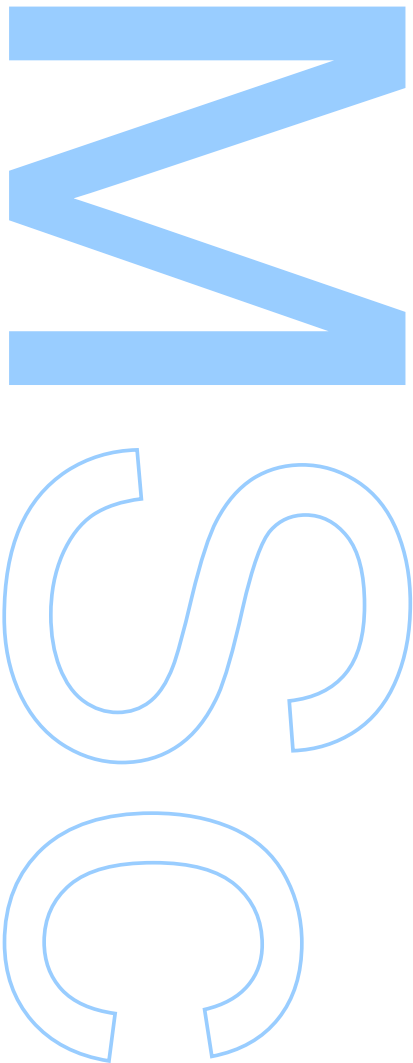
Injector de faltas para teste de aplicações

Miguel Silva
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2021

Orientador
João Soares, Faculdade de Ciências

Coorientador
Rolando Martins, Faculdade de Ciências



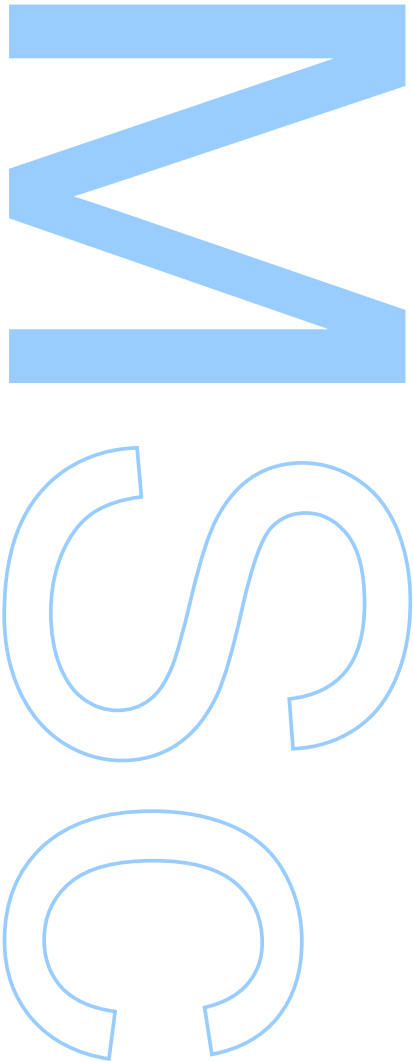
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Universidade do Porto

Masters Thesis

Fault Injector for application testing

Author:
Miguel Silva

Supervisor:
João Soares

Co-supervisor:
Rolando Martins

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Computer Security

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

January 28, 2022

Acknowledgements

I would like to say thank you to my thesis supervisor Professor João Soares, for steering me in where I should take this project and for providing me with valuable insights and ideas on how to solve the problems that arose.

I also want to express my gratitude to my friends and family, for providing me with support, patience and much needed distractions.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Computer Security

Fault Injector for application testing

by Miguel Silva

Assuring the resilience of software is an ever increasing necessity brought forward by the reliance on these systems by the general population. Part of this resilience comes from fault tolerant techniques and protocols which give a system the capability of sustaining faults that are introduced by unpredictable factors. Fault injection tools required to test fault tolerance at the software level have not been developed at the same pace as the needs imposed by the industry. The ones that have been created are very focused on testing the specific protocol for which they were designed, something that is especially true in regards to software that is not composed by a single structure like distributed or concurrent application.

In this project we will present Proteus, a fault injector that builds on top of its predecessors, Hermes and Zermia, with the focus of generalizing these systems through the creation of a Domain Specific Language responsible for generating the code required for each test.

Our tool performs fault injection through the use of aspect based functions, a versatile mechanism that does not require source code manipulation. These faults communicate with the client server architecture that is responsible for their coordination across several locations, such as multiple nodes of a distributed system or threads in a concurrent application.

The Domain Specific Language is the main focus of this project, it is a dynamic tool that improves the systems adaptability by abstracting testers of its inner working. Its structure consists of a dichotomy between general faults offered by Proteus, and custom faults designed by the testers for their specific protocol. Throughout this documents we

will analyse the design, present the syntax and explain how to use it. Additionally, we describe the underlying client-server architecture used by our language for features related to coordination and log management. Finally, we present examples of its implementation, and analyse the use cases where our tool brings the most benefit.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

Mestrado em Segurança Informática

Injector de faltas para teste de aplicações

por Miguel Silva

Garantir a resiliência de software é cada vez mais uma necessidade, devido à dependência destes sistemas pela população. Uma parte desta resiliência advém de protocolos e técnicas tolerantes de faltas, que providenciam a um sistema a capacidade de resistir a faltas provenientes de fatores imprevisíveis. As ferramentas necessárias para testar a tolerância de faltas a nível do software não têm sido desenvolvidas ao ritmo necessário para acompanhar o desenvolvimento deste sector. As que têm vindo a ser desenvolvidas focam-se em demasia no teste dos protocolos específicos para as quais foram desenhadas, algo que é especialmente notável em software que não é composto por uma única estrutura como aplicações distribuídas ou concorrentes.

Neste projeto vamos apresentar e discutir o Proteus, um injetor de faltas que complementa o trabalho desenvolvido pelos seus predecessores, Hermes e Zermia, com um especial foco na generalização destes sistemas através da criação de uma Linguagem de Domínio Específico responsável pela geração do código necessário para cada teste.

A nossa ferramenta realiza a injeção de faltas usando funções baseadas em aspectos, um mecanismo versátil que não requer a alteração do código fonte. Estas faltas comunicam com a nossa arquitetura cliente servidor, responsável pela sua coordenação entre várias localizações, como com múltiplos nós de um sistema distribuído ou *threads* de um sistema concorrente.

A Linguagem de Domínio Específico foi o principal foco do nosso projeto, tratando-se de uma ferramenta dinâmica que melhora a adaptabilidade do sistema, permitindo assim que os utilizadores se abstraiam do seu funcionamento. A sua estrutura consiste numa dualidade entre faltas genéricas oferecidas pelo Proteuse customizadas pelos utilizadores para

um protocolo específico. Ao longo deste projeto analisamos o seu desenho, apresentamos a sua sintaxe e explicamos como usá-la. De seguida, descrevemos a nossa arquitetura de cliente servidor utilizada pela nossa linguagem para funções relacionadas com coordenação e gerenciamento de logs. Para terminar, apresentamos exemplos da sua implementação e analisamos os casos de uso nos quais o uso da nossa ferramenta traz mais benefícios.

Contents

List of Figures

Glossary

DSL	Domain Specific Language
BFT	Byzantine Fault Tolerant
AOP	Aspect Oriented Programming
RPC	Remote Procedure Call
SFI	Software Fault Injection
SWIFI	Software-implemented fault injection
SQL	Structured Query Language
GUI	Graphical User Interface
SCIFI	Scan-Chain implemented Fault Injection
CPU	Central Processing Unit
SWG	Synthetic Workload Generator
EGM	Experiment Generation Module
ECM	Experiment Control Module
DCM	Data Control Modules
HMON	Hardware Monitor
FIA	Fault Injection Agent
WL	Work Load
AWS	Amazon Web Services
SSH	Secure Shell
SMR	State Machine Replication
DDOS	Distributed Denial of Service

JVM	Java Virtual Machine
GPL	General Purpose Language
JSON	JavaScript Object Notation
LOC	Lines of Code

Chapter 1

Introduction

High availability is a requirement for modern systems, that stems from the reliance we have in the computational structures that power our world. The difficulty in achieving this requirement is exacerbated by the ever increasing complexity of mission critical software. This complexity contributes to the presence of faults in programs either through defects, imperfections or flaws. These faults lead to errors which can cause incorrect states, with unpredictable results, that can range from system failure to propagation of errors through the entire program.

Since the presence of faults is inherent to computer systems, fault tolerant protocols and techniques must be incorporated into applications. Their implementation must be then thoroughly tested in order to assess if the fault is being handled in a correct manner, however, conducting these tests is a complex endeavour since faults are unpredictable by nature. These tests are usually conducted using fault injectors that induce a faulty state in a system and analyse the impact that it has in the programs operation. These kind of tools are usually composed of three parts: a load generator, that creates the inputs for the program under test, an injector, that loads the faults for a particular run, usually called the fault load, and a monitor component, responsible for registering the information relative to the systems operation with the objective of assessing the impact of a given fault.

The type of faults that should be used by a fault injector to achieve the most impact on the application depend on the type of program that is under test. However, there are several common faults that every system should endure, like handling the exhaustion of resources. Fault injectors should go beyond this and allow for the creation of custom faults, that take into account the underlying software that is under test.

There is a parcel of these computational structures that must be put under special scrutiny due to their complicated functioning. Examples of this are systems which are composed of several components that operate with a degree of independence from each other, such as distributed or concurrent applications. These programs are susceptible to unpredictable errors that can propagate through the system and are difficult to replicate. Distributed applications have special protocols to handle these errors known as Byzantine Fault Tolerant protocols, which are designed to guarantee a correct operation under the presence of arbitrary faults.

1.1 Motivation

Currently, existing fault injection tools do not match the requirements in terms of availability that are nowadays necessary in mission critical systems. Being able to handle unpredictable faults through the implementation of fault tolerant protocols is an ever increasing necessity, but the tools required to test these protocols are lagging behind or have high implementation costs.

The fault injectors that have been developed are very protocol dependant and lack the adaptability required to thoroughly analyse a system without having to invest in adapting it. The generalized tools that have been developed up to this point mainly focus on a low level approach, that derive from the origins of many of these fault injectors which was the injection of hardware faults at the software level.

Hardware related fault injectors are more advanced and have existed for longer time which has given them time to mature. Comparable progress has not yet been made at the software level where many critical faults occur [1]. Research and development in tools for fault injection at the software level would shorten this gap and allow for more secure and stable systems.

Some fault injectors also focus on statistical analysis to assess the performance of a system when it is injected with generalized faults that affect the execution, including exhaustion of resources or altering data at a lower level. This approach, although relevant, falls more in the job of chaos engineering, a recent approach that tests systems in production environments.

Software fault injectors that are commonly developed are usually tightly integrated with a specific application. They fulfill their roll and discover relevant faults that should be tested across different protocols but are not versatile enough to allow them to be used

in different applications. Fault injectors for distributed applications are an area where this problem is more prevalent [2, 3].

Finally, more generic tools often attach to a specific aspect of the implementation or are bound to a specific language, and do not offer the degree of freedom that a tester desires. By limiting the interactions with the program, they fail to strike a balance between usability and expressiveness, i.e., allowing the tester to develop scenarios tailored to their application.

Our work focuses on offering a software testing and validation framework, based on fault injection, that requires minimal development cost and effort. However, it must be flexible enough for allowing a high degree of customization, allowing it to be used by different protocols and application, and under different use cases.

1.2 Proposed Solution

In this dissertation, we will present Proteus, a versatile fault injector framework that improves on the previous iterations (Hermes [3] and Zermia [4]). Both works have proved the potential of injecting faults through the use of aspects (Aspect-Oriented Programming), along with a client-server model to test Byzantine fault tolerant protocols, in specific the BFT-SMaRt protocol implementation.

Proteus builds on top of these works, generalizing their functioning so that both the faults injection mechanism and the infrastructure that supports it may be adapted to function across several protocols.

One of the most prominent ways that we improved the fault injector is through the creation of a novel way of interacting with it. All faults and the setup for each run is specified by the tester through the use of a Domain Specific Language (DSL). The fault injection mechanism provides a set of generic faults that mainly involve resource exhaustion, while also allowing the creation of custom faults that can be tailored by the tester to fit a specific protocol. The code generated by the DSL integrates into any system through the use of *Aspects*, a special code that interweaves with the source application without the need for altering the source code, and causing minimal impact on the underlying application execution.

The previous interactions of the infrastructure will receive an overhaul so that it accommodates our more versatile approach. While we still follow a client-server architecture,

where the communication is made through the use of gRPC for versatility, the changes made to the infrastructure are mainly related with creating a more independent client.

1.3 Contributions

This project's contribution to the area of fault injection can be described as follows:

- Identification of where the Zermia protocol was limited in terms of extensibility;
- Design and development of a modular fault injection system;
- Design and development of a DSL that can be used to create the base code for fault injection;
- Modularization of the infrastructure, by developing an independent client-side (called *Agent*) and developing the server (*Coordinator*) capabilities required to handle *Agent* coordination and log management.

The work developed in this project combined with the work done in Zermia [4] resulted in the publication of a paper in an international conference [5].

1.4 Outline

The remainder of this document is structured as follows.

Chapter 2 presents and discusses the state of the art, exploring existing previous work in the field of general fault injectors and explore the tools that are used throughout our system in order to better understand their functioning. Here, we also analyse the previous iterations of this project and present their findings.

Chapter 3 presents the design of Proteus and discusses the general structure of our tool by giving an overview of all our functionalities and an explanation at a high level of what is expected of each feature. It also analyses and discusses some trade-offs made in the design and justifies the final decision.

Chapter 4 discusses implementation and constitutes our biggest chapter, which is divided into 3 major sections. The first section, Grammar, describes our DSL, the second section describes the process of generating code from the DSL, and the last section describes the infrastructure that supports the tool.

In Chapter 5, Evaluation, we adapt some tests made in the previous iterations and analyse the gained efficiency from our system. We follow this in Chapter 6, by analysing the results and exploring the future work that could be developed.

Chapter 2

State of the art

In this Chapter we will describe and discuss fault injection mechanisms. We start by discussing the reasons why we believe fault injectors are needed during the development and testing of applications, and what is their role in the development of more reliable systems. Next, we describe several injection tools and the purpose they are meant to fulfil. Moreover, we will also explore the two predecessors of this tool, Hermes, the first iteration that created the structure and architecture, and Zermia, the improved version which was developed to be more generic and have wider capabilities. Finally, we will delve into the concept of DSLs, their potential and usability, while covering some technologies that can be used for their development.

2.1 Fault Injection

A fault is a problem that can be present in a hardware or software component, it can arise from defects, imperfections or flaws [6]. Faults can lead to errors that can result in an unexpected state of a component, unexpected behaviour, unavailability, or other unpredictable results. Testing is an important part of software validation for minimizing faults. However, testing cases/units are not exhaustive, either by not covering 100% of the code, or by not considering faults introduced by external components (e.g., hardware). A typical approach to deal with these problems is the use of fault tolerant systems.

While fault tolerant systems are built with mechanisms to handle possible component faults, they introduce non-negligible overhead that compromises their adoption. Additionally, testing fault tolerant systems is not an easy task. Fault injection is a technique that

allows testing the resilience of a system by forcing faults that could otherwise occur naturally in an application, so that their result and impact can be assessed in order to identify dependability weaknesses.

2.1.1 Hardware-Based

Hardware components are susceptible to a wide variety of faults that can arise from errors in their architecture, manufacturing or environmental factors. This has long been an problem, with substantial amount of literature around this subject [7–9]. Although not the focus of this research project, some lessons can be learned from the development of these fault injectors.

These fault injectors try to find permanent, transient or intermittent faults, by contact or by having an external factor like power fluctuation. The downside of hardware based fault injectors is that they need the system to be built, which can carry many costs. To be able to test in a design phase, simulation or emulation based fault injectors are used [10].

2.1.2 Software Based

Software Fault Injection (SFI) is a type of fault injection that is highly sought after, since many failures of today's technologies are due to errors in the software they run and can be attributed to lack of testing [1]. They are also used in hardware, known as Software-implemented fault injection (SWIFI), where hardware faults are emulated by injecting the effect of a fault on the software. These faults are sometimes preferred because although they lack fidelity they are highly adaptable, simpler and have a much lower implementation cost.

Multiple tools have been proposed to test several types of systems. However, most tend to be designed and built for a specific use case, but when stripped down to its most essential components they all follow the same core principles associated with software based fault injection. Figure 2.1 shows a conceptual representation of these core principles. Three main components that are used by the controller to analyse the *target* [11], namely the *load generator*, *injector* and *monitor*. The load generator is responsible for feeding the inputs to the target, which are usually referred to as *workloads*, and can be supplied by the tester or generated through some rules. The injector loads the faults to be injected, called the fault load, these faults are chosen by the tester and are combined with the selection of inputs that are fed to the system. The monitor entity collects data, using readouts or

measurements, and it can also be responsible for a varying amount of analysis in order to be able to present information, sometimes in real time, to the tester. These analyses are often done in comparison to a *golden* or fault-free run, where a standard load is given as an input by the input generator and the injector does not load any faults.

