiple roles (even all of them), they must know how many acceptors a majority is, and they cannot forget what they accepted.

A Paxos run aims at reaching a single consensus, so the algorithm can only agree on one value. Once the algorithm reaches on a certain value, it wont progress to any other consensus

(i.e., any other different value). Which means, if we want agreement on multiple values, differents Paxos run must happen (Multi-Paxos implementation). The solution to do this is to have a replicated positions log and whether we want an agreement on another value, the algorithm will start a new Paxos run that will be on next log position. Thus, the set of logs will record all agreement values that were made by the nodes.

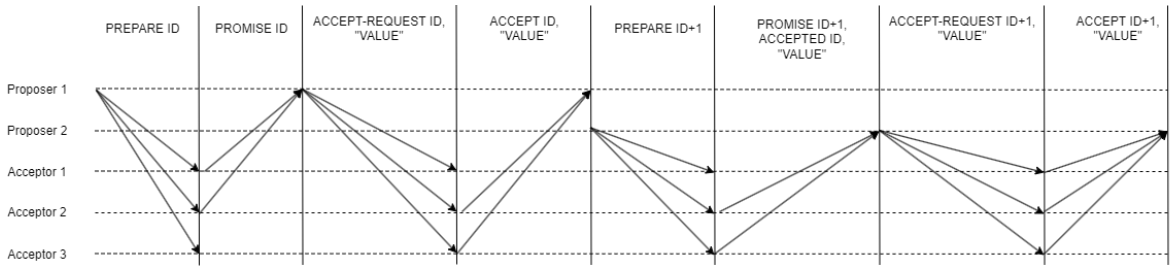


Figure 2.5: Paxos agreement mechanism using three nodes with all roles and supporting one node crash

Figure 2.5 shows how the Paxos algorithm works. Proposer 1 wants to propose a certain value. It will need to select a unique identifier *ID* (usually timestamps are used) and it sends a *PREPARE* request using that identifier. After sending, it will wait for responses and, if in a certain time (i.e., a timeout occurred) it does not receive any response, this proposer will retry the *PREPARE* message with a new higher identifier.

Acceptors receives the *PREPARE* message using *ID* as identifier and if that identifier is greater than another identifier received by *PREPARE* messages, then they reply to the request with a promise that they will not accept any more proposals using identifier less than *ID* (i.e., they will ignore *PREPARE* messages using less identifiers). When Proposer 1 gets a majority of *PROMISE* messages for the specific identifier *ID*, it sends *ACCEPT-REQUEST* messages to acceptors using promised identifier *ID* and the value *VALUE* that it wants to propose to the system. Acceptors receives *ACCEPT-REQUEST* message and, if they have not promised to ignore the message's identifier *ID*, they reply with an *ACCEPT* message containing the identifier *ID* and the value *VALUE*. After that they relay this message for all learners' nodes. If a acceptors majority accept the *ID,VALUE* association, an consensus is reached on value *VALUE*. When a proposer/learner gets accept majority (in Figure 2.5 scenario is when you get two accept messages) for a specific identifier *ID*, they know that a consensus has been reached on that value *VALUE*.

In Figure 2.5 after the *PREPARE ID+1* message we illustrate the case when the Proposer 2, that is not consensus aware, wants to propose a value into the system. If it sends a *PREPARE* message with an identifier smaller than ID, the acceptors will ignore the message and it will have to increase the identifier. If it sends a *PREPARE* message containing a higher identifier *ID+1*, the acceptors have not promised to ignore this *ID+1*, and they respond with a PROMISE message containing the identifier *ID+1*, the value *VALUE* and the identifier *ID* that they have accepted in the past. As soon as the proposer receives a *PROMISE* messages majority for the identifier *ID+1*, it checks if any values have been accepted in these messages. If no values have been accepted, then it can propose the value it wants. Since there was a previously accepted

value, Proposer 2 must choose the value containing the highest identifier, which in this case will be the identifier *ID* and the value *VALUE*. Then, it sends *ACCEPT-REQUEST* messages with the new identifier *ID+1* but with the previously chosen value *VALUE*.

Finally and again, acceptors receives the *ACCEPT-REQUEST* messages using an identifier greater than one they know, so they can't ignore this messages, and reply with an *ACCEPT* message using the identifier *ID+1* and the value *VALUE*. Lastly, Proposer 2 receives *ACCEPT* messages and figures out what the consensus was.

2.4.3.2 Raft

Raft[2] is a Paxos successor, and it is an algorithm that is more easily explainable and efficient. This algorithm implements consensus by electing a distinguished leader and giving him complete responsibility for managing the replicated log. The leader accepts log entries from clients then, when its safe to apply log entries to their state machines, replicates them on other servers. Raft defines three entities on system: leader, candidate, and follower. The leader manages all client's requests. Candidates have the responsibility to elect a new leader. Followers is a passive entity, and they simply reply to request from leaders and candidates.

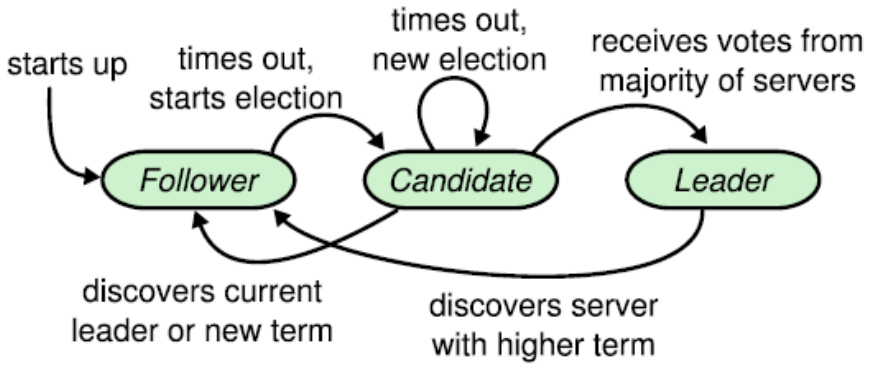


Figure 2.6: Raft server states[2]

Figure 2.6 represents the required leader election procedures and leader modification in case an elected leader node presents faulty behaviors.

To detect faults, this algorithm uses constant timeouts by reply. Particularly, a heartbeat mechanism to trigger leader election. When one node becomes Candidate and starts a leader election, there is a risk of getting more than one leader simultaneously in system overview, which is quite dangerous because they can take contradictory actions. To avoid this situation, Raft uses term sequencers. A leader node election occurs ins a certain term. Whenever there are new elections, the candidate's nodes will increment the current term. Using this approach, it is possible to restrict leaders in certain term to a maximum of one. A fault in leader election can happen, which means that a term has no leader at all, but there will never be more than one leader in any term.

In this election procedure, each node in system has chance to vote only once in each term and there is no chance to change the vote, since it would imply to be voting again in the same term. Thus, to elect a leader in the system requires a majority votes in which a quorum of nodes vote for a particular leader node in a specific term. That said, someone should be careful about who has permissions to insert new nodes into the system's set of nodes. Otherwise, the system can be vulnerable to Sybil attacks[30], where malicious actors create multiple anonymous nodes to form a quorum of nodes that can achieve a majority vote and thus control and manipulate consensus decisions.

Raft guarantees that there is only one leader node in each term, however it cannot guarantee that there will not be multiple leaders in different terms. When a leader node for whatever reason cannot communicate with other nodes, he does not know that there is a new elected leader and naturally assumes he is the leader, since the last communication suggested this scenario. Thus, we have one leader in term t and another leader in term $t + 1$. To deal to the fact that there may be different leaders on different terms at the same time in system, there is a constraint on when a leader wants to communicate to other nodes. In this circumstance, the leader does not decide exclusively for himself and will again have to ask a majority quorum of nodes if in fact he the leader of all the nodes is still, and thus has authority to deliver new messages with new values and elements for system information. The remaining nodes will reply affirmatively, if they do not have a different leader, who will naturally have a longer term (i.e., more recent).

2.4.4 Byzantine Fault Tolerance

Byzantine Fault Tolerance (BFT) appeared in the Byzantine Generals Problem paper[15]. Like we explained above, the analogy also fits the consensus topic because the generals must act together. This paper comes with a conclusion that the generals can not make the decision unless the number of generals is greater than three times the number of malicious actors. If we correlate the malicious actors with faults tolerated by the system, and the general's number with system nodes number, then we can claim the system cannot work safely unless $n \geq 3f + 1$.

The key differences to CFT model are in assumptions and fault model, CFT can withstand up to $\frac{n}{2}$system faulty nodes, while no guarantees on adversary nodes. BFT provides guarantees to withstand and correctly reach consensus in presence of $\frac{n}{3}$ failures of any kind including Byzantine. . This kind of protocols assume that faulty nodes are not able to break cryptographic primitives and consequently cannot impersonate correct nodes.

Recently, blockchain applications[11, 12, 31, 32] adopt this BFT approach. In a blockchain, consensus is also paramount. The blockchain power is a shared source of truth. If distinct parts of the network have different blockchain states, they can no longer work together.

2.4.4.1 PBFT

Miguel Castro et Barbara Liskov[3, 4] published a solution to the Byzantine Generals problem[15]. Pratical Byzantine Fault Tolerance (PBFT) can solve consensus when considering Byzantine faults and it tolerates less than $\frac{1}{3}$ Byzantine faults. Traditionally, the system can tolerate f Byzantine faults, where there are $n \geq 3f + 1$ total nodes.

Practical Byzantine Fault Tolerance algorithm can implement any deterministic replicated service with a state (a state machine that is replicated across different nodes in a distributed system) and some operations. In this service, clients issue requests to invoke operations and block waiting for a reply.

The original paper showed that it is also efficient when integrated with standard unreplicated Network File System (NFS). The resulting BFT-NFS is only 3% slower, even though it can now withstand Byzantine faults. However, the algorithm does not address the problem of fault-tolerant privacy[33], where a faulty replica may leak information to an attacker.

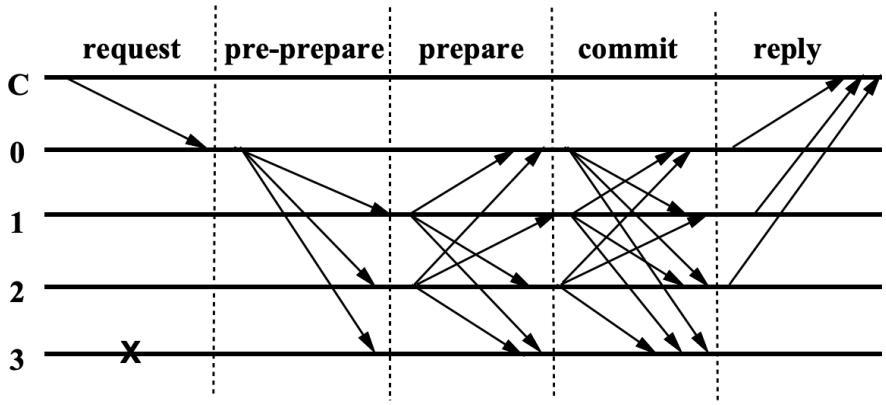


Figure 2.7: PBFT normal case operation using four replicas and one faulty node[3]

As it can be seen in Figure 2.7, PBFT algorithm consists of three phases on normal case operation: pre-prepare, prepare and commit. During normal operation, there is a primary replica that is associated with the corresponding view number. The view number is for tracking the current primary node of the system (like the Raft terms).

It begins when clients submit a request to primary node containing a timestamp and current view number. The primary node is responsible for receive requests from the client and to broadcast them to the remaining replicas nodes. If primary does not broadcast it, the other nodes will suspect her of misbehavior, causing a view change and another replica tries to take over his place by starting an election process and increasing the view number.

In pre-prepare phase, primary node 0 sends out *Pre-Prepare* messages to every replica in the network. A replica accepts the *Pre-Prepare* message so long as its valid. Messages contain a unique sequence number assign to each of their view, like the proposer identifier in Paxos. They also contain signatures that lets nodes determine message validity. If a node accepts a *Pre-Prepare* message, it follows up by sending out a *Prepare* message to every node. . The

receiving nodes accepts the *Prepare* messages so long as they are valid, again based on sequence numbers and signatures

PBFT considered a node prepared where it has seen the original request from the primary node, has pre-prepared and has seen $2f$ (where f is the number of Byzantine faults) *Prepare* messages that match it pre-prepare phase. These two first phases guarantee that non-faulty replicas agree on a total order for the requests within a view.

After nodes have prepared, they send out a Commit message. If a node receives $f + 1$ valid Commit messages, then they conduct the client request and, finally send out the client reply. Since the system allows for at most f faults, the client needs to wait for $f + 1$ of the same reply and this ensures the response to be valid. After this, client gets the correct response.

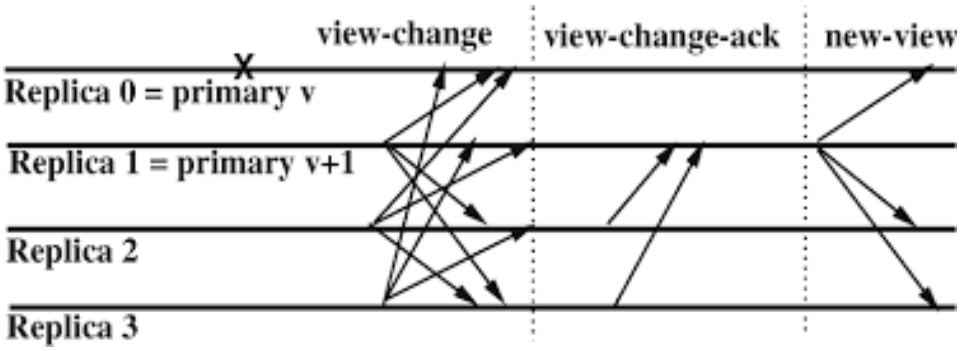


Figure 2.8: PBFT view change using four replicas node[4]

In order to handling leader crashes or doing some malicious activity, to kept the protocol to progress after failing, the view change presented in Figure 2.8, gets triggered by timeouts that prevent replicas from waiting indefinitely for requests. A timer starts after receiving a pre-prepare message. After it expires, the replica moves the system to view $v + 1$. It stops accepting messages and broadcast a View-Change message to every replica. After receiving *2f* responses, the replica should be the new primary and for this work, it will broadcast a New-View message followed by a set of Pre-Prepare messages. The replicas will receive these messages, and they will broadcast them to all other replicas if they accept the received messages.

2.4.4.2 BFT-SMaRt

BFT-SMaRt[5, 6, 34] is a BFT leader-based state machine replication protocol and it was the protocol used in previous research to assess the developed fault injection tools and fault tolerance BFT-SMaRt behavior. Considering a system composed by $n \geq 3f + 1$, where it can tolerate a maximum of f replicas subjected to Byzantine faults. It has a partial synchrony model because it requires synchrony to provide liveness. So, only when the system goes through synchronous states is that it guarantees the termination of protocol execution. Also, it assumes that all processes communicate through reliable and authenticated channels (i.e., fair links with retransmission and cryptographic operation that provide digital signatures[35] and message authentication codes[36]).

Similar to PBFT, the normal operation of BFT-SMaRt consensus instance uses the three-phase commit procedure we saw earlier to agree on a proposed value, as illustrated in Figure 2.9. When a consensus instance has decided a value, each replica replicated log will record the commit operations, along digital signatures of the involved replicas.

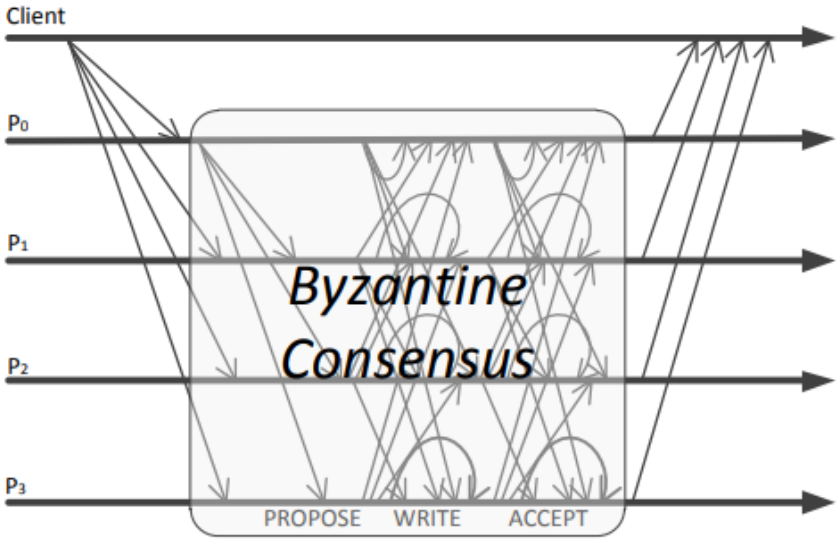


Figure 2.9: BFT-SMART normal phase message pattern[5]

Initially, normal phase starts with a client sending requests to all replicas containing a signature and a sequence number. Correct replicas will consider requests only if the message contains a valid signature and the expected sequence number. That said, the first phase starts when the leader sends a *PROPOSE* message with a batch of operations to be decided for other replicas. During each of the remaining phases, the procedure goes through an all-to-all communication exchange where the replicas exchange *WRITE* and *ACCEPT* messages, containing a cryptographic hash relative to the batch of proposed operations. Finally, all the correct replicas sent a reply to the client.

