Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Impianti di Elaborazione

An approach to failure mode analysis of cloud computing infrastructures

Anno Accademico 2017/2018

relatori
Ch.mo Prof. Domenico Cotroneo
Ch.mo Prof. Roberto Natella

correlatore
Ing. Luigi De Simone

candidato
Pietro Liguori
matr. M63000556

Ai miei nonni, ai miei genitori, a Luana ...

Abstract

Fault injection is a technique for valuating the dependability of computer systems. It is one of the most important challenges in distributed systems and thus for maintaining the reliability requirements and for ensuring that services are correct and available despite faults.

Like any other distributed system, cloud management stacks are susceptible to a variety of complex faults, most of which are often hard to diagnose. Furthermore failure mode analysis is very difficult and time consuming, even for expert developers, due to many problems.

We propose *PyF*, an innovative approach for analyzing the behavior of complex software systems under fault injection. The driving idea is to analyze the distributed system as a set of *black-box components* that interact through public *service interfaces*; and to apply an *anomaly detection* algorithm on these interactions, in order to identify how the system behaves in the case of failures. The proposed anomaly detection algorithm leverages dynamic programming algorithms and probabilistic models in order to solve the problem of identifying when a fault has occurred, of determining the type of fault and its location and of understanding how it propagated through time and space. PyF does not rely on any particular system, though the main interest lies in complex distributed systems, and it is fast and accurate in producing the results. The results are provided in a graphical and interactive form in order to be easily interpreted not only by expert developers.

Contents

1 Introduction to Fault Injection problems **1**

1.1 Faults and Failures in complex systems 2

1.2 Overview of Fault Injection 3

1.3 The fundamental issues of Failure Mode Analysis 7

1.4 Thesis Contributions 11

2 PyF: an approach to failure mode analysis **13**

2.1 Fault Injection Tool 15

2.2 Data Collection 16

2.2.1 Zipkin Framework 16

2.2.2 Idle Trace, Fault Free Traces and Faulty Trace 18

2.3 Data Analysis 20

2.4 Parsing 21

2.5 Noise Extraction 23

2.6 Sequence Creation 24

2.7 Longest Common Subsequence (LCS) 25

2.8 Variable order Markov Model (VMM) 32

2.8.1 Prediction by Partial Match (PPM) 35

2.9 PyF Algorithm 40

2.10 Visualization 45

3 Experimental Analysis on OpenStack **49**

3.1 Experimental Setup 49

3.2 Experimental results 56

3.2.1 Experiment number 1 - API error only 56

3.2.2 Experiment number 2 - Assertion Failure and API error 62

3.2.3 Experiment number 3 - Assertion Failure only 67

3.2.4 Experiment number 4 - No Assertion Failure and No API error . . 72

3.3 Performance evaluation of the approach 75

3.3.1 Accuracy 75

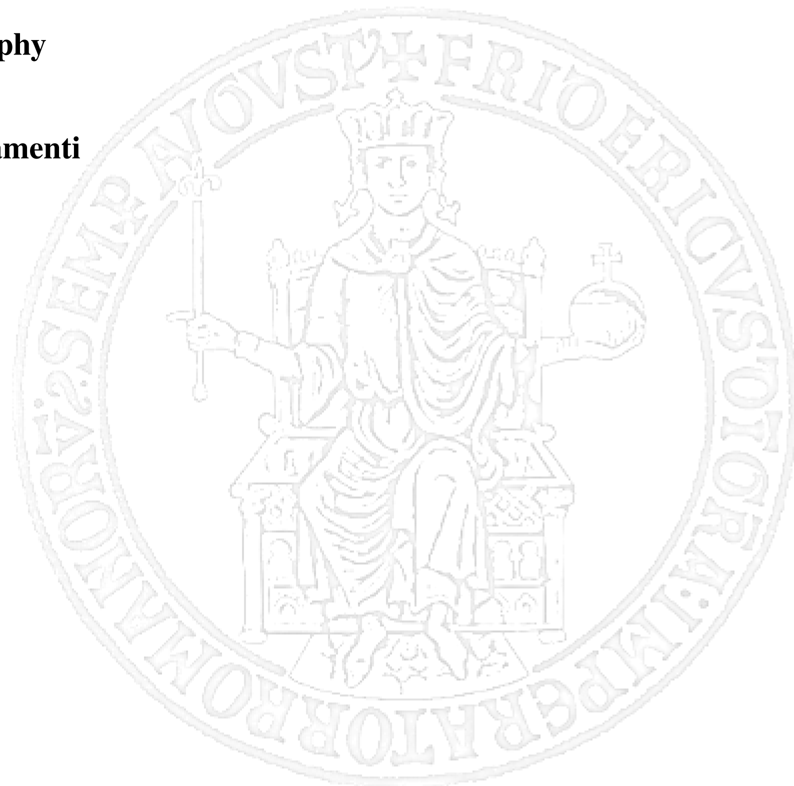
3.3.2 Computational Time 78

Conclusion **82**

A Zipkin **85**

Bibliography **89**

Ringraziamenti **91**



List of Figures

1.1.1 Chain reaction fault, error and failure 2
1.2.1 Fault Injection System 5
1.2.2 Workflow of SFI 7
1.3.1 Fault, Error and Failure issues 8

2.0.1 Architecture of the approach 14
2.2.1 Workflow Data Collection 19
2.3.1 PyF workflow 22
2.5.1 Noise Extraction workflow 24
2.6.1 Workflow of the creation of sequences 25
2.7.1 Probabilistic model to validate or not the differences 33
2.8.1 PPM-C trie 40
2.9.1 Steps of PyF Algorithm 41
2.10.1Plot example 46
2.10.2Mpld3 plugin 47
2.10.3Plot example - Debug Mode 48

3.1.1 Experimental Design 50
3.2.1 PyF Visualization - Experiment 1 58
3.2.2 Experiment 1 - Last faulty trace event 59

3.2.3 Experiment 1 - Missing event - 1 59
3.2.4 Experiment 1 - Missing event - 2 60
3.2.5 Experiment 1 - Missing event - 3 60
3.2.6 Experiment 1 - Missing event - 4 61
3.2.7 PyF Visualization - Experiment 2 64
3.2.8 Experiment 2 - First event in the faulty trace after fault activation 64
3.2.9 Experiment 2 - Last event in the faulty trace after fault activation 65
3.2.10 Experiment 2 - Missing events 66
3.2.11 PyF Visualization - Experiment 3 69
3.2.12 Experiment 3 - Missing event - 1 70
3.2.13 Experiment 3 - Missing event- 3 71
3.2.14 Experiment 3 - Missing event - 3 71
3.2.15 PyF Visualization - Experiment 4 73
3.2.16 Experiment 4 - Missing Event 74
3.2.17 PyF Visualization - Experiment 4 - Debug mode 74
3.3.1 Steps to identify False Negatives 76
3.3.2 Improvement of the Accuracy 79
3.3.3 Graph of the computational times when the number of traces increases . . 81
3.3.4 Graph of the computational times when the size of the traces increases . . 81

A.0.1 Zipkin flow diagram 86
A.0.2 Example sequence of http tracing 87

List of Tables

2.1 Noise Dictionary of Openstack Pike 23
2.2 LCS Table 28
2.3 LCS table storing length 29
2.4 Traceback example 29
2.5 PPM Table 39
2.6 PPM probabilities 39
2.7 Results of PPM-C on a diff symbol 44

3.1 Campaign on Nova subsystem 55
3.2 Campaign on Cinder subsystem 55
3.3 Experiment 1 - Fault Injection Point Info 57
3.4 Assertion Result and API error 62
3.5 Experiment 2 - Fault Injection Point Info 63
3.6 Assertion Result and API error 67
3.7 Severity Error Log 67
3.8 Experiment 3 - Fault Injection Point Info 68
3.9 Experiment 4 - Fault Injection Point Info 72
3.10 False positives 76
3.11 Distribution of the false positives 77

3.12 Campaign Nova - False Negatives 77
3.13 Campaign Cinder - False Negatives 78
3.14 False Negatives Statistic 78
3.15 Improvement of the Accuracy 79
3.16 Computational Time as the number of traces increases 80
3.17 Computational Time as the number of traces increases 80

Chapter 1

Introduction to Fault Injection problems

One of the major problems in developing a dependable system is how to evaluate its dependability. Numerous approaches have been proposed to validate and evaluate system dependability, including formal proofs, analytical modeling, and experimental methods. As computer systems become more complex, evaluation by modeling may become an intractable issue [9].

Fault injection is a technique for valuating the dependability of computer systems. It is one of the most important challenges in distributed systems and thus for maintaining the reliability requirements and for ensuring that services are correct and available despite faults. However, detecting the effects of a fault after its injection into the system can be a quite difficult practice.

This chapter presents an overview on fault injection and illustrates the issues related to this approach.

1.1 Faults and Failures in complex systems

A *fault* is the hypothesized cause of an incorrect system state, which is referred to as *error*. The *service* delivered by a system (in its role as a provider) is its behavior as it is perceived by its user(s); a *user* is another system that receives service from the provider. A *failure* is an event that occurs when the delivered service deviates from correct service. The period of delivery of incorrect service is a *service outage*, while the system under analysis is named *target*.

A fault is defined *active* when it produces an error, otherwise it is *dormant*. Faults can be *internal* or *external* of a system. An error is detected if its presence is specified by an error message or error signal. Errors in the system can be not detected: such errors are called *latent errors* [1].

Faults, errors and failures operate according to a mechanism. This mechanism is sometimes known as the *chain of threats* [1] or *fault pathology*, as shown in Figure 1.1.1.

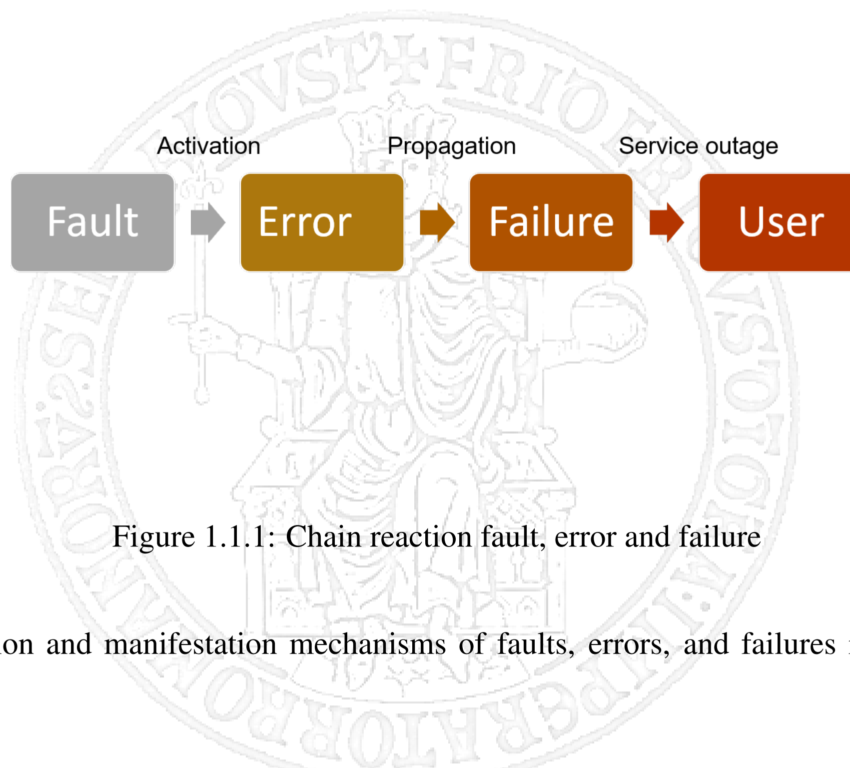


Figure 1.1.1: Chain reaction fault, error and failure

The creation and manifestation mechanisms of faults, errors, and failures is described below:

- A fault is active when it produces an error; otherwise, it is dormant . An active fault is either an internal fault that was previously dormant and that has been activated

by the computation process or environmental conditions, or an external fault. Fault activation is the application of an input to a component that causes a dormant fault to become active. Most internal faults cycle between their dormant and active states.

- Error propagation within a given component (i.e., internal propagation) is caused by the computation process: an error is successively transformed into other errors. Error propagation from component A to component B that receives service from A (i.e., external propagation) occurs when, through internal propagation, an error reaches the service interface of component A. At this time, service delivered by A to B becomes incorrect, and the ensuing service failure of A appears as an external fault to B and propagates the error into B via its use interface.

- A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. The failure of a component causes a permanent or transient fault in the system that contains the component. Service failure of a system causes a permanent or transient external fault for the other system(s) that receive service from the given system.

These mechanisms enable the chain of threats to be completed. The arrows in this chain express a causality relationship between faults, errors, and failures. By propagation, several errors can be generated before a failure occurs.

1.2 Overview of Fault Injection

The evaluation of the dependability of systems involves the study of failures and errors, but the destructive nature of a crash, the unpredictability of faults and long error latency make it difficult to identify the causes of failures in the operational environment [10].

In engineering, fault injection approach is often used for evaluating the dependability of a system, defined as "*the process of introducing faults in a system in order to assess its behavior and to measure the efficiency (i.e., coverage and latency) of fault tolerance*

mechanism" [6]. Dependability evaluation with fault injection experiments has therefore become a popular alternative. This approach is aimed at accelerating the occurrence of faults in order to assess the effectiveness of fault-tolerance mechanisms while executing realistic programs on the target system.

The first type of fault injection was based on the introduction of faults at a hardware level: for this reason, this approach of fault injection is called *HWIFI* (Hardware Implemented Fault Injection). HWIFI has become difficult with the growing of complexity at hardware level [6] and so a new approach, based on introduction of faults by software technique, has been found. The name of this method of fault injection is *SWIFI* (Software implemented fault injection). SWIFI tools were used to inject errors both in the program state and in the program code. Unfortunately it is not possible to accurately emulate the effects of of real faults in a complex system [6].

Software Fault Injection (SFI) is a recent approach that introduces faults at software level by adding small code changes to the target program. Such an approach provides different versions of the target program that will be exercised by a workload, and faults are inserted into specific software components of the target system. The main goal is to observe how the system behaves in the presence of the injected faults. SFI is used to validate the effectiveness and to quantify the coverage of software fault tolerance, but also to assess risk and to perform dependability evaluation.

SFI is less expensive than hardware fault injection, because it does not require additional hardware support. It is much easier to repeat experiments with SFI and collect sufficient amounts of data. A further aspect to consider is the risk of permanently damaging the target system that results to be very low with SFI [9].

A recurring problem in software fault injection is the time spent conducting a fault injection experiment, especially if the injected faults are rarely activated.

The *error injection* approach is based on the injection of errors rather than faults in order to accelerate the failure process, but this would require a mapping between representative software faults and injectable errors [5]. A well known technique of error injection is the

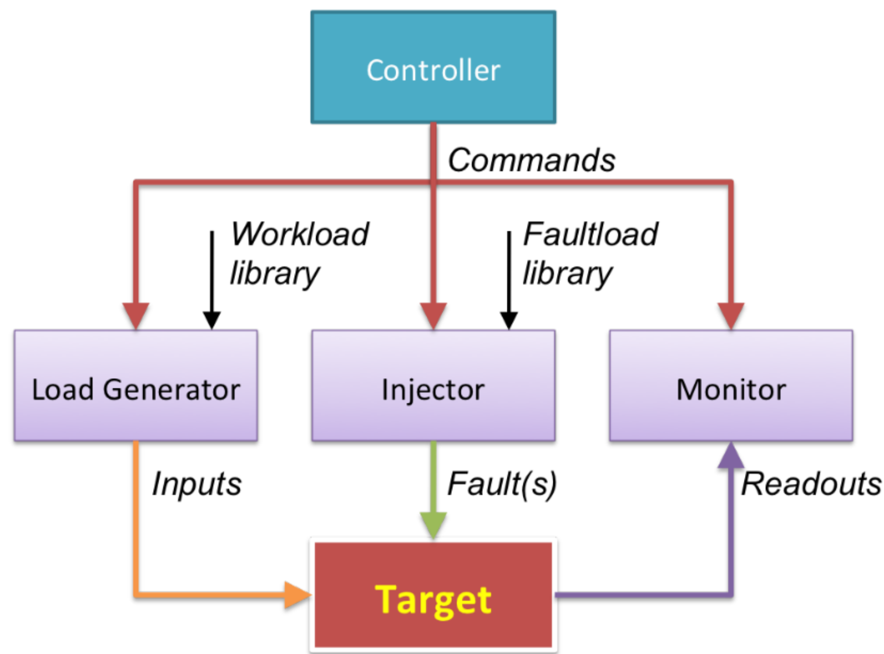


Figure 1.2.1: Fault Injection System

SWIFI, discussed above.