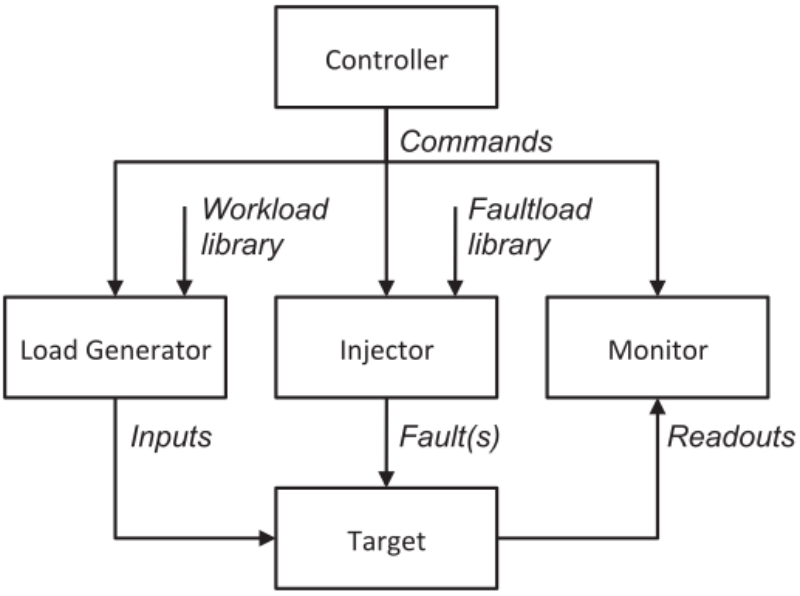


Figure 2.1: Conceptual schema of fault injection [10]

We will now go into detail about the more prevalent systems that have been designed for both SFI and SWIFI applications.

2.1.2.1 GOOFI

Generic Object-Oriented Fault Injection (GOOFI) is a portable generic fault injection tool capable of running in several platforms due to being written in Java. It is designed using a generic architecture that can be easily adapted to a new target [12]. The monitoring component manages and saves all data to an SQL database. It implements a Graphical User Interface (GUI) to create a user-friendly environment to visualize data and faults. It also contains a plugin system through which it is possible to create new fault injection algorithms by using abstract methods that belong to the GOOFI framework as building blocks.

This tool is capable of two fault injection techniques, the pre-runtime SWIFI where data is injected into the program and its data areas before it starts to execute, and the Scan-Chain implemented Fault Injection (SCIFI) technique that is used to test faults in

a physical system using built in test logic [13], these faults are often found in integrated circuits.

2.1.2.2 DOCTOR

IntegrateD Software Fault InjeCTiOn EnviRonment for Distributed Real-time Systems is a software implemented on a distributed real time system called Hexagonal Architecture for Real-Time Systems (HARTS) [14]. It allows developers to inject CPU, memory and network communication faults into a system, which can be transient, intermittent or permanent [15]. For fault scheduling and monitoring it provides a GUI that allows users to specify fault injection timing. The focus of this tool is testing the dependability of distributed systems.

To be able to evaluate the resilience of a distributed system, an appropriate and consistent workload must be used so that dependability parameters such as detection coverage and latency of execution can be measured. Along with DOCTOR, a Synthetic Workload Generator (SWG) was developed so that users can easily create and repeat payloads [16].

An interesting approach used by this tool to reduce the overhead caused by the fault injector is reducing the number of functions performed on the same processor that the target system is running on. Their monitoring capabilities also have the option of being offloaded to a dedicated hardware that is able to obtain high-resolution timing data, improve the accuracy and reduce the overhead of data collection.

The fault injector is divided into three modules: Experiment Generation Module (EGM), Experiment Control Module (ECM) and Fault Injection Agent (FIA). The EGM parses the experiment description file created by the user into separate files, that are interpreted by the ECM, which is also responsible for generating executable images of workloads. FIA is a separate module that receives commands via ethernet and performs their actions in a separate process that runs in the same processor that the workload is running. It reports these actions to Data Control Modules (DCM) or Hardware Monitor (HMON), and it has three distinct triggering mechanisms: time-out, trap and code modification. ECM is an external controller that coordinates the nodes and several FIAs, it sets up the environment, synchronizes the nodes and pulls data from DCMs or the systems software [15].

2.1.2.3 Xception

Xception [17] is a Software fault injection and monitoring environment with minimal interference. Instead of altering the target application, it uses advanced debugging and performance monitoring tools available in modern processors to inject realistic faults and monitor their impact on the target system behaviour. Because faults operate at the exception handler level, they can affect any processor and can be used to inject faults in the operating system, which also allows for the injection of faults when the source code is not available. Their monitoring component provides spatial and temporal fault triggers as well as triggers related to the manipulation of data in memory.

2.1.2.4 FERRARI

The Fault and ERRor Automatic Real-time Injector is a SWIFI tool that emulates hardware faults using software traps to inject CPU, memory and bus faults [18]. It is divided into four modules: 1) the initializer and activator, prepare the target program for the fault injection, it collects information through statical analyses of the executable and afterwards it dynamically analyzes a fault free run to collect certain execution behaviours; 2) Is the User Information Module that obtains experiment parameters supplied by the tester like: experiment modes, which define if it should be user or automatic control; fault and error types that specify the exact kind of fault; fault and error classes which define a general kind of possible faults where user defined classes are possible; 3) Fault and error injection is the mechanism by witch permanent faults and transient errors are injected, the way of injecting is similar the only difference is the duration of injected fault or error, to control the application, it uses a parallel daemon on the host processor that can inject faults into the address, data or the control lines of the processor; 4) Is the Data Collection and Analysis, where every run is logged.

2.1.2.5 FIAT

Described as an automatic real-time distributed accelerated fault injection environment the Fault Injection Automated Testing (FIAT) environment is a tool that can inject faults into user application code and data, as well as into messages, tasks and timers [19]. The Workload (WL) system is employed to manage the tasks to be performed in each machine, it is described as a observable set of real time communicating tasks, a WL can run in one machine or distributed across several computers. It has a component that allows for

the definition of fault classes (correlating faults and error patterns that they cause) by specifying where, when, and for how long errors will strike, and how they interact with object code or data. Lastly it has a Data Collection/Analysis part that supports two entities, histories that record normal functional and performance events and error reports, composed by a record of exceptions and abnormal events.

2.1.2.6 PROPANE

PROPagation ANalysis Environment is a tool for profiling and conducting SFI, whose main focus is software for single-process user applications on desktop systems, where the process may be multithreaded [20]. This suite of tools consists of the PROPANE Library (PL) that is used by the target system to access the probing and injection functionality. The PROPANE Campaign Driver (PCD) that handles the actual execution of experiments, it also supports a GUI capable of controlling and following the experiments. The PROPANE Data Extractor (PDE) extracts traces of various logged variables and memory areas and can compare these results with the non-faulty executions. The PROPANE Setup Creator (PSC) creates the setup files that the system uses.

The tool is able of injecting software faults through the mutation of source code, and the injection of data errors by manipulating variable and memory contents. The generic injection and logging mechanisms are provided in a static C-library, which means that the language used for the targets source code must be able to interface with libraries implemented in the C programming language.

2.1.2.7 Loki

Loki is a state driven fault injector for distributed systems that injects faults based on a partial view of the global system state [21]. It argues that distributed applications fail in subtle ways that depend on the state of multiple parts of a system, which means that the fault injection must also have this into account. The system obtains a partial view of the global system state with notifications that represent state changes, and faults are triggered according to these state changes, they present an example where the injection status of a fault in one node can depend on the state of other nodes. After faults are injected, they are analysed using offline clock synchronization events and injections are put on a single global timeline. It has a GUI that can specify a state machine, and a fault injection campaign to verify if the faults were correctly injected.

A Loki runtime runs alongside the distributed system maintaining the partial view of the global state necessary for fault injection, and performs the fault injection when the system arrives at the desired state. The injected faults is called a probe and consist of three parameters, the fault name, a fault expression that determines when it should be injected and a flag that specifies whether the fault should be injected once or repeatedly. The probe is written in C++, the user must implement the *injectFault()* function that contains the code that will be injected.

2.1.2.8 ORCHESTRA

ORCHESTRA is a fault injection environment for testing dependability and timing properties of distributed protocols that claims to focus in finding specific problems instead of assessing system dependability through statistical metrics such as fault coverage [22]. It uses a script-driven probing and fault injection approach that looks at a distributed system as an abstraction through which many participants exchange messages. This approach leads to an architecture that is detached from the target protocol, and creates a clean separation between the two. This is able to reduce intrusiveness in the system by decreasing the overhead and helping maintain logical and timing correctness.

The scripts perform operations in the messages in three distinct ways: message filtering, for intercepting and analysing messages; message manipulation, for dropping, delaying, reordering, duplicating, or modifying a message; and message injection, for probing a participant by introducing a new message into the system. It also has a model of failures to identify how a protocol deviates from its correct specification, which are: process and link crash failures, send and receive omission failures, timing/performance failures, and arbitrary byzantine faults.

2.1.2.9 JACA and FIRE

JACA [23] is a SWIFI tool that injects faults into a high level application written in Java using the javassist toolkit [24]. It's predecessor is the FIRE tool, that used reflective programming to inject faults into C++ applications [25]. JACA introduces the ability to inject faults at the byte code level instead of requiring the source code. Its architecture bases itself on a set of patterns it calls Fault Injection Pattern System, which is composed of a Controller that coordinates the components, a fault injector to the system under test, a monitoring system that collects data, an activator for the target system and a GUI

that allows the user to specify experiments. JACAs fault injection capabilities include the ability to change the values of parameters, attributes and returned variables.

The tool mentioned in this paper is responsible for the fault injection, whereas javassist allows for simple java bytecode manipulation [24]. It can be used to generate new Java Classes, or alter the bytecode of an existent class.

2.1.2.10 Discussion

Many of the tools presented have an approach that focuses on how the software responds to lower level faults, such as memory or registry data corruption.

In terms of fault injection methods, two tools follow a similar approach to ours, namely JACA and PROPANE. Both of them manipulate source or byte code, but the tooling they use is, in our vision, more limited than ours, because they are tightly integrated into their respective language, and porting their behaviour into other programming languages would be a complicated process.

In relation to the faults that are injected, many tools provide similar capabilities to the ones we offer, and some even go beyond, like FIAT or DOCTOR. In its own specific use case, the ORCHESTRA method of altering the messages also presents an interesting approach, although restricted to a specific type of systems.

Several tools offer a generic fault injector component, similarly to the one we provide, however most of them fail to provide predefined faults and we would argue that our method of extending the capabilities is more versatile.

Loki is very interesting to us and their method of triggering faults was a great inspiration, since it is very versatile and has in mind the system as a whole. Their method of creating custom faults is also similar to ours.

We draw inspiration from several aspects mentioned during this section, but one big change that we implemented is how to interact with the tool. Many of the described works use a GUI to interactively manage faults, instead, we decided to use a DSL to transform the instructions introduced by the tester into code that runs the faults. The tooling used by our system binds to the program and executes custom along with predefined faults, while also allowing for several triggering options that proportionate users with a versatile and extensible tool.

2.1.3 Chaos Engineering

In the topic of fault injection, it is important to understand a recent approach to testing that is becoming increasingly famous. We will discuss its purpose, capabilities, and how it relates to our work.

Chaos engineering is the discipline of experimenting on a software system that is in production in order to build confidence in the system's capability to withstand turbulent and unexpected conditions [26]. This is a growing discipline, and a different way of looking at fault injection. This methodology of testing has gained popularity in large scale distributed software due to, in part, an ever-increasing use of the cloud. The goal is to automatize the introduction of faults in the system and analyse the results to continuously improve, all while in production. An important concept in this discipline is the blast radius, a measure of how much the service will be affected and, ultimately, the impact it will have on the customer; while some allowance is necessary, it must be reduced to a minimum.

Several tools have been developed to test applications this way. The most prominent is the Simian Army [27], a suite of tools developed by Netflix to test their Amazon Web Services (AWS) infrastructure, this project has since been disbanded and divided into several tools like the still supported Chaos Monkey [28]. Since then, several tools have been developed such as Mangle [29], a tool that requires very little configuration and enables the injection of faults in several platforms like Docker [30] or ssh enabled remote machine. Chaos Mesh [31], an open-source chaos engineering platform written in Go [32], used to orchestrate chaos in Kubernets [33] environments. Litmus Chaos [34] is similar to chaos Mesh but written in type script.

This method of thinking and testing is increasingly necessary, but it does not remove the need to test applications in a controlled environment, Chaos Engineering should be viewed as an additional way of testing that in no way obsoletes common fault injection mechanisms. Besides this, there are many critical systems that cannot risk this kind of test in a production environment.

2.2 Hermes and Zermia

Our work will expand on the previous work done by Hermes [3] and Zermia [4]. Hermes was the first iteration and was designed to show the potential of SFI combined with AOP tools to automate testing of BFT (Byzantine Fault Tolerant) protocol implementations.

Zermia improved Hermes and made the design more versatile while improving the testing capabilities.

2.2.1 BFT

To understand the way our fault injector works it is important to understand BFT protocols, the original testing objective of our system. These protocols are designed to guarantee the correct operation of a distributed system in the presence of arbitrary faults. They achieve this through several techniques, the most common being the State Machine Replication (SMR), where multiple instances replicate the application state and execute the same set of operations in the same order. A BFT protocol has the objective of ensuring both safety and liveness. Safety ensures that a non-faulty replica that executes the same set of operations in the same order will achieve the same final state, and liveness guarantees that non-faulty replicas eventually execute all requests from non-faulty clients.

Hermes fault injector was developed to address research that looked at the inefficiencies of BFT protocol implementations to handle Byzantine, as well as benign faults. In [35] a comparative in-depth analysis of several protocols is made.

2.2.2 Hermes

The first iteration of the fault injector that was developed during our project was presented in the paper [3]. Here it introduced an innovative SFI framework created to improve and automate the testing of Byzantine Fault Tolerant (BFT) protocol implementations. It also allows developers to get an insight into a sometimes overlooked aspect of BFT protocols, the performance of the implementation when presented with several kinds of faults.

The presented system was shown to be flexible and allow protocol independent faults (like crash faults and network faults) as well as protocol dependant faults (like corrupt headers, forged signatures). Hermes was used to assess the behaviour of BFT-SMaRt [36], a well-known BFT protocol implementation.

The fault injector used in Hermes, and the subsequent versions, take advantage of features present in Aspect Oriented Programming (AOP) tools, in particular AspectJ for the JAVA runtime and AspectC++ for the C++ runtime. This will be discussed further in Section 2.2.4.

In Figure 2.2 we can observe an overview of the Hermes architecture. The fault injection is governed by an orchestrator, which is the component that dictates and coordinates

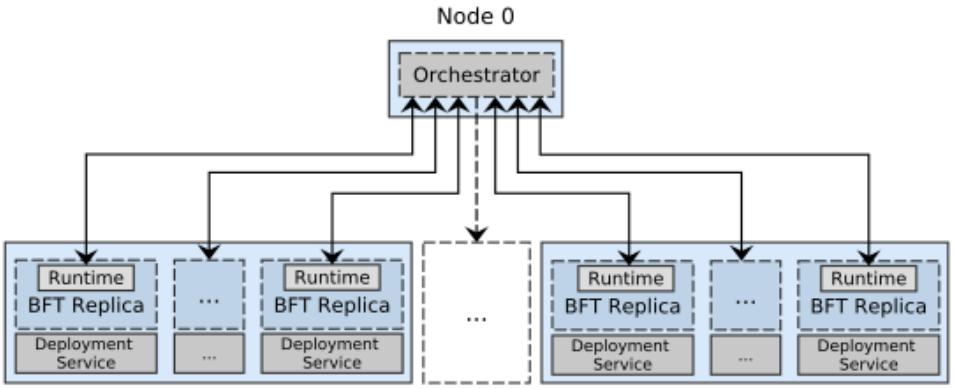


Figure 2.2: Overview of the Hermes architecture

the injection of faults. The runtime that is integrated into the BFT replica is responsible for the fault injection. These two components communicate using three communication primitives: Remote Action, Action and Notification. The Action communication primitive is used by the runtime to verify if a fault about to be injected is enabled and, if all the conditions are set, its main purpose is to act as a synchronization barrier. The remote action communication primitive is used by the orchestrator to perform a remote procedure call in the replica's runtime. The notification communication primitive allows for asynchronous messages between the runtime and the orchestrator, which can be used to reduce the overhead associated with synchronous operations.

The runtime is capable of two types of faults, namely Context-Free and Context-Dependant, these faults were chosen by taking into consideration the protocols being tested. The implemented Context-Free Faults are CPU Load, Crash, Sleep and Drop Packet. For the Context-Dependant Faults there are Corrupt header, Corrupt payload, Forge Signatures and DDOS (Distributed Denial of Service).

Using some combination of the aforementioned faults, several experiments were developed to test the BFT-SMaRt implementation:

- Attack 1 - crashed several nodes simultaneously;
- Attack 2 - change the payload size to MAX_INT;
- Attack 3 - Delay the prepare message to 90% of the timeout that is currently in use;
- Attack 4 - Delay prepare message by 5 times the value of the current timeout.

The analysis of the logs generated in each run revealed two implementation bugs. At the low communication level, insufficient validation in the size of the received packet opens

the possibility of an overflow or underflow attack, the second bug was related to an ever-increasing timeout value within the leader-change sub-protocol. An improvement was also recommended that triggered a leader change under the suspicion of a malicious leader. Alterations to the source code were proposed and the tests were repeated. The fixed version showed significant improvements in terms of performance. These changes were committed to BFT-SMaRt public repository.