BFT-SMaRt assumes that correct replicas concurrently respond to the current consensus instance i and the previous consensus instance $i-1$. This ensures that a correct replica that is running instance $i-1$ and is not running instance i will be able to finish instance $i-1$ because there will be $n-f$ correct replicas running instance $i-1$. However, due to the asynchrony of the system, there is a possibility that the replicas receive messages regarding instances of different consensus (i.e., smaller than $i-1$ or larger than i). Replicas store early messages (messages with consensus instance greater than i) for future processing and discard outdated messages (messages with consensus instance lower than i). As soon as these conditions are not satisfiable, the synchronization phase starts.

The synchronization phase, illustrated in Figure 2.10, starts when two timeouts are triggered referring to a request made by a client, or when the system goes through an asynchronous period. To try to prevent leader changes, when the first timeout occurs, replicas forward the requests to all replicas. The reason for trigger timeouts may have to do with a faulty client, which may have

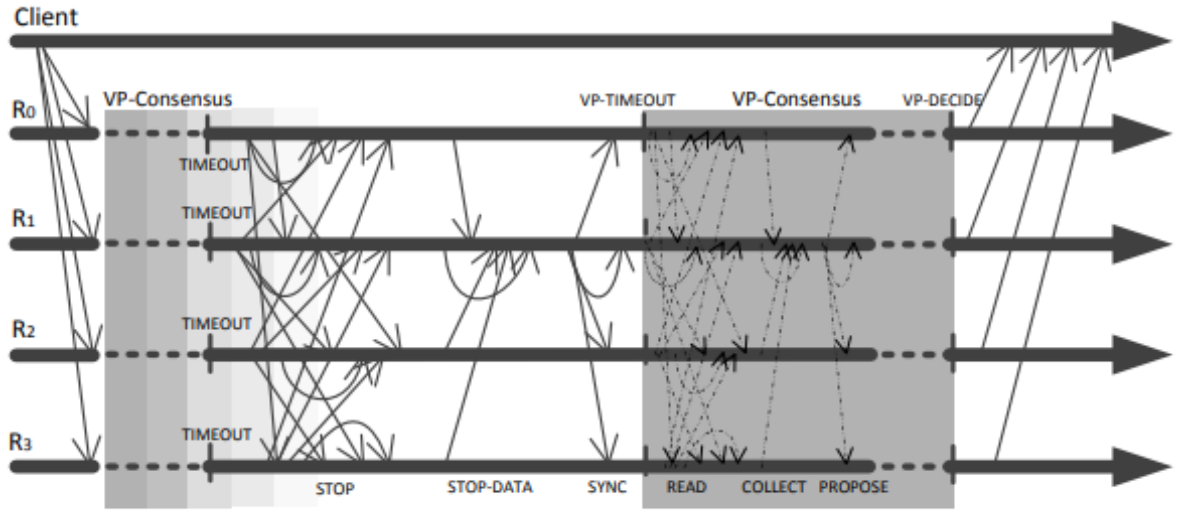


Figure 2.10: BFT-SMART synchronization phase message pattern[6]

sent the request only to a subset of replicas smaller than $n - f$ (which is not sufficient to ensure agreement progress). This forward procedure forces requests to reach all replicas in the system.

If there is a second timeout for the same request it is because leader has not delivered the client requests to other replicas. This phase causes a regency exchange (regencies is equivalent to views in PBFT) and forces a state synchronization between the correct replicas, i.e., makes all correct replicas be in a same consensus instance state. Consequently, as we see in Figure 2.10, the decision procedure (i.e. normal phase) stops by sending a *STOP* message to all replicas, which indicates the new leader replica. As soon as a replica receives more than f *STOP* messages, it knows that at least one correct replica has started the view change.

When a replica receives more than $2f$ *STOP* messages, it will proceed locally to the regency change. However, it is necessary that all replicas are at the same state to make a peaceful change. To synchronize their logs, all replicas, including the new leader, provide their decision logs by sending a *STOPDATA* message to the new regency leader. The leader checks if all decision values in logs are valid and when it receives at least $n - f$ valid *STOPDATA* messages, it sends a *SYNC* message to all replicas containing all information collected about consensus instances. After a replica receives a *SYNC* message, it performs those same operations provided by the leader. Finally, all correct replicas are at the same state and normal phase can resume.

2.5 Fault Injection

Faults may derive from hardware or software problems. At any moment, these faults will trigger due to user activities or by internal activities within the hardware/software systems architecture. The outcome that a fault can lead to depends on its severity and the system type. Independently of the developed system, we should be aware of which kind of faults that exposes the system.

We should assess the faults impact to determine if a system is vulnerable[37].

In fault-tolerant systems this testing procedure becomes even more important. We can gear the systems up to react to faulty adversities and respond with correct behaviors. However, the creation of well-coordinated fault chains intended to deceive and manipulate such systems' algorithms is always difficult to predict. For this reason, assessing such systems' algorithms in presence of recurrent faulty environment is vital to ensure any system reliability[38]. Fault injection testing is a technique that tests system resiliency in real-world scenarios that may occur through forced fault injection, to provide an evaluation of the outcome (i.e., the systems' recovery paths) and a risk analysis to identify vulnerabilities in systems design and implementation.

Fault Injection tools (FI) can be hardware-based, software-based, or use a hybrid hardware/software approach. Hardware methods can inject faults at the physical level into chip pins and internal components, such as combinational circuits and registers that are not software addressable[39–41]. Software faults are known as a major cause of computational systems defects nowadays. The relevant fault injection tools for this research are software based. They are extremely useful because they can emulate faults caused by hardware errors[42–44], with low intrusiveness and because it can monitor the systems by observing behavior in faulty or normal scenarios to determine if it continues to operate as expected.

Typically, Software Fault Injection (SFI) consists of three entities. The load generator, the injector, and the monitor[38]. The load generator stimulates the system using fault-triggering inputs and constraints provided by the tester. The injector introduces faults into the system, usually at pre-runtime (e.g compile-time, load-time, executable image manipulation[45]) or at runtime[39, 46]. The monitor gathers system readings and measurements, which may include target outputs, fault tolerance relevant computations, benchmarking techniques and/or comparisons with fault-free runs.

However, the existing FI tools are quite specific to the programming environment[47, 48] or to target application[49]. There is a lack of generic FI tools, especially for concurrent and distributed environments. Below we deeply describe in more detail the most influential examples of these types of tools.

2.5.1 Xception

Xception[50] is a fault injector that uses debugging and processor performance monitoring mechanisms. Its structure is composed of three modules. The Experiment Manager Module which contains a user interface to produce the faults and displays test results. A module that loads faults' configurations and libraries. And lastly, the kernel module that injects faults through system calls. Through remarkably close hardware-level operations, injected faults have an impact on any running process and can even affect the kernel.

This tool can collect detailed information on the number of clock cycles, the number of memory read and write cycles, and instructions executed from the injection of the fault until

other subsequent event, for instance the detection of an error.

2.5.2 FERRARI

FERRARI[51] emulates hardware faults from software. Through traps and system calls, it can corrupt the program execution state, making the system's behavior like a Central Process Unit (CPU), memory, or bus fault. It brought innovation with the possibility of introducing temporary and permanent faults, where cycle instructions define the fault duration. In addition to emulating hardware faults, it can also create flow control errors through target properties such as the time, location, type, and duration of a fault.

To do this, this injector concurrently executes a process for fault injection and the target process that will suffer fault injections. This tool manages the fault injection through communications between the two processes. The injection process changes the target program execution state by executing system calls.

It is capable of injecting faults in the address, data, or the control lines of the processor. Additionally, it has modules for initializing faults and collecting information from the tests performed.

2.5.3 FIAT

FIAT[52] stands for Fault Injection-based Automated Testing environment and creates a software environment that can fault inject user application code and data. It can manipulate messages (corrupted, lost, delayed), tasks (delayed or abnormal termination) and timers.

Through workloads, allows fault classes definition (relations between faults and the error patterns that they cause) within an application context, where the user should configure parameters regarding where, when and for how long errors will occur. The workloads are an observable set of real-time communicating tasks, and they could run in one machine or in distributed computers. For analysis purposes, this tool contains event performance logs and error reports such as exception logs and abnormal events.

2.5.4 Ftape

The fault injector in FTAPE[53] uses software to emulate the effects of underlying physical faults and can inject them throughout the target system, including in CPU registers, memory, and the disk system. The approach used is bit-flip faults to corrupt stored data. The fault configurations can be concrete or randomly location-based and/or time-based properties.

It has one workload generator that provide a controllable workload that can propagate the stress conditions on the machine, and it has a measure tool to monitor the workload activity

that present results like performance degradation, number of system crashes and fault ratio.

2.5.5 DOCTOR

DOCTOR[45] is an IntegrateD Software Fault InjeCTiOn EnviRonment for Distributed Real-time systems that aims the HARTS[54] distributed system. This tool can inject communication faults as well as traditional hardware faults such as memory and CPU faults. The user can choose any combination of these three types to assess the target system dependable properties. Effects such as making processes slow or fast, terminating or suspending processes, corrupting clock/timer services, corrupting system-call services, delay or lose messages adding the capability of distributed errors injection. The specification of a fault may be as transient, intermittent, or permanent. Additionally, this tool has a user interface, and it also presents a synthetic workload (replaces a real program) generation tool that allows the user to specify the fault injection timing and the rest payloads configurations.

The fault injector consists of three modules: Experiment Generation Module (EGM), Experiment Control Module (ECM), and Fault Injection Agent (FIA). The EGM is to generate executable images of workloads, and these executables will run on a single processing node or on a set of nodes. It is also to parse the experiment fault plan for each node supplied by the user. After that, ECM uses these files. The FIA receives commands from ECM via Ethernet and executes them by injecting faults. ECM functions as an external controller and its responsability is to set up an experiment environment by downloading executable images of the workload and synchronize the begin/end of each run among the set of nodes. Also, Data Collection Module (DCM) collects experimental data during each experiment supplied by FIA.

One distinct feature of this injection approach is the segregation of the host computer components from the target system components. Because each component runs separately and the target system executes essential components only, it has the advantage of reducing the run-time interference.

2.5.6 PROPANE

The Propagation Analysis Environment[55] is a fault injector tool that targets software modules (i.e., software functions which can be multithreading) on a desktop computer. It has injection and logging mechanisms provided by static C-libraries and the targets are software developed for single-process applications on embedded systems that can interact with libraries implemented in the C programming language. The injections support the mutation of source code and the manipulation variables and memory contents.

By design, it consists of different components to create and run test experiments. Namely, the PROPANE Setup Creater (PSC), the PROPANE Campaign Driver (PCD), the PROPANE Library (PL), and the PROPANE Data Extractor (PDE). The target system uses the PL,

written in the C programming language, for injection functionality. The PCD is responsible for managing the actual execution of injection experiments and it has a user interface through which the user can control and follow the experiments. During this analysis, the user may use the PDE to extract specific data from the test experiment. The PSC creates setup files needed during test execution with general information (regarding faults, probes, and injection locations) and fault triggers that may be based on time, frequency, probability distribution or user-based properties triggers. Also, it is responsible to spawns a new process for each experiment and PL is which performs the actual fault injection.

2.5.7 Loki

Loki[56] injects faults in a distributed system based on a partial view of its global. To achieve this view, a state machine specifies each component of the distributed system, and it can go forward and backward to various states. The global state is the vector of all component's states. A partial view is where you abstractly define specific states for each machine/node that will be the triggers aspects to fault injection.

This tool has the Loki runtime attached to each node, it maintains the partial view configuration for each node and inject faults when the states restritions in partial view of a node are true. Therefore, state changes trigger the faults, which allows you to define that a fault injected in one node can depend on the state of other nodes.

After an injection, Loki uses notifications to inform other system nodes of the state transitions made in a certain node. However, the Loki runtime does not block the system while these notifications are on way, which can lead to incorrect fault injections. In terms of features, it also allows monitor records of state changes and fault injections along with their times of occurrence.

Finally, the user must implement a probe component used in Loki. In this probe component is where the user select the type of faults to inject into the system and where the Loki perform the fault injections through the user's implementation of the *injectFault()* method, which contains the code that will be injected. The implementation of Loki runtime was in C++, so the implementation of this probe component also must be in C++. To facilitate user's work, this tool contains a Graphical User Interface (GUI) that can specify a state machine and a fault injection campaign.

2.5.8 J-SWFIT

The Java Software Fault Injection Tool[57] is based on the G-SWFIT[37] and allows software faults injection in Java systems. The injection is directly at bytecode level using the Objectweb ASM framework[58], which means that does not need the source code of target system to perform the injection. Firstly, it tries to find suitable location by Assembly operations recognition in a way to deduce the high-level programming source code and then perform modifications in the

bytecode by inserting faulty bytecode. It allows operator injection that represents the lack of an *IF* structure and its block of code and the lack of a function call.

It has a user interface that facilitates the fault construction but also the monitoring of faulty result. Firstly, this tool loads the java system classes, recognize of if structures and function call's fault patterns, inject faults and then monitor the system.

2.5.9 LLFI

LLFI is a program level fault-injection tool that uses a collection of compiler tools framework called Low Level Virtual Machine (LLVM)[59] for fault injection mechanisms. LLVM provides higher level intermediate code than assembly code and encodes information such as address computations of loads and stores.

Anna Thomas et Karthik Pattabiraman[60] target applications called soft computing applications (e.g, multimedia application) and argue terms like Egregous Data Corruptions (EDC) to denote outcomes whose quality deviates from the fidelity system metrics. They focus is to formulate source-level heuristics on detecting errors under user assumptions. LLFI allows fault injections performed at specific program points and into specific data types by a single bit flip into the destination registers.

2.5.10 Discussion

The tools we have covered present different approaches to perform fault injections. Some use low-level injection mechanisms to control CPU processing, such as Xception and Ferrari, or use bit-flip mechanisms to corrupt the data stored in the memory registers, such as Ftape and LLFI. Others we have discussed have source or byte code manipulation mechanisms but are heavily dependent on the native language (for example, C or Java), as is the case of PROPANE and J-SWFIT.

The closest tools to our implementation are those that are designed to assess distributed systems such as FIAT, DOCTOR and LOKI. FIAT can inject similar faults as we do, such as delayed messages or processing, and creates an interesting fault relationship. DOCTOR also provides the similar type of faults to us. However, it is specifically for assessing a distributed system implementation called Harts. Loki uses an interesting and similar approach to ours where it defines independent states for each node as the triggering method for faults in the system.

Many of these tool's features served as inspiration for our research, however we feel that there is no generic tool that enables the user to freely choose the programming language, the type of system, the protocol it uses, and fault customization.

Chapter 3

Prior Work

Due to the huge complexity involved in protocols that implement distributed systems, it is difficult to cover and investigate all attack vectors to which can expose such algorithm. The purpose of beginning a fault injector investigation is to create a tool that detect system vulnerabilities and improve the system resilience of such complex distributed system.

In this chapter we present the tools used by the Fault Injector in our research and in the research that precedes our work. Next, we present the research work on Fault Injector tools that precedes this document. Hermes was the initial tool which combined Aspect Oriented Programming (AOP) and Software Fault Injection (SFI) approaches to create faulty tests in Byzantine Fault Tolerance (BFT) protocol implementations. Zermia came next to improve the testing capabilities of Hermes. Lastly, Proteus brought the innovation of automating the code development process for a target application by creating a Domain Specific Language (DSL).

3.1 Aspect Oriented Programming

The tool we have developed, and tools before ours, rely on AOP implementation mechanisms for fault injection in a modularized way.

AOP describe a programming technique and a way of thinking about the construction of software applications that complements the forms of expression found in object-oriented programming. Aspect oriented programs comprise of a mixture of objects and aspects and the goal of AOP is to improve the modularity of software applications[61].

The behavior in an aspect can execute at points in the runtime of the program determined by the aspect's specification. This simple idea turns out to be enormously powerful at modularizing the implementation of additional code on top of the application code without necessarily having to edit the application code. This allows in our case to create a fault injector with an extremely low intrusiveness.

In its operation, it uses four main concepts:

- JoinPoint - Join points are events that occur during the runtime execution of a program (for example, the initialization of a class, the execution of a method, the handling of an exception, or the updating of a field).
- Pointcut - Pointcuts are predicates (usually class names or methods) that match join points.
- Advice - Any join point matched by a pointcut expression executes blocks of code known as advice.
- Aspect - Aspects are modular units of crosscutting implementation. Aspect declarations defines an Aspect, which have a form like class declarations. Aspect declarations may include pointcut declarations, advice declarations, as well as all other kinds of declarations permitted in class declarations.

There are several ways to trigger an advice associated to a particular pointcut:

- Before - runs before the called predicate.
- After - runs after the predicate returned a result.
- Around - This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point by returning its own return value or throwing an exception.

AspectJ implements AOP for the Java runtime and will be the tool we will use in our injector. It provides three types of weaving[62]:

- Compile-time weaving - It requires access to the source code of an application and then, the AspectJ compiler, called ajc, will compile from source and produce woven class files as output. The aspects themselves may be in source or binary form.
- • Post-compile weaving - Also sometimes called binary weaving, it does the weaving of existing class files and Java ARchive (JAR) files. Like the compile-time case, the aspects used for weaving may be in source or binary form and may aspects weave them.
- Load-time weaving - Defers from binary weaving at the point that a class loader loads a class file and defines the class to the Java Virtual Machine (JVM).

We will use the post-compile weaving method for weaving and annotations in the code implementation on aspects regarding a particular fault.