A *fault injection environment* typically consists of the target system plus the *fault injection system* [10].

Software fault injection follow the common schema presented in [7], and shown in Figure 1.2.1. The components of the schema are described below:

- *Injector*: injects fault into the target system.
- *Load Generator*: exercises the target system with inputs that will be processed during a fault injection experiment.
- *Faultload library*: stores different fault types, fault locations, fault times, and appropriate hardware semantics or software structures.
- *Workload library*: stores test workloads for the target system.
- *Monitor*: it is an entity that collects from the target raw data (*readouts* or *measurements*) that are needed to evaluate the effects of injected faults.

- *Controller*: it is responsible for iterating fault injection experiments and for storing the results of each experiment. It orchestrates the described entities.

Fault injection usually involves the execution of several *experiments* or *runs*, which form a *fault injection campaign*. In order to obtain information about the outcome of an experiment, readouts are usually compared to the readouts obtained from fault-free experiments (referred to as *fault-free run* or *fault-free traces*) [6].

The accuracy of results of a fault injection campaign depends on several properties of the experiments, namely:

- **Representativeness**: refers to ability to represent the real faults and inputs that the system will be experience during operation.
- **Non-intrusiveness**: requires that the instrumentation adopted in the fault injection process (such as fault insertion and data collection) should not significantly alter the behavior of actual system.
- **Repeatability**: is the property that guarantees statistically equivalent results when a fault injection campaign is executed more then once using the same procedure in the same environment.
- **Practicability**: refers to the effectiveness of fault injection in term of cost and time. The factor include the time required to implement and setup the fault injection environment, the time to execute the experiments, and the time for the analysis of the results.
- **Portability**: requires that a fault injection technique or tool is applicable with low effort to different systems.

The Figure 1.2.2 shows a generic workflow of SFI. It can be divided in five phases:

1. Injection of the fault in the target system: fault is inserted into specific software components by changing program code.

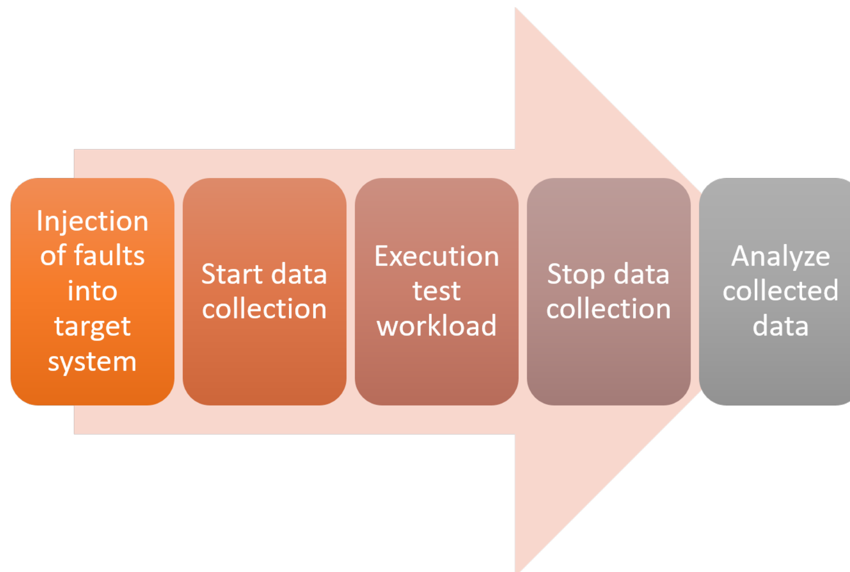


Figure 1.2.2: Workflow of SFI

2. Start collection of data: to understand how the system reacts after the fault injection, it is necessary to collect data.

3. Execution of a test workload on the target system: the target is exercised with a given test workload in order to evaluate how the injection of the fault has affected the target system.

4. Stop data collection: after the execution of the test workload, the collection of data is stopped.

5. The last phase consists in analyzing the data collected during the workload execution. This phase is crucial because a correct analysis can detect which problematic the fault has introduced, whereas a wrong analysis can not detect the effects of the injected fault.

1.3 The fundamental issues of Failure Mode Analysis

Fault injection is a key approach in the dependability assessment of systems. However complex, large scale distributed system are susceptible to a variety of complex faults,

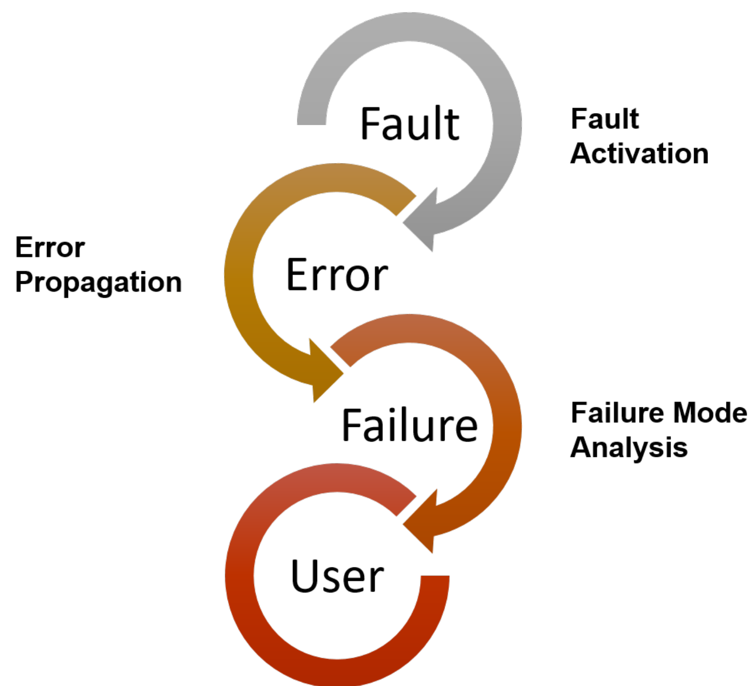


Figure 1.3.1: Fault, Error and Failure issues

most of which are often hard to diagnose. Furthermore failure mode analysis is very difficult and time consuming, even for expert developers [11], due to many problems. These problems are both related to all three stages of the chain of threats, as shown in Figure 1.3.1, and to to other aspects that will be analyzed below. Let's illustrate some examples.

Fault Activation When a fault is injected in the target system, it not necessarily generates an error. In this case, the fault is defined *dormant*. From the perspective of fault injection, dormant faults are problematic, since the target system does not experience any error or failure; thus, the fault-tolerance mechanisms cannot be tested, and the fault injection experiment is wasted. In the case of software faults, a fault can be dormant because of two reasons: (i) the buggy code path (i.e., the program statement where the bug was injected) is not covered during the experiment, and (ii) even if the buggy path is covered, the bug affects the software only when the program inputs, internal variables and environmental conditions assume specific values (the set of conditions that turn a bug into an error are called the *trigger* of the fault). Addressing these problems is important to achieve

a high efficiency, that is, to generate a good number of software failures with a limited number of experimental runs. To address these problems, we adopt a fault injection technique that focuses the injection on the program statements that are actually covered by the workload (thus avoiding to inject faults that are not covered); and the injected bugs are designed to guarantee fault activation once the buggy code is covered, by explicitly overwriting the program state with incorrect values in order to force an error.

Error Propagation Once a fault has been activated, one important issue is to understand if the effects of the fault cause any impact on the target system. For example, the corrupted program state may, or may not, have an influence on the behavior of the target system (e.g., there may be no influence if the erroneous data are not processed by the software during the course of the experiment). It is important to identify which faults have an impact on the software, in order to focus the analysis of fault-tolerance on these faults, and to ignore the faults that do not have any effect (errors not activated during the experiment are named *latent*). Another problem is to identify which are the effects of the faults, both in terms of *spatial propagation* and of *temporal propagation*. Analyzing error propagation is useful to improve fault-tolerance mechanisms. For example, in the case of temporal propagation, the error spreads its effects on the system and its users over a long period of time, thus exacerbating the severity of the fault. Identifying this kind of temporal propagation is useful for the developers to introduce additional error checks inside their system and to achieve a *fail-stop* behavior (that is, the system stops as soon as an error occurs, in order to avoid worse consequences). In the case of spatial propagation, an error propagates across several parts of the target system: for example, between different processes in a distributed system. As for temporal propagation, the spatial propagation increases the risk of further damages on the system (such as, the corruption causes cascading effects across components). The solution presented in this work aims to support the analysis of spatial and temporal error propagation, in order to ease the improvement of fault-tolerance mechanisms to prevent the propagation.

Failure Mode Analysis Analyzing the failure modes of the system (that is, the impact of the fault on the service provided by the system) is another challenging aspect. It is important to note that software systems are nowadays so complex that they can exhibit a large number of failure modes, and that not all these failure modes are known, even by the developers. In the trivial case, the system can fail by exhibiting an outage (i.e., the system stops). However, more subtle cases are also possible: for example, the system can still be available, but it can provide a *wrong* service to the users, by returning wrong data, by exhibiting poor performance, by performing wrong operations, etc. Moreover, it is important to understand whether the system is able to point out the presence of the failures: ideally, the system should clearly point out the presence of the failure, by generating an explicit error signal (e.g., by generating an exception that can be handled elsewhere), or by logging as much useful information as possible to let the system administrators to recover the system and to repair the fault. However, in the worst case, the failure may go unnoticed, thus exacerbating the severity of the problem. For these reasons, it is important to discover these failure modes during the development process, and to revise fault-tolerance mechanisms to detect the failures and to reduce their severity.

Another challenge that makes difficult to analyze failures is the *non-deterministic* behavior of complex systems. The behavior of the system (for example, the timing and the order of messages) cannot be predicted, and can change even if there is no failure. Therefore, it is difficult to distinguish between variations of the behavior that are caused by genuine failures, and variations that are caused by non-determinism and do not impact on the quality of service. In this work, we aim to support developers at identifying behavioral deviations that are actually caused by failures, by taking into account the possibility of non-deterministic behaviors that may sporadically impact on the system.

Independence of Experiments From a technical perspective, fault injection experiments are very difficult to conduct in a correct way. In particular, one challenge is to assure that the fault injection system (described in the Section 1.2) can completely auto-

mate the execution of the experiments (since there is often a large number of experiments to perform), while assuring that the injected fault does not propagate from the target system to the fault injection system. Moreover, after a fault injection experiment, the fault injection system must clean-up any residual effect of the experiment, and should restore the original state of the target system, in order to prepare the target system for the next experiment, with another injection different from the previous one (we refer to this property as the *independence among experiments*). This is not trivial since the target system can corrupt its environment in subtle and unexpected ways. For example, if the target system is unavailable due an injected fault, then it would not make sense to inject a new fault in order to evaluate how it affects the system. It is important to note that cleaning-up the residual effects of an experiment can take a significant amount of time (such as, to restore the state of a database, to restart every part of the target system, to clean-up the the environment of the target system, etc.), thus increasing the duration of the experiments. If the duration of the experiments is too long, it may be not affordable by developers. Therefore, a further aspect to consider is to find a trade-off between independence of the experiments and their duration. We consider this problem in our design of the fault injection workflow.

1.4 Thesis Contributions

Like any other distributed system, cloud management stacks are susceptible to faults whose root cause is often hard to diagnose and may take hours or days to fix [8]. Therefore, it is fundamental to have a technique of fault injection accurate in its diagnosis and in identifying of the fault.

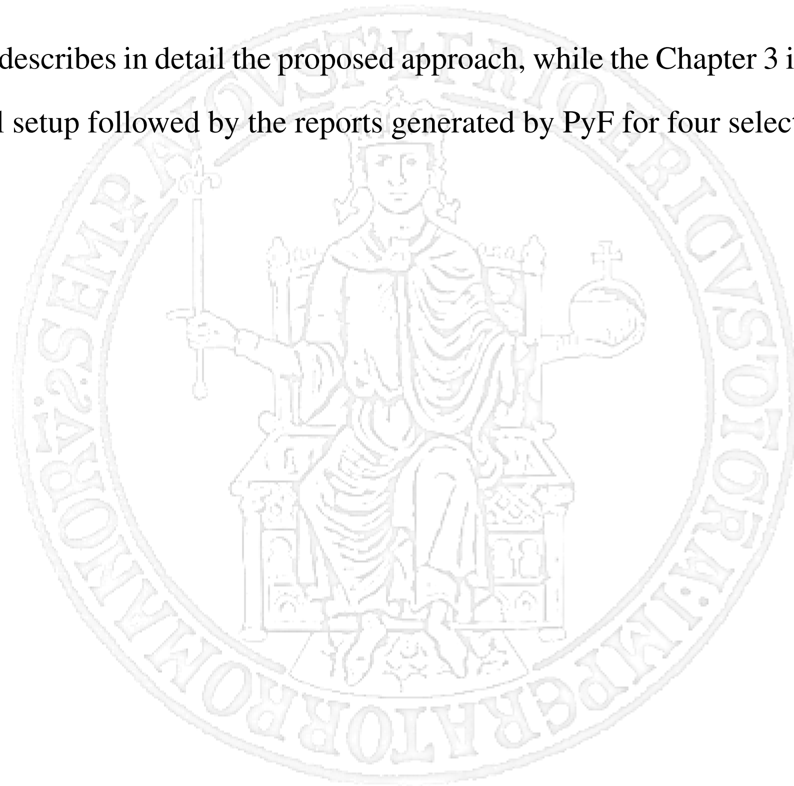
We propose *PyF*, an innovative approach to failure mode analysis implemented in Python. The approach leverages dynamic programming algorithms and probabilistic models in order to solve the problem of identifying when a fault has occurred and of determining the type of fault and its location. Its name is a tribute to his two creators.

PyF has been used for detecting faults on OpenStack, a free and open-source software

platform for cloud computing written in Python, after a large number of fault injections. The results produced are accurate and the approach is fast and scalable, thus increasing also the practicability.

The previously implemented SFI methods usually have the common restriction that they were developed for specific target systems. That is, portability, an important merit of SFI, has not been figured into their design [9]. The portability of a SFI tool can be increased by minimizing its dependence on the hardware architecture and operating system. This a fundamental aspect of the SFI, because changing every time a tool in order to adapt it to different hardware architectures is very expansive and time consuming. Given that the complete independence of a SFI from hardware and system software is almost impossible to achieve, this approach does not rely on any particular system (though the main interest lies in complex distributed systems). The portability is guaranteed through the use of a powerful and flexible data collection/analysis tool. PyF guarantees also a high level of non-intrusiveness, through the addition of very few changes only in the data collection phase.

Chapter 2 describes in detail the proposed approach, while the Chapter 3 illustrates the experimental setup followed by the reports generated by PyF for four selected experiments.



Chapter 2

PyF: an approach to failure mode analysis

PyF is an innovative approach for analyzing the behavior of complex software systems under fault injection. The driving idea is to analyze the distributed system as a set of *black-box components* that interact through public *service interfaces*; and to apply an *anomaly detection* algorithm on these interactions, in order to identify how the system behaves in the case of failures. The proposed anomaly detection algorithm is based on dynamic programming algorithm and on probabilistic model in order to analyze the sequence of interactions during an experiment, and to identify when and where a failure behavior occurred, and how it propagated through time and space.

After collecting data as a set of events, PyF analyzes them and shows suspicious events to users in the form of a HTML chart. Input data or traces are JSON format files collected by a distributed tracing system. The collected traces are sets of events. We define an *event* a set of information associated with the execution of a specific method tracked during the test workload.

PyF takes in input three types of different traces, as shown in the Figure 2.0.1:

- *Idle trace*: it is a set of data collected while the system is in an idle state (i.e., while the system is not executing operations intended as user commands specified by the

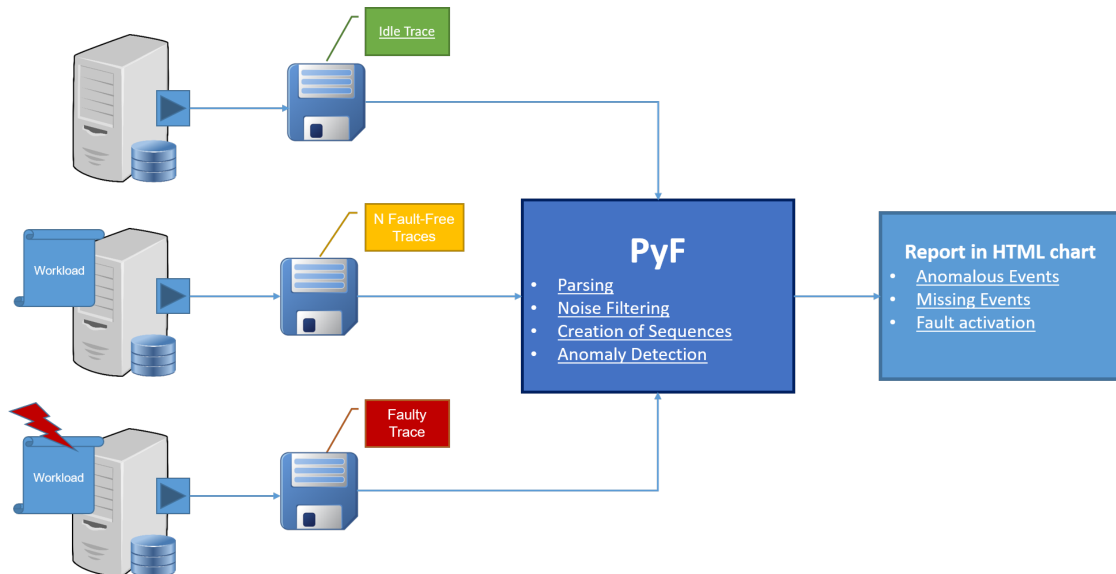


Figure 2.0.1: Architecture of the approach

user). These data are necessary to characterize the background noise in the traces, such as periodic events that are not correlated with the workload, in order to remove them from further analysis.