2.2.3 Zermia

Zermia expands on the previous design of Hermes, improving its flexibility, extensibility, and by introducing new faults [4].

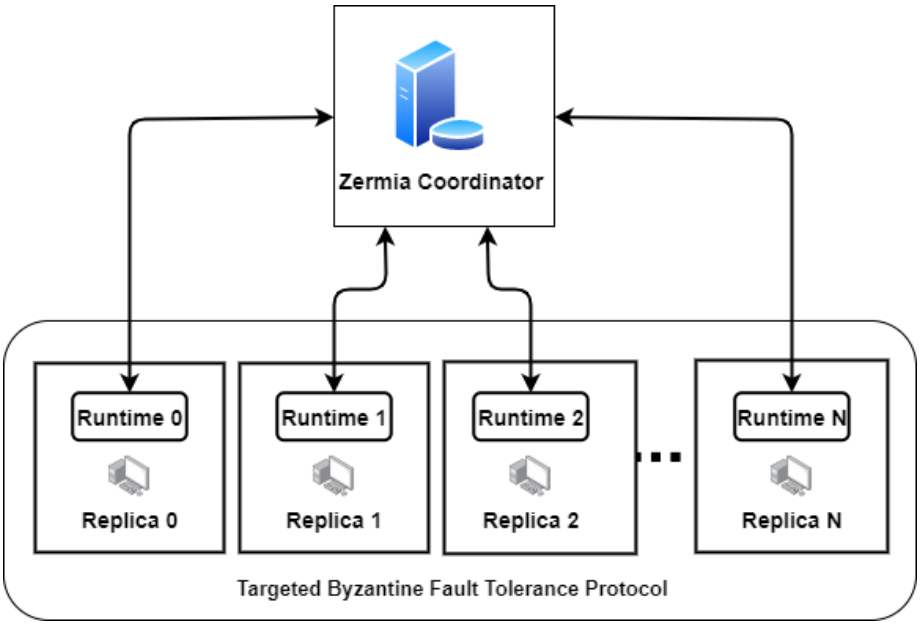


FIGURE 2.3: Overview of the Zermia architecture

As we can see in Figure 2.3, Zermia has a very similar architecture to Hermes. The system is built in Java, and has two distinct components, the *Coordinator*, responsible for managing the *Agents* by distributing fault schedules and synchronising their execution. The *Agent* runs alongside the target application and is responsible for triggering faults using AspectJ code weaving mechanisms. It also gather metrics which are then sent to the *Coordinator*. This sharing is done via Remote Procedure Calls (RPC) with the use of gRPC.

This system is capable of injecting several more faults than the previous iteration, the faults it can inject are:

- Thread Delay - Delays the thread by using sleep cycles, where the amount of sleep, is defined by the user (in milliseconds);
- Packet Dropping - Replicas drop random messages based on a probability defined by the user, which can be between 0 - 100;
- Empty Messages - Similar to the previous fault, but the replicas send an empty message;
- Random content in Messages - Similar to the two previous faults, but replicas send messages with the same size but with random data;
- Flood - Several messages are sent per call to simulate a Denial of Service attack;
- Crash - Suddenly crashes the replica;
- Memory Leak - Increases the usage of memory by creating an array with a variable size. If used too often it can lead to a crash;
- CPU loader - Increases the CPU usage by employing a simple algorithm. The user is able to decide how many threads are going to be used to cause the delay.

This framework also has a priority system to handle fault collision. If multiple faults are to happen at the same time, it will define an order of triggering that tries to increase the impact in the application under test. A specific priority associated with each fault can also be created by the user.

The Zermia framework improved Hermes in several other ways, such as modularizing the system. The use of the gRPC system helps the modularization of the program and opens the possibility of extending the framework to several languages by only requiring the agents to be ported.

However, it still has several shortcomings, which will be discussed further when we approach the design decisions of the most recent version. To summarize, the system is too dependant on the *Coordinator*, which brings unnecessary overhead to the program, and distances it from a real non-faulty run. The other big aspect is that, despite the efforts to modularize the system, it is still closely related with the testing of BFT protocol implementations, in particular BFT-SMaRt. Because of this, it would still be difficult for a developer to port this system to a new use-case, without having to rewrite substantial parts of the code.

2.2.4 Aspect Oriented Programming and AspectJ

Hermes, Zermia and the framework that is going to be discussed in this paper take advantage of tools designed to help users implement the AOP (Aspect Oriented Programming) paradigm, which are AspectJ for the JAVA runtime and AspectC++ for the C++ runtime.

Aspect Oriented Programming (AOP) [37] is a design principle aimed at separating a computer program into separate parts, in which each one addresses a concern, and, if properly implemented, it creates a modular program.

The problem arises when we need to implement cross cutting concerns, a concern that is present throughout the program and that even intercepts other created concerns, the two classical examples are logging and security. These tools fix this problem by adding additional behaviour to existing code without modification of the code itself. They do this by weaving the additional code at a position defined by the user [38].

There are three types of weaving [39]:

- Compile-Time Weaving - also called build-time weaving, is the simplest approach. It requires access to the source code of the application and, depending on how it is implemented, the compilation may require the AspectJ compiler, called ajc, where the aspects themselves can be in source or binary form. The AspectJ compiler will compile the source code and produce a woven class as output. Upon execution, this woven class is loaded into JVM as a normal JAVA class;

- Post-Compile Weaving - also called binary weaving, it is used to weave existing Class and jar files. Like in Compile time weaving, aspects can be in source or binary form, and may themselves be woven by aspects;

- Load-Time Weaving - is binary weaving deferred to when the class loader loads a class file and defines the Class to the JVM.

AspectJ implementation is different from normal programming languages, to understand the way it functions four core concepts are crucial:

- Aspect - A modularization of a concern that cuts across multiple objects. Each aspect focuses on a specific crosscutting functionality;

- Advice - A class of functions that add additional behaviour to existing code. The aspect runs the advice at a particular join point, where additional behaviour can be usefully joined;

- Pointcut - Specifies where exactly to apply the advice using a regular expression that uses syntax like the base language (usually class names or methods). An advice is associated with a pointcut expression and runs at any join point that matches the pointcut;
- JoinPoint - A point during execution, such as the execution of a method or property access.

There are five ways that an advice can alter the execution of an application [40].

- Before, intercepts the executions before the method is called.
- After, runs after the method returned a result.
- AfterReturning is the same as After but intercepts the returned result, and can manipulate it
- AfterThrowing, runs only if the method throws an exception.
- Around is the most versatile advice, it encompasses all of the previous and is capable of altering the functions execution, it's downsides are the impact on the application efficiency.

The last feature present in AspectJ required to understand the implementation of Zermia and Proteus is the concept of annotations. Introduced in AspectJ 5, it is a different and, in our opinion, a friendlier way of writing Aspects. It also removes the need for the *ajc* compiler and makes it possible to compile the system using a regular java compiler that is above version 5 [41].

Listing 2.1 shows an example of how to use annotation-based AspectJ to create an aspect:

```
@Aspect
public class LoggingAspect {

    @After("execution(* exp.main.*(..))")
    public void logAfterAllMethods(JoinPoint joinPoint) { ... }

   @AfterReturning( pointcut = (void exp.main.Main.main(..), returning = "retval"))
    public void AfterExecution(Object retval) { ... }
}
```

Listing 2.1: Example of an aspect using AspectJ [42]

In Listing 2.1 we can see an *Aspect* being created with the purpose of logging information, in which the *Pointcuts* specify where the *JoinPoints* will be inserted. In this example, the first *Pointcut* specifies that a *JoinPoint* should be inserted after the execution of all functions that start with *exp.main.**; we then specify the function that will be executed, in this case *logAfterAllMethods*, which receives an argument with the object *JoinPoint*, this object holds all information accessible to us related to the function that it has been woven into. This includes the arguments of the function, the executing context, among others [43].

In the method *@AfterExecution*, we can see a *Pointcut* that catches the returning value of the function, the generic object *retval*, which can then be accessed to, for example, log the return value of the function. Beyond this, this method also allows the generic object *retval* to be altered, according to the developer's needs.

In Figure 2.2 we demonstrate the *@Around* injection method and its capabilities.

```
@Aspect
public class LoggingAspect {

    @Around("execution(void exp.main.Main.main(..))")
    public void alterMethod(ProceedingJoinPoint joinPoint) {
        Object[] args = joinpoint.getArgs();
        args[1] = null;
        Object result = joinpoint.proceed(args);
        result++;
        return result;
    }
}
```

LISTING 2.2: Example of the Around injection point

We use a *ProceedingJoinPoint*, an extension of the *JoinPoint* interface, that gives us access to the *proceed()* method. As previously mentioned, we can access the arguments received in the method by using *joinpoint.getArgs()* and altering them according to our needs. We then continue the execution of the method by using the *joinpoint.proceed* method and providing it with the altered arguments and catching the return object of the execution, at which point we change this result and return the altered value. Although the *@Around* advice is significantly more intrusive, it is capable of altering the execution of the program giving a lot of possibilities to developers, including modifying arguments, prevent method execution and/or modifying the return object.

2.2.5 gRPC

Another important tool used in Zermia and Proteus is gRPC [44], it is an open source Remote Procedure Call (RPC) framework that can run in multiple languages. it is used in the communication between the *Coordinator* and it's *Agents*.

The system uses a *.proto* file to define the Remote procedure calls. The definitions have their specific syntax, an example can be seen in Listing 2.3.

```
syntax = "proto3";

service example{
    rpc Hello (sendMessage) returns (receivedMessage) {}
}

message sendMessage{
    int64 nrNames = 1;
    repeated string names = 2;
}

message receivedMessage{
    bool acknowlegment = 1;
}
```

Listing 2.3: Example of a simple proto file

Here we can see an RPC call being created using proto3 syntax, where a list of names and their amount is sent and a boolean value that confirms the reception of the data is received. The proto syntax allows for various data structures; more examples can be found in their documentation [45].

2.3 Domain Specific Language

A Domain Specific Language is a programming language with a higher level of abstraction that is designed to solve a specific problem [46]. It allows users to abstract themselves from the inner workings of the tools that they are using by allowing its functions to be implemented through a simple language. All DSLs usually have common characteristics among them, in particular [47]:

1. An unambiguous grammar, where there should only be a single way of achieving a purpose;

2. A single abstraction level, where a language should have a defined and limited scope;

3. A solid tooling system that is reliable and does not need further tinkering from the developer;

4. It should not compile to machine code but to another GPL.

There are also two distinct design philosophies usually followed by DSLs [48, 49]:

1. External DSL - A language that is parsed independently of a General Purpose Language (GPL). It still uses a GPL as its host underneath but it is not bound by its syntax which makes it more independent, it can have a wider scope but it faces difficulties in relation to support from IDEs and the need to implement some of a GPLs base utilities. It is also limited by the capabilities of the parser.

2. Internal DSL - Also called fluent interfaces. Are often described as a different idiom of the base GPL, they still maintain the same structure and adhere to the same syntax rules of the base GPL, this allows for great support from IDEs and simple development, but their capabilities are limited, and their main purpose are to function as an extension of the base language.

The language we have developed is, in its essence, an external DSL it is independent from the base GPL and is also parsed independently, this was needed because it is one of the objectives of this project to be able to extend itself to other languages in the future so we could not depend on a single base language.

We circumvent the problems present in external DSLs by using a great tooling mechanism that we will explain in Section 2.3.2. Our system does use some of the concepts present in Internal DSLs, more specifically its reliance on the syntax of a base GPL, the syntax of our language is similar to its current base language JAVA, we developed it this way so that users would find its syntax easier and predictable.

Several tools can be used for the development of DSLs, the ones we considered all act as a compiler and follow a predetermined set of steps that are important to know, in order to understand how they function [50].

The first step a compiler must do is lexing, also known as tokenization, which is the breakup of a language into tokens that can be words, identifiers or symbols. The type of tokens is entirely dependant on the language. These tokens are then fed into a parser, who imposes the syntax of the language (semicolon use, bracket placement, etc). The guide of how a parser should act is defined by the formal language description called a grammar,

which is designed to be consumed by the users and allow them to abstract themselves from the inner workings of the parser.

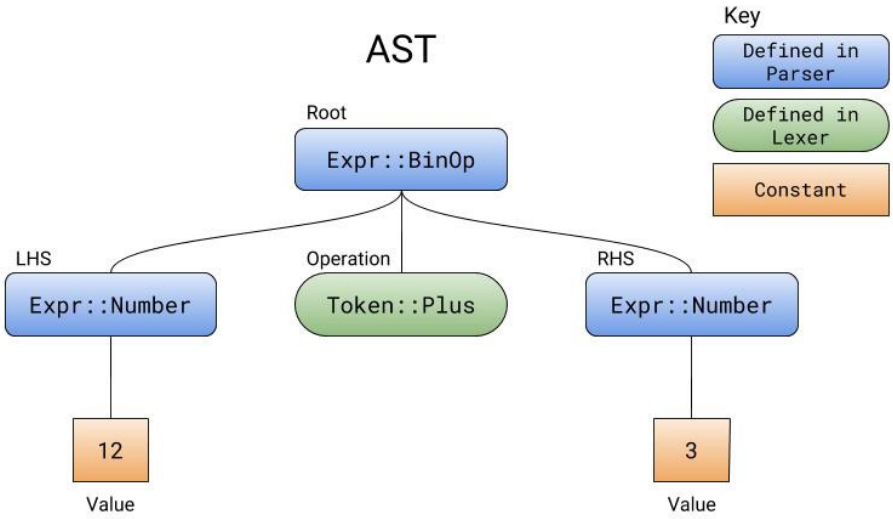


Figure 2.4: Example of an AST tree

The analysis done by the parser results in an AST, as can be seen in Figure 2.4. There are essentially two ways of producing this tree: top-down parsing, where tokens are consumed from left to right; and bottom-up parsing, where a parser tries to find the most basic elements and construct the tree from there.

The code generator takes this AST and outputs the desired code, which in a GPL is usually assembly instructions, and in a DSL the output is usually a GPL.

2.3.1 ANTLR

ANTLR is a parse generator used to build DSLs [51]. It outputs a parser that is capable of building an AST tree from a given input, by following the rules defined in the grammar. It's also capable of creating a tree walker that can be used to visit specific nodes and execute certain predetermined instructions. It is written in Java, but is capable of outputting a parser in several languages.

A popular tool with a dedicated community and a thorough documentation that can be found in the literature [52]. It is employed by a lot of companies, such as: Twitter search, who uses it for query parsing over 2 billion queries a day; the languages for Hive and Pig, data warehouse and analysis system for Hadoop both use it; NetBeans IDE parses C++ with ANTLR; among other examples. It is also very useful in the development of small

tools like configuration file readers, legacy code converters, JSON parsers, and any tool that could benefit from a DSL [53].

This tool does not manage the last part of the compiler known as the Code-Generation, which is done by design, since the AST tree walker can be used to walk the tree and generate code. This gives a lot of flexibility but also requires significantly more development time. Because of this, a pure implementation with ANTLR was not chosen.

2.3.2 Xtext

Xtext is an open-source software framework developed by The Eclipse Foundation [54] for the development of programming languages and DSLs. Users can define their language using the syntax of the Xtext grammar language, which then produces the entire infrastructure, including a parser, linker, typechecker, compiler, as well as a customizable Eclipse-based IDE.

The inner workings of the system are significantly abstracted, but it is nonetheless important to understand them in case we need some more fine-grained adjustments. Like it was previously mentioned, this tool is developed by The Eclipse foundation, so it uses design standards and concepts used in several other of their tools. One of the Eclipse projects that is heavily used internally is the Eclipse Modeling framework (EMF) [55].

The EMF Project is a modelling framework and code generation facility that is used to build tools and applications based on a structured data model, which is described in a XML Metadata Interchange (XMI). EMF provides tools and runtime support to produce a set of java classes for the model, as well as a set of classes that allows interaction with the model. It is the foundation for interoperability with other EMF based tools. Ecore is the core metamodel, and other models are defined using its constructs [56].

Xtext uses EMF models for the representation of parsed text files. An example of this can be seen in Figure 2.5. This EMF model is essentially an AST tree that represents the essence of the textual models. It is an abstraction of the syntactical information and is used in later processing stages. The model is made up of connected EObjects, an EObject is an instance of an EClass, and a group of EClasses can be contained in an EPackage; these are core concepts of Ecore. These are important concepts because when handling the code generation, it is necessary to interact with objects that have their roots in the EMF package. The code generated to interact with the objects derived from the grammar

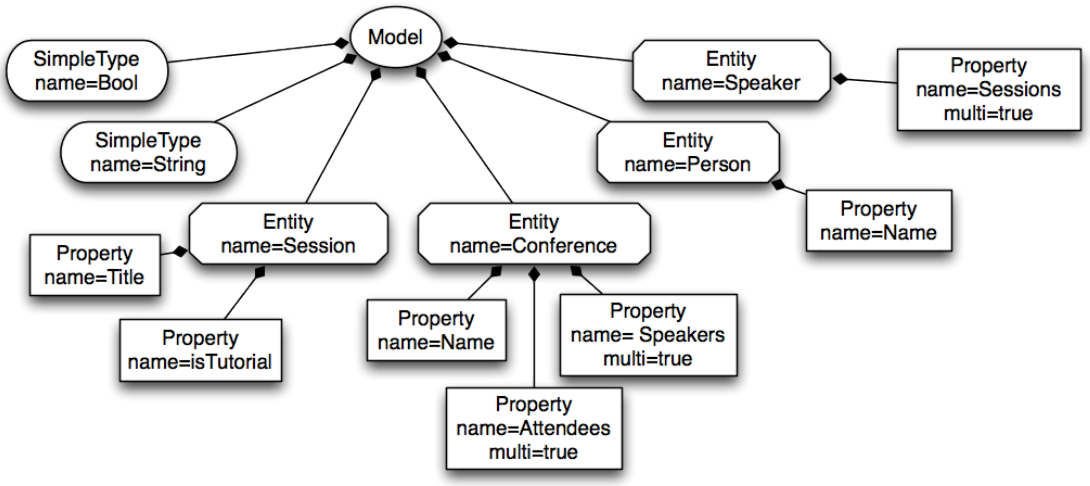


Figure 2.5: Example of an EMF Model, xtext representation of an AST tree [55]

extend the EObject, and some methods are derived from it. Although not used in our system, it is also possible to create new meta-models for the system, using EPackages.