Listing 3.1, shows a simple example of an implementation of aspects. The *@Aspect* annotation declares the *HelloAspect* class as an aspect. The pointcut refers to the *main()* method of the *HelloWorld* class that is in the *com.example* package. The *@After* annotation executes the advice after this predicate pointcut returned a result. The *helloWorldMainAdvice()* is the advice (block of code) associated with this pointcut. Lastly, the *joinPoint* has information associated with the pointcut's method call (for example, the arguments to the main function).

```
@Aspect
public class HelloAspect {
    @After("execution(* com.example.HelloWorld.main(..)")
    public void helloWorldMainAdvice(JoinPoint joinPoint) {
        ... // some instructions executed after main()
    }
}
```

Listing 3.1: Simple example of an aspect implementation using AspectJ

3.2 gRPC

gRPC[7] is an open-source high performance Remote Procedure Call (RPC) framework developed by Google that can run in multiple common languages. This tool is essential for the communication of the different Zermia components. This makes it possible to implement Zermia components in other languages without affecting our architecture.

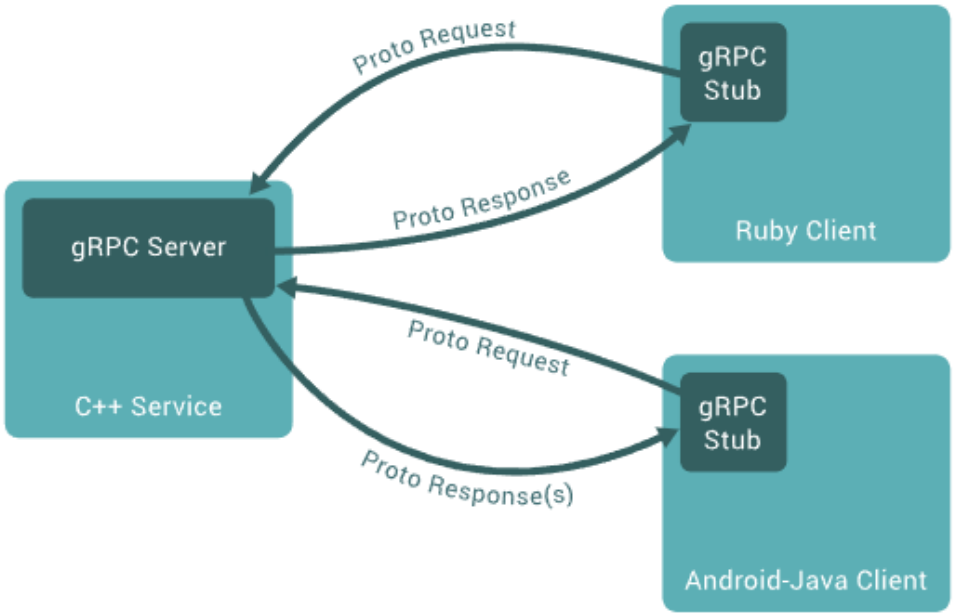


Figure 3.1: gRPC overview[7]

Figure 3.1 shows how gRPC works. A client application can directly call a method on a server application on a different machine. A server application can be developed on Java and a client application implementation can be written in Python. Even so, they can communicate through *grpc Server* and *grpc Stub*, which derived from Protocol Buffers, Google's mature open-source mechanism for serializing data structures. gRPC uses *protoc* with a special gRPC plugin to generate code from proto files. It will generate gRPC client and server code, as well as the regular protocol buffer code for populating, serializing, and retrieving message types.

For demonstration, we can see in Listing 3.2 an RPC call to say hello written in *protoc*,

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Listing 3.2: Service declaration from proto files[7]

where a message is sent containing thw user's name and then replied with the response message containing the greetings.

3.3 Hermes

Hermes[8] is a software fault injection tool that through AOP[63] can intrinsically weave the entire injection infrastructure into the target application's code. The aspect-oriented programming approach decreases intrusiveness because it allows to avoid modifications in the source-code of the respective application. After the definition of injection points, the faults need to be compiled into the injection points in the target source-code. Applications developed in programming languages like Java and C++ are the ones that can use this tool, since the tool relies on technologies like AspectJ[61] and AspectC++[64] to implement the aspect-oriented programming approach.

The Hermes architecture, shown in Figure 3.2, has one *Orchestrator* that manages all the remote nodes of the distributed system and allows the coordination of multiple fault injections. The *Hermes Runtime* is a component that is present on each node, symbolizes a fault configuration and performs the actual fault injection through communications with the Orchestrator, which acts as a synchronization layer. These two components have three communication primitives: *RemoteAction*, *Action* and *Notification*. The *Hermes Runtime* sent an *Action* to *Orchestrator* to check if the fault is ready to be executed. The *Orchestrator* sent an *RemoteAction* to a particular *Hermes Runtime* to manipulate its execution state. A *Notification* is an asynchronous message that sends *Orchestrator* information about a replica's state.

Regarding faults, Hermes considers that there are protocol context independent faults, where it is not necessary for the developer to be familiar with BFT protocol implementations

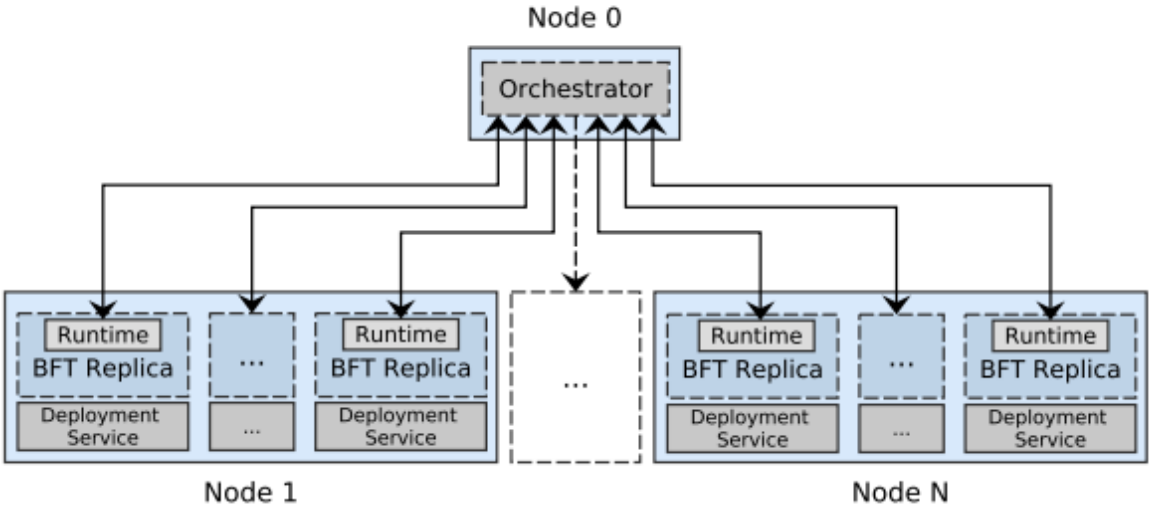


Figure 3.2: Hermes's architecture[8]

to successfully inject faults, and protocol context dependent faults, where it is necessary for the developer to access protocol data and thus must make modifications to the source code. Context-independent faults are Central Process Unit (CPU) Load, Crash, Sleep and Drop packet. Context-dependent faults are Corrupt header, Corrupt payload, Forge signatures and Distributed Denial of Service (DDoS).

As a trial foundation, Hermes assessed the BFT-SMaRt protocol implementation and through synchronous combination of several faults revealed bugs in the validation of incoming packet sizes and the increases in timeouts values in leader exchange phase.

3.4 Zermia

Zermia[9] is a Hermes evolution and shows many similarities both in architecture and in technologies it uses to implement fault injection mechanisms (such as the AOP approach).

It uses a client-server model, shown in Figure 3.3, that was developed in Java and has two main components: the independent *Coordinator* and an *Agent* for each replica/client node from the target system. The *Coordinator* is responsible for managing Agent instances by providing their user-defined fault schedules. However, it also coordinates the fault injection moments through synchronization mechanisms of the various *Agents* involved. The *Agents* maintain application-independent states to monitor fault injection moments depending on the application state, and through the AOP approach by using AspectJ code weaving mechanisms, they take over locations in the target protocol where they can perform actions such as changing the node state and execution flow by accessing to method calls and variable modifications.

This independent state mechanism allows the comparison of the current execution state of

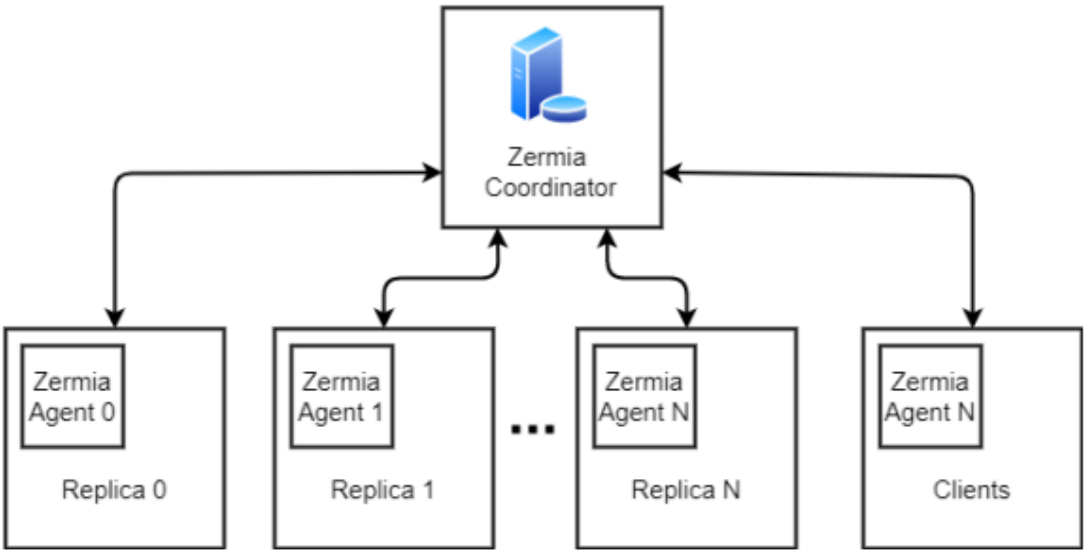


Figure 3.3: Zermia architecture[9]

a node with the intended state when a fault is ready to trigger. Zermia has also developed a priority mechanism to control fault collisions at the same instant, where a specific priority is associated with each fault type to increase the faulty behavior impact on the application. The fault types that Zermia is capable of injecting are:

- **Crash** - Suddently crashes the client or replica.
- **Thread Delay** - Delays the thread by using sleep method, where the user defines the time.
- **Message Dropping** - Replicas or clients drop message by preventing the send target method call. The user can define the execution based on a probability (between 0-100).
- **Message Flood** - Sends several messages per call, where the user defines the number of messages sent, to simulate a Denial-of-Service attack.
- **Message Modification** - Replace message contents by modifying argument values from method call. The user defines the message content.
- **Resource exhaustion** - Increases the usage of CPU or memory, which can lead to crashes and memory leaks. The user can specify the number of threads for CPU tasks and the amount memory used.

```
java -jar Zermia.jar Replica 0 TdelayAll 100 5000 10000 Replica 2 TdelayAll 100 5000 10000
Crash 1 20000 1 Clients 5 PNRS 500 100
```

Figure 3.4: Fault schedule example through command line [9]

Figure 3.4 shows a test experiment configuration, where it defines the fault schedules for two server replicas and one client. *Replica 0* will receive a delay fault for 100 milliseconds, and it will

trigger when consensus instance 5000 is running and consecutively in the next 10000 consensus instances. *Replica 2* will have the same delay fault configuration as *Replica 0* and, additionally has a crash fault configured that it will trigger at consensus instance 20000. Finally, *Client 5* stops sending messages only to the primary replica for 100 consecutive rounds, once he reaches the sequence number of 500.

Comparatively to Hermes, Zermia improved aspects such as flexibility, extensibility and eventually introduced new fault types to assess BFT system protocols. It also introduced the gRPC[65] technology that is a tool created by Google to perform RPC which opens the ability to extend the tool to other languages through Agent portability. However, the system has a complex implementation and is very dependent on the *Coordinator*, which leads to unnecessary overhead to the target application that distances it from functioning with its normal performance. It also remains extremely focused on BFT-SMaRt protocol like Hermes, which makes it quite difficult to port this tool to other BFT protocols and other programming languages.

3.5 Proteus

Proteus[10] is the evolution of the previous tools Hermes and Zermia, consequently it has a similar architecture, as shown in Figure 3.5.

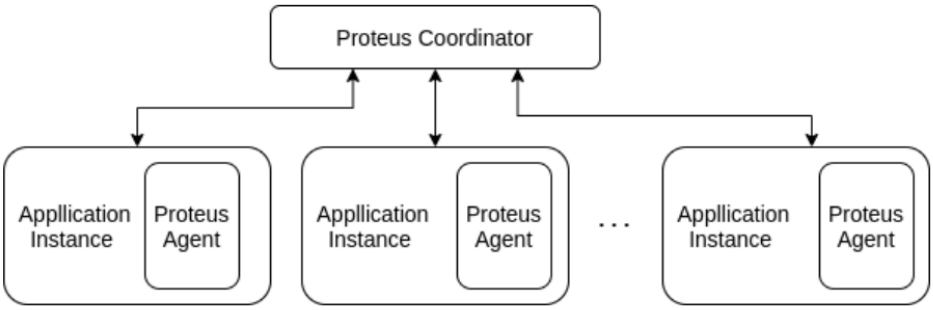


Figure 3.5: Proteus architecture [10]

The biggest innovation that this tool brought was the creation of DSL[66] code that facilitates the method of creating fault schedules by the user. By defining grammars and using the Xtext framework, an extension of the Java programming language, Proteus can generate Java source-code that will use AspectJ as a mechanism to alter the execution state of the target application. This makes it possible to create fault schedules for each of the system's replicas, simply by defining configuration parameters for the behavior of the faults and the injection location of the application. The DSL transform the described schedule to the corresponding code that implements it.

The purpose of Proteus is the creation of a standardized way of defining and creating objects using a higher-level abstraction, which allows for simpler code and facilitates any future expansions into a new programming language, since the code generator allows the code development of any

language.

Chapter 4

Design

In this chapter we describe our design for a latest version of *Zermia* in which it is possible to assess a variety of distributed and concurrent applications. First, we describe the main objectives and analyze each of the components of our tool, referencing their functionalities. Next, we describe the primary features that lead us to achieve the proposed objectives, such as the designation of faults, schedulers, and triggers. Finally, we discuss the design ideas we implemented, touch on other paths we could have taken in our approach, and we make a comparison with previous tools *Hermes*[8],*Zermia*[9] and *Proteus*[10].

4.1 Zermia Design

Zermia is a fault injector framework designed for concurrent and distributed applications which focuses on flexibility, extensibility, and efficiency. It aims at evaluating the behavior of distributed/concurrent systems to assess whether they behave properly in the presence of certain adversities. In this latest version of the injector, we decided to redesign the previous versions so that the injector is extensible to assess various fault tolerance protocols, to porting our *Agents* to other programming languages because the capability of injecting faults in different nodes that have different languages is interesting for the testing of distributed applications. Also, we want to provide freedom for the users to customize the faults they want to assess in their application (while still providing predefined faults).

This implementation follows client-server model to inject predefined and customized faults on a concurrent or distributed application, as presented in Figure 4.1. A client is a target application node from tested system which has *Zermia* libraries hooked to manage and inject *Fault Hooks* that will produce faulty behavior in the application. The server is composed by *Agents* and one *Coordinator* (as needed to inject faults that depend on the state of several application nodes). All communications between components (*Coordinator*, *Agents*, and *Fault Hooks*) are through Remote Procedure Call (RPC) by the gRPC tool, as we describe in Section 3.2.

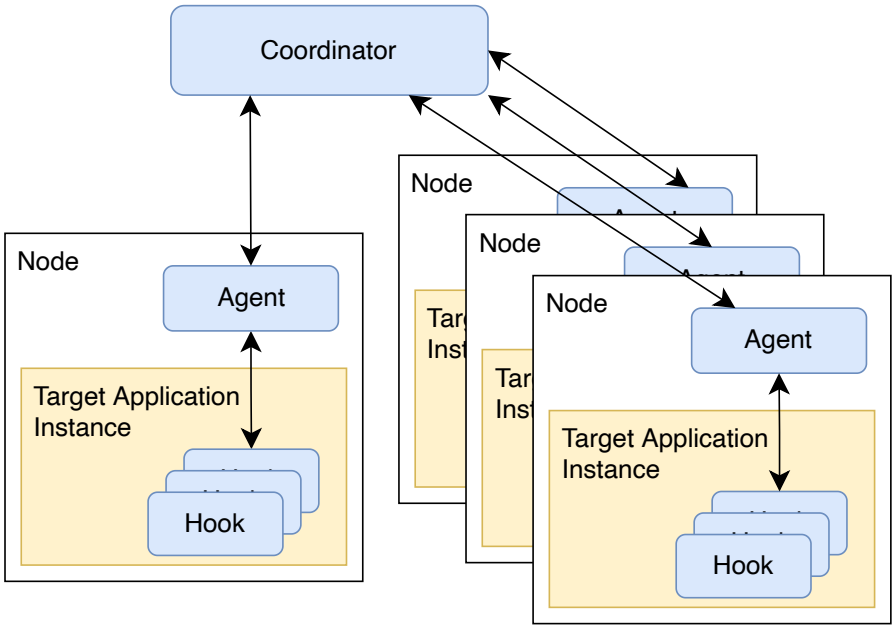


Figure 4.1: Overview of the latest Zermia architecture

4.1.1 Server

The main functions of the servers are to serve the client applications nodes with the fault scheduler configuration and to receive notifications of the injected faults. With those notifications, servers' instances can check dependencies between faults. These dependencies can be internal (faults injected within the same application node), external (faults injected in another application node) or independent (no dependencies).