- *Fault free traces*: it is a set of traces collected while the system is executing properly a workload.
- *Faulty trace*: it is a trace collected while the system is executing a workload after a fault injection into target system.

In the next sections the developed approach will be illustrated in detail. The target system taken as a reference in this thesis work is OpenStack Pike, the 16th release of the most widely deployed open source infrastructure software.

The next sections will present in detail the proposed approach. The discussion will include examples in the context of the OpenStack project (version Pike, the 16th release of the most widely deployed open source infrastructure software), that has also been considered for developing a prototype of the proposed approach, and for the experimental evaluation.

2.1 Fault Injection Tool

A key aspect in the fault-injection is the necessity of other support tools for fault-injection experiments. The main tools needed for fault-injection experiments are:

- the *fault injector tool*;
- the *data collection tool*;
- the *data analysis tool*.

The fault injector tool to which this thesis refers has been developed by the Dessert Research Group of the University Federico II of Naples. The tool provides a fault injection workflow to assist test engineers at applying software fault injection on systems written in Python.

In a first phase, named *SCAN phase*, the tool interacts with the user in order to define a fault injection test plan. During the scan phase, no fault is yet injected. Instead, in this phase the tool analyzes the source code of the system, and identifies all of the fault injection points in the source code. A fault injection point is a statement in the source code where the tool can inject a software bug. It looks for statements such as arithmetic/boolean expressions, method and function calls, variable initialization, etc.

After scanning the source code, the tool offers to the user a list of the fault injection points that were found. The user can then interact with the tool, by selecting a subset of the fault injection points according to his/her preferences. For example, the user may want to inject faults only on a specific source code file inside the system; or the user may want to focus on a specific type of bug. The selection results in the fault injection test plan.

The second phase is called *TEST EXECUTION*. In this phase, the tool iterates over the items in the fault injection test plan. At each iteration, the tool perform one test with one fault injected in the source code. In each test, the original Python source code is transformed into a mutated source code, which is identical to the original code except for few statements that are deleted or modified (in order to deliberately introduce a bug). For

example, when injecting a bug in a method call, the tool modifies an existing statement with a method call, by replacing it with another call to the same method but with different input parameters; or the method call is deleted (in order to emulate an omission by the programmer). The mutated source code is then deployed and executed in the test environment. In order to make the bug to surface (that is, to trigger a component failure caused by the bug), the system is exercised with a workload.

Finally, after the execution of the test, the tool collects log files from the system. These steps are then repeated for every test in the test plan.

2.2 Data Collection

One of the major issues related to the data collection phase is to understand which points in the system to monitor. This phase is closely linked to the architecture of the target system. Two monitoring points have been identified in OpenStack:

- the *Message Queue*, used to allow communication (through *RPC* messages) internally to the components of the target system;
- the *REST API*, used so that the components of the target system can receive requests from the outside (CLI, Dashboard, etc.)

Both the REST interfaces and the Message Queues are diffused communication technologies in cloud computing systems.

2.2.1 Zipkin Framework

The proposed approach is based on the tracing of events of the distributed system under analysis. In particular, the prototype of the approach uses *Zipkin* framework. A full description of *Zipkin* is given in the Appendix A.

Since the target system of this work is OpenStack Pike, we could have used *OSProfiler*, a tiny library that is used by most OpenStack projects and their python clients. It provides

```
1  from py_zipkin.zipkin import zipkin_span
2  def some_function(a, b):
3      with zipkin_span(
4          service_name='my_service',
5          span_name='my_span_name',
6          transport_handler=some_handler,
7          port=42,
8          sample_rate=0.05, # Value between 0.0 and 100.0
9      ):
10         do_stuff(a, b)
```

```
1  def some_function(a, b):
2      with zipkin_span(
3          service_name='my_service',
4          span_name='some_function',
5          transport_handler=some_handler,
6          port=42,
7          sample_rate=0.05,
8      ) as zipkin_context:
9          result = do_stuff(a, b)
10         zipkin_context.update_binary_annotations({'result': result})
```

functionality to generate one trace per request, that goes through all involved services. However, when the target system differs from OpenStack, obviously it is not possible to make use of this library. Thus it is preferable not to be dependent on the target system, also in order to increase the portability of the fault injection: this is the main reason that prompted us to utilize Zipkin.

Since the target system is written in Python, the *py_zipkin* library was used. This libray provides a context manager/decorator along with some utilities to facilitate the usage of Zipkin in Python applications. py_zipkin requires a transport_handler object that handles logging zipkin messages to a central logging service. The trace data collected is called a *Span*.

py_zipkin.zipkin.zipkin_span is the main tool for starting zipkin traces or logging spans inside an ongoing trace. zipkin_span can be used as a context manager or a decorator as shown in the following program listing:

zipkin_span.update_binary_annotations() can be used inside a zipkin trace to add to the existing set of binary annotations as follows:

Since we are interested not only in the service provided by one of the components of the distributed system, but also in the code (and the related call stack) that makes the request for service from another component in the system, we used the method *getouterframes* of Python *inspect* module. This function get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of frame: the first entry in the returned list represents frame; the last entry represents the outermost call on frame's stack.

2.2.2 Idle Trace, Fault Free Traces and Faulty Trace

The instrumented component that sends data to Zipkin is called a *Reporter*. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. In order to start the collection of data, it is needed to start the Zipkin sever listening on the port 9411. We will refer to this step as *enabling tracing*. When we want to stop the data collection, it is enough to stop Zipkin server: this step will be called *stop tracing*. The time between the enable tracing and the stop tracing represents to the data collection phase. In this phase data received by Zipkin collectors are saved in a JSON format.

We need to collect three types of traces (a set of events), as shown in Figure 2.2.1: idle, fault-free and faulty.

An *idle trace* is a trace collected while the target system is not executing specific commands given by the users. This trace is useful to understand which components are not linked to the execution of the test workload because they are periodic. These components will be called *noisy components*. Examples of these components in OpenStack Pike are the calls to *notify_service_capabilities* and to *update_service_capabilities* made by *cinder-scheduler*.

To collect the idle trace we need to:

1. Enable Tracing

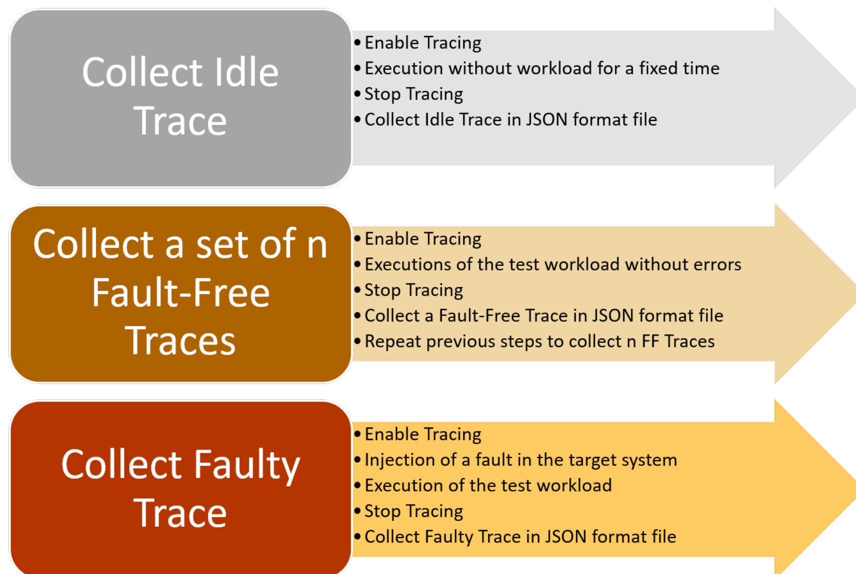


Figure 2.2.1: Workflow Data Collection

2. Collect the data for a fixed time (e.g. 60 seconds) while the system is in an idle state (i.e., it isn't executing specific command entered by users). By so doing, it is possible to collect a set of periodic events that will be removed from the analysis.

3. After a fixed time, stop the tracing.

4. At this point, a set of events composing the idle trace are saved in a JSON format file.

To evaluate the results of an experiment following the fault injection and to be able to identify the location of a fault and its consequences, we need a set of traces used to be compared with the faulty one. These traces are named *faulty-free traces*. In order to perform a better comparison, it is recommended to have more fault-free traces. The steps needed to collect a set of n fault-free are the following:

1. Enable Tracing.

2. Execution with no errors of a test workload without injection of a fault into target system.

3. Stop Tracing.

4. At this point, a set of events composing a fault-free trace are saved in a JSON format file.
5. Repeat the previous steps to collect a set of n fault-free traces.

The last trace to collect is the *faulty* one. This trace is collected following the injection of a fault into target system. The steps related to this last data collection are described above:

1. Enable Tracing.
2. Injection of a fault into the target system.
3. Execution of a test workload.
4. Stop Tracing.
5. At this point, a set of events composing a faulty trace are saved in a JSON format file.

After describing how to collect the data necessary for the analysis, it is possible to explain in detail the approach of failure mode analysis.

2.3 Data Analysis

The operation of this approach is based on the collaboration of several components that work sequentially in different stages of the workflow, as shown in the Figure 2.3.1. In these stages, input data are transformed and analyzed in order to detect *suspicious* events in the faulty trace, that is to detect those events whose presence (or absence) within the trace is hypothesized to be a consequence of the injected fault. The stages are the following:

1. *Parsing*: in this stage, input data are transformed into a fixed structure (CSV file format) containing the information needed for the analysis.

2. *Noise Extraction*: data are filtered removing noisy events from the CSV file created in the previous stage.

3. *Sequence Creati*on: in this phase a series of operations are performed in order to associate the collected data of each trace with a sequence of symbols.

4. *LCS*: the sequences created in the previous stage are compared by computing the *Longest Common Subsequence*. In this way it is also possible to obtain the differences between the compared sequences.

- *VOMM*: the differences obtained in the previous stage are evaluated by a probabilistic model in order to confirm or not them as real differences. These differences are the suspicious events to focus on.

- *Visualization*: in the last stage, a bar chart is generated showing a sequence of events to users. Different colors are used to separate suspicious events from those non-suspicious.

In the following sections the various stages presented will be analyzed in detail.

2.4 Parsing

In the first stage, a parser written in Python extracts the components of the collected data that are of interest for analysis and saves them in tabular form in a CSV format file. The parser represents the part most dependent on the target system and on the monitoring points since the construction of the data table is strongly linked to the data collected in the data collection phase. For this reason, the parser must be built ad hoc.

By focusing on our target system, information about events is dependent on monitoring points. In the case of RPC messages, the information we can gather is as follows:

- *Timestamp*: it is represents the initial time of an event expressed in the UNIX time stamp.

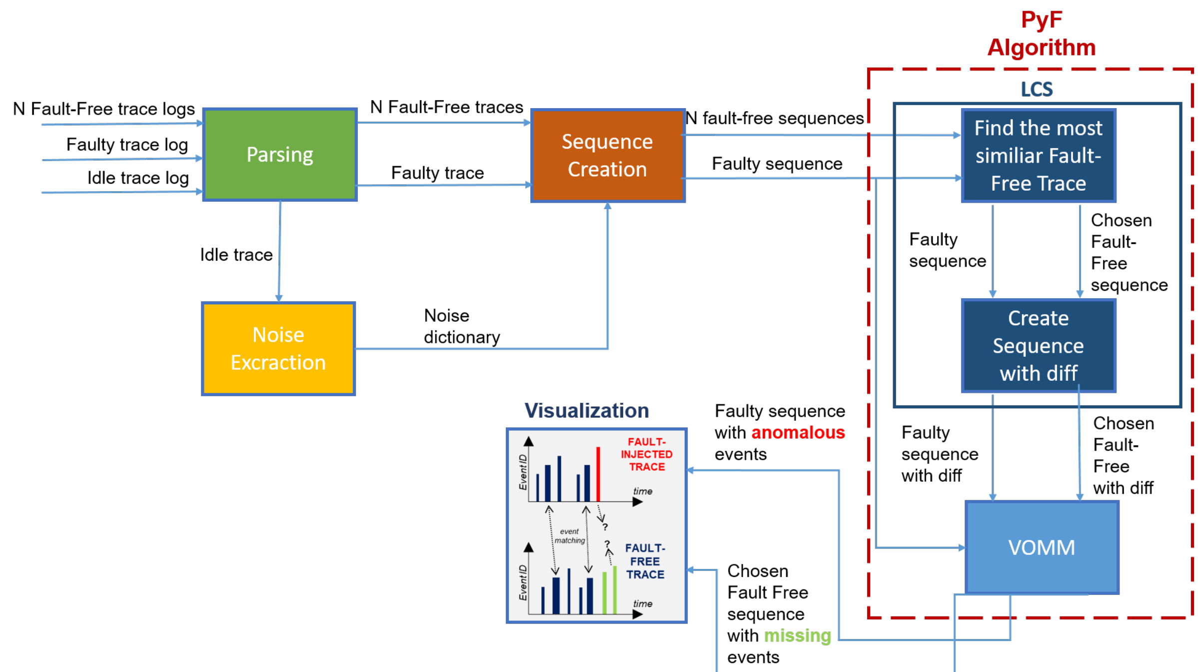


Figure 2.3.1: PyF workflow

- *Duration*: it is the duration of an event expressed in μs.
- *Method*: it is the name of the method invoked by current event.
- *Target*: it is the name of target component invoking the method. The *Method* and the *Target* refer to a service provided by one of the components of the distributed system (in particular, the function within the component that corresponds to the service).
- *Caller*: it refers to the code (and the related call stack) that makes the service request from another component in the system.

As for the REST API, we do not have the same information available for RPC messages. In this case the information collected is composed by Timestamp, Duration, Method ("request" method) and Target (e.g. "cinderclient", "novaclient", "neutronclient").

Thus, after the collection in a JSON format file of the idle trace, of a set of n fault-free traces and of the faulty one, the parser extract only the components of interest and save them in a CSV file format.

Table 2.1: Noise Dictionary of Openstack Pike

TargetName_InvokedMethod
cinder-scheduler_notify_service_capabilities
cinder-scheduler_update_service_capabilities
conductor_object_action
conductor_object_class_action_versions
q-metering-plugin_get_sync_data_metering
scheduler_sync_instance_info

2.5 Noise Extraction

The JSON format files saved during data collection phase contain a very large set of events. One of the problems due to data collection is that not all the collected data are related to the execution of the workload.

For example, in complex distributed infrastructures there is a series of periodic services whose calls are not dependent on the test workload. Including such events in the analysis makes the comparison of the sequences and the further fault detection very difficult. For this reason it is necessary to have a mechanism capable of removing events not related to the test workload. As described above, PyF approach takes in input also an idle trace that is a set of events collected while the system is not executing commands given by the user. The idea is to collect data while the system is in an idle state in order to understand what are the periodic services that are performed during the collection. These periodic events, which will also be called noisy events or noise, must be removed from the analysis.

In this stage, after the parsing of the idle trace in input as described in the Section 2.4, it is created a file containing target name and invoked method of the noisy events in the form *TargetName_InvokedMethod.* This file, called *noise dictionary*, will be used to generate sequence of events noise free in the next stages.

Figure 2.5.1 shows the workflow of the Noise Extraction phase. The noise dictionary created for the target system OpenStack Pike is illustrated in the Table 2.1.