2.3.3 Grammar

In Xtext, grammars are interpreted using the top-down parsing technique, this simply means that the parser consumes tokens from left to right, in a descending order, more information can be found in the discussion of DSLs in Section 2.3. This tree structure is reflected in the way grammars are created. In Listing 2.4 we present an example grammar that incorporates many of the major components present in our grammar.

There are three essential structures in Xtend grammar, these are *ParserRules*, *EnumRules* and *TerminalRules* [57].

- *ParserRules* are the branches of the produced AST tree, they do not produce any atomic terminal token, but instead they produce a tree of terminal and non-terminal tokens;
- *EnumRules* is one of the possible leafs of the tree, it returns an enumeration literal, from a provided list;
- *TerminalRules*, also known as token rules, are the leafs of the tree. Each one of this rules return an atomic value (an *EDataType*). By default it is an instance of *ecore::EString*. They are described using a regular expression, with some extended features. There are several predefined terminal rules, easily used by a grammar, the

most common ones are INT, STRING and ID, a full list can be found on the Xtext documentation [57]. They have this names because there is an informal naming convention that names of terminal rules are all upper-case.

```
Model:
    (elements+=Entity)*
;
Entity:
    Vehicle | Person
;
Vehicle:
    'vehicle' '(' number=INT ',' team=Teams ')'
;
Person:
    'person' '(' name=STRING (',' team=Teams)? ')'
;
enum Teams:
    ENGINEERING='Engineering' | RESEARCH='Research' | LEADERSHIP='Leadership'
;
```

LISTING 2.4: Example of the grammar syntax

In the example (Listing 2.4) all created objects are *ParserRules* except the *Teams* object which is an *EnumRule*. The *Model* object is the root of the tree, and contains a list of entities, this entities can be of two types, as described by the *Entity* parameter. This types then have a specific syntax, an example of this syntax can be seen in example 2.5. The *Vehicle* type contains a predefined *TerminalRule* that returns an integer that is then stored in the variable called number, it also contains one of the types of *Teams EnumRule*. The *Person* object is similar but the name is given as a String and the team is optional, optional statements are surrounded by parenthesis and end in a question mark.

In Listing 2.5 we can see an application of the syntax described in the Listing 2.4.

```
vehicle (5,Engineering)
person("John",Research)
person("Clara")
```

LISTING 2.5: Valid example using the Grammar syntax in 2.4

2.3.4 Xtend

Xtend is a dialect of Java, with a more concise syntax and additional functionalities like type inference, extension method, operator overloading, template expressions, lambda expressions, among others [58]. It is designed using Xtext and is compiled into Java 8 code

and, because of this, integrates seamlessly with all existing Java Libraries. There is also support for syntax highlighting, refactoring, navigation and debugging embedded in the Eclipse IDE. This is the language used for our process of generating code from the grammar so it is important to understand some aspects of its functioning, which will in turn help to comprehend code snippets presented throughout the project.

In Listing 2.6 we can see a snippet of code that resembles the code generation that is found in our DSL. The first function type inference is used to deduce the type of name and the return type from the context. Semicolons as we can see are optional.

The second function is a template expression, which are a type of function that returns a multi-line string, something that is heavily used in our implementation of the DSL, because it is an intuitive way of generating code. As we can see in Listing 2.6, we can write the static parts explicitly and use the french quotes to execute code or access variables.

```
package example

import java.util.List

class hello{
    def helloAll(List<String> names} {
        for(name: names){
            println(name.sayHello)
        }
    }

    def sayHello(Sring name) {
        '''
        Hello «name»!
        '''
    }
}
```

Listing 2.6: Example of Xtext

Template expressions can do much more that the example in Listing 2.6. In Listing 2.7 we can see some of its features, more features of template expressions and Xtend can be found in the tools documentation [59].

```
 package example

import java.util.List

class hello{
    def helloAll(List<String> names} {
```

```
        '''
        «FOR name : names»
            «IF name !== null»
                Hello «name»!
            «ENDIF»
        «ENDFOR»
        '''
    }
}
```

LISTING 2.7: Example of Xtext relying on template expressions

Chapter 3

Design

In this Chapter we describe the design and discuss decisions made throughout the process of designing a modular fault injection mechanism called Proteus. This system is the evolution of the previous tools Hermes [3] and Zermia [4], during the analysis of our design we will draw parallelisms with the previous iterations so we can learn from the design decisions they made.

We will first discuss the general objective and the targeted use cases. Then, we elaborate on the specifics of our system, namely: schedules, and which entity should be responsible for coordinating them; triggers, who should be responsible for them and how to achieve the correct balance; finally, we will explore the parameters required to describe a fault. In the end, we will go over our DSLs requirements and the general structure it follows.

3.1 System Design

Proteus is a fault injector intended for testing concurrent and distributed application. It is designed to be flexible, extensible and easy to use. It uses a client-server model where the server, called the *Coordinator*, manages one or more clients, known as *Agents*, who run alongside the target applications instances that are under test.

Agents share the process runtime of the target application instance and inject faults during its execution, while also sending metrics to the *Coordinator*. The *Agents* can alter the execution of the target application by attaching to the program at specific execution points, defined by the testers. When it attaches it is capable of altering the target's execution and state by adding additional behaviour to methods and/or altering the arguments and/or return values. *Agents* are also capable of keeping state dependent variables across

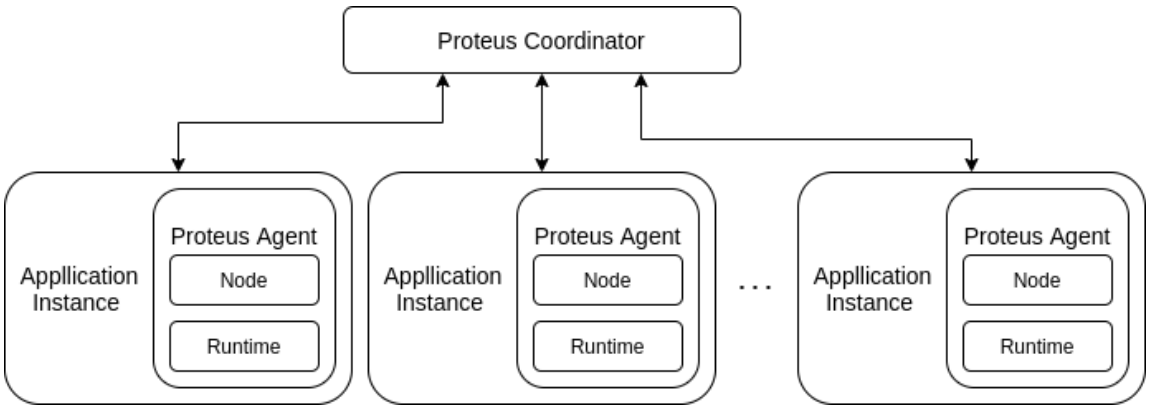


FIGURE 3.1: Overview of our system

the several fault injections, as well as share information with the *Coordinator*. In our work we divide the *Agent* into two parts, the *runtime* and the *node*. The runtime is static while the node is generated by the DSL and is responsible for fault injection.

Our *Agent's* runtime do not communicate with eachother so the *Coordinator* was developed with the objective of being a centralized structure responsible for coordinating the *Agents*, it is also responsible for managing the log data that is gathered during and after each run. It is independent from the Agent's and is not influenced by the DSL, the interaction with the Agent's is through RPC calls initiated by the clients.

3.1.1 Codebase adaptability

The design of our system separates the several components and allows us to implement them in different languages. Additionally the use of aspects to interact with the base system is a technique that exists in several programming languages. The use of RPC calls is also away to differentiate the client server component this would allow This adaptability allows us to test systems with distinct codebases, giving greater versitility to our tool. It does although complicate the deployment process since different languages have different necessities.

These design decisions were also made because in the future we aim at porting our *Agents* to other programming languages, and the capability of injecting faults in different nodes that have different languages is very interesting for the testing of distributed applications.

3.2 Schedules

Schedules are a plan that define a set of zero or more faults that will be injected into an application during a particular test run. This planning constitutes a fundamental aspect of fault injection. Our scheduling systems was designed to support many different execution scenarios, including:

- Independent fault schedule for each *Agent*, including fault free schedules (i.e., schedules with no faults);
- State based schedules, with conditions managed by *Agents* for triggering faults, the main forms of this are timers or rounds;
- Schedules with fault dependencies within an *Agent*. The capability of executing or not executing faults depending on the triggering status of other faults.
- Schedules with fault dependencies across different *Agents*. Similar to the previous fault but in a wider specter, that encompasses faults outside an *Agent*.

3.2.1 Schedule management

Regarding schedule management, Proteus offers two different approaches; Coordinator-based scheduling and Agent-based scheduling. These are necessary for supporting the different scheduling capabilities.

Coordinator-based schedules: the *Coordinator* is the entity responsible for distributing the schedules, using this approach the bootstrap process of each *Agent* requests the schedule to the *Coordinator*.

Agent-based Scheduling: the *Agent* has direct access to the schedule from some pre distribution method.

The Coordinator-based approach allows for a faster change in each run schedule since the schedule can be altered and sent, whereas agent-based has a simpler bootstrap process and, as a result of it, moves the complexity away from the *Coordinator* and into the *Agent*, which is useful because it makes the *Coordinator* simpler and more versatile. In Zermia, the Coordinator is responsible for sending the schedules, whereas in Proteus we opted for an agent-based approach, this decision was heavily influenced by the use of a DSL since its code generation allowed to embed the schedule directly into the code.

3.2.2 Triggers

In our work, a trigger is a set of conditions that, when met, lead to the injection of a fault. Triggers are distributed in the form of a schedule using one of the techniques mentioned previously.

The possible trigger scenarios, can be grouped into two types: state-based (dependant of certain application states); or event-based (dependant on some event), which can be further subdivided, into events that occur inside the current *Agent* or events that are triggered by the action of external *Agents*.

There are several ways of keeping track of trigger status. Particularly, three trigger models can be used for this purpose. These can be managed by: the *Coordinator*; the *Agent*, or using a *hybrid approach* where the *Coordinator* and *Agent* are involved.

Coordinator based triggering model

In this approach, the *Coordinator* has the responsibility of managing and triggering faults. *Agents* must query the *Coordinator* to understand the triggering status of each fault before its execution. The fault is injected or not depending on the response given by the *Coordinator* as presented in Figure 3.2.

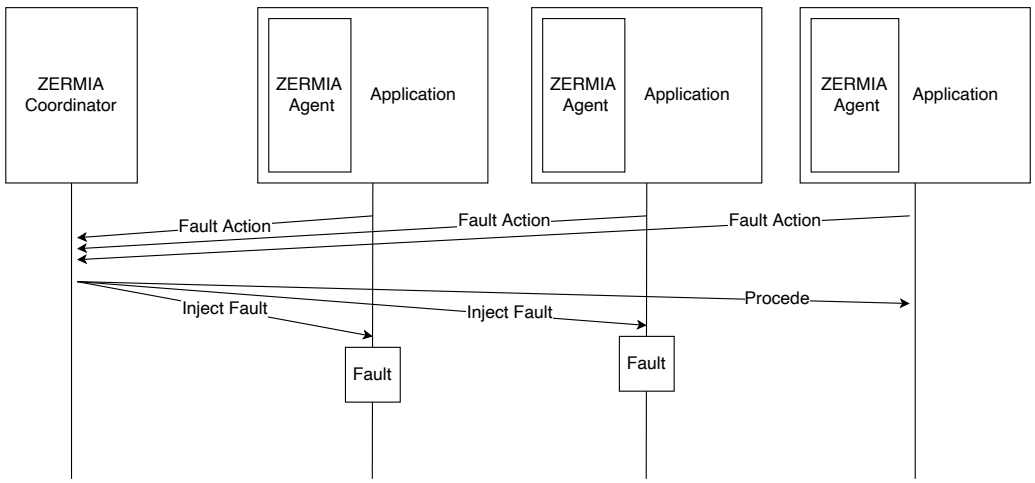


FIGURE 3.2: Coordinator based triggering model [5]

This approach is suitable when we want to track faults that have been triggered across different *Agents*, creating a global view that allows for the implementation of fault dependencies across different nodes. On the other hand, it creates a bigger overhead on the *Agents* since the application execution flow is altered due to the added synchronization step.

Agent Based Triggering model

In this approach the *Agent* is responsible for its own management and the coordination with all other *Agents*. This allows for an independent system that handles all its functions, to achieve this each *Agent* is required to keep track of the events and/or the state needed to manage their triggers.

The benefit of this system is the reduction in overhead, which leads to a less intrusive *Agent* that interferes significantly less with the normal execution flow of the system. To achieve coordination between *Agents* one possible approach would be to have them communicate directly, in a peer to peer way. This would require a greater amount of messages than if there was a centralized structure. A downside to both of the previous points is added complexity in the *Agent* implementation.

Hybrid Triggering model

The objective of this approach is reaching a balance between the previous two, so we can reduce the overhead created by the Coordinator-based approach, but have a centralized structure to handle coordination.

We achieved this balance by tasking the *Agent* with the management of all faults that do not depend on external factors, and sending the events that other faults depend on to the *Coordinator*. When it encounters a fault that requires an external trigger, a query is sent to the *Coordinator* to determine if the fault should be run.

Our implementation follows this approach and tries to reach a balance between reduction in overhead and improved control mechanisms.

3.2.3 Determinism

An objective present in fault injectors is achieving determinism, faults should always run in the same order so that runs with the same configuration yield the same results. The replication of findings is crucial to have a reliable system. Implementing methods through which we can reach determinism is especially true in our case where we handle several independent *Agents*.

One way to achieve predictability is to determine how the system will behave if several faults are to be injected by the same trigger. Zermia solves this by having a fault priority system that tries to find the biggest disruption to the correct operation of the base application, and also gives users the ability of specifying the priority of faults.

Our system does not have a fault priority system because of the degree of flexibility that we wanted to provide for our users. It is possible to specify the order of the faults to be executed and dependencies across them, which still allows the user to specify the priority of the faults, but this is done through the detailed writing of the faults, which, although more complex, it further reduces non-determinism, and makes the order of the faults execution much more transparent.

3.3 DSL Design

The largest innovation brought by Proteus is the creation of a DSL that facilitates the creation of faults and schedules. Through its specific grammar, testers can create fault schedules and configure test runs that execute them.

3.3.1 Fault

A *Fault* is a high level abstraction that is associated with a specific location in the code. It is composed by a set of *fault execution parameters* that complement the required information for each fault. We designed the system with this separation so that a location in the code can have several simultaneous faults.

Faults are characterized according to four different parameters or characteristics. These serve as the building blocks of our DSL, and include:

- **Who** is to run this fault, specifies what *Agents* are supposed to run the fault;
- **When** should a fault be injected, specifies the condition that must be met for the fault to be injected;
- **Where** in the application under test should the fault be injected, specifies the point in which the fault is to be injected;
- **What** is the code to be injected, specifies the code associated with the fault;

The execution parameters in conjunction with the *Fault* must define three of the four fault characteristics: *What* is to be executed; *When* should it be executed; and *Where* should the fault be executed.

The *What* parameter has the fault code to be injected. It seamlessly integrate with the code present in the application under test by using *AspectJ* and the capabilities associated

with it. It can use predefined faults, with their specific arguments, or custom faults with their specifically designed code;

The *Where* parameter has two components, the method in the code base where the fault should run, and if the injection should occur *before* or *after* its execution. This is a significant difference because different objects are accessible before and after the execution. With *before* you can alter the arguments that the function will use, while *after* you can change the return values.

The *When* parameter defines when should a fault be injected (i.e., triggered). It fulfills many of the requirements presented in Section 3.2. It handles state based conditions like rounds, timers and custom conditions. It manages fault dependencies in relation to faults inside and outside the current *Agent*.

One feature present in our design is the re-usability of each construct. These are identified by an unique ID, which can be seen as a variable name. These IDs have several uses: in the case of faults they are used to allow them to be referenced in distinct parts of the schedule; similarly to the execution parameters that allows for a custom built parameter to be referenced at different locations of the schedule, making it easier for testers to reuse personalized parameters.

3.3.2 Fault Conditions

This component allows a fault's execution to depend on the triggering status of other faults inside the current *Agent*. We represent the other faults through fault conditions, these are events that when met signal other faults.