One of *Zermia's* characteristics is its flexibility in these types of dependencies, distinguishing the server's execution flow according to each type of fault dependencies. To ensure this flexibility, the server splits into two main components. They are the *Agents* and the *Coordinator*. Each application node requires one *Agent* because it is through him that the application node receives the information regarding the faults it should inject, but in terms of dependencies, *Agents* does internal fault dependency checks. The *Coordinator* (which coordinates all agents) does external fault dependency checks and the user can exclude him from a test experiment if he does not want to check dependencies between multiple *Agents*.

4.1.1.1 Agent

Agents runs in the same node of the target application. It interacts with a target application node by monitoring its execution and deciding when to inject faults based on a fault scheduler. As stated above, it is reponsible for serving the application node with the configuration of his fault scheduler, where *Zermia* data structures hooked into the application will stores it. For this purpose, an *Agent* can get the scheduler from the *Coordinator*, or it can have its own fault scheduler. If it gets its scheduler from the *Coordinator*, during an *Agent* bootstrap it must

register with the *Coordinator* to get its corresponding scheduler.

That said, it plays a key role because it performs internal fault dependencies. To do this, the *Agent* stores which faults for its application node have already been injected.

4.1.1.2 Coordinator

Coordinator is an *Agent's* handler. It is responsible to communicate with *Agents*, distribute their respective fault scheduler, synchronizing and coordinating *Agents* and *Fault Hooks*. It plays a key role because it allows dependencies checks on external faults. That is, when we need to know some execution state of another external application node, the *Coordinator* can inform about the running state of other nodes. For this and like the *Agent*, it stores the faults that have already been injected. However, the *Coordinator* stores the faults of all *Agents*.

4.1.2 Client

Clients are all the nodes (client's application, replicas, server, processes, or threads) that are part of the target application, and that will be exposed to faulty behavior or have an influence on the fault injection process in other application nodes. To integrate with *Zermia*, they interact with *Zermia* libraries that allows the communication with their respective *Agent* and controls when a fault should be injected according to their fault schedule defined in *Agent*. During a target application node bootstrap, it must register with their respective *Agent* to instantiate the required *Zermia* data structures for maintaining its state and to get its corresponding scheduler.

4.1.2.1 Fault Hook

Fault Hooks are a set with zero or more concrete faults attach to target application node and sharing that process's runtime to produce faulty behavior. They use the *Zermia* library that can maintain additional state, independent of the application, which allows the definition of parameters for triggering faults. Through Aspect Oriented Programming (AOP) mechanisms as we describe in Section 3.1, they can alter the target execution state and flow, by acessing method calls, arguments and return values.

4.2 Faults

In Zermia, we identify faults through a unique identifier and define faults as a high-level structure through four characteristics:

- ***What*** is the code to be injected.
- ***When*** should inject the fault, the conditions required to trigger the fault.

- ***Where*** should inject the fault, the location in the application context.
- ***How*** to trigger the fault, if is to be triggered before or after, and if it prevents the target from execution.

Since we want to provide freedom to the user to customize the faults they want to inject, we chose to define these characteristics in two phases. On the server side of *Zermia* (*Coordinator/Agent*) we define the *When* characteristic and specify the Agent to whom the fault belongs, which is associated with the application node.

The conditions needed to trigger a fault can be based on application/timing state conditions or based on fault dependencies. Those based on application/timing state conditions can have multiple purposes. For example, we may want to trigger a fault at a specific time instant, consensus instance, specific thread, we may set a time or consensus range of instances, we may set a certain value for a variable to perform actions whenever a variable loads that value, or we may have several of the above conditions that when all have been met it creates a specific state so that the fault can be injected. Because of the plethora of possibilities that we can have as trigger conditions, we chose to abstractly define the conditions and let the user define the specifics according to the tests they want to run.

In addition to the above, we can also add the possibility that we want a fault to be triggered after a certain behavior of the system or before/after another fault has been injected. For example, we may want a fault to be triggered before another fault. So, we define dependencies through *after* and *before* which are sequences of zero or more fault identifiers, where in the *after* you define the fault identifiers that must happen previously for a fault to be triggered, and in the *before* you define the fault identifiers that cannot happen previously for a fault to be triggered.

On the client side, when implementing a *Fault Hook*, the user must define the remaining *Where*, *What* and *How* characteristics. A function and its code block define the code to be injected. JoinPoints define the location in the application context and After, Before or Around annotations (AOP features, as presented in Section 3.1) define how to trigger the fault.

4.3 Schedulers

Schedulers are a sequence of zero or more faults. It includes information about the *Agent* that will apply the scheduler, and which is associated with a specific application node. The user creates these schedulers and *Zermia* was designing to provide scheduling flexibility, offering support for:

- Independent fault schedulers for each *Agent*, including fault free schedules (i.e., schedulers with zero faults).
- Fault dependencies within the same scheduler, i.e., dependencies between faults triggered by

a single *Agent*. The capability of triggering faults is affected by whether internal faults are triggered (e.g., prevent a fault from triggering if another fault has already been triggered).

- Fault dependencies between schedules, i.e., dependencies between faults triggered by different *Agents*. Like the previous model but in extending restrictions involving several nodes. The capability of triggering faults is affected by wether or not external faults are triggered (e.g., only inject a fault after some other *Agent* has injected its own scheduler).

4.4 Triggers

Like we said above, in *Zermia*, fault triggers can be state (dependant of target application's) and/or event based. There are several ways of monitoring these triggers. *Zermia* includes different trigger modules, and these can be managed by the *Coordinador*, the *Agent*, a hybrid approach managed by both *Agent* and *Coordinator*, and a indepedent model where a fault does not need to be managed by any *Agent* or the *Coordinator* (the *Fault Hook* manages itself).

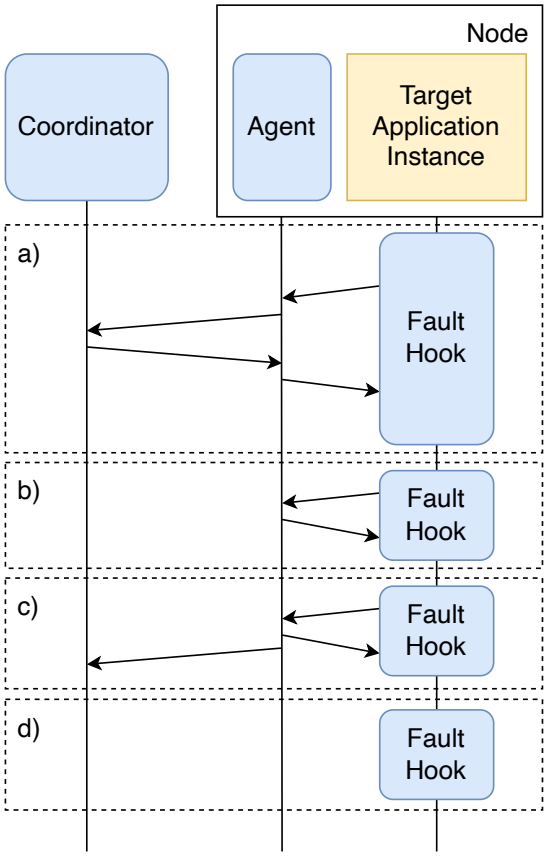


Figure 4.2: Zermia's fault triggering models

4.4.1 Coordinator-based Triggering Model

In the Coordinator-Based approach, the *Coordinator* is responsible for managing the triggering of faults. *Agents* query the *Coordinator* before each fault is triggered to check the triggering conditions. The fault is injected or not depending on the response given by the *Coordinator*, as presented in Figure 4.2, case a).

This approach makes it possible to trace faults from multiple *Agents* and enables fault dependency between multiple *Agents*. Conversely, the *Agent* must constantly communicate with the *Coordinator*, which will consequently mean more communications and more overhead for the target application.

4.4.2 Agent-based Triggering Model

The Agent-Based approach does not require the *Agent* to contact the *Coordinator* to determine if a fault is to be triggered or not, leaving the Agent responsible for managing internally the triggering of faults, as presented in Figure 4.2, case b).

The benefit of this approach is the reduction of communications, and therefore the reduction of overhead for the target application. However, this approach makes it impossible to depend on faults between multiple *Agents*, since there is no synchronization with the *Coordinator*. We could implement direct communications between multiple *Agents*, however this would lead to enormous complexity and an increase in the number of communications throughout the system.

4.4.3 Hybrid Triggering Model

In the Hybrid model, as presented in Figure 4.2, case c), *Agent* can trigger faults autonomously, notifying the *Coordinator* after triggering faults if needed, or synchronise with the *Coordinator* for triggering faults with external dependencies.

The objective of this approach is reaching a balance between the previous two models. We achieve this balance by letting the *Agent* perceive and determine whether the fault contains internal dependencies, external dependencies, or both. Since it has access to its fault scheduler, it knows faults assigned to it and checks if the fault depends on any fault that it knows about. If it depends on an internal fault, it checks if the fault can be triggered according to the internal dependency and returns a response. If it depends on a fault that the *Agent* does not know about, then it has an external dependency and contacts the *Coordinator* to perform the verification.

Although it is a more complex approach, compared to the Coordinator-based model it reduces overhead, and compared to the Agent-based model it increases flexibility since faults can contain external dependencies.

4.4.4 Independent Triggering Model

The Independent model approach does not require any contact with the *Agent* or the *Coordinator* to determine if a fault is to be triggered or not. The *Fault Hooks* maintain the necessary state information for managing their own triggering conditions, as presented in Figure 4.2, case d). However, this model is quite restricted because it only supports state-based trigger conditions. Any event-based condition, i.e., any dependency, this model cannot support it.

This increases complexity in the *Fault Hook*, since it is responsible for managing target application state. However, this approach reduces overhead, compared with other models, since no external communication.

Reducing overhead is important, since an intrusive fault injector can compromise the target application's execution flow, which in turn compromises test run validity. To being able to simulate a wide range of execution scenarios possible, *Zermia* implements a triggering model according to the hybrid and independent approaches. We try to reach a balance between overhead and flexibility providing management of state-based triggering conditions by *Fault Hooks* and management of event-based (i.e., dependencies) trigger conditions by *Agents* and/or the *Coordinator*.

4.5 Determinism

Concurrent and distributed applications typically suffer from non-deterministic execution traces due to different thread/process interleaving. Fault injection tools must find deterministic mechanisms to execution the faults in the same order, where the configuration is the same.

Our hybrid triggering model reduces non-determinism by synchronizing internal faults or synchronizing *Agents* with the *Coordinator* before triggering faults. This allows the user to assess multiple coordinated event scenarios based on fault dependencies.

The previous version of *Zermia* developed a system based on priorities, which allows users to assign priorities to faults to determine fault execution when two or more faults are triggered concurrently. We have no mechanism in this version for this purpose. However, the user can set up a fault priority and order manually using conditions and fault dependencies.

4.6 Discussion

To add flexibility and efficiency to *Zermia* it was necessary to modify the design that had accompanied the last investigations of our group. We could have chosen an architecture without the *Agent* components, so that the functions performed by the *Agent* would be on the *Zermia* library present in an application node and used by the *Fault Hooks*. However, this would imply the implementation of functions to communicate with the *Coordinator* within the application

context, and our intention is to guarantee, without major complexities, the portability of the tool to test applications of several languages.

The significant difference was that we chose to remove the *Agent* functions from the application context, placing the *Agent* independent of the application but residing on an application node. This change makes it possible to port *Fault Hooks* and the *Zermia* library used by *Fault Hooks* to other languages. Therefore, the tool is structurally prepared to assess a node of any programming language, without having to patch any functionality of the injector. Thus, the user will have to develop the faults regarding the tests they want to perform and will have to use the *Zermia* library to communicate with our injector's structure to take full advantage of the functionalities we offer.

Another change we had to make was in the definition of the fault schedules. In the previous version of *Zermia*[9], the definition of the fault schedules was a little complex and the schedules did not allow flexibility to the user's tests. In *Proteus*[10] there was an evolution in this aspect by adding flexibility in the fault triggering conditions, which makes it like our implementation. However, due to the use of code generation mechanisms, *Proteus* differs from our injector because it defines the location in the application context where the fault should be injected and the location of the code that should be injected in the *Coordinator*. In our implementation, we want to guarantee the user's ability to inject faulty code that is already developed (i.e., predefined faults) or their own custom code (i.e., custom faults). To achieve this goal, we had to change the schedule definition syntax and we have implemented an abstraction of a *Fault Hook* to enable the creation of custom faults while implementing as little code as possible to use our injector's functionality. We will cover this implementation in more detail in Chapter 5.

Since we can set faults to run during a range of instances, the fault notifications should only be sent when a fault was executed for the last time. Otherwise, we may be making semantic errors that could lead to faulty behavior outside the intended context. Therefore, sending notifications is responsibility of the user.

Those injected fault notifications are sent to the corresponding *Agent* and, if necessary, to the *Coordinator*. We could have instead opted for the possibility of broadcasting to all *Agents* when a fault is injected, obtaining a decentralized way to control fault dependencies (i.e., without the need for a *Coordinator*). However, we felt that this approach would result in reduced communications for applications with few nodes, and that it is not a scalable solution, because with many nodes the communications would be exponentially more than having to check the dependencies in a centralized way, with the *Coordinator's* support.

Our fault model and schedules enable assessing several types of applications, independent of the concurrent model. *Zermia* can be used with a simple *Agent* that is responsible for fault injection, while in concurrent applications, each process or thread can have its own *Agent*. This allows faults to be triggered on independent processes/threads, or to coordinate faults between processes/threads. The same principles apply to distributed and replicated applications.

Chapter 5

Implementation details

In this chapter we describe the implementation details that led us to a latest version of *Zermia*. First, we present the details of the *Coordinator* and *Agents* and how they are booted. Next, we present the details of faults in an abstract way, predefined faults, and how we can define faults from that abstraction in two programming languages case studies (Java and Python). We address some thoughts on how to achieve fault tolerance testing protocols. Finally, we present a test automation script to initialize *Coordinator*, *Agents* and application nodes to demonstrate the faulty experiences derived from a configuration.

5.1 Coordinator

Coordinator was developed in Java and provides an Remote Procedure Call (RPC) communications service to interact with Agents. As can be seen in Listing 5.1, to be instantiated it needs 3 inputs:

- The port number to listen for communications made by Agents.
- The total number of Agents.
- The configuration file with the fault schedules for each Agent.

```
$ java -jar coordinator.jar -p 9090 -a 4 -f agents_config.json
```

Listing 5.1: Running the Coordinator on port 9090, using 4 agents and the settings for the fault schedules in the agents_config.json file

Internally it stores the schedulers of the respective Agents configured via a JavaScript Object Notation (JSON) file. This configuration must contain the *agents* field with a sequence of zero or more agents where each *Agent* must be defined by three fields. The *agent_id* identifier, the *faulty* logic property, and its *scheduler* fault schedule.

The *faulty* field, which characterizes whether the *Agent* is a faulty or a correct *Agent*, distinguishes *Agents* that can inject faults (i.e., a faulty *Agent*). This distinction is designed to allow that one correct *Agent* can use a faulty configuration to monitor correct events that may happen in the application, in order to deterministically reach a certain application state, and thereby be able to use the injector to send notifications when it reaches the desired states, without injecting any kind of fault.

The *scheduler* field is a sequence of zero or more faults where each fault must be defined by a *fault_id* identifier and which optionally can contain both the *fault_trigger_conditions* field and the *fault_dependencies* field.

The *fault_trigger_conditions* field is used to specify the triggering constraints that will define the intended application state to inject the fault. Since all applications have their own specifications, the user can define any constraint or a set of constraints that define the intended application state. However, for serialization reasons in the injector structure, the constraints and the values of the constraints must be of *String* type. When it is necessary to compare application states, the constraints should be deserialized to their native data structure in the application. Thus, the key-value or key-values method defines a trigger condition. In the first case we define the trigger condition with the syntax *"key":"value"*. In the second case we define the condition with the syntax *"key": ["value1", "value2", "valueN"]*.