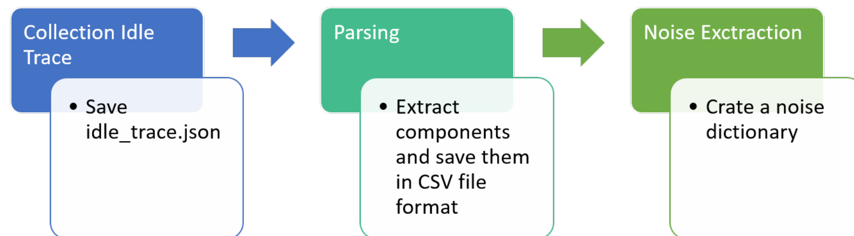


Figure 2.5.1: Noise Extraction workflow

2.6 Sequence Creation

This phase generates the sequences that will be compared in the next stages. Data obtained after the parsing are transformed making the following operations using a shell script:

- Step 1: remove the noisy events from data structure. This operation is accomplished finding for each event if the value *TargetName_InvokedMethod* matches one of the value *TargetName_InvokedMethod* in the file created during the noise extractor. If a value matches, then the event is cataloged as a noisy event and it is removed from structure, that is it will not be considered in the further creation of a sequence.
- Step 2: sort events in the table by increasing timestamp.
- Step 3: transforming timestamps from absolute into relative by subtracting from the timestamp of each event the minimum timestamp value present in the filtered data structure.
- Step 4: Converts the timestamp and the duration of each events from μs to s by dividing by one million these components in order to have a final bar chart more easily readable.

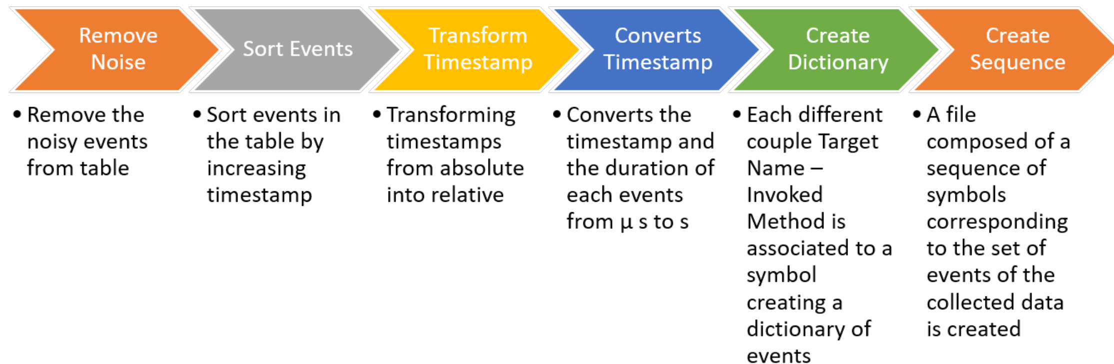


Figure 2.6.1: Workflow of the creation of sequences

- Step 5: Each different couple target name - method name is associated to a symbol (an incremental number) creating a dictionary of events. Obviously if different events have the same couple target name - method name then they will be identified by the same symbol.
- Step 6: A file composed of a sequence of symbols corresponding to the set of events of the collected data is created. This file will be used in next stages.

Figure 2.6.1 shows the workflow of this stage.

2.7 Longest Common Subsequence (LCS)

The sequences created in the previous stages are compared for finding anomalous events in the faulty traces. The basic idea is to use a string comparison technique in order to compare the sequence generated from the data collected during the fault injection (i.e., the faulty trace) with the sequences generated from the fault-free traces.

String comparison is a central operation in various environments: a spelling error correction program tries to find the dictionary entry which resembles most a given word, in molecular biology it is used to compare two DNA or protein sequences to learn how homologous they are, etc.

An obvious measure for the closeness of two strings is to find the maximum number of identical symbols in them (preserving the symbol order). This, by definition, is the *longest common subsequence* of the strings [3]. The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff utility, and has applications in bioinformatics. It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection of files.

Formally, we are comparing two input strings, $X[1\ldots m]$ and $Y[1\ldots n]$, which are elements of the set Σ^*; where Σ denotes the input alphabet containing σ symbols.

A *subsequence* $S[1\ldots s]$ of $X[1\ldots m]$ is obtained by deleting $m-s$ symbols from X. A common subsequence (cs) of $X[1\ldots m]$ and $Y[1\ldots n]$ is a subsequence which occurs in both strings. The longest common subsequence of strings X and Y, LCS(X, Y), is a common subsequence of maximal length. The *length* of LCS(X, Y) is denoted by r(X, Y).

The LCS problem has an optimal substructure: the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial. The LCS problem also has overlapping subproblems: the solution to high-level subproblems often reuse lower level subproblems. Problems with these two properties (optimal substructure and overlapping subproblems) can be approached by a problem-solving technique called dynamic programming, in which subproblem solutions are memoized rather than computed over and over. The procedure requires memoization, i.e. saving the solutions to one level of subproblem in a table (analogous to writing them to a memo, hence the name) so that the solutions are available to the next level of subproblems.

The subproblems become simpler as the sequences become shorter. Shorter sequences are conveniently described using the term *prefix*. A prefix of a sequence is the sequence with the end cut off. Let S be the sequence (AGCA). Then, the sequence (AG) is one of the prefixes of S. Prefixes are denoted with the name of the sequence, followed by a subscript to indicate how many characters the prefix contains. The prefix (AG) is denoted S_2, since it contains the first 2 elements of S. The possible prefixes of S are:

- S_1 = (A)
- S_2 = (AG)
- S_3 = (AGC)
- S_4 = (AGCA).

LCS(X,Y) is typically solved by filling an $m \times n$ table, that is used to store the LCS sequence for each step of the calculation. The task is to find the longest path between the vertices in the upper left and lower right corner of the table.

The traditional technique for solving $LCS\left(X\left[1\ldots m\right], Y\left[1\ldots n\right]\right)$ is to determine the longest common subsequence for all possible prefix combinations of the input strings. Let two sequences be defined as follows: $X = (x_1, x_2, \ldots x_m)$ and $Y = (y_1, y_2, \ldots y_n)$. The recurrence relation for extending the length of the LCS for each prefix pair $(X_{1,2,\ldots m}, Y_{1,2,\ldots n})$ is the following:

$$LCS\left[X_i, , Y_j\right] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS\left[X_{i-1},\ Y_{j-1}\right] + 1 & \text{if } x_i = y_j \\ max\left\{LCS\left[X_{i-1},\ Y_j\right],\ LCS\left[X_i,\ Y_{j-1}\right]\right\} & \text{if } x_i \neq y_j \end{cases}$$

To find the longest subsequences common to X_i and Y_j, compare the elements x_i and y_j:

- If they are equal, then the sequence $LCS\left[X_{i-1},\ Y_{j-1}\right]$ is extended by that element, x_i.

Table 2.2: LCS Table

	Ø	**A**	**G**	**C**	**A**	**T**
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	←↑Ø	↖G	⟵G	⟵G	⟵G
A	Ø	↖A	←↑A & G	←↑A & G	↖GA	⟵GA
C	Ø	↑A	←↑A & G	↖AC & GC	←↑AC & GC & GA	←↑AC & GC & GA

- If they are not equal:
  - if the two sequences $LCS\left[X_{i-1},\, Y_j\right]$ and $LCS\left[X_i,\, Y_{j-1}\right]$ are not the same length then the longer of these two sequences is retained.
  - if the two sequences$LCS\left[X_{i-1},\, Y_j\right]$ and $LCS\left[X_i,\, Y_{j-1}\right]$ are both the same length, but not identical, then both are retained.

Example In order to understand well the principle of operation of the algorithm, let's show a simple example based on two short strings X=(GAC) and Y=(GCAT). The final filled table is shown in the Table 2.2.

Because the LCS function uses a "zeroth" element, it is convenient to define zero prefixes that are empty for these sequences: $X_0 = Ø$; and $Y_0 = Ø$. The first column and first row have been filled in with Ø, because when an empty sequence is compared with a non-empty sequence, the longest common subsequence is always an empty sequence.

The final result is that the last cell contains all the longest subsequences common to (AGCAT) and (GAC); these are (AC), (GC), and (GA). Notice that the LCS is not necessarily unique, as shown in the example. Indeed, the LCS problem is often defined to be finding all common subsequences of a maximum length. This problem inherently has higher complexity, as the number of such subsequences is exponential in the worst case, even for only two input strings. For this reason we will limit to find only one longest common subsequence.

Table 2.3: LCS table storing length

	Ø	**A**	**G**	**C**	**A**	**T**
Ø	0	0	0	0	0	0
G	0	←↑0	↖1	⟵1	⟵1	⟵1
A	0	↖1	←↑1	←↑1	↖2	⟵2
C	Ø	↑1	←↑1	↖2	←↑2	←↑2

Table 2.4: Traceback example

	Ø	**A**	**G**	**C**	**A**	**T**
Ø	0	**0**	0	0	0	0
G	0	←↑0	↖**1**	⟵**1**	⟵1	⟵1
A	0	↖1	←↑1	←↑1	↖**2**	⟵**2**
C	Ø	↑1	←↑1	↖2	←↑2	←↑**2**

Traceback Approach Calculating the LCS of a row of the LCS table requires only the solutions to the current row and the previous row. Still, for long sequences, these sequences can get numerous and long, requiring a lot of storage space. Storage space can be saved by saving not the actual subsequences, but the length of the subsequence and the direction of the arrows, as in the Table 2.3.

The actual subsequences are deduced in a *traceback* procedure that follows the arrows backwards, starting from the last cell in the table. When the length decreases, the sequences must have had a common element. Several paths are possible when two arrows are shown in a cell. The Table 2.4 is used for such an analysis, with colored cells where the length is about to decrease. The bold numbers trace out one of longest common subsequence, (GA).

Normalized length An important aspect for the analysis is the length of the LCS $r(X,Y)$. In fact, through it, it is possible to calculate the *normalized length* of the LCS (nLCS), a metric used as the similarity measure for comparing two sequences [4]. Given two sequences X and Y of lengths l_x and l_y, respectively, it is possible to calculate the nLCS(X, Y) by the formula:

$$nLCS(X,Y) = \frac{r(X,Y)}{\sqrt{l_x \cdot l_y}} \tag{2.7.1}$$

The normalized length of the LCS is very helpful for compare the similarity between the faulty trace and the fault-free traces. In this way it is possible to estimate a faulty execution is dissimilar from the fault-free executions.

Print differences Calculating the LCS on sequences helps to find the elements in common but above all to focus on differences. It is possible, in fact, to write the compared sequence with differences, understood as those symbols not belonging to the LCS. Let's show an example based on the strings X= (ABCDBB) and Y=(CBACBAABA):

$$\begin{cases} X = & A, B, C, D, B, B \\ Y = & C, B, A, C, B, A, A, B, A \end{cases} \Rightarrow \tag{2.7.2}$$

$$LCS(X,Y) = B, C, B, B$$

$$diff = -A, +C, B, +A, C, -D, B, +A, +A, B, +A$$

The sequence *diff* is the union of two strings compared and must be interpreted as follows:

- the minus sign preceding a symbol in the sequence diff indicates that the current symbol is not present in the LCS output because it refers to the the first sequence but not to the second one;
- the plus sign preceding a symbol in the sequence diff indicates that the current symbol is not present in the LCS output because it refers to the the second sequence but not to the first one;
- no sign preceding a symbol in the sequence diff indicates that the current symbol is present in the LCS output because it refers to both sequences.

For the analysis it is useful to write the input sequences with sign of the differences in order to maintain a reference to each difference in the sequences, as follows:

$$\begin{cases} X = & -A,B,C,-D,B,B \\ Y = & +C,B,+A,C,B,+A,+A,B,+A \end{cases} \tag{2.7.3}$$

In this way it is possible to retain symbols (thus the the corresponding events) of both compared sequences not belonging to the LCS. It is precisely on these events that attention needs to be focused. We will refer to these symbols (i.e., to the symbols of the sequences preceded by a sign) as *diff symbols*.

Thus, by comparing the sequence of events of a faulty trace with a sequence of events of a fault free trace, it is possible to classify the corresponding events as follows:

- *Anomalous Events*: the events of the faulty trace that should not occur at that time of the workload execution. An event is anomalous when the associated symbol of the faulty trace is not present in the LCS (a symbol of the faulty trace preceded by minus sign; e.g. the first symbol of sequence X in 2.7.3). If an event of this type appears, then the behavior of the system is diverging from the correct behavior (e.g. the system is doing a wrong operation on a resource).
- *Missing Events*: events of the faulty trace that should have occurred at that time of the workload execution but which did not occur. An event is missing when the associated symbols of fault free-trace is not present in the LCS (a symbol of the fault-free trace preceded by plus sign; e.g. the first symbol of sequence Y in 2.7.3).
- *Non-Suspicious Events*: events common to both execution (faulty and fault-free). An event is common when the associated symbol of the faulty trace and of the fault free trace is present in the LCS (a symbol not preceded by a sign; e.g. the second symbol of the sequences X and Y in 2.7.3).

Anomalous and missing events will also be named *suspicious events*, that is events to

be explored. However, the differences shown with the LCS do not allow us to draw the conclusion that those non-aligned symbols correspond to events that need to be considered as suspicious. In fact it could simply happen that they do not belong to the longest path for the construction of the longest common subsequence. So they are not necessarily anomalous or missing event. As simple example, just think that the application of the LCS is not necessarily unique. So using a longest path of the table instead of another makes a symbol that can be perfectly aligned as an event not common. Another issue that makes difficult to identify suspicious event with the LCS is the *non-deterministic* behavior of complex systems. The behavior of the system cannot be predicted, and can change even if there is no failure. Therefore, it is difficult to distinguish between variations of the behavior that are caused by genuine failures, and variations that are caused by non-determinism and do not impact on the quality of service.

Therefore the only use of the LCS to identify the effects of a fault injection is not enough. Then it is necessary to use a probabilistic model that can validate or not the results obtained by the algorithm, as shown in the Figure 2.7.1.

In this work, we aim to support developers at identifying behavioral deviations that are actually caused by failures, by taking into account the possibility of non-deterministic behaviors that may sporadically impact on the system.

2.8 Variable order Markov Model (VMM)

As described in the previous section, the problem in applying only the LCS is to identify differences in the sequences that represent benign changes in behavior due to non-determinism (i.e., identify the false positives), and not to failure.

Learning of sequential data continues to be a fundamental task and a challenge in pattern recognition and machine learning. Applications involving sequential data may require prediction of new events, generation of new sequences, or decision making such as classification of sequences or sub-sequences. The classic application of discrete sequence pre-

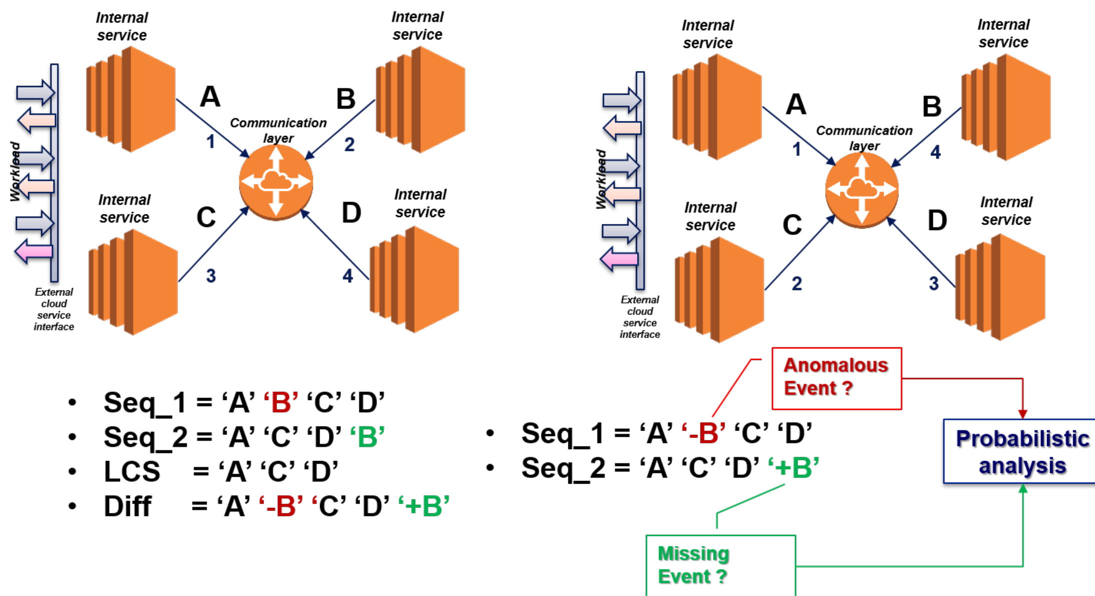


Figure 2.7.1: Probabilistic model to validate or not the differences

diction algorithms is lossless compression, but there are numerous other applications involving sequential data, which can be directly solved based on effective prediction of discrete sequences. Examples of such applications are biological sequence analysis, speech and language modeling, text analysis and extraction, music generation and classification, hand-writing recognition and behavior metric identification.