These component was designed to be similar to a *Fault* so as to simplify the DSL. It differs in two ways, it does not have a *What* parameter, since it isn't responsible for injecting code and, it is a single unified structure that is not divided using execution parameters, this was made since a more atomic structure is easier to reference across the system.

Besides the scheduling capabilities it is also used in synchronization points and the logging system. It was designed this way as to more deeply integrate these components with the rest of the system.

3.3.3 Syncpoints

The requirement of a trigger that is dependent of faults in different nodes is fulfilled through the use of synchronization points, referred to as *Syncpoints*. This synchronization process

requires interaction with the *Coordinator*.

Syncpoints are represented by a list of nodes and corresponding fault conditions. The fault conditions must be met in all nodes of the list for the *Syncpoint* to be reached.

Syncpoints create a synchronization/interaction between the Agents and the Coordinator, relying on the response from the latter to execute, and are referenced within the *When* parameter of a particular fault.

A *Syncpoint* can appear in multiple *When* parameters. This combination of multiple dependants allows the system to have a group of nodes perform an action only when another group has completed theirs. This can be escalated so that everyone depends on everyone, which effectively stops everyone until they are at the same position. This feature allows for a great degree of coordination between several nodes.

3.3.4 Logging

We designed Proteus to log every action we perform in locations we control, along with the default logging the user may define additional logging through the use of the *Logging* component.

Our system saves execution data through the use of log entries sent from the *Agents* to the *Coordinator*. Our objective is to send relevant general fault data by default, and allow the tester to include custom data as needed. The *Coordinator* will be the centralized structure that receives all logs and performs some basic analysis and saves them for later analyses by the testers.

3.3.5 Test Run Configuration

This is where we fulfill the last requirement of our schedule design, with the capability of each node to have it's own set of customized faults.

We also use this structure to gather relevant variables that each node requires like the network location of the *Coordinator* and *Agents*, and the base method where the *Agent* runtime environment should attach to.

The schedule is defined using a list of faults that will be executed in each node. We also allow for the creation of logging locations, which is mainly useful in nodes with no faults, where we would still like to log information for comparison.

3.3.6 Language Format

The format of our language was a design decision made earlier in the development. Two possibilities were considered for this: first, the JSON schema format, the pros of this approach being its wide adoption, where the added familiarity allows for a faster learning curve, and the fact that it has also been used in projects that implement code generation, like openAPI [60]. The cons are the simplicity of the JSON schema, although it is one of it's greatest strengths, it proved limiting for our implementation. In Listing 3.1 we can see a prototype that was developed based on the JSON schema.

```
{
"CoordinatorLocation" : "10.10.10.10:8080",
[
    "Node" : {
        "nodeLocation" : "10.10.10.11:8080",
        "id" : "nodeA",
        "faults": {
            "ID": "simpleInjection"
            "pointcut" : "(void exp.main.Main.main(..)",
            "execParams" : [
                {
                "what": "sleep",
                "params": {
                    "duration": 10,
                },
                "when": {
                    "type" : "round",
                    "start": 5,
                    "end": 50,
                    "interval": 5
                },
                "where": "before"
                }
            ]
        }
    }
]
}
```

Listing 3.1: Example of the DSL using a JSON schema

The second option was a syntax developed by us that gets inspiration from the JAVA syntax, which is more modular and easier to add new features because it is not tied to a predefined schema. It has a higher learning curve because of it's unfamiliarity, which we

tried to offset by having a consistent syntax across it's several functions that resembles JAVA. In Listing 3.2 we can see an early prototype of this approach.

```
RunConfiguration{
    RuntimePackage = "package exp.aspect;";
    CoordinatorLocation = "10.10.10.10:8080";
    Node nodeA (firstAttachPoint = "bftsmart.demo.counter.CounterServer.main"
        ; location = "10.10.10.11:8080" ; simpleInjection)
}

Fault simpleInjection {
    pointcut = "(void exp.main.Main.main(..)";
    ExecParams sleepPeriodically {
        where = Before ;
        whatToExecute{sleep(10)}
        whenToExecute{betweenRound(5,50,5)}
    }
}
```

LISTING 3.2: Example of the DSL using JAVA like syntax

The presented examples describe the essential same situation, while not an exact translation, since in the first example we are using arrays to describe the objects and in the second we use variable names as references. This is just a small look at the capabilities of our language, the full syntax and its detailed explanation will be present in Chapter 4.

3.4 Agent-Coordinator communication

The *Coordinator* and the *Agent* interact through the use of RPC calls, three messages were defined to allow this interaction, they represent each of its functions. In the diagram presented in Figure 3.3 we display a representative interaction between the *Agent* and the *Coordinator*, the filled lines represent the transmission of essential data, the doted lines mean that non essential data is being transmitted.

The *First Connection* enunciates the presence of the *Agent* and informs the *Coordinator* of the necessary synchronization points related information. The *Logging messages* send information relative to the execution of the fault. The *Syncpoint* message represents a request to the state of a synchronization point made by the *When* parameter to the *Coordinator*.

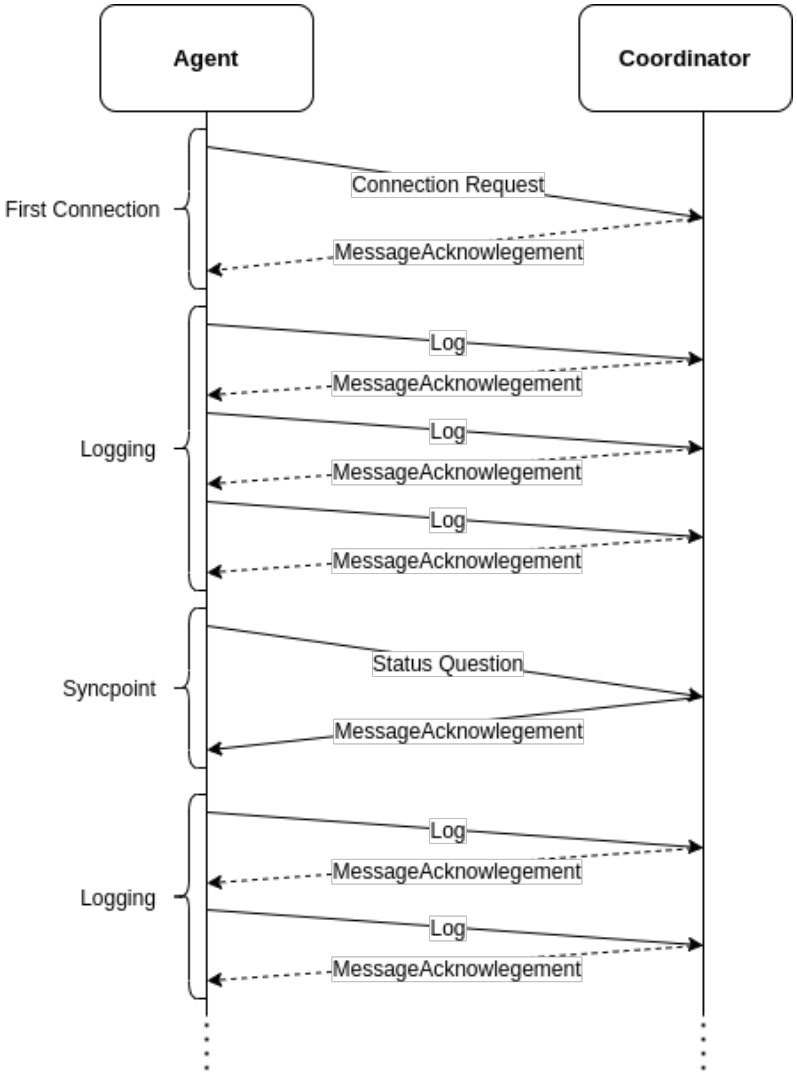


Figure 3.3: Diagram demonstrating example interaction between *Agent* and *Coordinator*

Chapter 4

Implementation

We will now describe some of the implementation details of Proteus. First, we describe how the DSL was built, its grammar and all it's capabilities. Next, we discuss relevant processes present in the code generation phase, responsible for creating the code necessary for the operation of our system. Finally, we explore the infrastructure of the system, the communications required to make the constructs presented in the DSL work, and how are the main structures implemented.

4.1 Grammar

Several tools are used to build Proteus. Starting from top to bottom, we first have *Xtext*. This is the framework used to generate the code. The first part of this process is parsing the file according to the grammar that we will show throughout this section.

As mentioned before (see Section 2.3.3), our grammar follows a tree like structure. Our explanation will follow the structure of the tree, similar to the path taken by the parser.

4.1.1 Root

This tree like structure starts at the top with a meta model that is used to enumerate the several root types in our language. These root types are described by the *Types* structure, as presented in Listing 4.1.

```
Types:
    RunConfiguration | Fault | FaultCond | Where
        | CodeBlock | Condition | Syncpoint | Log
;
```

Listing 4.1: Root types of the Grammar

The base structure is divided into two types: the base elements of the language, that can be referenced independently, mainly *RunConfiguration* and *Syncpoint*; and the structures that can only be created as a root element, to make them easier to reference inside the objects from the first type.

The *RunConfiguration* is the starting point for the configuration of the system, its independence makes it a base type that can not be referenced in any other place. The *Syncpoint* is another base type because it is a feature that influences many locations and is tied to many other structures, so declaring it independently makes it easier to understand its relationships with other structures.

4.1.2 RunConfiguration

The run configurations defines the necessary information needed for the execution of the framework, including the *Coordinator* and *Agents* network address and port number, names and ids, etc. It also defines the *Node* structures, used to identify the *Agents* nodes, all other components are IDs that reference other structures in our grammar, a design that simplifies and improves the readability of the schedule. Every node must have a single *RunConfiguration* structure.

Nodes are composed by:

- *firstAttachPoint* - The base function of the program where the runtime of our application should bind in order to run alongside the rest of the application;
- *nodeLocation* - The location of the *Agent*, hardcoded by the tester using a similar structure of IP:Port;
- *logData* - The *Log* structures that we wish to include alongside our *Agent*;
- *faultIDs* - The list of fault IDs that we wish to integrate with a node. The keywords 'none' and 'all' are valid and can be used to represent none or all faults.

Listing 4.2 presents the grammar that describes the *RunConfiguration* structure.

```
RunConfiguration:
    'RunConfiguration' '{' 'RuntimePackage' '=' runtimePackage = STRING ';'
    'CoordinatorLocation' '=' CoordinatorLocation = STRING ';' nodes+=(Node)* '}'
;
Node:
    'Node' name=ID '(' 'firstAttachPoint' '=' firstAttachPoint = STRING ';'
        'location' '=' nodeLocation=STRING (';' 'logData' '=' logLocations+=ID*)? ';'
```

```
        faultIDs+=(ID)* ')'
;
```

Listing 4.2: Grammar that describes a RunConfiguration

4.1.3 Fault

The *Fault* is the fundamental structure of the system and is referenced in the *RunConfiguration* through its unique identifier.

All faults have a unique identifier, a location called *pointcut* and a list of *ExecParams* structures. The *pointcut* specifies the method where the fault should execute, it uses the *pointcut* syntax [61] and is specified by a string. The last value it receives is a list of *ExecParams*, short for execution parameters.

The grammar used to define a *Fault* is presented in Listing 4.3.

```
Fault:
    'Fault' name=ID '{' 'pointcut' '=' pointcut=STRING ';' (execParams+=ExecParams)* '}'
;
```

Listing 4.3: Grammar that describes a Fault

4.1.4 ExecParams

The name of the *ExecParams* structure is optional which leads to several possible naming schemas that will be explained in Section 4.1.8.

The *ExecParams* receives three objects, the *Where*, *When* and *What* parameter.

The *Where* parameter can have two values, 'Before' or 'After' as specified in the *EnumRule JOIN_TYPE*, which tells the injector if the fault should be executed before or after the execution of the method that was specified in the faults pointcut. The effects of both options will be explored in Section 4.2.

The other two parameters will be explained in the following sections.

The grammar that describes an execution parameter is presented in Listing 4.4.

```
ExecParams:
    'ExecParams' '(' name=ID ')'
        | 'ExecParams' (name=ID)? '{' 'where' '=' where=JOIN_TYPE
        (';' 'whenToExecute{' when=When '}')? ';'
        'whatToExecute{' what=What '}'  '}'
;
enum JOIN_TYPE:
    BEFORE='Before' | AFTER='After'
```

```
;
```

LISTING 4.4: Possible Root types of the Grammar

What

The *What* parameter is where we manage the fault code to be injected, it can use predefined or custom faults. There are six distinct predefined faults that can be parameterized using arguments.

This construct holds several *ParserRules*, which are the leafs of the tree, since no *ParserRules* derive from them. Therefore, the *What* parameter can be seen as a terminal structure that completes a branch of the AST tree.

To understand the predefined faults and their respective arguments there is a concept that should be referenced, namely that in every fault the tester has access to the arguments of the original method. When in this grammar we refer to the *INT* of the *originalData*, we require the tester to introduce a number that represents which argument should be altered. This will further be discussed in Section 4.2.

The faults that have been implemented, and the parameters they receive are as follow:

- *AlterPacketFault* - receives two arguments, the *OriginalData* and the kind of alteration, there are two possible kinds, in one we fill the packet with random data, in the other the packet is filled with the zero byte value;
- *BigPacketFault* - increases the size of an object, it receives the *OriginalData*, and another integer that represent the size increase desired;
- *LeakFault* - increases the usage of memory, this is achieved through the use of a large array of bytes. It receives an argument that specifies the number of leaks that should occur;
- *Exit* - stops the execution of the program, the argument received is the exit type to be returned.
- *Sleep* - sleeps the thread by an amount of seconds specified by the tester;
- *CPULoader* - increases the usage of the CPU, an argument is received with an INT that regulates how intense the usage should be.

- *skipExecution* - takes advantage of the *AspectJ* capabilities to bypass the target function execution. This may lead to problems whenever the target function returns a value, since it could lead to an inconsistent state. However, it is useful in the end of an execution flow where a certain action, like sending data, wants to be skipped. It must be placed as the last execution parameter since no further execution happens beyond it.

- *RepeatExecution* - repeatedly executes the target method by taking advantage of *AspectJ* capabilities. It receives an integer as an argument that dictates how many times should the loop be executed.

The custom faults are described by two rules, the *ParserRule CodeBlock* and the *TerminalRule CODE_BOX*, as can be seen in the grammar presented in Listing 4.5. Their functioning will be explored in Section 4.1.8.

The grammar that describes this structure is presented in Listing 4.5.

```
What:
    codeBlock = CodeBlock | sleep = Sleep | exit = Exit
        | bigPacketFault = BigPacketFault | leakFault = LeakFault
        | alterPacketFault = AlterPacketFault | skipExecution=SkipExecution
        | repeatExecution=RepeatExecution
;
RepeatExecution:
    'repeatExecution' '(' times=INT ')'
;
SkipExecution:
     'skipExecution'
;
AlterPacketFault:
    'alterPacketFault' '(' originalData = INT ','
        kindOfAlteration = AlterPacketFaultType ')'
;
enum AlterPacketFaultType:
    RANDOM='random' | ZERO='zero'
;
LeakFault:
    'leakFault' '(' nrofLeaks = INT ')'
;
BigPacketFault:
    'bigPacketFault' '(' originalData = INT ',' timesLarger = INT ')'
;
Exit:
    'exit' '(' type=INT ')'
;
```

```
Sleep:
    'sleep' '(' time=INT ')'
;
CPULoader:
    'cpuLoader' '(' amount=INT ')'
;
CodeBlock:
    ('BlockID' '(' name=ID ')')
        | ('CodeBlock' (name=ID)? '{' (('imports' '{' imports=CODE_BOX '}' ';'
        'code' '{' code=CODE_BOX '}''}')|( code=CODE_BOX '}')))
;
terminal CODE_BOX:
    '///' -> '///'
;
```

LISTING 4.5: Grammar that describes a What parameter

When

The *When* parameter specifies the fault trigger conditions. We offer a lot of flexibility that allows for several ways of triggering faults we will explore them in this section.

The *ParserRule Condition* is the structure that allows for the creation of custom conditions by the tester, the user introduces a block of code that must return a statement of true or false. The *Condition* is a referenceable structure that can not function alone so it must be used in conjunction with one of the structures that we will now analyze.

The three simpler structures created using this system are the between conditions and the random chance, we will introduce them first:

- *random* - triggers the fault with some probability based on a normal distribution. This receives an integer with a value that must be between 0 and 100, that defines the probability of triggering;

- *betweenSeconds* - triggers the fault between some period of time, using the internal system clock. The period is specified using two variables, a start and an end, counted in relation to the beginning of the fault. Each of the parameters can have a value of zero, being considered *null*. This allows for faults that should be triggered after some instant and until the end of the program; triggered since the start of the program and until some specific instant; or, triggered between some predefined start and end instances;

- *betweenRounds* - this trigger is heavily influenced by Zermia because it was used as the main conditional format. In our implementation we consider a round to be completed every time the execution reaches this condition. Similar to *betweenSeconds* it receives a start and end argument, with similar interpretation in relation to when to trigger, it also has an additional argument called interval, that only triggers a fault every X amount of times, being X defined by the tester. Another feature is the option to receive a *Condition* as an argument, if this is set than the round is only incremented if the condition returns true. Finally it can also receive a *random* structure as an argument, which is identical to the one that was previously explained, it is inserted after the round check, as an added condition.