The *fault_dependencies* field defines the synchronization moments that a fault should have with other faults. Since we will instantiate the *Coordinator*, this synchronization can refer to the same *Agent* or to different *Agents*. If we did not have a *Coordinator*, we could only synchronize a fault with the faults that may happen in the same *Agent*. To define this synchronization, optionally the user can define the *after* field and the *before* field. The *after* field is a sequence of zero or more fault identifiers that means that all the faults that are defined in this field will have to be injected earlier, so that the fault we are configuring can be injected after these faults. The *before* field is also a sequence of zero or more fault identifiers that mean that all the faults that are set in this field will have to be injected later, so that the fault we are setting up can be injected before these faults.

```
{
  "agents": [
    {
      "agent_id": "0",
      "faulty": true,
      "scheduler": [
        {
          "fault_id": "delayFault_0",
          "fault_trigger_conditions": {
            "consensus_instance": [
              "5500",
              "5700",
              "5600"
            ],
```

```
            "consecutive_rounds": "20"
          },
          "fault_dependencies": {
            "before": [
              "crashFault_0"
            ]
          }
        },
        {
          "fault_id": "delayFault_1",
          "fault_trigger_conditions": {
            "consensus_instance_end": "6500"
          },
          "fault_dependencies": {
            "after": [
              "crashFault_0"
            ]
          }
        }
      ]
    },
    {
      "agent_id": "1",
      "faulty": true,
      "scheduler": [
        {
          "fault_id": "crashFault_0",
          "fault_trigger_conditions": {
            "consensus_instance": "6000"
          },
        }
      ]
    }
  ]
}
```

Listing 5.2: An Agent Fault Schedule Configuration File Example

As we mentioned in Section 4.2, fault configuration is divided into two processes. The process shown in Listing 5.2 refers to describing the application's state context and the synchronizations that are required to inject the fault. The application location, the faulty code to execute and how it will execute will be covered later in Section 5.3.1. In the configuration example presented in Listing 5.2, we intend to inject faults into two *Agents* in a distributed context where consensus mechanisms exist in a fault tolerance protocol. We intend *Agent 1* to crash as soon as it is at consensus instance 6000. That said, we want *Agent 0* to synchronize with *Agent 1's* state to execute delay faults with different delay times. Thus, we intend that before *Agent 1* crashes, *Agent 0* delays execution for a brief period on several consensus instances. Respectively in instances 5500, 5600 and 5700 and in the following 20 instances. Therefore, it is intended that the delay be executed in the intervals of instances [5500:5519], [5600:5619] and [5700:5719]. The

fault setting with *delayFault_0* identifier symbolizes this delay. Then, we want that after *Agent 1* crashes, *Agent 0* delays execution for an extended period on all instances until it halts delaying execution on instance 6500, which is what symbolizes the *delayFault_1* fault. With this setting, the user only needs to know the current execution consensus instance and compare it with the settings in the fault schedule of the respective *Agent*. We will see in Section 5.3.3 how this can be done.

With a total number of agents, a, all *Agents* that are not identified in the agent configuration file, between 0 and $a - 1$, will be left with the default configuration that will be characterized as correct and faultless agents in their schedule.

If required and for the sake of widespread use, both the trigger conditions and the dependencies can be defined in a generic way for all faults in a scheduler of a respective agent, by the *scheduler_trigger_conditions* and *scheduler_dependencies* fields, or in schedulers of all agents, by the *agents_trigger_conditions* e *agents_dependencies* fields.

```
{
  "agents_trigger_conditions": {
    "messages":[
        "PREPARE",
        "COMMIT"
    ]
  },
  "agents": [
    {
      "agent_id": "0",
      "faulty": true,
      "scheduler_trigger_conditions": {
        "consensus_intance_start": "1000"
        "consensus_instance_end": "I'm gonna be changed"
      }
      "scheduler_dependencies": {
        "before": [
              "crashFault_0"
            ]
      }
      "scheduler": [
        {
          "fault_id": "delayFault_1",
          "fault_trigger_conditions": {
            "consensus_instance_end": "2000"
          },
        },
        ...
      ]
    }
    ...
  ]
```

```
}
```

Listing 5.3: Widespread configurations example

The end configuration of a fault is determined by the depth level of the configurations. If there is a widespread like we shown above in Listing 5.3, and if we define other configurations for a specific fault within the fault scheduler, this fault will then be configured with the deepest configuration that replaces some of the configurations defined by their upper layers. The fault *delayFault_1* of *Agent 0* on Listing 5.3, will inherit the generalized trigger conditions for agents regarding *messages*, will inherit the generalized trigger condition for *Agent 0* regarding *consensus_instance_start*, will replace *consensus_instance_end* with a new value and finally will inherit the generalized dependencies for *Agent 0*.

5.1.1 gRPC service

The *Coordinator* establish *Agent* interactions through the gRPC interface presented in Listing 5.4. These interactions can be done to provide *Agent* schedulers, to check the current states of several *Agents* or to receive a notification that reports the injection of a fault in a respective *Agent*. The interaction to provide the *Agent* scheduler is not synchronized as it has no interference with the actions that must be taken when it comes to triggering a fault. However, the interaction of checking other *Agents'* states and the fault notification interaction have a direct influence with the faults that has been injected on all *Agents*. Thus, it is needed to deal with multi-threads problems, such as race conditions. Therefore, any method that need to deal with triggered faults list, must be prevented by a synchronization method. In our implementation we use ReentrantLocks to prevent multiple threads from entering these processes simultaneously.

```
service CoordinatorServices {
    rpc AgentConnection (AgentId) (AgentConnectionReply) {}
    rpc FaultDependenciesValidation (DependenciesServerMessage) returns
        (InfoReply) {}
    rpc NotifyFaultExecution (NotifyFaultExecutionRequest) () {}
}
```

Listing 5.4: Coordinator service interface (gRPC methods)

In *AgentConnection* request, the *Coordinator* receives the *Agent* identifier and based on the identifier sends the faulty property declaration and the corresponding fault scheduler, as we can see in Listing 5.5.

```
message AgentConnectionReply {
   bool faulty_agent = 1;
   repeated FaultServerMessage scheduler = 2;
}

message FaultServerMessage{
   string fault_id = 1;
```

```
    repeated TriggerConditionsServerMessage trigger_conditions = 2;
    optional DependenciesServerMessage dependencies = 3;
}

message TriggerConditionsServerMessage {
    string condition_key = 1;
    repeated string condition_values = 2;
}

message DependenciesServerMessage {
    repeated string before = 1;
    repeated string after = 2;
}
```

Listing 5.5: Protocol Buffers messages for AgentConnectionReply

In *FaultDependenciesValidation* request, *Coordinator* receives all the dependencies that refer to a particular fault in the Agent's schedule, as we can see in Listing 5.6. The *Coordinator* maintains a list with the faults that have already been triggered, and thus checks whether the dependencies it receives *before* and *after* are met in the current state. If there is no hindrance, *Coordinator* validates the dependencies by sending a successful response or an unsuccessful response otherwise.

```
message DependenciesServerMessage {
    repeated string before = 3;
    repeated string after = 4;
}

message InfoReply {
    bool successful_operation = 1;
}
```

Listing 5.6: Protocol Buffers messages for dependencies validation

In *NotifyFaultExecution* request, an *Agent* notifies that a fault has been injected. *Coordinator* receives the fault identifier and the agent identifier, as we can see in Listing 5.7. It stores this information internally in the list of injected faults and does not send anything to the *Agent*.

```
message NotifyFaultExecutionRequest {
    string fault_name = 1;
    string agent_id = 2;
}
```

Listing 5.7: Protocol Buffers messages for fault execution notifications

5.2 Agent

Agent is also developed in Java and implements a gRPC stub to serialize communications with the Coordinator services and also provides an RPC communications service to interact with the application nodes. Agent can be instantiated in two ways but in both initialization cases it will always need the port number definition to listen for communications made by the application nodes. The first way to instantiate the Agent is by defining the Coordinator, i.e. the Internet Protocol (IP) address and the listening port) as we see in Listing 5.8, which after the agent interaction will be responsible for delivering the configuration of his respective fault schedule.

```
$ java -jar agent.jar -p 9000 -c 10.10.10.10 -cp 9090
```

Listing 5.8: Running Agent on port 9000, defining the IP and port to listen on by the Coordinator

The other way is to replace the Coordinator definition with the definition of its own Agent fault schedule via a JSON file, as we see in Listing 5.9. However, this second option assumes an Agent-based Triggering Model as we described in section 4.4.2, and for this reason it is not possible to validate fault dependencies between multiple Agents.

```
$ java -jar agent.jar -p 9000 -f agent_1_scheduler_config.json
```

Listing 5.9: Running Agent on port 9000, with fault schedule settings in the agents_config.json file and assuming an Agent-based Triggering Model

In this case, the JSON file follows the same layout as was demonstrated for the Coordinator in Section 5.1, as we can see in Listining 5.10. The only difference is that only one specific Agent is defined.

```
{
  "agent_id": "1",
  "faulty": true,
  "scheduler": [
    {
      "fault_id": "delayFault_0",
      "fault_trigger_conditions": {
        "consensus_instance_start": "500"
      }
    },
    {
      "fault_id": "crashFault_0",
      "fault_trigger_conditions": {
        "consensus_instance_start": "650"
      },
      "fault_dependencies": {
        "after": [
          "delayFault_0"
        ]
```

```
        }
      }
    ]
}
```

Listing 5.10: Agent configuration file example agents_config.json

5.2.1 gRPC service

Interaction with an application node is established through the gRPC interface presented in Listing 5.11. This interaction can be to provide fault schedulers to the application, to check the dependencies that a fault has are valid, or to receive a notification that informs of the injection of a fault. The messages for providing a schedule (Listing 5.5) and fault injection notifications (Listing 5.7]) is very similar to what we have seen for the *Coordinator*.

```
service AgentServices {
    rpc ApplicationConnection (AgentId) (ApplicationConnectionReply) {}
    rpc VerifyDependencies (VerifyDependenciesRequest) returns (InfoReply) {}
    rpc NotifyFaultExecution (NotifyFaultExecutionRequest) () {}
}
```

Listing 5.11: Agent service interface (gRPC methods)

What differs from the *Coordinator* is the *VerifyDependencies* service. Although the messages are the same as we showed in Listing 5.6, when an application node asks the *Agent* for this service, the *Agent* is in charge of figuring out if the dependencies sent are internal dependencies and so it knows them because they are in its scheduler. Otherwise, if the *Coordinator* is defined it asks to the *Coordinator* to perform the check for external dependencies. For this to be possible, *Agent* keeps a list of its injected faults, and so checks the dependencies it receives "before" and "after" are fulfilled in their current state. After doing its check if it is not successful it immediately returns a unsuccessfull response to the application. If it is successful, in case there are still dependencies that it does not know about, then it communicates with the *Coordinator* to finish the check and sends the response depending on the *Coordinator's* response.

As happen in the *Coordinator*, the dependency checking process and the fault notification process have to be prevented by synchronization mechanisms to deal with multi-threads problems.

5.3 Hook

To finish the fault configuration process and to inject the faults into the target application, we use Aspect Oriented Programming (AOP) mechanisms such as advices and joinpoints, as we saw in Section 3.1. Based on these concepts, we can get the interaction of the faults with the application using *joinpoints* with *advices*. That is, we specify a target method, which through an execution preceding or following the method call, will execute faulty code in the application.

It is through these mechanisms that the fault configuration defines the *what*, *how* and *where* properties described in the Section 4.2, and that complements the existing configuration in the *Agent*, which defines the *when* property.

Zermia hooked the faults into the application code after going through a compilation (if necessary) and before the application starts running. The development of the faults will be as intrusive in the application code as the AOP mechanism used in the application language.

To establish communication from the hooked faults to the *Agent*, it was necessary to develop a grpc stub to communicate with the Agent's services. Since we implemented an Independent Triggering Model to trigger faults that does not have dependencies on other faults, and therefore does not need to communicate with its *Agent*, it was necessary to develop mechanisms to achieve an application-independent state on each node of the application.

These two mechanisms represent the *Zermia* framework that is present and runs along with the application. The Figure 5.1 show the Unified Modeling Language (UML) Class Diagram regarding the data structures used to implement the *Zermia* framework in the application, which supports the fault implementation. The *AgentStub* structure has the task of performing the communications with the *Agent* services and sanitizing the responses received. The *ZermiaRuntime* structure stores the application's independent state and is responsible for determining whether can it enable the Independent Triggering Model. It is based on a Singleton class, which can be used in multiple threads and locations, but its instance will be unique.

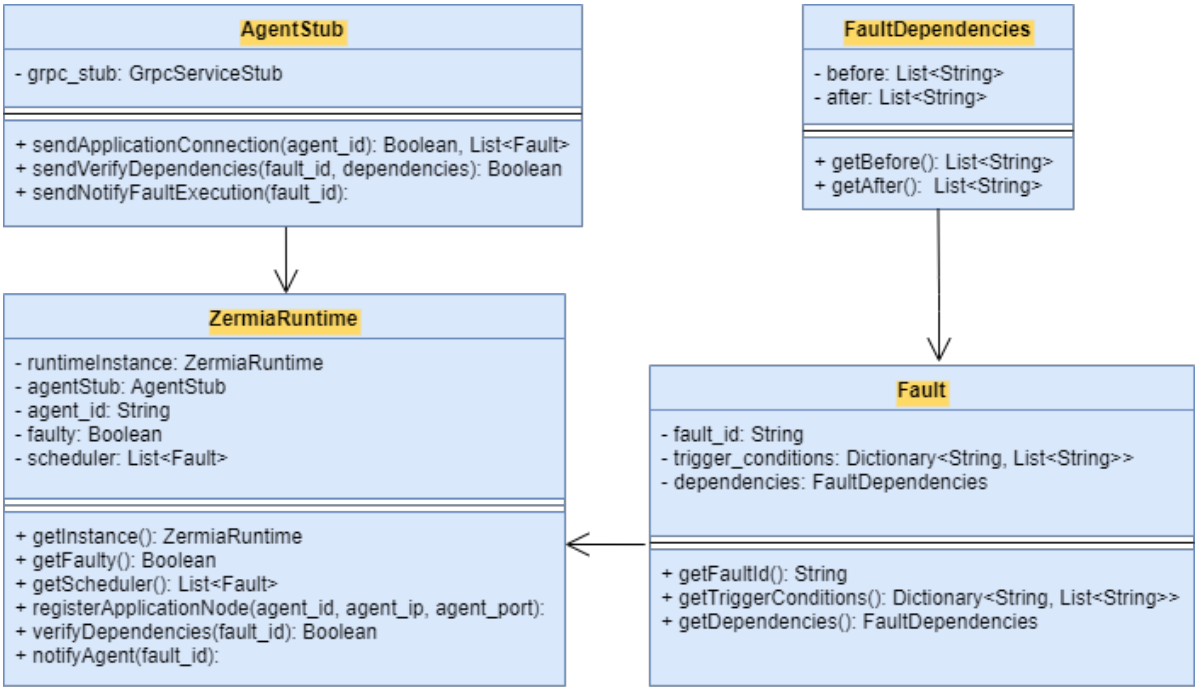


Figure 5.1: Zermia framework hooked in application

5.3.1 Faults

The fault procedure follows the information gathering regarding the current application state or other application-independent information. Whether there is a match of the gathered information with the application independent *Zermia* state that contains the exact information that defines when the fault should be injected, the application executes the faulty code injected by *Zermia* and notifications are sent via *Zermia* framework hooked in application.

Regardless of the information gathered and the code that it will be injected, we define a fault as an object, using an Object Oriented Programming (OOP) approach. The faults must implement an interface that symbolizes the method for checking the trigger conditions and the method that will inject the code into the application, respectively the *canTrigger()* method and the *executeFault()* method. To facilitate the faults development to the user, we developed an *InjectableFault* class that is characterized by an fault identifier and contains several methods that interact with the *Zermia* framework that facilitate the implementation of the *canTrigger()* method, as we see in Figure 5.2.

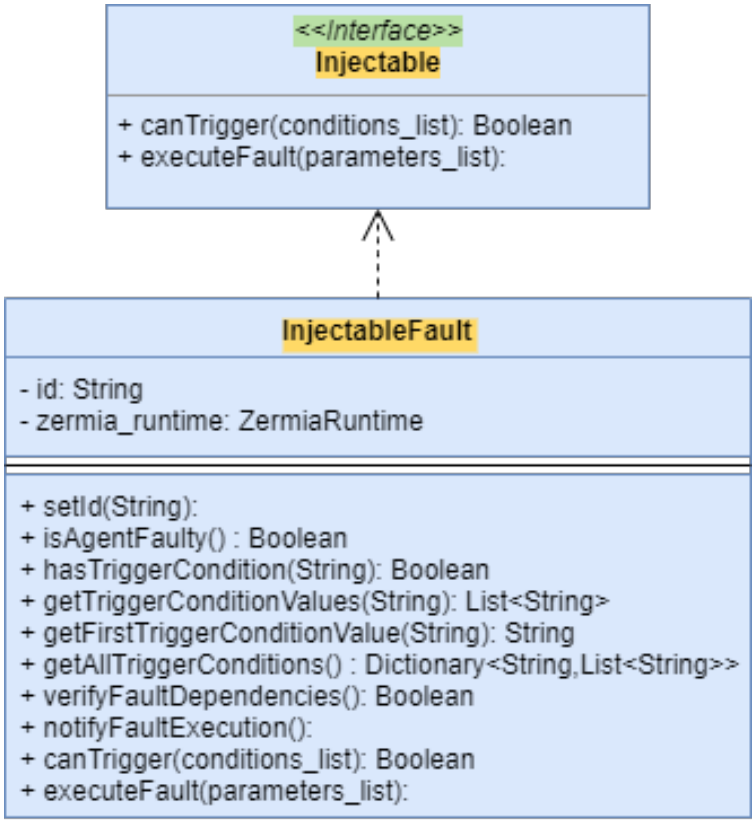


Figure 5.2: Fault abstraction

Therefore the user will have to develop a class, characterized as an *Aspect*, that extends *InjectableFault* and implements the *canTrigger()* and *executeFault()* methods. In the implementation of the *canTrigger()* method, the user should use the *InjectableFault* methods for the triggering conditions to be able to match the application's state instance with the configuration made in

the *Zermia* framework. These are:

- *hasTriggerCondition()* - Receives the condition (key) identifier and returns true if the condition was configurated on Agent side in the fault trigger conditions associated with fault identifier.
- getTriggerConditionValues() - Receives the condition (key) identifier and retrieve all trigger condition values that was configurated on Agent side associated with fault identifier.
- getFirstTriggerConditionValue() - Receives the condition (key) identifier and retrieve the first trigger condition value that was configurated on Agent side associated with fault identifier.
- getAllTriggerConditions() - Retrieves all trigger conditions configurations associated with fault identifier.