The literature on prediction algorithms for discrete sequences is rather extensive and offers various approaches to analyze and predict sequences over finite alphabets. Perhaps the most commonly used techniques are based on Hidden Markov Models (HMMs). HMMs provide flexible structures that can model complex sources of sequential data. However, dealing with HMMs typically requires considerable understanding of and insight into the problem domain in order to restrict possible model architectures. Also, due to their flexibility, successful training of HMMs usually requires very large training samples. In this thesis we focus on general-purpose prediction algorithms, based on learning Variable order Markov Models (VMMs) over a finite alphabet Σ. VMMs are widely used in the study of anomaly detection on sequences of syscall and it is suitable for our purposes to identify false positives.

In contrast to the Markov chain models, where each random variable in a sequence with a

Markov property depends on a fixed number of random variables, in VMMs this number of conditioning random variables may vary depending on the specific observed realization. This realization sequence is often called the *context*; therefore the VMMs are also called *context trees*.

Let's begin with some preliminary definitions. In this section it is used the terminology defined in [2].

Let Σ be a finite alphabet. A learner is given a training sequence $x_1^n = x_1 x_2 \ldots x_n$, where $x_i \in \Sigma$ and $x_i x_{i+1}$ is the concatenation of x_i and x_{i+1}. Based on x_1^n, the goal is to learn a model $\hat{P}$ that provides a probability assignment for any future outcome given some past. Specifically, for any context s $\in \Sigma^*$ and symbol $\sigma \in \Sigma$ the learner should generate a conditional probability$\hat{P}(\sigma|s)$.

Prediction performance is measured via the *average log-loss* $\ell\left(\hat{P}, x_1^T\right)$ of $\hat{P}(\cdot|\cdot)$, with respect to a test sequence $x_1^T = x_1 \ldots x_T$,

$$\ell\left(\hat{P}, x_1^T\right) = -\frac{1}{T}\sum_{i=1}^{T} \log \hat{P}(x_i | x_{1 \ldots} x_{i-1}) \tag{2.8.1}$$

where the logarithm is taken to base 2. Given the occurrence of the actual sequence x_1^T, the average log-loss represents the probability of occurrence of that sequence according to the predictor.

There are several VMM predection algorithms, such as PPM (discussed in Subsection 2.8.1).

In general, most algorithms for learning VMMs include three components: *counting*, *smoothing* (probabilities of unobserved events) and v*ariable-length modeling*.

Specifically, all such algorithms base their probability estimates on counts of the number of occurrences of symbols σ appearing after contexts s in the training sequence. These counts provide the basis for generating the predictor $\hat{P}$.

The smoothing component defines how to handle unobserved events (with zero value counts). Smoothing attributes a non-zero score to unobserved contexts, by taking into

account the scores of observed contexts that are similar to the unobserved one. The existence of such events is also called the *zero frequency problem*. Not handling the zero frequency problem is harmful because the log-loss of an unobserved but possible event, which is assigned a zero probability by $\hat{P}$, is infinite.

Finally, variable length modeling can be done in many ways. Some of the algorithms discussed here construct a single model and some construct several models and average them. The models themselves can be bounded by a predetermined constant bound, which means that the algorithm does not consider contexts that are longer than the bound. Alternatively, models may not be bounded a-priori, in which case the maximal context size is data-driven.

2.8.1 Prediction by Partial Match (PPM)

Prediction by Partial Match (PPM) was developed in 1984 by Cleary and Witten. This was followed by a series of improvements resulting in the version PPM-C. The algorithm uses a finite context Markov Model to predict the occurrence of the next symbol based on the occurrences of the previous symbols in the text used to train the model. The method blends together multiple lower order Markov models in the case when a given order is unsuitable for the prediction. Thus it recursively falls onto lower order models until the least order where every symbol occurs with a probability of $1/|\Sigma|$ where Σ is the total number of symbols in the alphabet. This accounts for the escape probability in case a symbol was never before seen in the input text used to generate the model statistics.

Though the initial applications of the PPM algorithm were to enable better compression, it has been applied to text prediction, language identification, dialect identification, language segmentation, word segmentation, text categorization etc.

PPM uses the Markov model principle to perform the compression and the prediction. Each symbol is encoded based on the probability of the symbol calculated in the context of the n previous symbols. The value n acts as the order of the model. PPM performs

a blending of different order models through the process of exclusion to remove lower order predictions for a given symbol. This is achieved by calculating probabilities for all orders lower than the highest order specified. Most importantly a 0 order distribution is calculated from the unconditioned probabilities of the symbols and a -1 order model is also constructed where the probabilities of the symbols are the reciprocal of the alphabet size leading to very small but finite probability of occurrence. Each of the probabilities in all the order models are calculated by considering the frequency counts of the symbols.

The blending mechanism of PPM can be implemented at different levels depending on the token size. This can be either character based, word based or even sentence based.

Escape and Exclusion mechanisms PPM handles the zero frequency problem using two mechanisms called *escape* and *exclusion*.

There are several PPM variants distinguished by the implementation of the escape mechanism. In all variants the escape mechanism works as follows. For each context s of length k $\leq$ D, the algorithm allocates a probability mass $\hat{P}_k(escape|s)$ for all symbols that did not appear after the context s (in the training sequence). The remaining mass $1-\hat{P}_k(escape|s)$ is distributed among all other symbols that have non-zero counts for this context. The particular mass allocation for escape and the particular mass distribution $\hat{P}_k(\sigma|s)$, over these other symbols σ, determine the PPM variant.

The exclusion mechanism is used to enhance the escape estimation. It is based on the observation that if a symbol σ appears after the context s of length k $\leq$ D, there is no need to consider σ as part of the alphabet when calculating $\hat{P}_k(\cdot|s')$ for all s' suffix of s. Therefore, the estimates $\hat{P}_k(\cdot|s')$ are potentially more accurate since they are based on a smaller (observed) alphabet.

PPM-C PPM-C or Method C is a variant of PPM differing from PPM itself in that it uses a different scheme to evaluate the escape probability when PPM switches to a lower order model to make a prediction. PPM-C uses the number of new symbols that were

already observed in the given stream to calculate the escape probability.

For each sequence s and symbol σ let N(sσ) denote the number of occurrences of sσ in the training sequence. Let Σ_s be the set of symbols appearing after the context s (in the training sequence); that is, $\Sigma_s = \{\sigma : N(s\sigma) > 0\}$.

PPM-C is based on a trie data structure (also called digital tree and sometimes radix tree or prefix tree). In the learning phase the algorithm constructs a trie T from the training sequence q_1^n. Each node in T is associated with a symbol and has a counter. The maximal depth of T is D + 1. The algorithm starts with a root node corresponding to the empty sequence ε and incrementally parses the training sequence, one symbol at a time. Each parsed symbol q_i and its D-sized context, x_{i-D}^{i-1} , define a potential path in T , which is constructed, if it does not yet exist. Note that after parsing the first D symbols, each newly constructed path is of length D + 1. The counters along this path are incremented. Therefore, the counter of any node, with a corresponding path sσ (where σ is the symbol associated with the node) is N (sσ). Upon completion, the resulting trie induces the probability $\hat{P}(\sigma|s)$ for each symbol σ and context s with $|s| \leq D$. To compute $\hat{P}(\sigma|s)$ we start from the root and traverse the tree according to the longest suffix of s, denoted s' , such that $s'\sigma$ corresponds to a complete path from the root to a leaf. We then use the counters N (s' σ') to compute $\Sigma_{s'}$ and the estimates.

PPM uses the previous k symbols to calculate the probability of occurrence of the next symbol. To achieve this, the algorithm stores all length-k subsequences of characters and calculates a probability distribution of these subsequences for different values of k. The algorithm follows a recursive procedure to determine the highest order K which can be used to perform arithmetic coding and generate a probability for the current symbol. Thus it contiguously switches from order k to lower orders until an order which is most suitable for prediction is reached.

In case a character that appears in the stream is novel to order k then it cannot be assigned a probability with a model of that order. Hence the k-th order model becomes unsuitable and the next lower order model is chosen. In the case that even this model is unsuitable

(i.e., the character is novel to this model also), then the next lower model is triggered etc. To ensure that the recursion does not proceed indefinitely, a final model is present that contains all the characters in the given alphabet. Each time the decoder switches from one higher order model to a lower order model it generates an escape sequence with a specific probability calculated for that model. Thus the algorithm makes sure that all characters are assigned a probability value.

Example Let us consider an input text "abracadabra". We shall now observe how PPM will use the symbol frequencies to generate models of order k=2, 1, 0, and -1 and use these distributions to predict the probability of occurrence of the next symbol in the context.

The Table 2.5 is generated by first enumerating the different length-k subsequence strings in the input. With k=2 we get 7 different strings as in the first column of the table. With order k=1 we get 5 strings etc. Each column has an escape sequence whose count is equal to the number of distinct symbols seen in the input until then. Thus the third column of order k=0 has a count of 5 associated with the escape sequence as 5 character were observed in that order. The final column has the lowest order model that would be used if a character was totally unseen. Consider three characters which may appear following the input symbols "abracadabra". The probabilities of occurrence of these characters are shown in Table 2.6.

If the next character in the stream is "c" then it matches the first column with k=2 where "c" followed the context "ra" previously. Therefore "c" would be encoded with a probability of $\frac{1}{2}$. Consider the character "d" appearing following "abracadabra". We can see that "d" never followed the characters "ra" previously. Therefore the k=2 order model becomes unsuitable and an escape sequence is generated with a probability of $\frac{1}{2}$. Following this the next lower order i.e. k=1 model is chosen and it can be seen that "d" followed "a" with a probability of $\frac{1}{7}$. Therefore the combined probability for "d" to appear is the product of the two probabilities and equals $\frac{1}{14}$. Finally, let us consider the case where a symbol was never seen in the text stream. Consider the character "t". Since "t" did

Table 2.5: PPM Table

Order k = 2			**Order k = 1**			**Order k = 0**			**Order k = -1**		
Prediction	**c**	**p**	**Prediction**	**c**	**p**	**Prediction**	**c**	**p**	**Prediction**	**c**	**p**
ab			a			–> a	5	5/16	–> Σ	1	1/ |Σ|
–> r	2	2/3	–> b	2	2/7						
–> esc	1	1/3	–> c	1	1/7						
			–> d	1	1/7						
			–> esc	3	3/7						
ac			b			–> b	2	2/16			
–> a	1	1/2	–> r	2	2/3						
–> esc	1	1/2	–> esc	1	1/3						
ad			c			–> c	1	1/16			
–> a	1	1/2	–>	1	1/2						
–> esc	1	1/2	–> esc	1	1/2						
ba			d			–> d	1	1/16			
–> a	2	2/3	–> a	1	1/2						
–> esc	1	1/3	–>	1	1/2						
ca			r			–> r	2	2/16			
–> d	1	1/2	–> a	2	2/3						
–> esc	1	1/2	–> esc	1	1/3						
da						–> esc	5	5/16			
–> b	1	1/2									
–> esc	1	1/2									
ra											
–> c	1	1/2									
–> esc	1	1/2									

Table 2.6: PPM probabilities

Character	Probabilities without exclusion	Probabilities with exclusion
c	$\frac{1}{2}$	$\frac{1}{2}$
d	$\frac{1}{2} \cdot \frac{1}{7}$	$\frac{1}{2} \cdot \frac{1}{6}$
t	$\frac{1}{2} \cdot \frac{3}{7} \cdot \frac{5}{16} \cdot \frac{1}{\lvert\Sigma\rvert}$	$\frac{1}{2} \cdot \frac{3}{6} \cdot \frac{5}{12} \cdot \frac{1}{\lvert\Sigma\rvert-5}$

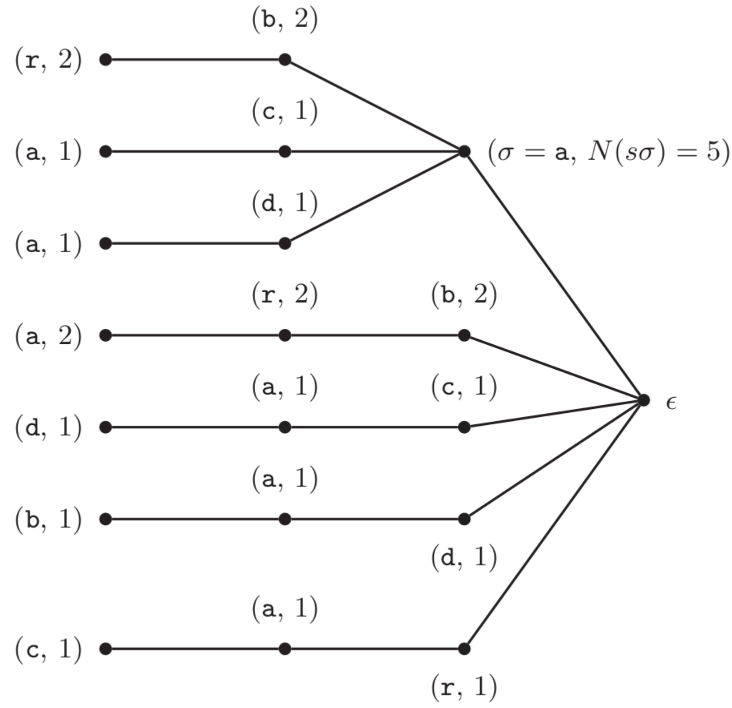


Figure 2.8.1: PPM-C trie

not occur at all in the input stream it would generate an escape sequence for all models irrespective of k. However it will finally end with k=-1 where the probability of finding "t" would be $\frac{1}{|\Sigma|}$ or equal to $\frac{1}{256}$. By multiplying with escape sequence probabilities of all models prior to this, we see that "t" has a probability of $\frac{1}{3822.33}$.

The Figure 2.8.1 shows the tree constructed by PPM-C using the training sequence "abracadabra".

$\hat{P}(d|ra) = \hat{P}(escape|ra) \cdot \hat{P}(d|a) = \frac{1}{2} \cdot \frac{1}{4+3} = 0.07$ without using the exclusion mechanism. When using the exclusion, observe that $N(rac) > 0$; therefore, we can exclude c when evaluating $\hat{P}(d|ra) = \hat{P}(escape|ra) \cdot \hat{P}(d|a) = \frac{1}{2} \cdot \frac{1}{6} = 0.08$.

2.9 PyF Algorithm

PPM, and in particular its PPM-C variant, is an excellent prediction algorithm. However, it remains to understand how to apply this algorithm to the differences obtained as a result of the use of the LCS on the compared sequences, in order to validate or not the results

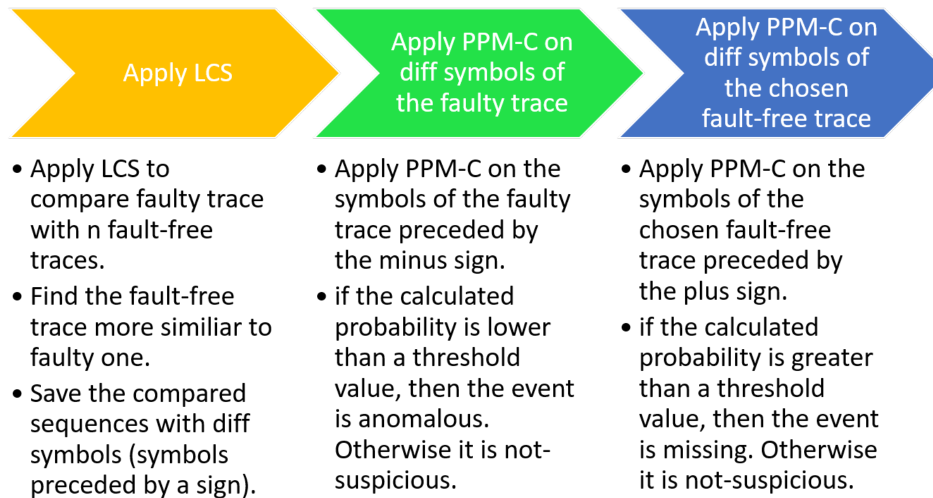


Figure 2.9.1: Steps of PyF Algorithm

obtained.

PyF algorithm is the heart of the approach. It applies two algorithms written in Python implementing LCS and PPM-C respectively in order to detect symbols of compared sequence corresponding to suspicious events.

An overview of the steps of the algorithm is shown in Figure 2.9.1.

First, after the sequences of symbols of the faulty trace and of the set of n fault-free traces have been created, the LCS is applied to compare the faulty trace with each of the fault-free traces. In this way, through the *normalized length* (eq. 2.7.1), a comparison is made in order to identify which fault-free trace is the most similar to the faulty one. The fault-free trace most similar to the faulty one will be also called *chosen fault-free trace* or simply *chosen ff*. This step is fundamental because the chosen faulty free trace will be used for a direct comparison.