The last trigger is the most direct way of using the *Condition* rule. The *ifCondition* receives a *Condition*, and executes the *What* parameter if the *Condition* returns true.

The grammar that describes this structure is presented in Listing 4.6.

```
When:
     ('ID' name=ID)?
        ( betweenRounds = betweenRounds | betweenSeconds = betweenSeconds
        | random = Random | 'ifCondition' '(' ifCondition=Condition ')'
        | faultCond=FaultCond | 'Syncpoint' '=' syncpoint=ID)
;
Random:
    'random' '(' chance = INT ')'
;
betweenSeconds:
    'betweenSeconds' '(' start = INT ',' end = INT ')'
;
betweenRounds:
    'betweenRound' '(' start = INT ',' end = INT ',' interval = INT
        (',' condition=Condition)? (',' random = Random)? ')'
;
Condition:
Condition:
    ('CondID' '(' name=ID ')')
        | ('Condition' (name=ID)? '{' (('imports' '{' imports=CODE_BOX '}' ';'
        'code' '{' code=CODE_BOX '}''}')|( code=CODE_BOX '}')))
;
;
```

Listing 4.6: Grammar that describes a When parameter

4.1.5 Fault Conditions

A fault condition is a structure used to define a *Fault* that does not inject the fault code, they are used by several features, but their original objective was creating fault dependencies inside the *Agent*.

This is a referenceable structure which means the naming is optional. The *Where* parameter is the combination of the pointcut and the 'before' or 'after' option, as described in Section 4.1.3. The *When* parameter is the same as the explanation given in Section 4.1.4, an *extraData* parameter was added to allow users to log custom data using the logging capabilities of our system which will be further analysed in Section 4.1.7.

A *FaultCond* may be used in conjunction with the *When* parameter. This allows this structure to be used to create fault dependencies in an *Agent*, e.g., only triggers a fault if some previous fault has been triggered.

The grammar that describes this structure is presented in Listing 4.7.

```
FaultCond:
    ('FaultCond' name=ID)
        | ('FaultCond' (name=ID)? '{' where=Where ';'
        'whenToIncrement{' when=When '}'
        (extraData=CodeBlock)? '}')
;
Where:
    'whereToExecute' '{'(joinType=JOIN_TYPE ';')? name=STRING '}'
;
```

LISTING 4.7: Grammar that describes a Fault Condition

4.1.6 Syncpoints

Synchronization points are a complex system that can create fault dependencies between *Agents*, the information they require is obtained using the *Syncpoint* structure in conjunction with the option available in the *When* parameter.

A *Syncpoint* depends on several nodes as was explained in Section 3.3.3, this nodes are listed using their unique identifier. There are two types of synchronization points, *recurrent* and *oneTime*, as well as two ways of dealing with a false response, *continue* or *retry*. The *Syncpoint* is reached when the fault conditions of all the nodes that it depends on is met.

In the *Where* parameter we can use the option *syncpoint* with the desired *Syncpoint* ID. This creates a place where in order to execute the *What* parameter the synchronization

point must be true, if it is false the execution can continue or retry after a predetermined time, depending on the value set in the *Syncpoint* parameter *syncpointOnFail*.

The *syncpointType* describes two different types of *Syncpoint*, *oneTime* is only met once and becomes true, the *recurrent* option creates a *Syncpoint* that can be reached multiple times, the specifics of how this works will be explained in Section **??**.

Listing 4.8 presents the grammar of this structure.

```
Syncpoint:
    'Syncpoint' name=ID '(' nodesDependant+=ID* ';'
        'type' '=' type=syncpointsType ';'
        'onFailure' '=' onFail=syncpointOnFail ';'
        faultCond=FaultCond ')'
;
enum syncpointOnFail:
    continue='continue' | retry = 'retry'
;
enum syncpointType:
    oneTime='onetime' | Recurrent='recurrent'
;
```

Listing 4.8: Grammar that describes a Syncpoint

4.1.7 Logging

This component of the logging system can be used by testers to create custom logging locations, these locations are defined by *FaultConds*.

This structure receives a *FaultCond* and two *CodeBlocks* one with extra data to be logged, and another with custom state based variables created by the tester.

The *FaultCond* is mandatory and defines this component's main correlation. A *FaultCond* is used because it is able to interact with the program without affecting it and has predefined logging positions. The other two components are optional and enhance the logging capabilities.

The *extraData* structure is optional and allows the tester to add custom logging to the *FaultCond* by using a *CodeBlock*, how this data is added will be explored in Section 4.2.5.

Using the structure *stateBasedVariables* the tester can create custom state based variables that can be manipulated in future code boxes, it receives a *CodeBlock* with the initialization of those variables.

The grammar that describes this structure is presented in:

```
Log:
```

```
    'Logging' name=ID '{' faultCond=FaultCond
        (';' 'extraData' '=' extraData=CodeBlock)?
        (';' 'stateBasedVariables' '=' stateBased=CodeBlock)? '}'
;
```

LISTING 4.9: Grammar that describes a Log

4.1.8 Custom code and referenceable objects

In this section we will give an overview of how the referenceable objects can be represented, to explain this we present an example with the *CodeBlock* object:

```
BlockID ( createdCode1 )

CodeBlock {
///
    /*custom fault code*/
///
}

CodeBlock createdCode1 {
///
    /*custom fault code*/
///
}
}
```

LISTING 4.10: Example of possible naming schemes

As can be seen Listing 4.10 this structure can be written in 4 different forms, being the first three common across many referenceable structures.

Referenceable structures allow testers to determine if the structure should be local or referenceable at future points. For the code to be local testers can use the second method, since it does not have a name this code is not referenceable in the future. If a name is supplied than this object can be referenced in the future, as seen in example 3, an example of this reusability is presented in the first example.

The actual code of the faults represented in Listing 4.10, with the comment */*custom fault code*/* is a custom *TerminalRule* called *CodeBox* that has the purpose of holding code, it is also used in the *Condition* structure. It was designed by us since there is no standard preincluded *TerminalRule* that can handle this purpose. The optimal solution would be for *Xtext* to comprehend the brackets and understand when the code ends, but to the extent of our knowledge it is not capable of doing so through conventional methods,

because of this a set of symbols was used to delimit the beginning and ending of the code, we decided to use triple forward slashes, since it is a symbol combination with no purpose in a normal Java program.

```
CodeBlock createdCodewithImports {
imports {
///
    /*required imports*/
///
},
code {
///
    /*custom fault code*/
///
```

Listing 4.11: Example of the import feature with CodeBlocks

The example given in Listing 4.11 is regarding custom code, so it applies to *CodeBlocks* and *Conditions*. Here we have a depiction of the optional parameter used to manage imports and allow users to use objects from other locations in their custom code.

4.1.9 Discussion

As was mentioned in the state of the art Section 2.3, a DSL should only have one way of doing something, so one might question the referenceable structures presented in Section 4.1.8, we went against this principal in this feature, it is not a crucial feature but we consider that giving the possibility of code reuse is a healthy compromise since it can improve code readability and prevent code duplication, despite the added complexity in the code compilation phase.

A question that may arise is why we reuse the Fault Conditions in different circumstances instead of relying on the normal faults or creating specific structures for each component. This structure was reused because in our system there is essentially one way of interacting with the program under test, which is the use of aspects. The two structures that implement Aspects are the faults and the fault conditions. This led us to use fault conditions whenever we want to interact with the base application but do not need to alter its functioning.

4.2 Code Generation

Code generation transforms the schedule described using our DSL to the corresponding code that implements it. Throughout this section we will explain the inner workings of the code generation process and analyse the files that are created by this process. The code generation is implemented using *Xtend*, an extension of the Java programming language, that was presented in Section 2.3.4. We use code generation and compilation interchangeably, since the code generation results from the compilation process of the schedule written using our DSL.

The compilation follows the tree like structure of the grammar, where for each object we call the corresponding *compile* method that returns a string with the object corresponding source code. Since many objects are cross cutting and/or referenced in diverse locations, several globally accessible data structures are used throughout the object compilation.

4.2.1 Object analysis

The first step in the code generation process is a preliminary analysis of all the structures that are used in the *schedule*. We achieve this by mapping the objects into globally accessible *HashMaps* and *Arraylists* data structures, using the structure *name* as key and the respective generated code as a value. All structures without a name will be given one, this facilitates the creation of state variables.

```
1  HashMap<String, String> codeBlocksMap = newHashMap();
2  ...
3  var counter = 0;
4
5  for(block : resource.allContents.toIterable.filter(CodeBlock)){
6      if (block.getName === null){
7          block.setName("codeBlock"+counter);
8          counter ++;
9      }
10     codeBlocksMap.put(block.getName, block.code);
11 }
```

LISTING 4.12: Creating names for variables

Listing 4.12 presents the code that manages this process. When referencing *codeBlocks*, if the object does not have a name (line 7) it is assigned one by using the syntax *objectType+counter*. This ensures that no two objects have the same name. The referenced objects are put in a *HashMap* for constant access time across the code generation process.

Note that the *newHashMap()* method call (line 1) is the initialization of a *HashMap* object through the use of a function made available by the *Xtend* language.

4.2.2 Fault Compilation

The next step of the process is the compilation of the *Fault* objects into their respective code. While compiling we make a list of all the objects it depends on, to more easily manage them later in the node creation.

All faults are compiled and put into a *HashMap* with their name and their compiled version in the String format. The code used to compile the *Fault* object can be seen in Listing 4.14.

```
1  fault.compile
2
3  private def compile(Fault fault)
4  '''
5  «FOR ExecParams exec : fault.execParams»
6      «IF exec.when !== null && exec.when.betweenRounds !== null»
7          int «exec.getName»counter = 0;
8      «ENDIF»
9  «ENDFOR»
10 @Around("«fault.pointcut»")
11 public Object «fault.name»(ProceedingJoinPoint joinpoint){
12     Object retobj = null;
13     String randomLogData = "";
14     long startTimestamp = System.currentTimeMillis();
15     Object[] args = joinpoint.getArgs();
16     «execCompileConditions(fault.execParams, 'Before', fault.name)»
17     try {
18         retobj = joinpoint.proceed(args);
19     } catch (Throwable e) {
20         e.printStackTrace();
21         System.exit(0);
22     }
23     «execCompileConditions(fault.execParams, 'After', fault.name)»
24     ProteusRuntime.sendLogs(curNodeName, "«fault.name»", startTimestamp, randomLogData);
25     return retobj;
26 }
27 '''
```

Listing 4.13: Snippet of code responsible for fault compilation

The compilation starts with a call to the *compile* method (line 1). When compiling, *Faults* recursively call the *compile* method of all their respective objects.

The fault compilation starts with the initialization of all state based variables that will be required for the correct functioning of all the *When* parameters (lines 5-9). At this stage, the only state based variables required are iterators that can hold the number of rounds when the *betweenRound* option is used.

The pointcut specified by the tester is created, using the *@Around* pointcut notation (line 10).

All variables required for the *Fault's* execution are created (lines 12-15). The *args* array holds all the objects of the original method and the *retobj* the returned value of the method execution. The other two objects are related to logging and will be discussed in Section 4.2.5.

The original method is executed with the arguments provided (line 18), and is part of the *@Around* pointcut flow. We use this repeatedly in the *repeatExecution What* parameter, in order to repeat the execution of the method. If we return the function before its execution (line 18) than the function does not execute. This is the basis for the *skipExecution* parameter when used in the *What* attribute.

The original method return value is assigned to *retobj* (line 25), thus allowing altering this values if needed.

The *ExecParams* object is compiled using the *execCompileConditions()* function (line 16 and 23). The location where each *ExecParams* is compiled depends on their respective *Where* value, which can be 'before' or 'after' the execution of the method, the 'before' *ExecParams* can access and alter the arguments in their custom code, the 'after' *ExecParams* has access to the arguments and can alter the return object of the function in its custom code.

The *execCompileConditions()* function references the corresponding *ExecParams*. This, in term, compiles the *When* parameter, using the corresponding code.

The *When* parameter is responsible for creating code for all possible conditions. This includes: a general *if* statement, when using the *betweenRounds* along with managing the increment of the counter that keeps track of rounds; two *if* statements to handle the *random* and *betweenSeconds* options; another *if* statement for the fault condition, that integrates with the globally accessible variables created by *FaultCond* object (further discussed in Section 4.2.3); and, the *if* statement responsible for handling synchronization points, which involves interacting with the *Coordinator*.

4.2.3 Fault Condition

Fault conditions are used in several features of our code and constitute another way through which we interact with aspects. In *FaultConds* we use the *@Before* and *@afterReturning* pointcut types, depending on the location specified in the *Where* parameter.

These structures are similar to *Faults* without the state altering code of the *What* parameter, and are less intrusive to the system because of the distinct pointcut types through which it interacts with the base application.

The compilation of the *When* parameter in *FaultConds* is similar to the compilation of a *Fault*, with the added behaviour of controlling a globally accessible variable that is passed using the *What* parameter. This variable conveys the state of this *FaultCond* to other objects. It is a *Boolean* object that is set to the result of the *FaultCond*'s *When* parameter.

4.2.4 Syncpoint

Syncpoint objects can be referenced in two locations; the *Syncpoint* option present in the *When* Parameter which creates a communication with the *Coordinator* questioning the *Syncpoint* status; and the *FaultConds* associated with a *Syncpoint* that informs the *Coordinator* of the synchronization point status through the use of logs. The actual objects sent will be discussed in Section **??**.

When the interaction fails two outcomes can be defined, depending on the *syncpointOnFail* argument. If set to 'continue', the execution of the fault proceeds as normal. If it is set to 'recurrent', the system enters a loop polling the Coordinator to determine if the synchronization point state has been altered.

4.2.5 Logging

Code generation for logging information is divided into two parts. First, the logging that takes place during the execution of the fault injector. Second, the logging performed by the *Logging* object available in our grammar. We will explain this two parts in this section but the actual information and the structure of the objects sent to the *Coordinator* will be discussed in Section **??**.

Proteus collects logging data at the end of every *Fault* and *FaultCond*, as presented in Listing 4.14 (line 19). There are two types of logging data: *constant data*, containing the

fault description and the time when it was executed; and, the *ExecParams* data, logged every time a *When* parameter is evaluated as *true*.

We allow log data customization, where testers can input custom string values to the log as presented in Listing 4.14. This attribute is present in every *Fault* and *FaultCond*, and appends the string to the *randomLogData* variable.

The *Logging* object allows logging specific information. It is implemented as a *FaultCond*, so it automatically has access to the logging capabilities present in this structure. This capabilities can be complemented by using custom code introduced through the use of the *extraData* combined with the *randomLogData* string.

The *stateBased* parameter, is an optional *Logging* parameter that is handled in a separate part of the code generation. These are state based variables that are added to the header of the node's source file, as discussed in Section 4.2.6.

4.2.6 Node creation - Customizing an Agent

In Proteus, a *Node* represents the customization part of an *Agent*, that together with the *Agent runtime* executes alongside the application. A *Node* is responsible for assembling all structures that we have analysed so far into the corresponding source file. Figure 4.1 presents the essential organizational structures of this file, where each part is located, and how it is divided.

The header can be divided into four sections:

- Proteus dependencies, that maintains the necessary dependencies related to the Proteus runtime;
- Custom dependencies, that maintains the necessary dependencies related to the test application;
- State variables, that maintains the static variables and all data structures needed. The dynamic component holds variables which are accessed throughout the program, examples include the address of the *Node* and the *Coordinator*, the name of the *Node*, if it is faulty, the list of the *Syncpoints* related to this node, etc.;
- Custom state variables, maintain the *stateBased* arguments of *Logging* objects.

The Proteus runtime junction, is the Aspect that binds to the test application during boostrap. It initializes the runtime of the system and the communication infrastructure. In

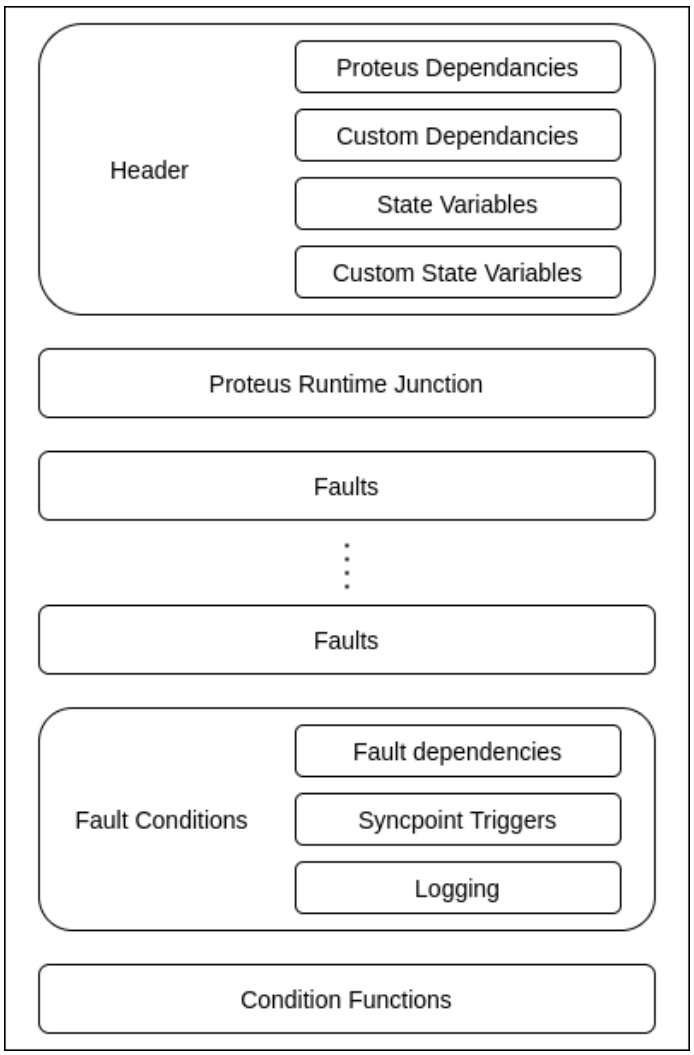


Figure 4.1: Diagram demonstrating the Node Structure

this function we create the first connection to the *Coordinator*, passing it the information required for the bootstrap process.