The *verifyFaultDependencies()* method is used to automatically verify the fault dependencies, so the user can call this method if they wish, and it will automatically determine which components they need to call to get a dependency verification.

5.3.2 Predefined Faults and Extensibility

Using the *InjectableFault* interface, the user must implement *executeFault()* which will have the code that will be injected into the application. To implement custom faults the user just need to implement *executeFault()* with the code they want. To simplify the developer interaction, *Zermia* provides a set of predefined faults that can be invoked on *executeFault()* method. These faults come from the previous version and were intended for assessing the properties of distributed systems as they are related to communication faults but can be adapted to assess any kind of application. They were described in Section 3.4.

5.3.3 Java implementation

We developed a *Zermia* library in Java, which implements the classes we present in the Figures 5.1 and 5.2. To assess Java applications, users can develop faults with the support of this library and using the AspectJ tool that uses AOP. In Listing 5.12 we can see an example of a *Crash Fault*. This example shows the continuity of the crash fault development referred in Listing 5.10, and we are gathering the consensus information through the arguments given in the application's target function shown in Line 2. After this, the consensus instance is compared with the settings we made in Listing 5.10. If the instance state is higher than the one that was configurated, the trigger condition check is successful, and, since this fault depends on another fault, we check if the dependencies are met. If all goes according to plan the *executeFault()* method is called which will eventually stop the execution of the Java process.

Finally, through the *joinPoint.proceed()* method the target method can proceed and execute its predestined code. Since we are experiencing a *Crash Fault*, when the application executes the faulty code, this instruction will no longer be executed because the process will terminate. However, this demonstrates the notion of how the execution would eventually follow with its previously defined instructions with the *joinPoint.proceed()* instruction, and how we can use this to execute the faulty code after or before the target method call.

```
@Aspect
public class CrashFault extends InjectableFault {
    @Around("execution(*
        bftsmart.communication.server.ServersCommunicationLayer.send*(..))")
    public void sendConsensusMessageEntryPoint(ProceedingJoinPoint joinPoint)
        throws Throwable {
        super.setId("crashFault_0");

        Object[] arg = joinPoint.getArgs();
        Integer consensus_instance = (Integer) joinPoint.getArgs()[1];
        ArrayList<Object> conditions = new ArrayList<>();
        conditions.add(consensus_instance);

        if ( super.isAgentFaulty() && canTrigger(conditions) ) {
                super.notifyFaultExecution();
                executeFault(new ArrayList<>());
        }
        joinPoint.proceed(arg);
    }

    @Override
    public Boolean canTrigger(ArrayList<Object> conditions) {
        Integer current_instance = (Integer) conditions.get(0);
        if ( super.hasTriggerCondition("consensus_instance_start") ) {
            Integer initial_configured_instance =
                Integer.valueOf(super.getFirstTriggerConditionValue("consensus_instance_start"))
            if ( current_instance >= initial_configured_instance &&
                super.verifyFaultDependencies() )
                return true;
        }
        return false;
    }

    @Override
    public void executeFault(ArrayList<Object> params) {
        System.exit(-1);
    }
}
```

Listing 5.12: Crash Fault Java implementing following the example shown in Listing 5.10

5.3.3.1 Bootstrap Fault

To get the *Zermia* structures initialized in the application node, it is necessary to register the node in its *Agent*. For this purpose, during bootstrap and by using a special fault associated with the application node main method, the user must implement the *Boostrap Fault*, which is responsible to interact with the *Agent* (using the gRPC services) to instantiate the necessary *Zermia* data structures for maintaining its independent state.

```
@Aspect
public class BoostrapFault {
    ZermiaRuntime zermia_runtime = ZermiaRuntime.getInstance();

    @Before("execution(* *.main(..))")
    public void mainEntryPoint(JoinPoint joinPoint) {
        String[] args = (String[]) joinPoint.getArgs()[0];
        String agent_id = args[0];
        String agent_ip = "127.0.0.1";
        Integer agent_port = 0;

        if ( agent_id.equals("0") )
            agent_port = 9000;
        else if ( agent_id.equals("1") )
            agent_port = 9010;
        else if ( agent_id.equals("2") )
            agent_port = 9020;
        else if ( agent_id.equals("3") )
            agent_port = 9030;

        zermia_runtime.registerApplication(agent_id, agent_ip, agent_port);
    }
}
```

Listing 5.13: Bootstrap Fault Java example

In Listing 5.13, we have an example where we take the first argument of the main function (which corresponds to the identifier of the application node) and use this identifier as the identifier of the *Zermia Agent* used to communicate with our framework. If we have several nodes that must be registered to different Agents, we can use the node identifier to differentiate each *Agent* node, where they will communicate through the port they are listening on. Finally, to register the application node in the *Zermia Agent* we call the function *registerApplication()* from the *ZermiaRuntime* class where we take as arguments the *Agent* identifier, the IP address and the *Agent* port.

5.3.3.2 Compilation

The *Zermia* library binaries should be compiled by the same Java Development Kit (JDK) version as the target application to avoid conflicting version incidents. To compile the *Zermia*

library we use the Maven automation and compilation tool. We built a configuration file for compiling called *pom.xml*. By default, the library is compiled for *JDK* version 11. To change the *JDK* version you need to change the *pom.xml* file as shown in Listing 5.14.

```
<properties>
        <maven.compiler.source>desired-JDK-version-here</maven.compiler.source>
        <maven.compiler.target>desired-JDK-version-here</maven.compiler.target>
</properties>
```

Listing 5.14: Agent configuration file example agents_config.json

That said, once the maven installation, inside the directory containing the *pom.xml* file, we can run the command presented in Listing 5.15 which will lead to a new target folder containing the Java ARchive (JAR) with all the classes from the *Zermia* library.

```
mvn clean package
```

Listing 5.15: Zermia framework compile command for Java

5.3.4 Python implementation

We also developed a *Zermia* library in Python and it implements the classes we present in the Figures 5.1 and 5.2. Since we have not found any tool that implements an AOP approach in Python, such as AspectJ in Java, from the research we have done the best way to implement the AOP paradigm in Python is through decorators. The solution with decorators is very similar with the AOP approach, as we can see in Listing 5.16. We define a class that implements the *InjectableFault* interface and defines the *___call___()* method that makes the class instances behave like methods that can be called. This method takes as parameter another method that will be the target method of the application.

Inside this function the user can define the advice with the same arguments passed by the target method and that will be called by the *___call___()* method. This advice will be as a proxy method, so whenever the target method is called the user can invoke the target method or not. In Listing 5.16, we have the same crash fault implementation example that we performed in Java in Listing 5.12. In the *__proxy__advice()* method we start by collecting the information regarding the current consensus instance, which is later inserted into the *canTrigger()* method that checks the trigger conditions and dependencies of the *crashFault_0* fault. Finally we call the target application method *func()* that we receive as an argument by the *___call___()* method, store the returned variables so we can then return them and proceed with the normal operation of the application, just like *joinPoint.proceed()* in Java.

```
class CrashFault(InjectableFault):
    def __call__(self, func):
        self.setId("crashFault_0")
```

```
        def _proxy_advice(*args, **kwargs):
            instance = args[1]
            conditions = []
            conditions.append(instance)

            if super().isAgentFaulty() and self.canTrigger(conditions):
                self.notifyFaultExecution()
                self.executeFault(None)

            results = func(*args, **kwargs)
            return results
        return _proxy_advice

    def canTrigger(self, conditions):
        current_instance = conditions[0]

        if super().hasTriggerCondition("consensus_instance_start"):
            initial_configured_instance =
                int(super().getFirstTriggerConditionValue("consensus_instance_start"))
            if current_instance >= initial_configured_instance and
                super().verifyFaultDependencies():
                return True
        return False

    def executeFault(self, params):
        sys.exit()
```

Listing 5.16: Crash Fault Python implementing following the example shown in Listing 5.10

The difference between the Python and Java implementation is that in Python we have to manually develop the moment we want to execute the target function by calling the target method inside the advice, whereas in AspectJ we used *After*,*Before* or *Around* annotations and then used *joinPoint.proceed()* for continue with the normal execution application method. In this Python implementation we also have to be a bit more intrusive since we have to define the *JoinPoints* inside the application's source code whereas in AspectJ we would do the *JoinPoints* definition without having to mess with the source code. Despite this, this definition minimally affects the source code, since it is just a decorator in the target method signature annotated with the @ character followed by the developed class name, as we can see in Listing 5.17.

```
@CrashFault()
def do_something_important(x):
    ...
    return x,y
```

Listing 5.17: Associating a Crash Fault to a target method in Python

5.3.4.1 Bootstrap Fault

Also in Python, we must define a specific fault for the initialization of the *Zermia* structures on the application nodes. As we saw in Section 5.3.3.1, this special fault called *BootstrapFault* defines the *Agent* properties that the application node will interact with. Listing 5.18 illustrates the same bootstrap fault implementation example we saw in Listing 5.13 but now in Python.

```
class BootstrapFault():
    def __call__(self, func):
        def bootstrapAdvice(*args, **kwargs):
            agent_ip = "127.0.0.1"
            agent_port = 0
            agent_id = args[0]
            if agent_id == "0":
                agent_port = 9000
            elif agent_id == "1":
                agent_port = 9010
            elif agent_id == "2":
                agent_port = 9020
            elif agent_id == "3":
                agent_port = 9030

            ZermiaRuntime().get_instance().registerApplication(agent_id,
                agent_ip, agent_port)
            result = func(*args, **kwargs)
            return result
        return bootstrapAdvice
```

Listing 5.18: Bootstrap Fault Python example

As we saw in Listing 5.17, the user must inject the BootstrapFault in the main() method of the target application, as we can see in Listing 5.19.

```
@BootstrapFault()
def main(args):
    ...
```

Listing 5.19: Associating the Boostrap Fault to the main method in Python

5.4 How to assess the behavior of Fault Tolerance protocols

There are several ways to assess fault tolerance protocols. As each protocol has its own particularities it is difficult to characterize the conditions needed to evaluate the resilience of this protocols. A simple consensus instance can have different nomenclatures in different protocols. For this reason, we opted to let the *Zermia* user decide the trigger conditions.

However, there are patterns and scenarios for assessing the resiliency of these types of protocols

that are independent of their implementation. The *Zermia* predefined faults were designed with the intention of being used in a general way to assess these distributed applications. By injecting these faults, it is possible to evaluate the reactive communications that arise to circumvent malicious behaviors.

Thus, the correct behavior of the protocol must be defined, all phases must be determined, and for each phase, each of the predefined fault types defined in Zermia must be applied.

Other missing experiments that may help in assessing these systems are:

- Shutting down the leader server.
- Shut down one or more backup servers.
- Delaying the sending of packages regarding voting from the leader server election.
- Delay sending the periodic communications (heartbeat communications) that define the liveness of the nodes and establish leadership/authority.
- Try to break a cluster into different subsets and see if these creates two leads.

The next chapter will present some of these experiments by assessing the RAFT protocol.

5.5 Script setup for compiling and running

Zermia has an executable script written in Python3 that allows the automation of assessing fault injection experiments either in Java or Python. From a configuration file where information such as the *Zermia* server structure used (*Coordinator* and *Agents*), the target application and the code developed for the faults are provided.

The script creates an output directory which contains files with the output of the processes started by the script (*Coordinator*, *Agents*, application servers and application clients). For the assessing of Java applications, since we must compile the developed faults and then insert the compiled faults into the application via the AspectJ compiler, the output directory will also contain a directory with the compiled faults and a directory representing the application alongside the compiled faults. That said, it will be from this directory that we execute the Java application with the faults hooked. For Python, since we do not have to compile source code and the faults must be already within the application project, the output directory will only contain the process monitoring files generated by the script. Listing 5.20 shows example commands for running experiments from the script.

```
$ python3 setup.py -cfg config.ini  --run_python -o
    python_experiment_output_dir_name
$ python3 setup.py -cfg config.ini --run_java -o java_experiment_output_dir_name
```

Listing 5.20: Running experiences from setup Zermia script

Mandatory for running the script is to provide a configuration file in *.ini* format through the *-cfg* argument and a name for the output directory through the *-o* argument. You must also define the type of language that the application is going to use (Python or Java). The language differentiation is set using the *–run_python* or *–run_java* argument.

Optionally the user can define the binary paths of the programming languages that the script will eventually use, such as:

- Path to Java binary by the *–java* argument.
- Path to Java compiler by the *–javac* argument.
- Path to AspectJ compiler by the *–ajc* argument.
- Path to Python3 binary by the *–python* argument.

5.5.1 Configuration file

The configuration file must be organized by sections, where *[zermia]* represents the *zermia* section. The sections contain mandatory fields and optional fields. The different sections that user need to configure are:

- ***zermia*** - This is where needs to define the paths to the *Coordinator* and *Agent* binaries. The definition of the attribute referring to the *coordinator* is optional and should be set when the faulty experiments have multi-agent dependencies.
  - The zermia section fields are:
    - ***agent*** - Represents the *Zermia Agent* binary path.
    - ***n_agents*** - Represents the *Agents* number that experiment will use.
    - ***coordinator*** - Represents the *Zermia Coordinator* binary path.
    - ***boot_time*** - It has an optional field that represents the waiting time for booting *Zermia* infrastruture in seconds.
- ***coordinator*** - The coordinator section is defined when it is intended to inject faults with dependencies between multiple *Agents*.
  - The coordinador section fields are:
    - ***ip*** -Represents the ip address of the *Coordinator* host.
    - ***port*** - Represents the port that *Coordinator* is listening.
    - ***agents_scheduler*** - Represents the *Agents* fault schedulers configuration JSON file.
- ***agent-X*** - Each agent will have its own section identified by a number that symbolizes *X*. There will be as many sections as defined in the *n_agents* field in the *zermia* section.

  - The agent section fields are:
    * ***ip*** - Represents the ip address of the *Agent-X* host.
    * ***port*** - Represents the port that *Agent-X* is listening.
    * ***scheduler*** - Is is an optional field that must be defined when there is no *Coordinator* host. It represents fault scheduler configuration JSON file of *Agent-X*.

- ***aspects*** - This section is only used in Java application experiments and defines the source code developed for implementing the faults and the dependencies needed to compile that code.
  - The aspects section fields are:
    * ***source*** - Represents the root folder that has all packages of aspects source code.
    * ***dependencies*** - Represents all dependencies on the aspects source code. This field represents a list, and each item list can be defined as a directory with multiple JAR files, by a single var file, or by a compiled code package. Unavoidably, it must contain the AspectJ library and the *Zermia* library and may obviosly contain more dependencies.

- ***application*** - This section defines all the characteristics needed to be able to run the application. Mainly, the number of servers and clients that the experiment will use.
  - The application section fields are:
    * ***root_folder*** - This field is only used for testing Java applications and represents the root directory that contains the entire application project.
    * ***binary*** - This field is only used for testing Java applications and represents the compiled application Java code. It must be a JAR or a folder with the packages that contains the compiled Java classes.
    * ***dependencies*** - This field is only used for testing Java applications and represents all dependencies of target application. It represents a list, and each item list can be defined as a directory with multiple JAR files, by a single var file, or by a compiled code package. It is a must that a dependency folder is in this field and this is because it will be in this directory where we add the necessary dependencies to the fault implementation code.
    * ***n_replicas*** - It as an optional field and represents the replicas number needed by application server.
    * ***server_execution_path*** - It as an optional field and represents the working directory to execute server application.
    * ***server_boot_time*** - It as an optional field and represents the waiting time in seconds for proceeding execution after all replicas are up.
    * ***n_clients*** - It as an optional field and represents the clients number needed by application.

    - ***client_execution_path*** - It as an optional field and represents the working directory to execute client application.

- ***replica-X*** - Each server in the application will have its own section identified by a number that symbolizes *X*. There will be as many replica's sections as defined in the *n_replicas* field in the *application* section.
  - The replica section fields are:
    - ***execution_command*** - Represents the command that executes to the *Replica-X* and should follow the path described in the *server_execution_path* field of the application section.
    - ***boot_time*** - It is an optional field and represents waiting time in seconds for booting the *Replica-X*.

- ***client-X*** - Each client in the application will have its own section identified by a number that symbolizes *X*. There will be as many client's sections as defined in the *n_clients* field in the application section.
  - The client section fields are:
    - ***execution_command*** - Represents the command that executes the *Client-X* and should follow the path described in the *client_execution_path* field of the application section.
    - ***boot_time*** - It is an optional field and represents the execution time of the *Client-X*.