After that the chosen ff has been found, the sequences corresponding to the two traces (the faulty trace and the chosen ff) are written with the diffs as shown by way of example in the Equation 2.7.3 in order to retain symbols (thus the corresponding events) of the sequences that do not belong to the LCS. But, as described above, the differences

shown with the LCS does not allow to define that those non-aligned symbols necessarily correspond to event that actually represent suspicious events. Hence the need to apply a VMM prediction algorithm as PPM-C. In fact, VMMs allow us to assign a probability to the anomalous/missing events, thus determining if the events are actually suspicious. In order to do this, it is necessary to have a training sequence or training model and an observed sequence or test sequence. All such algorithms base their probability estimates on counts of the number of occurrences of symbols σ appearing after contexts s in the training sequence. These counts provide the basis for generating the predictor $\hat{P}$.

The PyF algorithm applies the PPM-C both on the sequences of the faulty trace and on the chosen ff in two rounds. In each round a different training sequence is constructed.

Round 1 In this round, the observed sequence corresponds to the faulty trace. The training model, instead, is the concatenation of all n fault-free traces collected. As the number of fault-free traces increases, the accuracy of the training model improves. However, a large number of fault-free data leads to a significant increase in data collection times as well as greater computational times in the analysis. So here too there is a trade-off between accuracy of results and costs (in terms of time to perform the experiments).

As above described, prediction performance is measured via the *average log-loss* shown in Eq. 2.8.1, with respect to a test sequence $x_1^T = x_1 \dots x_T$.

However, we do not want to focus on the entire test sequence, but only on those symbols of the observed sequence that are not present in the LCS. The goal is to understand if these symbols of the faulty sequence must be really categorized as anomalous events. To achieve this result, we calculate the individual probabilities of each diff symbol x_i of the test sequence x_1^T as the value of $\hat{P}(x_i | x_{1 \dots} x_{i-1})$:

- If such value is less than a minimum threshold value, then the probabilistic model states that there is low likelihood for the symbol x_i to be in that position of the test sequence. Then PPM-C confirms the result of the LCS for the symbol and the associated event is confirmed to be an anomalous event.

- If such value is greater than a minimum threshold value, then the probabilistic model states that the likelihood for the symbol x_i to be in that position of the test sequence is not low. Therefore PPM-C does not confirm the result of the LCS for the symbol and the associated event is not confirmed to be an anomalous event and and it is added to the list of non-suspicious events removing the minus sign before the symbol.

Round 2 In the second round, the observed sequence corresponds to the chosen fault free trace. The training model, instead, is the concatenation of the remaining n-1 fault-free traces (all the fault free traces except the chosen ff).

However, as for the faulty trace, we do not want to focus on the entire test sequence, but only on those symbols of the chosen fault free that are not present in the LCS in order to understand if these symbols must be really categorized as missing events. So the algorithm does not limit its analysis to the only faulty trace, but it also extends it to the most similar fault-free one. The goal is to understand for each diff symbol present in the chosen fault-free trace the likelihood of be in that position of the test sequence. To achieve this result, we calculate the value of individual probabilities of each diff symbol y_i of the test sequence $y_1^T = y_1 \dots y_T$ in the same way described in the previous round:

- If such value is less than a minimum threshold value, then the probabilistic model states that there is low likelihood for the symbol y_i to be in that position of the test sequence. Then PPM-C does not confirm the result of the LCS for the symbol and the associated event is not confirmed to be an missing event of the faulty trace and it is added to the list of non-suspicious events removing the plus sign before the symbol.

- If such value is greater than a minimum threshold value, then the probabilistic model states that the likelihood for the symbol y_i to be in that position of the test sequence is not low. Therefore PPM-C confirm the result of the LCS for the symbol and the associated event is confirmed to be a missing event.

Table 2.7: Results of PPM-C on a diff symbol

Observed sequence	**Training model**	**Comparison likelihood of i-th symbol with a threshold value ε**	**Result**	**Analyzed event**
faulty trace $x_1^M = x_1 \dots x_M$	Concatenation of n fault-free traces	$\hat{P}(x_i \mid x_{1 \dots} x_{i-1}) < \varepsilon$	PPM-C confirms LCS	x_i is an anomalous event
		$\hat{P}(x_i \mid x_{1 \dots} x_{i-1}) > \varepsilon$	PPM-C does not confirm LCS	x_i is a non-suspicious event
fault free trace $y_1^N = y_1 \dots y_N$	Concatenation of n-1 fault-free traces	$\hat{P}(y_i \mid y_{1 \dots} y_{i-1}) < \varepsilon$	PPM-C does not confirm LCS	y_i is a non-suspicious event
		$\hat{P}(y_i \mid y_{1 \dots} y_{i-1}) > \varepsilon$	PPM-C confirms LCS	y_i is a missing event

Notice that, although in the two rounds the comparisons with a threshold value ε are made in the same way, the results obtained are different and complementary. The Table 2.7 shows a summary of what has been described above as a simplification.

Thresholds calibration The results obtained by PPM-C depend very much on the value that is set as a threshold. A very high threshold value carries the risk of not distinguishing the false positives due to the non-deterministic nature of the system from events that are suspect due to system failure. Vice versa, a too low threshold value would make the approach too rigid in the evaluation of false positives, cataloging almost always the anomalous events highlighted by the LCS as not suspicious.

There is no single threshold value that can be set because it depends on the training model. Thus, the fundamental aspect is to have a good training model, i.e. to collect a good number of fault-free traces available (at least 10). With a good model, the probabilities calculated by PPM-C are very accurate, providing a very high probability value when the event is non-suspicious and a very low probability value when the event is suspicious

(usually less than 0.05).

2.10 Visualization

After the execution of the PyF algorithm, the sequences of the faulty trace and of the most similar fault-free trace will contain as diff symbols only those differences highlighted by both LCS and PPM-C.

The last stage of PyF is to generate an HTML chart that is easy to interpret. In fact, the goal is to provide a chart that is easily understandable and interpretable not only for skilled and experts developers but also for general users.

The basic idea is to show two charts through two adjacent subplots: the first is related to the faulty trace, while the second represents the fault-free trace used for the comparison. Each symbol of a sequence (i.e., each event of a trace) can be modeled as a vertical bar in which:

- timestamp of the event represents the start time of the corresponding vertical graph along the time axis, while the sum of the timestamp and of the duration of an event represents the end time of the corresponding vertical graph. The distance between the start time and the end time of an event shape the *width* of the corresponding vertical bar.
- the *height* of each vertical bar depends first of all by the type of call:
  - in the case of a RPC call, the height is related to the position of the corresponding event in the trace multiplied by an offset. The use of offset is necessary to better distinguish the heights of different bars. So the bars will have a height that grows directly proportional with the position of the event in the trace.
  - in the case of a REST call, the height has a prefixed dimension.

In order to distinguish the RPC calls from REST calls, the first ones are aligned with the

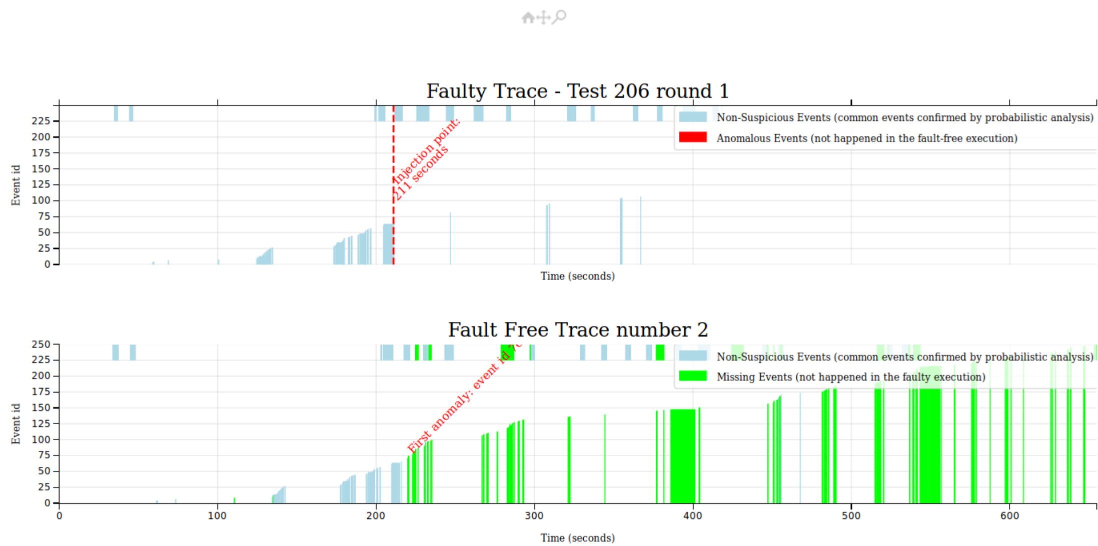


Figure 2.10.1: Plot example

time axis (horizontal axis), while the last ones are positioned on the upper part of the subplots.

The symbols of the compared traces that are not preceded by a sign are those events that are listed as non-suspicious from PyF algorithm. These events are represented with a vertical bar of the same color in both subplots (light blue). Symbols of the faulty trace preceded by a minus sign are those events classified as anomalous after the analysis in the previous stages: these events are represented through a red vertical bar in the first subplot. Lastly, symbols of the fault-free trace preceded by a plus sign are those events classified as missing after the previous analysis: these events are represented through a green vertical bar in the second subplot.

Furthermore, in the first subplot, a vertical dashed line is shown at the instant in which the first fault activation occurs during the workload test. In order to further help the user to understand where there are the first differences to focus on between the two traces compared, an annotations has also been inserted indicating the first of a series of consecutive events that differ in the two traces.

The Figure 2.10.1 shows a typical plot created with two sequences compared.

The HTML graph is created with *mpld3*, a project that brings together *Matplotlib*, the

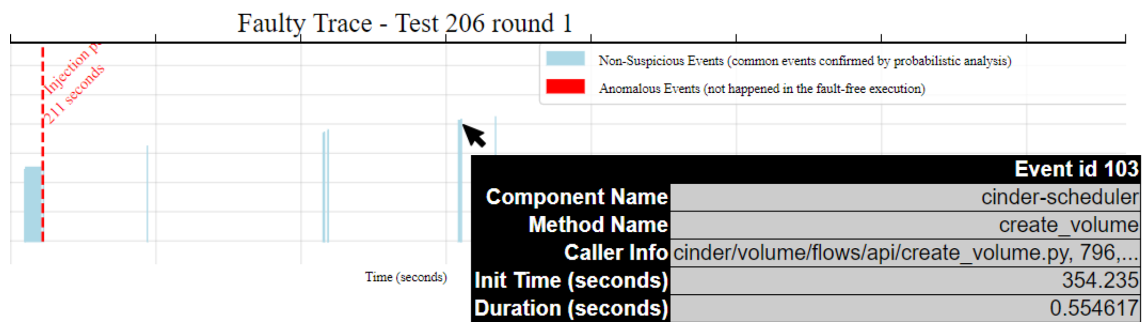


Figure 2.10.2: Mpld3 plugin

popular Python-based graphing library, and *D3js*, the popular JavaScript library for creating interactive data visualizations for the web. The result is a simple API for exporting Matplotlib graphics to HTML code which can be used within the browser, within standard web pages, blogs, or tools. Mpld3 offers a set of plugins allowing the definition of new interactive behaviors on Matplotlib plots through the use of JavaScript and D3js.

The potential offered by mpld3 makes the graph easy to interpret. For example, by the use of a plugin, when the user moves the mouse on a vertical bar, a table containing the main information of the corresponding event is shown, as illustrated in the Figure 2.10.2. In this way it is possible to get information of the event we want to focus on. The information that can be viewed by moving the

mouse pointer over a vertical bar are:

- **Component Name**: the name of the component calling the current method.
- **Method Name**: the name of the invoked method.
- **Caller Info**: a set of information about the previous caller of the current method.
- **Init Time**: the initial time of the call, expressed in seconds.
- **Duration**: the duration of the call, expressed in seconds.

It should be noted that between the two subplots there is no temporal link, both for the non-determinism of the analyzed systems, and because the faulty trace and the fault free traces refer to executions on different target machines, at different times.

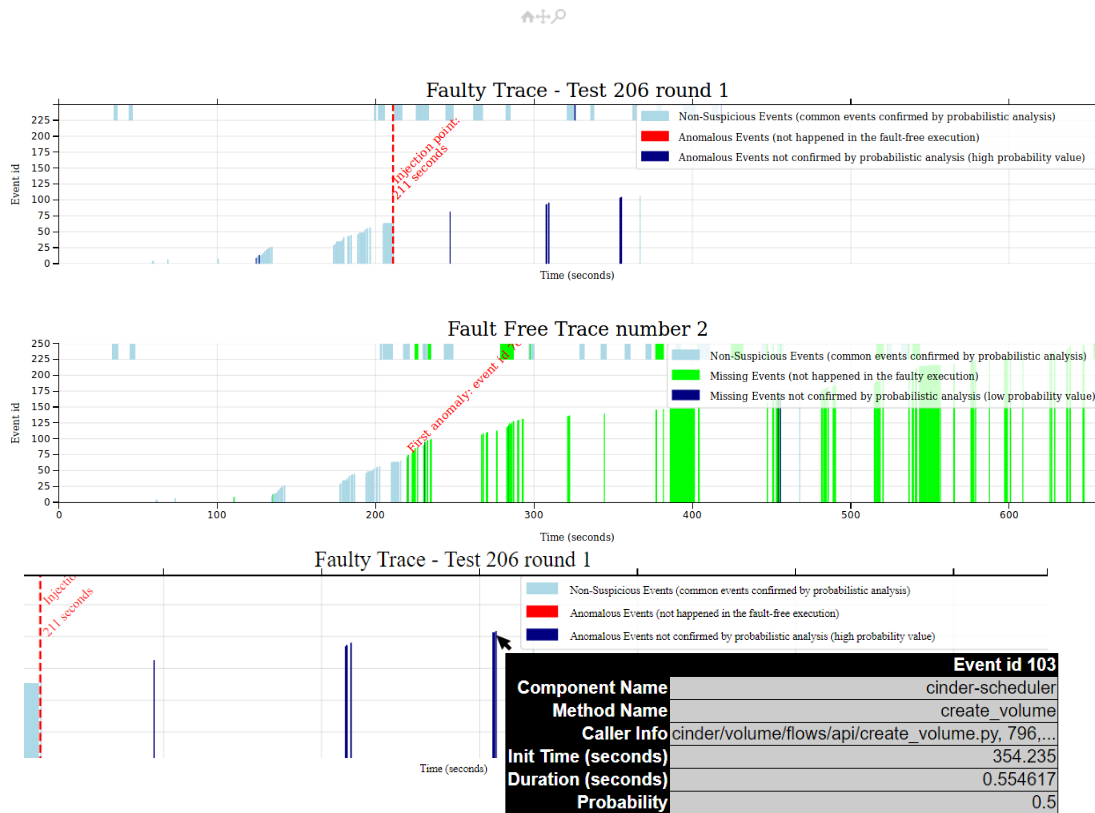


Figure 2.10.3: Plot example - Debug Mode

The plot can also be created in *debug mode* or *expert mode*, as shown in Figure 2.10.3. This mode, unlike the previously described mode (named *user mode*), implies that the events indicated as suspected by LCS and not confirmed by PPM-C (with consequent categorization of the event as not suspicious) are represented with a vertical bar of different color (dark blue). In this way it is possible to distinguish the analysis performed by LCS from that performed by the probabilistic model and to appreciate, moreover, the results obtained by the latter. In this mode, when we move the mouse cursor over a vertical bar, it is also shown the probability calculated by PPM-C that the event can be found at that point of the sequence in addition to the information shown in the user mode.

Chapter 3

Experimental Analysis on OpenStack

This chapter shows the results obtained by PyF in analyzing the failure modes of selected experiments in a campaign performed on the target system OpenStack Pike. Two campaigns of experiments were executed: one performed on the *Nova subsystem*[1], while the other on the *Cinder subsystem*[2].

First of all, the experimental setup is described followed by the reports generated by PyF for four selected experiments. The last section describes the evaluation of the performances of the approach.

3.1 Experimental Setup

The test workload directly exercises the injected components/services. It submits variegated inputs in order to cover as much code paths as possible, including the buggy execution path that was injected in the mutated source code. For example, in the case of OpenStack, the foreground workload should stimulate many of the OpenStack APIs, including the APIs related to the injected component (for example, *nova-api* in the case of the *Nova subsystem*, *cinder-volume* in the case of the *Cinder subsystem*, etc.). Further-

[1]Nova is the OpenStack project that provides a way to provision compute instances.