The *Faults* maintains the associated faults for the particular node, listed according to their *faultIDs*.

The *Fault Conditions*, unlike their sibling structure *Faults*, have not been previously compiled because there could be additional code associated with the *Log* object. Three distinct types of *FaultConds* are added:

- Fault dependencies, that include dependencies that are required for *When* parameters of the *Fault* objects;
- *Syncpoint* Triggers, that are enumerated in the *Syncpoint* objects; and,
- *Logging*, that are objects which are specified in the *RunConfiguration*.

Finally, the *Condition* functions include all necessary functions used in the *When* parameter when creating custom *Conditions*. During the implementation of every object, we keep track of what conditions are necessary for each node.

4.2.7 File Structure

The DSL's compilation of the grammar outputs several files. Three of them are static and are required for the execution of the *Agent*. Additionally a file for each *Agent* is created. These files can be described as follows:

- *ProtoRuntime.proto*, a static file that contains the model for the gRPC calls that are used throughout the system, the contents of this file will be explained in Section 4.3.1.
- *helper.java*, a static file composed of fault inducing code needed for the runtime. It contains several functions which are called from the dynamic files. However, since their results are fairly standard and predictable they are implemented in a separate file, this reduces clutter in the main file.
- *ProteusRuntime.java*, a static file responsible for the *Agent runtime* and interations with the *Coordinator*.
- *<nodeName>gen.java*, where *<nodeName>* is the name of the corresponding node. This file results from the code generation process.

All files are stored into a folder named according to the *Agent*, for easier management, the static files are replicated into each folder.

The files responsible for the execution of the *Coordinator* are completely independent and are not handled in the code generation. In our vision the *Coordinator* works out of the box and does not require any configuration files, the configurations for each run is received in the bootstrap process.

4.2.8 Discussion

The code generation process provides a separation between the grammar, the code generator and final source code. It is easily extensible and allow for the expansion of this system into a new programming language, since the code generator can be retrofitted to allow the generation of any language. This fulfills the task we set out to achieve in Section 3.1.1.

The *stateBasedVariables* feature of the *Log* object has some interesting repercussions across our system. While its main purpose is extending the logging capabilities of the *Logs* objects, these variables are accessible by any *CodeBox*. Thus, it also allow user controlled state based variables to exist across Proteus.

Regarding how we handle fault conditions dependencies, on the *Faults* side. We decided to set a *FaultCond* state to false whenever it is used by the *When* parameter of a *Fault*. in our vision when an entity acts according to a dependency, than that dependency is no longer true, and must be met again.

The pointcuts used in our system vary depending on if its a *Fault* or a *FaultCond*. *Faults* use the *@Around* pointcut and *FaultConds* use the *@Before* or *@afterReturning* depending on what's specified in the *Where* parameter. The *Faults* use the *@Around* pointcut because of its wide array of capabilities that allow us to standardize the structure of all *Faults*. *FaultConds* use more specific pointcuts that are not as impactful in the performance of the system. These pointcuts also always run before or after the *@Around* pointcut so we can be sure that the conditions will execute before/after the fault.

4.3 Infrastructure

In this section we will describe the implementation details for the *Coordinator*, the *Agent* runtime, and interactions between the two components by detailing the gRPC interface implementation and method calls.

4.3.1 gRPC Implementation

The *Coordinator* is implemented as a service (i.e., server) that listens for contacting *Agents* (clients), by implementing the gRPC interface presented in Listing 4.14. Its main functionalities are: i) registering *Agents* (line 2); ii) synchronizing fault injection (line 3); and, iii) maintaining the log components that registers fault execution in test runs (line 2).

```
1 service ZermiaServices{
2     rpc FirstConnection (ConnectionRequest) returns (ConnectionReply) {}
3     rpc Logging (Log) returns (ConnectionReply) {}
4     rpc SyncpointInfo (StatusQuestion) returns (ConnectionReply) {}
5
6     message ConnectionReply{
7         bool messageAcknowledgement = 1;
8     }
9 ...
```

LISTING 4.14: Proto File snippet containing the messages definition

Registering agents During an *Agent* bootstrap, a *ConnectionRequest* message is sent from the *Agent* runtime to the *Coordinator*. This message contains the *nodeID*, node address (in the form IP:port), a faulty flag, and the set of faults (represented as a list of Syncpoints), as presented in Listing 4.15.

```
message ConnectionRequest{
    string nodeID = 1;
    string nodeLocation = 2;
    bool faulty = 3;
    repeated Syncpoint syncpoint = 4;
}
```

LISTING 4.15: Proto file snipped containing bootstrap object

The faulty flag (represented as a boolean variable) informs the *Coordinator* whether the node is faulty or not. The list of Syncpoints is used by the *Coordinator* to update a map of Faults that are to be executed during the test run. This way, the *Coordinator* will have a global view of Fault dependencies for each test run. This information is used when synchronizing faults.

Synchronizing fault injection Fault synchronization is performed using the *SyncpointInfo* RPC. A *SyncpointInfo* message containing the information that characterizes a *Syncpoint* is sent to the *Coordinator*, as presented in Listing 4.16. This message includes: i) the *Syncpoint* name (line 2); ii) the fault condition name (line 3); iii) the sync type (line 8); iv) and, the node type (line 13).

```
1  message SyncpointInfo{
2      string name = 1;
3      string faultCondName = 2;
4      enum synctype {
5          onetime = 0;
6          recurrent = 1;
7      }
8      synctype syncType = 3;
9      enum nodetype {
10         trigger = 0;
11         dependant = 1;
12     }
13     nodetype nodeType = 4;
```

```
14 }
15
16 message StatusQuestion{
17     string nodeName = 1;
18     string syncpointName = 2;
19 }
```

Listing 4.16: Proto file snippet containing Syncpoint related objects

The name and fault condition name are used to identify a *Syncpoint* and the corresponding Fault Condition associated with it. The *syncType* is used to define the behaviour of the fault, i.e., if is to be executed one time or recurrently. The *nodeType* informs if the specified *Syncpoint* is a trigger or if its a dependancy of a trigger.

The *StatusQuestion* message is used to request the state of a *Syncpoint* relative to a particular node name. The response is a boolean value that represents the state (i.e., triggered or not) of a synchronization point.

Logging Logging is performed using the *Logging* RPC, that sends a Log message containing the information related to the fault and its execution parameters, as presented in Listing 4.17. This message includes: i) the Node name (line 2); ii) the fault name (line 3); iii) the log timestamp (line 4); iv) execution info (line 5), and, v) custom log information (line 6).

```
1  message Log{
2      string nodeName = 1;
3      string faultName = 2;
4      int64 timestamp = 3;
5      repeated ExecInfo execInfo = 4;
6      string randomLogInfo = 5;
7  }
8
9  message ExecInfo{
10     string execName = 1;
11     enum WhenType {
12         Round = 0;
13         Time = 1;
14         Boolean = 2;
15         Random = 3;
16         Sync = 4;
17         FaultCond = 5;
18         Custom = 6;
19     }
20     WhenType whenType = 2;
```

```
21      string whenValue = 3;
22  }
```

LISTING 4.17: Proto file snippet containing logging related objects

The *ExecInfo* contains the data relative to the *ExecParams* that have been executed in the current Fault/FaultCond, this includes: its name; the *whenType* that must be one of the possible options; the string *whenValue* that transmits relevant information like the number of rounds or the synchronization point used.

4.3.2 Coordinator

The *Coordinator's* main component is essentially a thread that acts as a constant listener. As it receives messages it creates a map of all the nodes, and *Syncpoint* dependencies. The logging message is the most recurring task, all received data is saved in an object that is then saved into a file so it can be further analysed by the tester. The most complex part of the system is the synchronization point handling.

4.3.3 Agent runtime

The *Agent* runtime is responsible for managing and monitoring fault injection and coordination with the *Coordinator* (by calling the corresponding gRPC methods).

During bootstrap, it establishes a connection and registers itself with the *Coordinator*, by calling the corresponding gRPC method (Listing 4.14). Whenever a Fault is to be triggered, the runtime is responsible for validating the triggering conditions, either locally or by contacting the *Coordinator*.

Chapter 5

Evaluation

The effectiveness of using the techniques presented in this paper to test applications has been proven by the two previous implementations of Hermes and Zermia, they showed the versatility and capabilities of fault injection that use aspects to inject code and an infrastructure supported by gRPC to coordinate the several entities. During the evaluation of Proteus we will focus in our objective which is to simplify the process of testing and demonstrate how our system can perform as well as the previous iterations, with a more clear and transparent interaction by the tester, all this while being adaptable and easily transferable to other applications.

A full analysis of a protocol in which to test our application is considered out of scope in this project, instead we will use the analysis made by the previous implementation, Zermia and replicate many of its tests using our system.

5.1 Testing

The tests conducted in Zermia are designed to find bugs in the implementation of the BFT-SMaRt protocol implementation, to achieve this they conduct a series of tests using several faults with different combinations of faulty nodes. The general fault setup involves two tests with 4 and 10 replicas, and 50 clients. The general pattern of fault injection involve introducing a fault at round 50000 when dealing with 4 replicas and also at round 100000 and 150000 when dealing with 10 replicas.

The tests are divided into two categories: general fault injection where we used the faults that we have already created and the protocol specific faults where we implement our own faults and conditions. The tests will showcase our grammar and compare the

implementation to the one used in Zermia, we will not go into detail relative to the protocol implementation so the tests relative to protocol specific faults will not be as complex as the ones performed in the original paper.

This specific use case fits into the purpose for which our system was designed, it is a distributed application that must be resilient against a variety of faults, it does not illustrate the full capabilities of our system but allows us to explore some interesting features. The main limiting factor is the fact that this protocol is based on replicas so a single code base is being tested. This implementation already contains a logging system, we will still use our logging system but have in mind that it is redundant since the metrics present in the program go into more detail due to its tailored design.

These test were conducted locally using virtual machines, each machine had 2 gigabytes of RAM, 1 core, 20 gigabytes of memory, and ran Ubuntu Server 20.04 LTS operating system.

5.1.1 Crash Fault

The first test creates a fault that crashes a replica at round 50000, the way we represent it using our grammar can be seen in Listing 5.1.

```
RunConfiguration{
    RuntimePackage = "package bftsmart.aspect.zermia;";
    CoordinatorLocation = "10.10.10.1:8080";
    Node faulty_replica (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.10:8080" ; crash50000)
    Node non_faulty_replica_1 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.11:8080" ; logData = log1; none )
    Node non_faulty_replica_2 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.12:8080" ; logData = log1; none )
    Node non_faulty_replica_3 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.13:8080" ; logData = log1; none )
}

Fault crash50000 {
    pointcut = "execution (* bftsmart.communication.server.ServerConnection.sendBytes*(..))";
    ExecParams{
        where = Before ;
        whenToExecute{betweenRound(50000,50000,0)};
        whatToExecute{exit(1)}
```

```
    }
}

FaultCond logSendingBytes {
    whereToExecute{ After ;
        "execution (* bftsmart.communication.server.ServerConnection.sendBytes*(..))"};
}

Logging log1 { logSendingBytes }
```

Listing 5.1: Grammar file for Crash fault with 4 replicas

The fault used in these test terminates the replica by calling *System.exit* method with the code 1, at round 50000. All other nodes have no faults but are tasked with sending the log data at a particular location.

This same test was also developed with 10 replicas, where on total 3 of them crash, one every 50000 rounds. The *RunConfiguration* object for the second test can be seen in Listing 5.2, the faults remain similar except for the round at which they are triggered, as can be seen by their name.

```
RunConfiguration{
    RuntimePackage = "package bftsmart.aspect.zermia;";
    CoordinatorLocation = "10.10.10.1:8080";
    Node faulty_replica_0 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.2:8080" ; crash50000)
    Node faulty_replica_1 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.3:8080" ; crash100000)
    Node faulty_replica_2 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.4:8080" ; crash150000)
    Node non_faulty_replica_0 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.10:8080" ; logData = log1; none )
    Node non_faulty_replica_1 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.11:8080" ; logData = log1; none )
    Node non_faulty_replica_2 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.12:8080" ; logData = log1; none )
    Node non_faulty_replica_3 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.13:8080" ; logData = log1; none )
    Node non_faulty_replica_4 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
```

```
        location = "10.10.10.14:8080" ; logData = log1; none )
    Node non_faulty_replica_5 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.15:8080" ; logData = log1; none )
    Node non_faulty_replica_6 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.16:8080" ; logData = log1; none )
}
```

LISTING 5.2: *RunConfiguration* object for Crash fault with 10 replicas

We require 10 lines to describe all the *Nodes*, more lines are present in the example due to the size constraints of this page.

5.1.2 Delay Fault

This attack delays all messages sent for an increasing amount of time each run, at round 50000 it delays all messages for 20 milliseconds, at round 100000 it delays all messages for 50 milliseconds and at round 150000 the messages are delayed for 100 milliseconds, this test allows us to take advantage of our *ExecParams* structure.

```
RunConfiguration{
    RuntimePackage = "package bftsmart.aspect.zermia;";
    CoordinatorLocation = "10.10.10.1:8080";
    Node faulty_replica_0 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.10:8080" ; all )
    Node faulty_replica_1 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.11:8080" ; all )
    Node faulty_replica_2 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.12:8080" ; all )
    Node faulty_replica_3 (
        firstAttachPoint = "bftsmart.demo.counter.CounterServer.main" ;
        location = "10.10.10.13:8080" ; all )
}

Fault delay_all {
    pointcut =
        "execution (* bftsmart.communication.server.ServerConnection.sendBytes*(..))";
    ExecParams delay_20_50000 {
        where = Before ;
        whenToExecute{betweenRound(50000,60000,0)};
        whatToExecute{sleep(20)}
    }
```

```
    ExecParams delay_50_100000 {
        where = Before ;
        whenToExecute{betweenRound(100000,110000,0)};
        whatToExecute{sleep(50)}
    }

    ExecParams delay_100_150000 {
        where = Before ;
        whenToExecute{betweenRound(150000,160000,0)};
        whatToExecute{sleep(100)}
    }
}
```

Listing 5.3: Grammar file for delay fault with 4 replicas

The system described in Listing 5.3, highlights the benefits of having a structure where *ExecParams* are used instead of a monolithic *Fault*, the pointcut in case is a prime target for fault injection so various faults are associated with it, in our example three distinct ones. Adapting each parameter requires little alterations, since both round intervals and sleep times can be independently adjusted.

5.1.3 Message Dropper and Flood Attack

These two tests are bundled in the same section because they both take advantage of features unique to *AspectJ*, the first test has a 70% random chance of dropping a packet between round 50000 and 100000. The grammar that can be used to implement it is as follows.

```
...

Fault dropper {
    pointcut =
        "execution (* bftsmart.communication.server.ServerConnection.sendBytes*(..))";
    ExecParams{
        where = Before ;
        whenToExecute{betweenRound(50000,100000,0,random(70))};
        whatToExecute{skipExecution}
    }
}
```

Listing 5.4: Grammar file snippet for message dropper

The example given in Listing 5.4 does not contain the *RunConfiguration* since it identical to the one presented in Listing 5.1.

The conducted test skips the execution of the method and therefore no message is sent, we can achieve this behaviour by using *Aspects* in particular the *@Around* method, which allows us to skip the execution by simply returning the method before using the *proceed* method of the *joinpoint*.

The grammar used in the flood attack is similar to the message dropper except for the fault used, it uses the *repeatExecution What* parameter, that also takes advantage of the *@Around* method capabilities by repeatedly calling the the *proceed* method of the *joinpoint*, which in turn causes the system to send a message several times. In our particular implementation we flood the other nodes with an adjustable amount of packets in our case 100 additional packets each time a packet is sent.