In Listing 5.22 we can see an example of a Java application testing experiment that will need two Agents, the Coordinator, four processes for the application server and one process for the application client. In Listing 5.21 we can observe an example of a Python application testing experiment that will need only one Agent and one process for the application server.

```
[zermia]
agent = "./agent/my_zermia_agent-1.42.1-jar-with-dependencies.jar"
n_agents = 1
boot_time = 0.5

[agent-0]
ip = "127.0.0.1"
port = 9000
scheduler = "./agent/agent-0_scheduler.json"

[application]
n_replicas = 1
server_execution_path = "/home/trainee/my-simple-app"
server_boot_time = 2
n_clients = 0
```

```
[replica-0]
execution_command = "runscripts/startReplicaYCSB.sh 0"
boot_time = 0.25
```

Listing 5.21: Script configuration file for Python applications

```
[zermia]
coordinator =
    "./coordinator/my_zermia_coordinator-1.42.1-jar-with-dependencies.jar"
agent = "./agent/my_zermia_agent-1.42.1-jar-with-dependencies.jar"
n_agents = 2
boot_time = 0.5

[coordinator]
ip = "127.0.0.1"
port = 9090
agents_scheduler = "./coordinator/crashFault_test_bftsmart.json"

[agent-1]
ip = "127.0.0.1"
port = 9010

[agent-3]
ip = "127.0.0.1"
port = 9030

[aspects]
source = "./fault_hooks/src/main/java"
dependencies = [
    "./fault_hooks/lib"
    ]

[application]
root_folder = "./bft-smart-test"
binary = "./bft-smart-test/bin/BFT-SMaRt.jar"
dependencies = [
    "./bft-smart-test/lib"
    ]
n_replicas = 4
server_execution_path = "./bft-smart-test"
server_boot_time = 8
n_clients = 1
client_execution_path = "./bft-smart-test"

[replica-0]
execution_command = "runscripts/startReplicaYCSB.sh 0"
boot_time = 0.25

[replica-1]
execution_command = "runscripts/startReplicaYCSB.sh 1"
boot_time = 0.25
```

```
[replica-2]
execution_command = "runscripts/startReplicaYCSB.sh 2"
boot_time = 0.25

[replica-3]
execution_command = "runscripts/startReplicaYCSB.sh 3"
boot_time = 0.25

[client-0]
execution_command = "runscripts/ycsbClient.sh"
```

Listing 5.22: Script configuration file for Java applications

Chapter 6

Experiments

In this section we present experiments that are meant to demonstrate the use of *Zermia* and its impact on applications. We decided to choose two different applications developed in different programming languages. The first experiment is a simple single-process application developed in Python, where we are injecting basic faults over time. The second experiment is an implementation of the RAFT protocol developed in Java where several faults were injected to validate the behavior and resilience of the protocol.

6.1 Simple Python application testing

We assessed the latest *Zermia* version with a single-process application developed in Python. The application is quite simple. Infinitely, by generating random numbers in the range [0:500000], it checks if the generated number is a prime number and if so, increments the count of primes found.

We chose to assess an application of such simplicity to demonstrate simple injection experiments. In terms of fault triggering conditions, we chose to define execution timings to determine the application's execution states.

6.1.1 Experimental setup

In this experiment we used Ubuntu 20.04.5 LTS virtual machine with 11GB of Random Access Memory (RAM) and eight Central Process Unit (CPU) core. The native machine has one AMD Ryzen 7 2700X Eight-Core 3.7GHz Processor and uses 16GB of RAM with a Windows 10 x64 operating system.

We do the fault injections locally on this machine with the initialization of one *Zermia Agent*.

6.1.2 Baseline

Figure 6.1 shows the evaluation of the application through various experiment measuring the throughput of the number of primes per second. In Figure 6.1 a), we can observe the behavior of the application in a fault-free experiment. This result derived from an average of ten runs of the application. We can observe that the common behavior of the application goes through the random generation of 100 to 125 prime numbers per second.

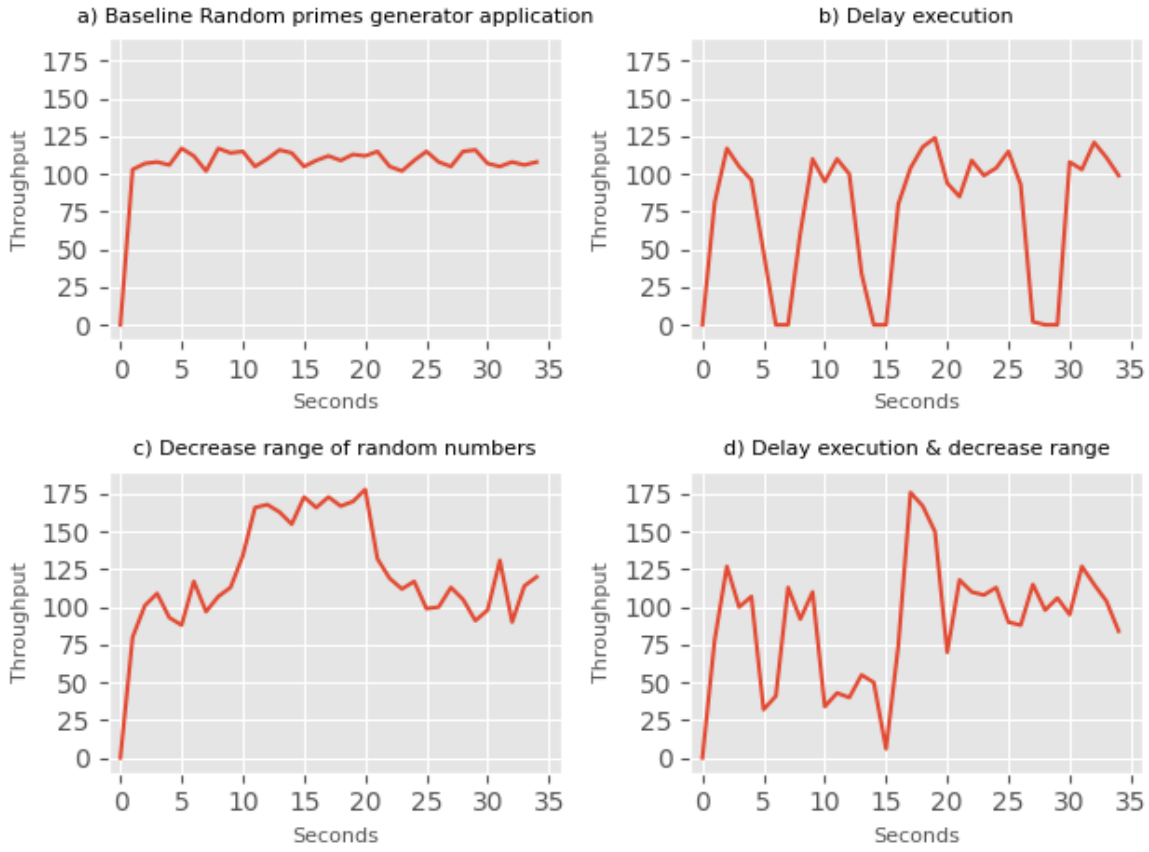


Figure 6.1: Simple Python application experiments

6.1.3 Delay Fault

In Figure 6.1 b) we can observe the experiment of delay fault injection in the execution. We triggered this fault at fifth, thirteenth and twenty-seventh seconds and it has delay intervals of three seconds. The result shows three instants where the random prime generation is zero, which coincides with instants fifth, thirteenth and twenty-seventh seconds.

6.1.4 Range Modification Fault

In Figure 6.1 c) we can observe the fault injection experiment of modifying the random number generation range. We triggered this fault from the tenth second until the twentieth second. During the fault execution the random number generation range becomes reduced to [0:150000]. This interval reduction increases the probability of randomly generating a prime number. The result shows that from the tenth second there is an exponential increase in the generation of prime numbers and that it remains constant until the twentieth second, where it finally decreases and returns to the standard throughput.

6.1.5 Delay and Range Modification Faults

In Figure 6.1 d), we can observe the experiment of injecting delay faults and modifying the random number generation range. The Delay fault is similar to the one we saw in Figure 6.1 b), but it only triggers at the fifth and fifteenth seconds, and now it will have delay intervals of 1.5 seconds. The Interval Modification fault is exactly the same as in Figure 6.1 c), at tenth second until the twentieth second, the random number generation interval becomes reduced to [0:150000]. However, additionally to the Range Modification Fault we will append an event dependency. This fault will only trigger after the Delay Fault has been executed. So, there is a dependency between internal faults.

The result of Figure 6.1 d) shows that at fifth second there is a local minimum coming from the Delay Fault. Then, from tenth second the throughput drops to half of the baseline value and remains constant at these values until the fifteenth second. This behavior comes from the dependency checking that the Range Modification Fault performs. Since the of the Range Modification fault execution is between the tenth and twentieth seconds, the Range Modification Fault contacts its *Agent* to know if the Delay Fault has already been injected. Communicating with the *Agent* requires the sharing of the machine's processing time between running the application and checking injector dependencies, which is noticeable between the tenth and twentieth seconds.

Again, at fifteenth second, the throughput has a local minimum coming from the Delay Fault. At this time, the Delay Fault is eventually terminated and notifies the Agent of the fault injection's completion. This unlocks the Range Modification fault which is noticeable between the fifteenth to twentieth seconds, as the throughput reaches values close to what it reached in Figure 6.1 c) at the same interval. After the twentieth second, the Range Modification Fault is eventually terminated, and we observe the application standard throughput from that instant onwards.

6.2 RAFT testing

Evan Sá[67] used *Zermia* in assessing the RAFT fault tolerance protocol. They relied on a Java implementation of RAFT and, evaluated the algorithm's performance in number of operations performed per second. For this evaluation they implemented a Yahoo! Cloud Serving Benchmark (YCSB)[68] benchmark, which builds a key-value store responsible for receiving and handling requests from the benchmark such as inserting, removing, or updating data.

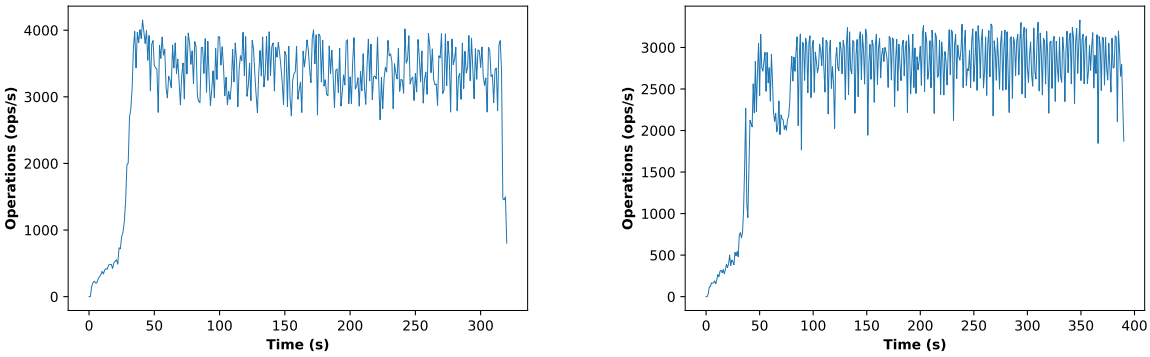
6.2.1 Experimental Configuration

The tests were conducted on a single machine on the Linux operating system that used the Ubuntu 22.04.1 LTS distribution. The embedded processor was the quad-core i7-7700HQ CPU and used 16GB of RAM.

As we saw in Section 2.4.3.2 the RAFT protocol needs $n \geq 2f + 1$ replicas to manage up to f faults. The experiment setup was divided into two clusters where one can manage $f = 1$ faults and the other $f = 2$ faults. Respectively the number of nodes in the clusters are, four replicas and another with six replicas. For easy identification in the following sections, we abbreviate each cluster with Cluster of four nodes (C4) and Cluster of six nodes (C6).

Finally, the YCSB configuration has fifty clients sending in total one million transactions into the system.

6.2.2 Baseline results



(a) Cluster of 4 replicas

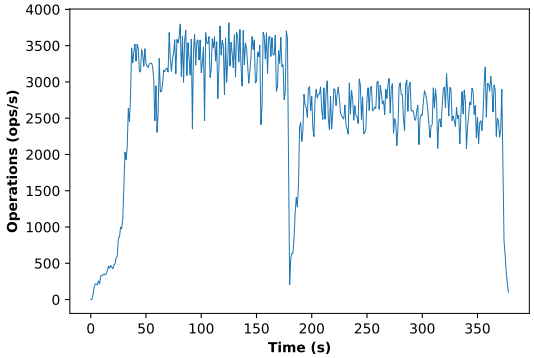
(b) Cluster of 6 replicas

Figure 6.2: Performance obtained by 1 million requests

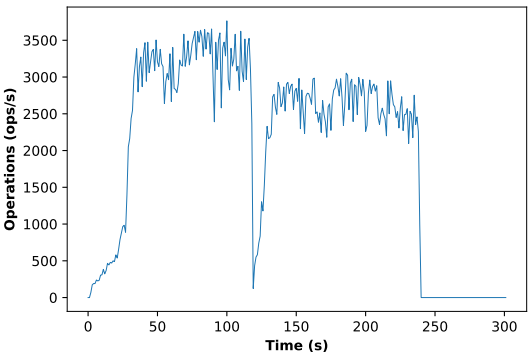
Figure 6.2 shows the standard performance obtained using the script setup mentioned in 5.5 without any fault injection. These results served as a baseline, to analyze the same setup but with the injection of certain faults and then try to observe the implications that each fault generates by comparison with the baseline.

6.2.3 Crash Fault

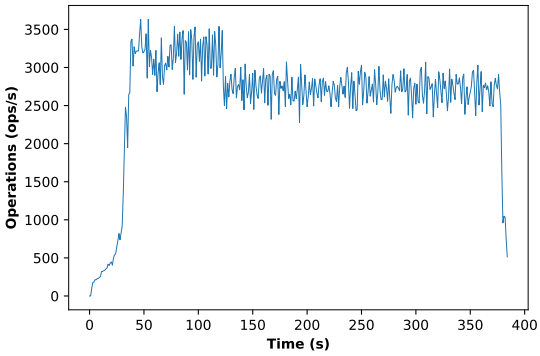
This experiment consists of forcing a replica to shut down and the trigger conditions are related to the size of the state machine log. Since per configuration, the clients will send one million requests in total, on the state machine log this reflects an index of the same length. The trigger condition used in this experiment was the last index used in the state machine log. This type of fault was tested by injecting it into the primary (Leader) and Follower replicas.



(a) Leader crash for trigger condition: 500000



(b) Leader crash for trigger condition: 300000 and 600000



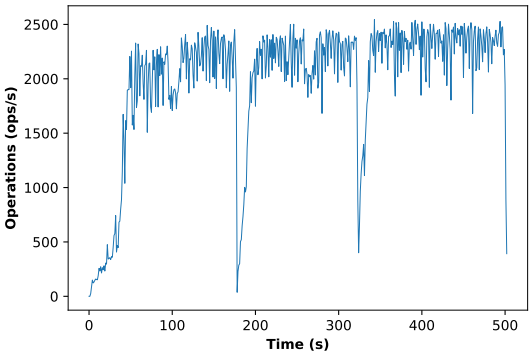
(c) Follower crash for trigger condition: 300000

Figure 6.3: Performance obtained by 1 million requests by injecting a crash at specific points (C4 configuration)

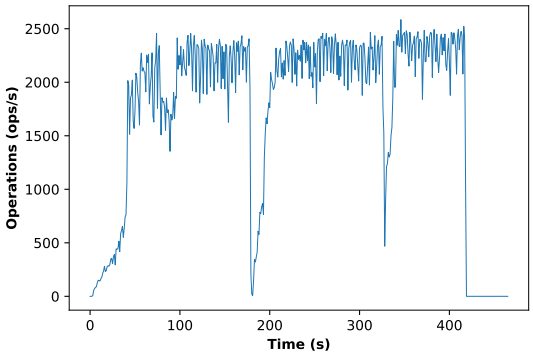
Figure 6.3, illustrates experiments of the C4 and shows that the crash of a server decreases the system performance to values close to zero for a few seconds. After the crash, the system never managing to recover the performance it had before the fault injection.

In Figures 6.3a and 6.3b, we illustrate the injection of crash faults on primary replicas. In Figure 6.3a, the Raft algorithm was quickly able to recover from first crash fault, electing a new leader and continue serving client requests. However, in Figure 6.3b a second fault was injected at log index six hundred thousand to crash the new leader server, remaining only two up of four servers, which breaks the protocol invariant $n \geq 2f + 1$. This is because to be able to tolerate two faults, the number of nodes in the system would have to be at least five. Figure 6.3c illustrate the

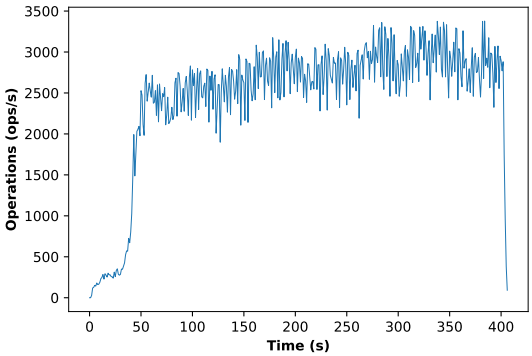
injection of crash fault on follower replica, and it display the triggering of the fault at log index three hundred thousand to a follower node, which does not a big impact on the throughput.