[2]Cinder s a Block Storage service for OpenStack. It is designed to present storage resources to end users that can be consumed by the OpenStack Compute Project (Nova).

more, the workload serves to detect service failures; for example, by looking for failed or stalled API calls, or by performing consistency checks in the workload (e.g., after creating a new instance, by checking that the instance was indeed created). The detection of service failures also serves to evaluate the recovery abilities of the system. The Figure 3.1.1 provides an overview of our experimental design, already extensively detailed in the Chapter 2.

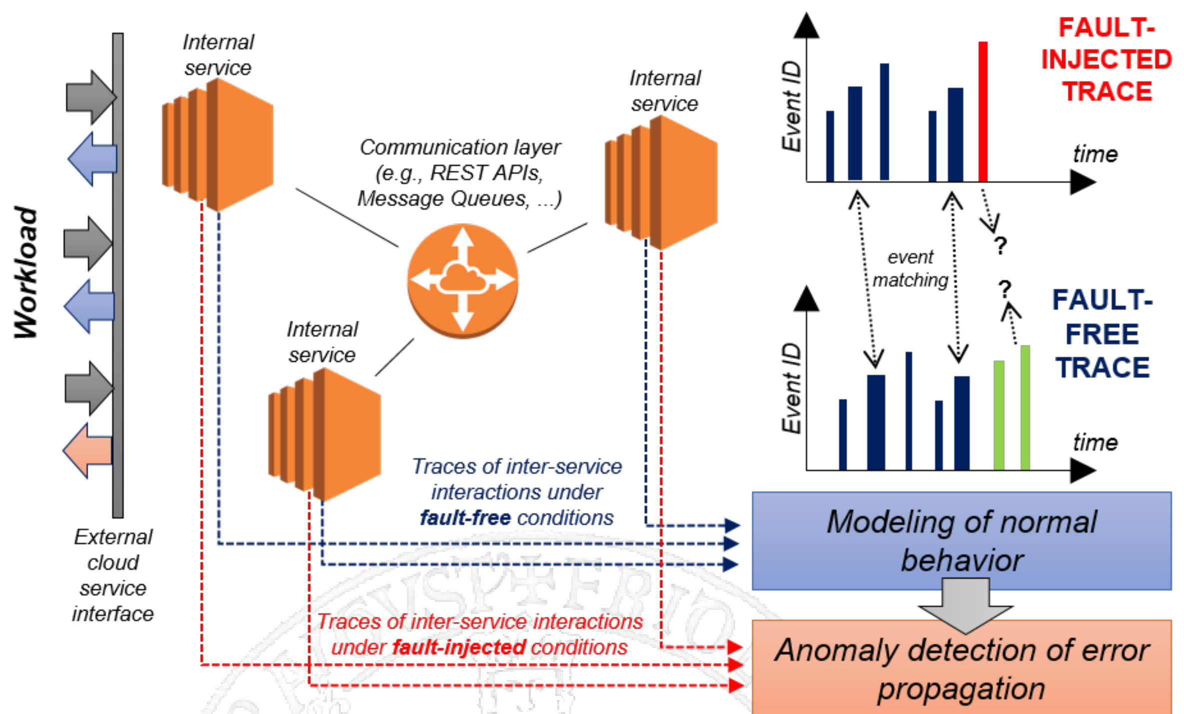


Figure 3.1.1: Experimental Design

The test workload performed on the target system consists of the following steps:

1. Create image.
2. Create key pair.
3. Boot instance with key pair and get list of instances.
4. Create volume and show list of volumes.
5. Attach volume to instance and get list of volumes.

6. Add IP to instance.
7. Create and add security group to instance.
8. Check SSH connection to instance.
9. Reboot instance.
10. Check SSH connection to instance after reboot.

To guarantee the independence among the experiments (issue discussed in the Section 1.3), a series of preliminary and subsequent operations to the workload execution are performed. These operations ensure that the effects due to the previous injection do not affect the execution of the workloads related to the subsequent injections.

First of all, we made a backup of all the files that will be changed due to the injection. Then the mutation of the files takes place introducing faults at software level, followed by a restart of all the system's services and a clean up of the environment in order to eliminate the effects of previous executions (e.g. image creation, volumes creation, etc.). After carrying out these preliminary operations, the test workload is executed to understand how the system reacts and, at the end of the execution, the log files of the target system and the log file related to the workload are saved. The system log files are useful to understand if the system has logged or not a high-severity message; the log file of the workload needs instead to understand if there are failures during the execution. At the end of the workload execution, the original file is restored by replacing the previously mutated one. Then a database restore is performed to resolve possible permanent effects on the target system database due to the activation of specific faults. The last operation is the restart of all the system's services.

To limit the problems related to the fault activation (issue already discussed in the section 1.3), a *coverage analysis* is performed in order to focus only on the program statements that are actually covered by the workload and thus avoiding to inject faults that are not covered (*skipped experiments*). So the injected bugs are designed to guarantee fault acti-

vation once the buggy code is covered, by explicitly overwriting the program state with incorrect values in order to force an error (e.g. an input parameter of a function is deleted or replaced with an invalid parameter such as *None*, a function call is omitted, etc.).

If the execution of one of the workload operations involves the generation of an *API error* by the target system, then it means that the system failed to perform the requested operation by returning an error message. In this case, the workload is terminated since it has been the target system itself to notice the failure. The API error message and its timestamp are saved in the log file related to the workload.

In each step, our fault injection system verifies that the requested operation has been successful. For example, after the request to create a volume, a check is made to understand the status of the volume created. If it is in an not available status, then an *assertion failure* is generated with its timestamp and will be saved in the log file related to the workload. The generation of an assertion failure does not interrupt the execution of the workload, since it was not the target system that reported an error through an API error. An assertion failure without the presence of API error allows both to evaluate the temporal and spatial propagation of a failure, and to identify how to improve the fault tolerance mechanisms of the target system.

From the presence or absence of assertion failure and API error in log file of the workload, it is possible to identify four cases:

1. **API error only**: the execution of the workload has been interrupted due to an API error message generated by the system. The absence of the assertion failure indicates that the system has immediately detected a failure, thus highlighting a good fault tolerance behavior. In these cases spatial and temporal propagation have been limited.

2. **Assertion Failure and API error**: the execution of the workload has been interrupted due to an API error message generated by the system. The presence of the assertion failure indicates that the system has not immediately detected a failure:

for this reason it is useful to deepen the spatial and temporal propagation of the faults.

3. **Assertion Failure only**: the execution of the workload has not been interrupted due to the absence of an API error message. These cases are probably the most interesting because the system has not realized that it has failed despite one or more workload operations have not been successful (the reason for which an assertion failure is generated). So the system failed but did not notify the user of the failure through an API error message. These cases are worth to be analyzed more in depth to look for spatial and temporal propagation, but above all to understand how it is possible to improve the system's fault tolerance mechanisms.

4. **No Assertion Failure and No API error**: these cases could almost be considered normal, in the sense that the execution of the workload does not necessarily have to produce failures, and therefore the workload can complete its execution and terminate correctly without generating assertion failures and API error. For example, the corrupted program state may not have an influence on the behavior of the target system (e.g., there may be no influence if the erroneous data are not processed by the software during the course of the experiment). These cases are important to identify the faults that have not an impact on the software.

The first three cases described above are used for analyzing the failure modes of the system, that is, the impact of the fault on the service provided by the system (a key aspect of the fault injection, as described in the Section 1.3). It is important to understand whether the system is able to point out the presence of the failures: the best case is described in the point number 1, where the system clearly notify the presence of the failure, by generating an explicit error signal in a reasonable time. The point number 2 can be seen as an extension of point 1, with the difference that the system notifies an error not immediately. So it is useful to understand in these cases the spatial and temporal propagation, and try to understand how to improve the fault tolerance mechanism. The worst case is represented

by case described in the point number 3, where the failure is unnoticed. In this case could be useful to deepen the analysis of the target system in order to understand if the system has logged any useful information to let the system administrators to recover the system and to repair the fault.

The last case, described in the point number 4, can be used to identify the faults that did not cause any impact on the software, i.e. the faults which activation generate the so named *latent errors.*

For these reasons, it is important to discover these failure modes during the development process, and to revise fault-tolerance mechanisms to detect the failures and to reduce their severity. In the next section are shown some experiments related to all the cases described above in order to show the results produced by PyF in each case and to verify the efficiency of this innovative approach of failure mode analysis.

Campaign statistics As written above, two campaigns were performed testing Nova and Cinder subsystem of the target system OpenStack Pike.

In the campaign on Nova, our fault injection system identified 5588 possible fault injection points during the scan phase. Of these faults, only 816 (14,6%) were covered by the workload, while 4472 (85,4%) were skipped (i.e., not executed) after the coverage analysis. The 816 experiments covered by the workload are divided as follows:

- 93 cases of API error only (11,4 %);
- 460 cases assertion failure and API error (56,4 %);
- 52 cases of assertion failure only (6,4 %);
- 211 no assertion failure and no API error (25,8 %).

The Table 3.1 shows a summary of the campaign on Nova subsystem.

In the campaign on Cinder, our fault injection system identified 6641 possible fault injection points. Only 371 (5,6%) were covered by the workload, while 6270 (94,4%) were

Table 3.1: Campaign on Nova subsystem

Case	Number of Experiments	Percentage
Experiments covered by the workload	816	14,6 %
Skipped experiments	4772	85,4 %
	5588	**100 %**

Case	Number of Experiments	Percentage
API error only	93	11,4 %
Assertion Failure and API error	460	56,4 %
Assertion Failure only	52	6,4 %
No Assertion Failure and No API error	211	25,8 %
	816	**100 %**

skipped. The 371 experiments covered by the workload are divided as follows:

- 3 cases of API error only (0,8 %);
- 343 cases assertion failure and API error (92,4 %);
- 1 cases of assertion failure only (0,3 %);
- 24 no assertion failure and no API error (6,5 %).

The Table 3.2 shows a summary of the campaign on Nova subsystem.

Table 3.2: Campaign on Cinder subsystem

Case	Number of Experiments	Percentage
Experiments covered by the workload	371	5,6 %
Skipped experiments	6270	94,4 %
	6641	**100 %**

Case	Number of Experiments	Percentage
API error only	3	0,8 %
Assertion Failure and API error	343	92,4 %
Assertion Failure only	1	0,3 %
No Assertion Failure and No API error	24	6,5 %
	371	**100 %**

3.2 Experimental results

This section shows four experiments with different outcomes related to the points described in the previous section: to maintain an association that is easy to remember, the number that identifies the experiment is consistent with the number of the point described above (e.g. experiment number 3 refers to the case described in the point number 3 of the Section 3.1).

3.2.1 Experiment number 1 - API error only

As a first experiment, we show the results of this approach in the best case, that is when the test workload is interrupted due to an API error without an assertion failure. This case indicates a good mechanism fault-tolerant behavior of the system, since the it returns to the user an API error message in a time reasoning, probably limiting the temporal and spatial propagation of the error.

During this experiment, a fault was injected into component *Cinder*. In particular, the method *show* of to the *VolumeController* class defined in the file */cinder/api/v2/volumes.py* was modified. In the following program listing is shown the original portion of code before the fault injection:

```
1 def show(self, req, id):
2         'Return data about the given volume.'
3         context = req.environ['cinder.context']
4         vol = self.volume_api.get(context, id, viewable_admin_meta=True
              )
5         req.cache_db_volume(vol)
6         utils.add_visible_admin_metadata(vol)
7         return self._view_builder.detail(req, vol)
```

The injected fault type is WRONG_PARAMETERS, i.e. the input parameter of the target method is set to a wrong value. The target method appears in this way after the mutation:

```
1 def show(self, req, id):
2         'Return data about the given volume.'
3         context = req.environ['cinder.context']
4         vol = self.volume_api.get(context, id, viewable_admin_meta=True
              )
5         req.cache_db_volume(pycc_corrupt(vol))
6         utils.add_visible_admin_metadata(vol)
7         return self._view_builder.detail(req, vol)
```

```
1 def pycc_corrupt(target, mode=None):
2     if isinstance(target, int):
3         return (-1)
4     elif isinstance(target, str):
5         return pycc_corrupt_string(target)
6     elif isinstance(target, dict):
7         return pycc_corrupt_dict_key(target)
8     elif isinstance(target, bool):
9         return (not target)
10    else:
11        return None
```

where the function *pyc_corrupt(),* shown in the following listing program, is used to corrupt the input parameters of the target method depending on the type:

The Table 3.3 shows the main information related to the fault injection point:

Table 3.3: Experiment 1 - Fault Injection Point Info

FAULT TYPE	WRONG_PARAMETERS
TARGET COMPONENT	/cinder/api/v2/volumes.py
TARGET CLASS	VolumeController
TARGET FUNCTION DEF	show+60;8
FAULT POINT	req.cache_db_volume(vol)

Once the information on the injected fault has been defined, we can show the results produced by PyF in the form of a bar chart in the Figure 3.2.1. The first subplot represents the events during the execution of the fault injection experiment number 508 (i.e., the experiment related to the fault above discussed), while the second subplot refers to the events of the execution of the fault-free number 7, that is, the fault-free trace most similar to the experiment number 508.

As we can see, the faulty trace (first subplot) does not contain anomalous events (i.e, red colored bars): therefore the injection did not cause different calls from what we can define

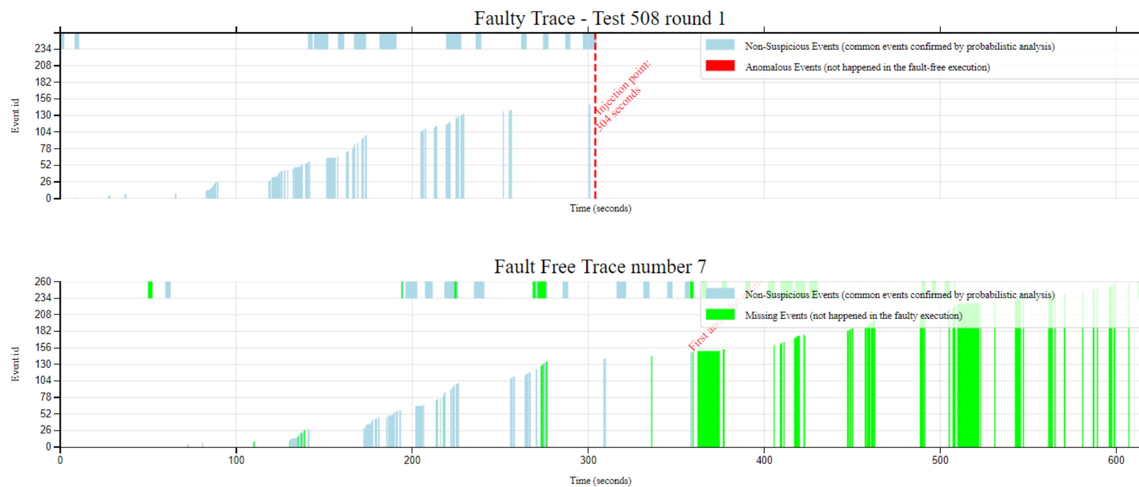


Figure 3.2.1: PyF Visualization - Experiment 1

a normal execution of the workload. However, after to the dashed vertical line indicating the injection activation time, we notice that the execution of the workload stopped immediately. Furthermore, there is a large set of green bars in the fault-free execution: these bars represent missing events in the faulty trace.

Moving the cursor of the mouse on the last event of the faulty trace we get the info shown in Figure 3.2.2. Thus, the last method invoked is *reserve_block_device_name* of *Nova Compute* component, and the Caller Info (i.e. the code, and the related call stack, that makes the service request from another component in the system) refers to the creation of a volume. Immediately after this event, the fault is activated. The interesting thing that we notice is that there is no event following the activation time of the fault: from this we deduce that the workload has stopped immediately due to API error message generated by the system, denoting a rapid response of the system in the become aware of the failure.

In order to better investigate, it is necessary to look at the subplot relative to the execution of the fault-free number 7 (second subplot), that is, the trace that was found to be most similar to the faulty one. It is easy to locate a large number of consecutive green bars from a certain instant of time onwards, which denote a series of missing events in the first subplot caused by the workload interruption of the faulty trace. And it is on these events that we have to focus on to identify the failure.