5.1.4 Protocol specific

The protocol specific faults take advantage of our custom faults and conditions to introduce application specific code, it is a simple example that without going into details about the system under test is able to showcase some implementation possibilities.

```
...

Fault alterConsensus {
    pointcut =
        "execution (* bftsmart.communication.server.ServersCommunicationLayer.send*(..))";
    ExecParams{
        where = Before ;
        whenToExecute{ifCondition(CondID(ifConsensus))};
        whatToExecute{BlockID(changeConsensusView)}
    }
}

Condition ifConsensus {
imports {
///
    import bftsmart.consensus.messages.ConsensusMessage;
    import bftsmart.communication.SystemMessage;
///
},
code {
///
    import bftsmart.consensus.messages.ConsensusMessage;
    import bftsmart.communication.SystemMessage;
    SystemMessage sysMessage = (SystemMessage) args[1];
    Boolean isInstance = (sysMessage instanceof ConsensusMessage);
    return isInstance;
```

```
///
}

CodeBlock changeConsensusView {
imports {
///
    import bftsmart.consensus.messages.ConsensusMessage;
    import bftsmart.communication.SystemMessage;
///
},
code {
///
    SystemMessage sysMessage = (SystemMessage) args[1];
    ConsensusMessage consMessage = (ConsensusMessage) sysMessage;
    Integer viewNumber = consMessage.getEpoch();
    consMessage.setEpoch(viewNumber+1);
///
}
```

Listing 5.5: Grammar file snippet for message alteration

The *Condition* only allows for a specific kind of message called *ConsensusMessage*, once this kind of message is identified we alter one of its parameters in the *What* parameter in an attempt to disrupt the system. As can be seen in Listing 5.5 we use code specific to the application under test both in the *Condition* as well as the *CodeBlock* by importing the specific packages using the *imports* option.

This also represents a specific kind of faults that we want to analyse which is not associated to statistical analysis, but tries to identify failures in the system through specific interactions.

5.1.5 Discussion

In Listing 5.2 we encounter a scaling problem, where the amount of nodes leads to a lot of code replication, and it ends up affecting the readability of our grammar. Our system is designed to detail every node, if in the future we wish the handle a significant amount of nodes, some sort of node management would have to be implemented.

The configuration presented in Listing 5.3 presents us with some interesting details, that showcase the strengths and weaknesses of our system, this fault contains three *execParams* that by design function independently and manage their own internal variables. In this particular use case this is a waste of resources since we are managing three counters that

all have the same value, this inefficiency is a problem associated with generalized systems where the goal of adapting to several distinct circumstance often introduce redundancies.

We could create a special use case for this test, but this would only solve this particular issue and would have limited use in other cases, a smarter compiler that identifies the redundancy could also be an avenue through which the redundancy could be mitigated. This is a future improvement but currently the system functions properly and the redundancy introduced does not significantly delay the execution of the base program.

The tests also show us the use cases which are more suited for our tool, when the tester uses the general faults implemented by our system, our tool is capable of saving a significant amount of work, this isn't as present in custom faults since they require the implementation of custom code, this is expected but the capability of using the infrastructure that supports our system, as well as the possibility of combining custom and general faults offsets the efficiency lost.

5.2 Efficiency

Analysing the efficiency brought forward by our tool is a difficult task, we would argue that the most relevant aspect is the readability that it provides and ease of implementation, but these are not directly analysable parameters. We try to evaluate its efficiency and its increase in productivity by analysing other aspects.

A measure that we feel like illustrates the improved efficiency brought by our tool is the Lines of Code (LOC) that it saves. First we have to justify why this is a valid metric. LOC are an infamous subject that has been discussed in great detail from various authors [62] [63], it is a complex measure that can't be taken at face value and must be analysed in a case by case basis, despite this it has been found to be a valid metric if not used by itself, and its nuances are taken into account [64]. In our first naive approach we consider the physical LOC, this includes all lines that are not comments or empty.

	Crash	Crash2	delay	dropper	flood	custom
Grammar	19	41	25	16	16	38
Generated	60	40	60	50	52	61

TABLE 5.1: Physical LOC comparison

This is the simplest comparison, and we can already observe significant improvements in some tests, although some caveats should be mentioned:

We consider physical LOC instead of logical, this means the code formatting influences the size, although both have similar formatting, upon code analysis we can observe that the Grammar takes advantage of this measure.

There is also an issue where less faults is advantageous to the Grammar, this is evident in the delay test where more *ExecParams* created a smaller separation, this is due to the initialization of variables and the import of dependencies that always happen.

Lastly when there are more *Nodes* the generated code has an advantage since this reflects in a higher amount of files and does not influence the generated code of a single file, this is clear in the *Crash* test with 10 replicas where the grammar had more lines than the generated code.

Because of the problems referenced above we altered the experiment to bring forward more relevant results, first the header of the generated code and the *RunConfigurations* of the Grammar will not be taken into account, adding to this we will consider logical LOC, considering a each line of code to be a single logical statement. This are the results that we obtained:

	Crash	Crash2	delay	dropper	flood	custom
Grammar	6	6	10	4	4	17
Generated	14	14	23	16	16	21

Table 5.2: Logical LOC comparison with adjustments

With this example we can have a better picture of one of the benefits in using our system, in average a test requires 2.21 times less LOC to be developed, this is obviously very dependant on things like the amount of custom faults used, since custom code is placed directly and uses the same amount of lines in the grammar and generated code. Despite this improvements we must mention that the code developed is not the most efficient and in some cases the gains would be much smaller if it was designed by a human, for example the flood attack could have saved at least five LOC if it was developed by a human, this lack of efficiency is to some point inevitable in generalized applications.

Chapter 6

Conclusion

Throughout this project we explored generalized fault injectors, aggregated the comparable ones and started to delineate a design of how a generalised fault injector should work and what our implementation can add to the work that has been done so far.

Our work is based on two previous fault injectors called Hermes and Zermia, we analysed these tools, identified their strengths and the potential points of improvement that would extend their capability and fit our objective of a generalised system. We followed this analysis with an explanation of the tools they use such as AspectJ and gRPC. We also analysed tools that could be used to create a DSL and explained *Xtext* and *Xtend*, the tools used for our implementation.

We went through the design of our system and how we organize it, as well as the objectives we set out to achieve. We explored the several structures needed to build a general fault injector, which includes *What* to be executed and how to allow for general along with custom faults, *When* to execute the faults exploring the various dependencies that testers can use, *Where* to execute a component by using a pointcut to specify the code location where a fault should be injected, and *Who* should execute by creating a schedule that defines the interactions of every node. While explaining each structure, we correlated them with the proposed design and focused on the decisions that were made in order to justify the path taken.

We also explained how we implemented the proposed design by presenting our syntax and exploring all the language capabilities. We then explored the code generation and focused on the features that would interest testers, like how they can interact with certain objects. We finished this by exploring how we adapted the underlying infrastructure to be more generalized and capable of handling synchronization points and log management.

The experimental evaluation performed using our tool was able to demonstrate how the process of creating a schedule run can be simplified by the use of a DSL that translates a high level explanation of the desired instructions into a specific implementation ready to execute once combined with the end application. The results show that several faults that were designed for a specific protocol can be translated using our tool.

The work done in this paper complemented by the research done in Zermia [4] led to the publication of an international paper [5].

6.1 Future Work

This tool can still be expanded in many directions, since there are still a variety of general faults that could be added along with timing systems that can adapt to more situations. The capability of stacking several kinds of conditions to create complex timing situations could also be explored, along with allowing for better management of user controlled state based variables. Being able to combine custom and general faults could also be an interesting possibility. As previously mentioned, a way of grouping several nodes would reduce the verbosity and lead to a simpler schedule description system.

An objective mentioned throughout this project is the ability to port Proteus to other languages, this steered several design decisions, so the adaptation of the system while still maintaining the language syntax would be very valuable. The system support for gRPC communication, along with the use of AOP, should facilitate this transition.

An interesting development would also be to streamline the integration of our system with the application that is being tested, if this integration was to be automated than we could also go into the realm of creating the inputs for each run and supplying them to the current application, thus automating the complete run.

Lastly, more testing should be conducted against several different protocols, since the testing that was performed was limited and should be expanded to understand what works and what areas warrant more development. Tests against different kinds of applications like concurrent systems would also be valuable and would further develop the generalization of the system.

Bibliography

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*. Seattle, WA: USENIX Association, Mar. 2003. [Online]. Available: https://www.usenix.org/conference/usits-03/why-do-internet-services-fail-and-what-can-be-done-about-it [Cited on pages 2 and 8.]

[2] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-glove approach for BFT testing," *CoRR*, vol. abs/2004.10617, 2020. [Online]. Available: https://arxiv.org/abs/2004.10617 [Cited on page 3.]

[3] R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo, "Experiences with fault-injection in a byzantine fault-tolerant protocol," in *Middleware 2013*, D. Eyers and K. Schwan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 41–61. [Cited on pages 3, 15, 16, and 31.]

[4] R. Fernandez, "Injecting faults in byzantine fault tolerant protocols," 2021. [Cited on pages 3, 4, 15, 18, 31, and 76.]

[5] J. Soares, R. Fernandez, M. Silva, T. Freitas, and R. Martins, "Zermia - a fault injector framework for testing byzantine fault tolerance protocols," in *15th International Conference on Network and System Security*, M. Yang, C. Chen, and L. Yang, Eds. Springer International Publishing, 2021. [Cited on pages xiii, 4, 34, and 76.]

[6] R. A. Haissam Ziade and R. Velazco, "A survey on fault injection techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171–186, Jul. 2004. [Cited on page 7.]

[7] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some

applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 340–347, 1990. [Cited on page 8.]

[8] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers.* Los Alamitos, CA, USA: IEEE Computer Society, jun 1989, pp. 340–347. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/FTCS.1989.105590

[9] J. Karlsson and P. Folkesson, "Application of three physical fault injection techniques to the experimental assessment of the mars architecture." IEEE Computer Society Press, 1995, pp. 267–287. [Cited on page 8.]

[10] T. K. T. M. C. Hsueh and R. K. Iyer, "Fault injection techniques and tools." in *EEE Computer 30, 4 (1997)*, pp. 75–82. [Cited on pages xiii, 8, and 9.]

[11] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, Feb. 2016. [Online]. Available: https://doi.org/10.1145/2841425 [Cited on page 8.]

[12] P. F. Joakim Aidemark, Jonny Vinter and J. Karlsson, "Goofi : Generic object-oriented fault injection tool," *IEEE Xplore*, pp. 83–88, 2001. [Cited on page 9.]

[13] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, 1998, pp. 284–293. [Cited on page 10.]

[14] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "Hartos: A distributed real-time operating system," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 3, p. 72–89, Jul. 1989. [Online]. Available: https://doi.org/10.1145/71021.71025 [Cited on page 10.]

[15] S. Han, K. Shin, and H. Rosenberg, "Doctor: an integrated software fault injection environment for distributed real-time systems," in *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, 1995, pp. 204–213. [Cited on page 10.]

[16] D. Kiskis, "Generation of synthetic workloads for distributed real- time computing systems," august 1992. [Cited on page 10.]

[17] H. M. João Carreira and J. G. Silva, "Xception: Software fault injection and monitoring in processor functional units." [Cited on page 11.]

[18] G. Kanawati, N. Kanawati, and J. Abraham, "Ferrari: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, 1995. [Cited on page 11.]

[19] Z. 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'.*, 1995, pp. 394–. [Cited on page 11.]

[20] M. Hiller, A. Jhumka, and N. Suri, "Propane: An environment for examining the propagation of errors in software," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, p. 81–85, Jul. 2002. [Online]. Available: https://doi.org/10.1145/566171.566184 [Cited on page 12.]

[21] R. Chandra, R. Lefever, M. Cukier, and W. Sanders, "Loki: a state-driven fault injector for distributed systems," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, 2000, pp. 237–242. [Cited on page 12.]

[22] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of fault-tolerant and real-time distributed systems via protocol fault injection," in *Proceedings of Annual Symposium on Fault Tolerant Computing*, 1996, pp. 404–414. [Cited on page 13.]

[23] E. Martins, C. Rubira, and N. Leme, "Jaca: a reflective fault injection tool based on patterns," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 483–487. [Cited on page 13.]

[24] S. Chiba, "Load-time structural reflection in java," in *ECOOP 2000 – Object-Oriented Programming, LNCS 1850, Springer Verlag*, pp. 313–336. [Cited on pages 13 and 14.]

[25] M. Martins and A. Rosa, "A fault injection approach based on reflective programming," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, 2000, pp. 407–416. [Cited on page 13.]

[26] (2019, March) Principles of chaos engineering. [Online]. Available: https://principlesofchaos.org/ [Cited on page 15.]

[27] N. Inc. (2016) Simian army. [Online]. Available: https://github.com/Netflix/SimianArmy [Cited on page 15.]

[28] ——. (2020) Chaos monkey. [Online]. Available: https://github.com/Netflix/chaosmonkey [Cited on page 15.]

[29] VMware. (2021) Mangle, orchestrating chaos engineering. [Online]. Available: https://vmware.github.io/mangle/ [Cited on page 15.]

[30] D. Inc. (2021) Docker. [Online]. Available: https://www.docker.com/ [Cited on page 15.]

[31] C. N. C. Foundation. (2021) Chaos mesh. [Online]. Available: https://chaos-mesh.org/ [Cited on page 15.]

[32] Google. (2021) Golang. [Online]. Available: https://golang.org/ [Cited on page 15.]

[33] C. N. C. Foundation. (2021) Kubernets. [Online]. Available: https://kubernetes.io/ [Cited on page 15.]

[34] ——. (2021) Litmus chaos. [Online]. Available: https://litmuschaos.io/ [Cited on page 15.]

[35] L. A. M. D. Allen Clement, Edmund Wong and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI '09*, 2009, p. 153–168. [Cited on page 16.]

[36] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with bft-smart," 06 2014, pp. 355–362. [Cited on page 16.]

[37] A. Restivo, "The case for aspect oriented programming," 2020. [Online]. Available: https://paginas.fe.up.pt/~prodei/comic06/p5.pdf [Cited on page 20.]

[38] Baeldung. (2020) Intro to aspectj. [Online]. Available: https://www.baeldung.com/aspectj [Cited on page 20.]

[39] T. E. Foundation. Chapter 5. load-time weaving. [Online]. Available: https://www.eclipse.org/aspectj/doc/released/devguide/ltw.html [Cited on page 20.]

[40] A. Rai. (2018) Spring aop + aspectj @before, @after, @afterreturning, @afterthrowing, and @around annotation example. [Online]. Available: https://www.websparrow.org/spring/spring-aop-aspectj-before-after-afterreturning-afterthrowing-and-around-annotation-example [Cited on page 21.]

[41] T. E. Foundation. Chapter 9. an annotation based development style. [Online]. Available: https://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj.html [Cited on page 21.]

[42] L. Gupta. Spring aop aspectj @after annotation example. [Online]. Available: https://howtodoinjava.com/spring-aop/aspectj-after-annotation-example/ [Cited on page 21.]

[43] T. E. Foundation. Interface joinpoint. [Online]. Available: https://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/JoinPoint.html [Cited on page 22.]

[44] Google. (2021) grpc. [Online]. Available: https://grpc.io/ [Cited on page 23.]

[45] ——. Language guide (proto3). [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto3 [Cited on page 23.]

[46] JetBrains. Domain-specific languages. [Online]. Available: https://www.jetbrains.com/mps/concepts/domain-specific-languages/ [Cited on page 23.]

[47] M. Eliasson. (2018) Building your own dsl does not have to be hard. [Online]. Available: https://markuseliasson.se/article/building-your-own-dsl/ [Cited on page 23.]

[48] R. D. Kelker, *Clojure for Domain-Specific Languages*. Packt Publishing, 2013. [Cited on page 24.]

[49] M. Fowler, *Domain-Specific Languages*. Upper Saddle River, NJ: Addison-Wesley, 2010. [Online]. Available: https://www.safaribooksonline.com/library/view/domain-specific-languages/9780132107549/ [Cited on page 24.]

[50] L. I. Wilson. (2018) Understanding compilers — for humans (version 2). [Online]. Available: https://towardsdatascience.com/understanding-compilers-for-humans-version-2-157f0edb02dd [Cited on page 24.]

[51] T. Parr. About the antlr parser generator. [Online]. Available: https://www.antlr.org/about.html [Cited on page 25.]

[52] ——, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013. [Cited on page 25.]

[53] Lexical analysis with antlr. [Online]. Available: https://web.mit.edu/dmaze/school/6.824/antlr-2.7.0/doc/lexer.html [Cited on page 26.]

[54] E. Foundation. (2021) Eclipse foundation. [Online]. Available: https://www.eclipse.org/org/foundation/ [Cited on page 26.]

[55] T. E. Foundation. Xtext integration with emf. [Online]. Available: https://www.eclipse.org/Xtext/documentation/308_emf_integration.html [Cited on pages xiii, 26, and 27.]

[56] ——. Eclipse modeling framework (emf). [Online]. Available: https://www.eclipse.org/modeling/emf/ [Cited on page 26.]

[57] ——. The grammar language. [Online]. Available: https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html [Cited on pages 27 and 28.]

[58] ——. Documentation. [Online]. Available: https://www.eclipse.org/xtend/documentation/index.html [Cited on page 28.]

[59] ——. Expressions. [Online]. Available: https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html [Cited on page 29.]

[60] Swagger. Openapi specification. [Online]. Available: https://swagger.io/specification/ [Cited on page 39.]

[61] T. E. Foundation. Join points and pointcuts, chapter 2. the aspectj language. [Online]. Available: https://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html [Cited on page 45.]

[62] C. Jones, *Software Assessments, Benchmarks, and Best Practices*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. [Cited on page 72.]

[63] A. Charnes, W. Cooper, and E. Rhodes, "Measuring the efficiency of decision making units," *European Journal of Operational Research*, vol. 2, no. 6, pp.

429–444, 1978. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377221778901388 [Cited on page 72.]

[64] B. M. Y. Parareda, "Measuring productivity using the infamous lines of code metric," 2007. [Cited on page 72.]