(a) Leader crash for trigger condition: 300000 and 600000



(b) Leader crash for trigger condition: 300000, 600000 and 800000



(c) Follower crash for trigger condition: 300000 and 600000

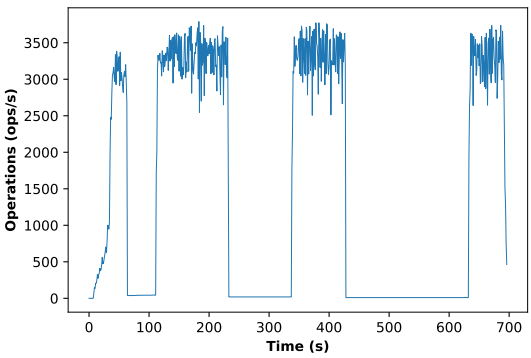
Figure 6.4: Performance obtained by 1 million requests by injecting a crash at specific points (C6 configuration)

Figure 6.4 shows results derived from the same experiment as above but on C6 and then we injected one more crash fault in each experiment. In comparison, we notice a slight improvement in throughput after the injection of the crash faults. In other words, system recovery is much more effective. In Figures 6.4a and 6.4b, the behavior during leader crashes is the same, but with an improvement in the performance when the protocol establishes a new leader. In Figure 6.4c, even with the loss of two followers nodes, it had no effect on system performance.

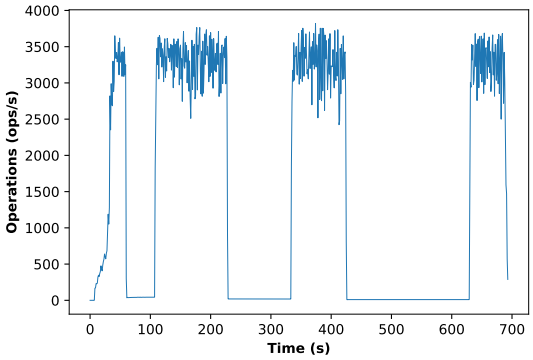
To conclude, in both clusters, we verify target RAFT implementations respects the security properties related to its configuration, i.e. the formula $n \geq 2f + 1$.

6.2.4 Delay Fault

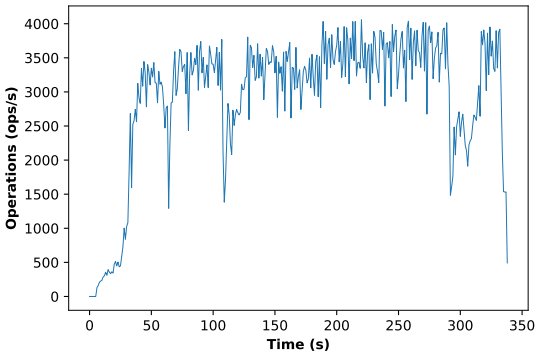
In this experiment, the objective is to inject delays on replicas execution before managing certain client requests. The trigger condition will be the same as in the previous Crash Fault, i.e., the last replicated log index. Different delays injected at separate times of the system execution will be shown, to get an idea of the system's latency because it is an essential attribute that Raft needs to have for practical systems.



(a) Leader delay 20,50 and 100 ms



(b) Delay all nodes 20,50 and 100 ms



(c) Delay all followers 20,50 and 100 ms

Figure 6.5: Results obtained through C4

In the three tests performed, we triggered the faults in three different instants. At log index number 100000, 500000 and 800000, and lasted for 2000 requests. For each interval we injected respectively delays of twenty milliseconds, fifty milliseconds, and one hundred milliseconds. The cluster used was the C4. Figure 6.5 shows the results obtained after injecting these delays. In Figure 6.5a the target was the leader replica, in Figure 6.5b all replicas (leader and followers) and in Figure 6.5c only the followers.

It is possible to verify that in the tests involving the leader replica (Figures 6.5a and 6.5b, the decrease in performance is extremely high.

6.2.5 Delaying the Leader Election

In this fault experiment we want to verify how long it takes for the first leader to be elected after injecting a certain delay. For this purpose, before the target application procedure *RequestVote* message sending, a delay execution is injected into it.

Tables 6.1 and 6.2 shows the results obtained after running the C4 and we performed six tests to have a more average value.

The values represented in each replica refer to the average time that a replica acknowledged the election of a leader or, in the replica leader, the time it took to become a leader. Concerning *Term* column, it represents how many elections rounds it took to elect a leader. The results in Table 6.1 were run without the presence of faults and the election of the leader almost occurs in the first term.

In Table 6.2 delays of 100, 250, and 280 milliseconds were injected before the vote request occured. The candidate timeout used by this RAFT implementation is equal to 300 milliseconds. The closer the delay is to this value, the longer it will take for system elects the replica leader. The results show that from 250ms to 280ms, the increase was substantial. Tests from 290ms and above already created infinite loops of request votes because the injected delay is longer than the candidate timeout.

In conclusion, we could disable the system with the injection of delays completely. Nevertheless, RAFT assumes a practicality where it cannnot manage Byzantine faults and only network faults, so it is unlikely to reach this point.

6.2.6 Delaying Heartbeats

In this experiment, the intention is to delay the sending of heartbeats by the leader, which guarantees the system's stability, resetting the election timeout of the follower nodes and thus avoiding recurrent elections.

Test	Replica 0	Replica 1	Replica 2	Replica 3	Term
T1	1144	1175	1035	1238	1
T2	1084	860	1067	777	1
T3	1145	1132	1139	1129	1
T4	1459	1293	1125	1478	2
T5	1389	1357	1377	1108	2
T6	1369	1527	1591	1681	2
AVG	**1265**	**1224**	**1222**	**1235**	**1,5**

Table 6.1: Baseline results for first leader elected after x milliseconds.

Injected delay	Replica 0	Replica 1	Replica 2	Replica 3	Term average
0 ms	1265	1224	1222	1235	1,5
100 ms	1615	1677	1739	1617	1,5
250 ms	3031	3021	3073	3116	6,3
280 ms	15477	15482	15378	15574	52,17

Table 6.2: Average results obtained after injected delays. In each "Injected delay", six tests were performed, and the result of the replicas are in milliseconds.

Delaying these messages for enough time can elapse the election timeouts and start a new election. However, if this delay is higher than the maximum election timeout, the leader elections will become infinite. The election timeouts in this RAFT implementation are something like 150 to 300 miliseconds.

The test results present in Table 6.3 used the C4 configuration, least for 60 seconds and we repeated the tests three times, by increasing the injected delay (mentioned as the F column) until it reached values exceeding the election timeout value. Since the maximum election timeout of a follower is 300 miliseconds, and if the heartbeats are frozen during that time (or longer), then the replica will elapse and transit to the candidate state. This process will happen on all followers in the cluster as no timeout reaches values greater than 300 ms leading to an infinite leader election that makes the system unusable. It is possible to evidence this for the total delay values of 300 milliseconds and 350 milliseconds in Table 6.3 and Table 6.4.

F(ms)	R0	R1	R2	R3	Total
25	1	0	0	0	1
50	0	1	0	0	1
75	0	0	1	0	1
100	1	1	0	0	2
125	5	6	6	4	21
150	9	7	11	8	35
175	12	7	6	12	37
200	8	12	10	13	43
225	18	21	15	2	56
250	18	15	22	16	71
275	18	18	10	28	74
300	113	0	0	113	226
350	0	0	110	110	220

F(ms)	R0	R1	R2	R3	Total
25	0	0	1	0	1
50	0	1	0	0	1
75	0	1	0	0	1
100	0	0	0	1	1
125	1	1	0	0	2
150	10	8	10	7	35
175	15	7	13	11	46
200	8	11	11	7	37
225	17	8	13	15	53
250	9	29	33	27	98
275	25	29	25	24	103
300	112	112	0	0	224
350	0	0	115	114	229

F(ms)	R0	R1	R2	R3	Total
25	0	0	1	0	1
50	0	0	0	1	1
75	1	0	0	0	1
100	0	0	0	1	1
125	1	0	0	1	2
150	9	9	9	10	37
175	12	10	6	11	39
200	9	11	10	9	39
225	19	14	15	12	60
250	11	34	19	30	94
275	77	59	10	7	153
300	0	0	111	111	222
350	113	0	0	113	226

Table 6.3: The total amount of candidates that were established leaders after injecting delays. Each column represents a replica and the number of times it got elected leader.

Regarding the minimum election timeout value of 150 ms, it is possible to see in both tables, Tables 5.3 and 5.4, that we are already starting to see unusual leader elections in 125 ms delay injection with an average of 8,3 leaders.

Fault (ms)	Leader Average
25	1
50	1
75	1
100	1,3
125	8,3
150	35,7
175	40.66
200	39.66
225	56,3
250	87,7
275	110
300	224
350	225

Table 6.4: The average results obtained from Table 6.3 that also represent the number of candidates that became leaders.

Chapter 7

Conclusion

Throughout this project, we explored mechanisms to generalize the tool to various fault tolerance protocols, multi-process, or single-process applications. In addition to this, we explored certain components that make it possible to extend the tool to several programming languages, using the gRPC tool and Aspect Oriented Programming (AOP) approaches. We also contributed to the implementation of intuitive fault dependency mechanisms that make it easy for the user to operate. The tool is flexible enough to evaluate the dependencies required in each test run and adapt its execution flow so that it minimizes the message swapping between components of the Zermia framework. Finally, we built a script that, through the fault development by the user, allows the injection test experiments by running the entire *Zermia* framework (*Coordinator*, *Agents*) and the target application together with the developed faults.

We present separate experiments of fault injection using Zermia. First, we used a simple single-process application developed in Python where we applied Delay and information Modification Faults pertaining to the application. In this case, we also demonstrated experiments that contained dependencies between faults. Then, we presented the use of the tool in an implementation of RAFT developed in Java where we applied crash, delays and more deeply leader election delays and heartbeats delays.

One limitation of our implementation is when we constantly checked internal and external dependencies between faults. The communication to the respective Agents and the Coordinator is done by only one thread/node simultaneously to avoid race conditions that could lead to unwanted injection states. For this reason, if Zermia checks dependencies for an extended period, this can lead to performance degradation in the application. We suggest that the users use the dependencies with awareness and if possible, in a combination of trigger conditions that minimize dependency checking so that the application has no performance consequences.

7.1 Future Work

This tool can be further improved to create a real automation and test injection tool. The study and develop a fault injection policies engine that analyzes existing code to determine the policies for injecting faults, would be a beneficial way to define the most important locations where a fault should be injected (for example, the methods that are called more frequently). In addition, through an adaptation of the procedure done in Proteus that generates the fault code automatically by the Domain Specific Language (DSL) definition, automation mechanisms could be created that would avoid the fault code development by the user. Thus, through these two mechanisms, we could automate the fault injection tests through configuration files.

The script for running injection experiments can also be extended to allow running experiments remotely. Currently, the script works for local testing. However, it can be adapted using the Secure Shell (SSH) tool to be able to access other machines remotely, we can run the Zermia Agents and application nodes remotely.

Another interesting development would be to create more predefined fault types by identifying patterns related to fault tolerance protocols.

Finally, since we can extend the usability of the tool to more programming languages and for the automation procedure reaches several applications in different programming languages, the development of the Zermia libraries and the fault development structure could be developed in other programming languages.

Bibliography

[1] Martin Kleppmann. Distributed systems. https://www.cl.cam.ac.uk/teaching/2021/ConcDisSys/dist-sys-notes.pdf. Online; Accessed December-2021.

[2] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[3] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

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

[6] Joao Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48. IEEE, 2012.

[7] Google. grpc. https://grpc.io/. [Online; Accessed July-2022].

[8] Rolando Martins, Rajeev Gandhi, Priya Narasimhan, Soila Pertet, António Casimiro, Diego Kreutz, and Paulo Veríssimo. Experiences with fault-injection in a byzantine fault-tolerant protocol. In *Acm/ifip/usenix international conference on distributed systems platforms and open distributed processing*, pages 41–61. Springer, 2013.

[9] Ricardo Jorge Alves Fernandez. Injecting faults in byzantine fault tolerant protocols. Master's thesis, Faculdade de Ciências da Universidade do Porto, September 2021.

[10] Miguel André Queirós Coelho da Silva. Injector de faltas para teste de aplicações. Master's thesis, Faculdade de Ciências da Universidade do Porto, September 2021.

[11] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[12] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[13] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16, 1998.

[14] Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 7 1982.

[16] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In *Formal models and semantics*, pages 1157–1199. Elsevier, 1990.

[17] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[18] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. Fault tolerance in distributed systems: A survey. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–5. IEEE, 2018.

[19] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[20] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[21] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474. IEEE, 2000.

[22] Peter Sobe. Combination of data deduplication and redundancy techniques in distributed systems. In *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, pages 1–6. VDE, 2016.

[23] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 200–211, 2019.

[24] Czesław Danilowicz and Ngoc Thanh Nguyen. Consensus methods for solving inconsistency of replicated data in distributed systems. *Distributed and Parallel Databases*, 14(1):53–69, 2003.

[25] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*, pages 127–140. Springer, 1983.

[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 1978.

[27] Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Fault-Tolerant Distributed Computing*, pages 201–208. Springer, 1990.

[28] Gul A Agha and Reza Ziaei. Security and fault-tolerance in distributed systems: an actor-based approach. *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions (Cat. No. 98EX358)*, pages 72–88, 1998.

[29] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32, 1985.

[30] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[31] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[32] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[33] Marek Jawurek and Florian Kerschbaum. Fault-tolerant privacy-preserving statistics. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 221–238. Springer, 2012.

[34] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the {Efficiency} of durable state machine replication. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 169–180, 2013.

[35] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[36] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[37] Joao A Duraes and Henrique S Madeira. Emulation of software faults: A field data study and a practical approach. *Ieee transactions on software engineering*, 32(11):849–867, 2006.

[38] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):1–55, 2016.

[39] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[40] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on software engineering*, 16(2):166–182, 1990.

[41] Ang Jin, Jianhui Jiang, Jiawei Hu, and Jungang Lou. A pin-based dynamic software fault injection system. In *2008 The 9th International Conference for Young Computer Scientists*, pages 2160–2167. IEEE, 2008.

[42] Eunjin Jeong, Namgoo Lee, Jinhan Kim, Duseok Kang, and Soonhoi Ha. Fifa: A kernel-level fault injection framework for arm-based embedded linux system. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 23–34. IEEE, 2017.

[43] J-C Fabre, Frédéric Salles, M Rodríguez Moreno, and Jean Arlat. Assessment of cots microkernels by fault injection. In *Dependable Computing for Critical Applications 7*, pages 25–44. IEEE, 1999.

[44] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro*, 33(3):58–66, 2013.

[45] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213. IEEE, 1995.

[46] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. Modifi: a model-implemented fault injection tool. In *International Conference on Computer Safety, Reliability, and Security*, pages 210–222. Springer, 2010.

[47] János Oláh and István Majzik. A model based framework for specifying and executing fault injection experiments. In *2009 Fourth International Conference on Dependability of Computer Systems*, pages 107–114. IEEE, 2009.

[48] Eliane Martins, Cecilia MF Rubira, and Nelson GM Leme. Jaca: A reflective fault injection tool based on patterns. In *Proceedings international conference on dependable systems and networks*, pages 483–487. IEEE, 2002.

[49] Yuting Fu, Andrei Terechko, Tjerk Bijlsma, Pieter JL Cuijpers, Jeroen Redegeld, and Ali Osman Örs. A retargetable fault injection framework for safety validation of autonomous vehicles. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 69–76. IEEE, 2019.

[50] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[51] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, 44(2):248–260, 1995.

[52] Zary Segall, D Vrsalovic, D Siewiorek, D Ysskin, J Kownacki, J Barton, R Dancey, A Robinson, and T Lin. Fiat-fault injection based automated testing environment. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,'Highlights from Twenty-Five Years'.*, page 394. IEEE, 1995.

[53] Timothy K Tsai and Ravishankar K Iyer. Measuring fault tolerance with the ftape fault injection tool. In *International conference on modelling techniques and tools for computer performance evaluation*, pages 26–40. Springer, 1995.

[54] Kang G. Shin. Harts: A distributed real-time architecture. *Computer*, 24(5):25–35, 1991.

[55] Martin Hiller, Arshad Jhumka, and Neeraj Suri. Propane: an environment for examining the propagation of errors in software. *ACM SIGSOFT Software Engineering Notes*, 27(4):81–85, 2002.

[56] Ramesh Chandra, Ryan M Lefever, Michel Cukier, and William H Sanders. Loki: A state-driven fault injector for distributed systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 237–242. IEEE, 2000.

[57] Bruno Pacheco Sanches, Tânia Basso, and Regina Moraes. J-swfit: A java software fault injection tool. In *2011 5th Latin-American Symposium on Dependable Computing*, pages 106–115. IEEE, 2011.

[58] Eugene Kuleshov. Using asm framework to implement common bytecode transformation patterns. *Proc. of the 6th AOSD, ACM Press*, 2007.

[59] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[60] Anna Thomas and Karthik Pattabiraman. Llfi: An intermediate code level fault injector for soft computing applications. 2013.

[61] Ivan Keselev. *Aspect-oriented programming with AspectJ.* Sams, 2003.

[62] Palo Alto Research Center. Eclipse. The aspectjtm development environment guide. https://www.eclipse.org/aspectj/doc/released/devguide/ltw.html. [Online; Accessed June-2022].

[63] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[64] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++ an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth*

International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, pages 53–60, 2002.

[65] Kasun Indrasiri and Danesh Kuruppu. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[66] Jetbrains. Domain-specific languages. https://www.jetbrains.com/mps/concepts/domain-specific-languages/. [Online; Accessed June-2022].

[67] Evan Diamantino Alves Dias Arezes de Sá. Raft under fire - fault injection. Master's thesis, Faculdade de Ciências da Universidade do Porto, September 2022.

[68] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.