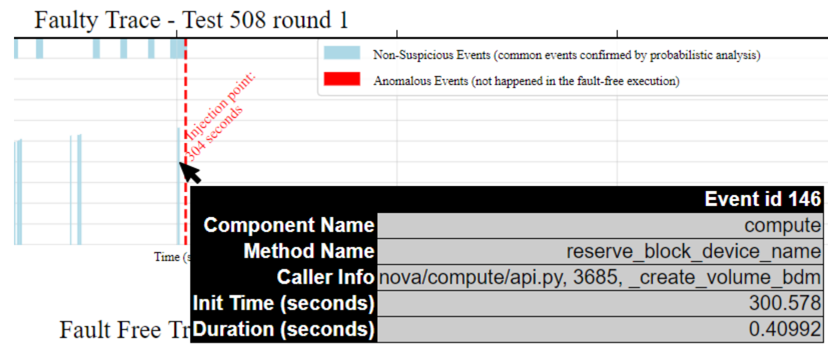


Figure 3.2.2: Experiment 1 - Last faulty trace event

Moving the mouse pointer over one of the first missing events as shown in the Figure 3.2.3, we note that these events correspond to a series of *get_ports* calls made by *q-plugin*. This method returns information regarding the port specified in the request URI. Usually, the methods called by *q-plugin* or more generally by *Neutron*[3] component are events that can be often not aligned after the computation of the LCS due the non-deterministic behavior of the system (and not due the failure). The same reasoning applies to the method *sync_instance_info* invoked by *scheduler.* Therefore, the suggestion that is given to the user is not to focus on these events when they are labeled as suspicious events.

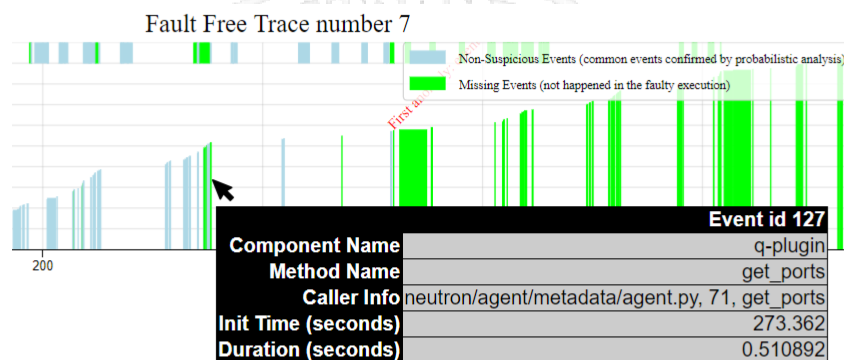


Figure 3.2.3: Experiment 1 - Missing event - 1

For this reason we focus on the following set of missing events, as shown in Figure 3.2.4. This event, as shown in the figure, corresponds to the method *attach_volume* invoked by the component *compute.* Continuing to deepen this area, we analyze the next missing event, corresponding to a large green block corresponding to a long duration call if

[3]Neutron is an OpenStack project to provide "networking as a service" between interface devices managed by other OpenStack services (e.g., *Nova*).

compared to the calls of the others events.

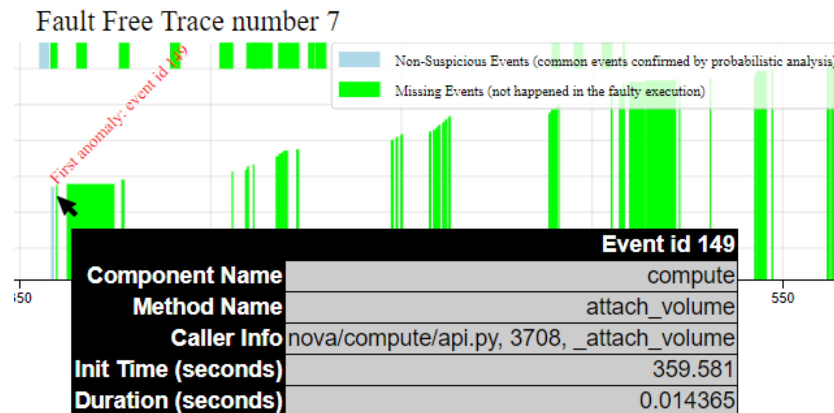


Figure 3.2.4: Experiment 1 - Missing event - 2

This event, as shown in the Figure 3.2.5, corresponds to the method *initialize_connection* invoked by the component *cinder-volume*. From the official API documentation of OpenStack, the method has two input parameters:

- *volume*: The volume to be attached.
- *connector*: Dictionary containing information about what is being connected to.

It allows connection to connector and return a dictionary of connection information. Thus, this call is closely linked to the attach of a volume.

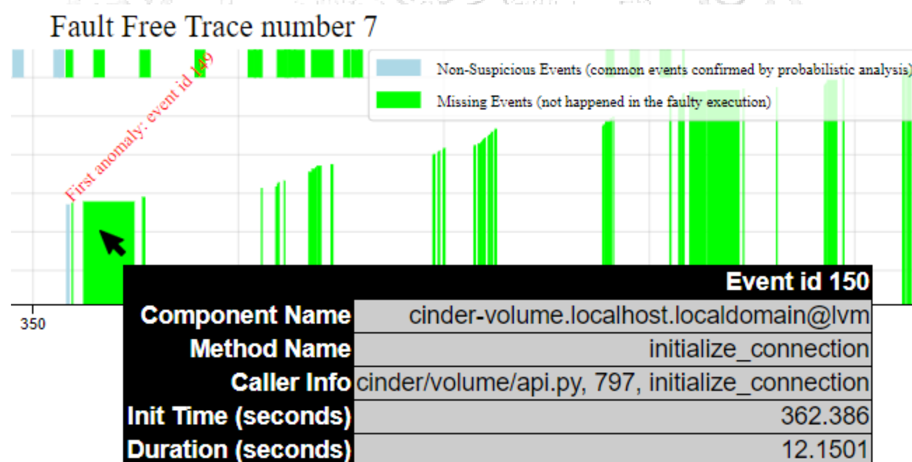


Figure 3.2.5: Experiment 1 - Missing event - 3

Continuing to deepen this area, we analyze the next missing event. As shown in Figure 3.2.6, this event corresponds to the call *attach_volume* always invoked by *cinder_volume.* This is a callback for volume attached to instance or host.

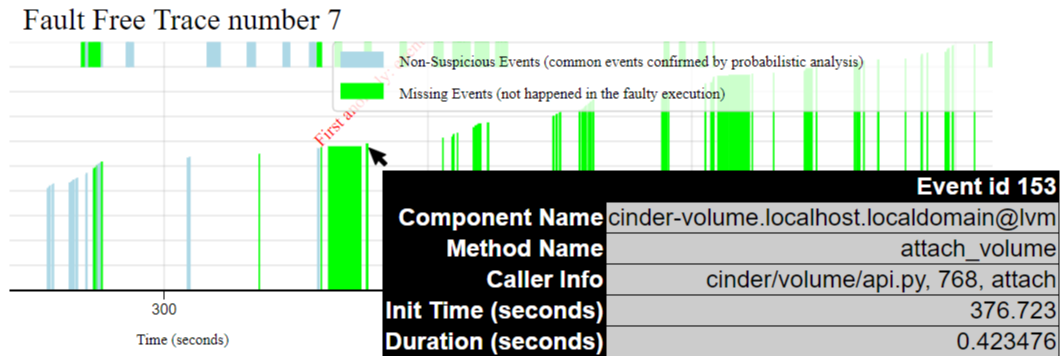


Figure 3.2.6: Experiment 1 - Missing event - 4

Based on what has just been shown, it is possible to state that the execution of the faulty trace was interrupted immediately after the activation of the fault, achieving a *fail-stop* behavior (that is, the system stops as soon as an error occurs, in order to avoid worse consequences). To confirm what we have just written, we can compare the timestamps of the activation of the fault and that of the API error message generated by the system shown in the Table 3.4. The first one is *2018-06-22 04:49:54.418836*, while the second is *2018-06-22 04:49:54.593547*: thus, the system notifies the user of the failure after a time of about 13 hundredths of a second.

The first missing events are related to the attach of a volume: in fact, focusing on the code of the mutated file, it is possible to affirm that the corruption of the volume information returned by the target method makes the attach of the created volume impossible to perform. In fact, the API ERROR message shown in the table confirms the results obtained through PyF.

In this first experiment we can affirm that this approach of failure mode analysis is able to (i) detect the behavior of the target system, (ii) locate the fault and (iii) identify its effects.

Table 3.4: Assertion Result and API error

	Timestamp	Message
Activation Fault	2018-06-22 04:49:54.418836	Injection activated
API Error	2018-06-22 04:49:54.593547	openstack server add volume tempest-INSTANCE_SAMPLE-1529635422 tempest-VOLUME_SAMPLE-1529635422 --device /dev/vdb ; Input ricevuto non valido: The server could not comply with the request since it is either malformed or otherwise incorrect. (HTTP 400) (Request-ID: req-8f6e88cb-6f4e-48af-867d-55c7ae966c94) (HTTP 400) (Request-ID: req-809c3c52-3748-4889-8c5a-9614c03b47fd)

```
1 def _notify_about_instance_usage(self, context, instance, event_suffix,
      network_info=None, system_metadata=None, extra_usage_info=None,
      fault=None):
2         compute_utils.notify_about_instance_usage(self.notifier,
              context, instance, event_suffix, network_info=network_info,
              system_metadata=system_metadata, extra_usage_info=
              extra_usage_info, fault=fault)
```

3.2.2 Experiment number 2 - Assertion Failure and API error

In the second experiment we analyze the case in which the test workload is interrupted due to an API error after that an assertion failure has been generated.

In this case, the API error message indicates that the system has detected failure but that it may have been non-reactive in doing so due to the assertion failure. In fact, if the time between the API error message and the assertion failure is long, then it is possible that spatial and temporal propagation can occur.

During this experiment, a fault was injected into component *NOVA*. The Table 3.5 shows the main information related to the fault injection point.

The target method is used to send a notification about the usage of an instance. In the following program listing is shown the original portion of code before the fault injection: The injected fault type is *MISSING_PARAMETERS*, i.e. the call to the target method

Table 3.5: Experiment 2 - Fault Injection Point Info

FAULT TYPE	MISSING_PARAMETERS
TARGET COMPONENT	/nova/compute/manager.py
TARGET CLASS	ComputeManager
TARGET FUNCTION DEF	_notify_about_instance_usage +1663;8
FAULT POINT	compute_utils.notify_about_instance_usage (self.notifier, context, instance, event_suffix, network_info=network_info, system_metadata=system_metadata, extra_usage_info=extra_usage_info, fault=fault)

```
1 def _notify_about_instance_usage(self, context, instance, event_suffix,
      network_info=None, system_metadata=None, extra_usage_info=None,
      fault=None):
2         compute_utils.notify_about_instance_usage(context, instance,
              event_suffix, network_info=network_info, system_metadata=
              system_metadata, extra_usage_info=extra_usage_info, fault=
              fault)
```

is made without an input parameter. The target method appears in this way after the mutation:

As we can notice, the call to *compute_utils.notify_about_instance_usage* is made without the first input parameter (i.e., *self*).

Once the information on the injected fault has been defined, we can show the results produced by PyF in the form of a bar chart in the Figure 3.2.7

The faulty trace (Test number 206) does not contain anomalous events (i.e, red colored bars): therefore the injection did not cause different calls from what we can define a normal execution of the workload. However, after to the dashed vertical line indicating the injection activation time, we notice that (i) the execution of the workload does not stop immediately and (ii) the presence of missing events between the last event of the faulty execution and the time of the fault activation.

First of all, we focus on those events close to the fault activation: before the the dotted red line the trend of the faulty trace follows that of the fault-free one, not denoting any anomalous event to investigate. The last of these non-suspicious events corresponds to

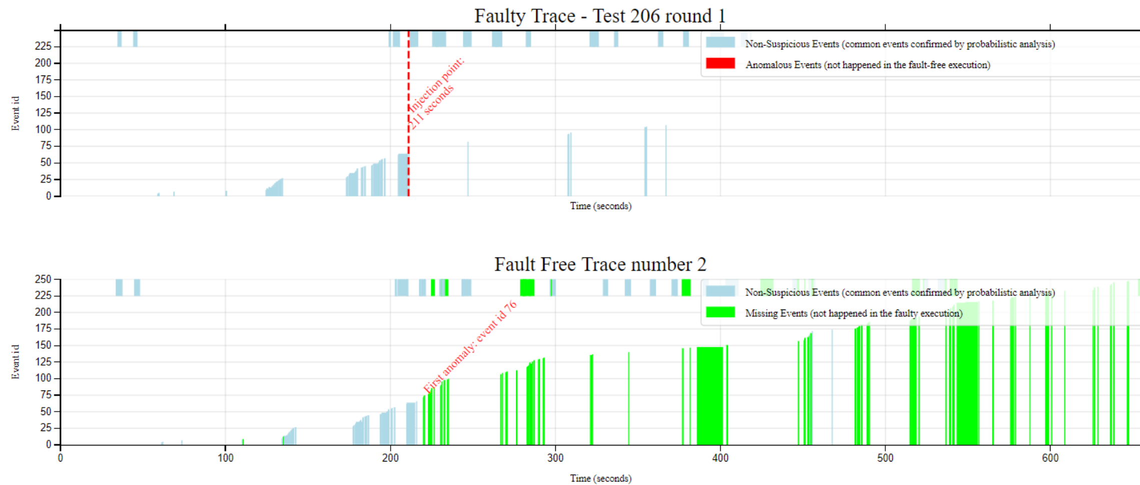


Figure 3.2.7: PyF Visualization - Experiment 2

the method *build_and_run_instance* of *Nova Compute,* as shown in Figure 3.2.8: this suggests that the test workload was correctly executed up to the step 3.

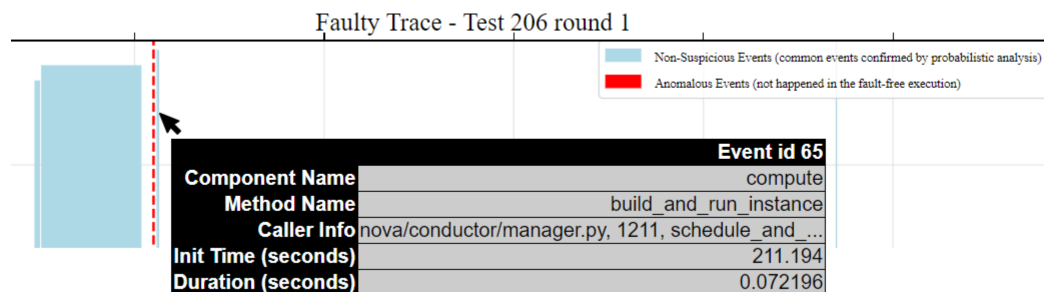


Figure 3.2.8: Experiment 2 - First event in the faulty trace after fault activation

After this event there are a series of white spaces up to the final events of the faulty trace, which correspond to the calls necessary for the creation of the volume, as illustrated in Figure 3.2.9. Only after this moment the system fails by returning an API error message.

By focusing on fault-free trace most similar to the faulty one (i.e., the fault-free trace number 2), there are a succession of missing events that follow the aligned event *build_and_run_instance* and suggest that something went wrong after this event. Near the interruption of the workload, we note that the attach of the volume has not happened by moving the cursor on the missing events as shown in Figure 3.2.10.

Although it is certainly a more difficult case to analyze than the one described above, we can state from the results of PyF that:

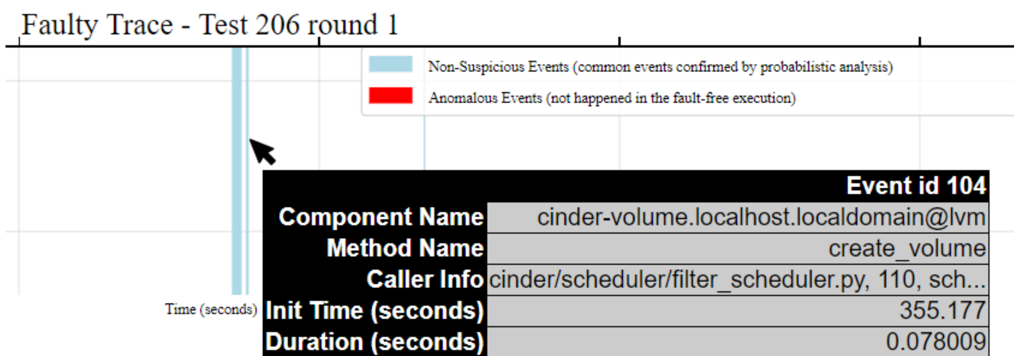


Figure 3.2.9: Experiment 2 - Last event in the faulty trace after fault activation

1. the effects of the fault occurred during the phase of building the instance;
2. the system does not behave in a fail-stop manner, continuing the execution of the workload and also creating a volume;
3. the system failed during the attach of the created volume.

To confirm what we have just written, we can analyze the assertion result generated during the execution of the workload and the API error message generated by the system, as shown in the Table 3.6.

The assertion result message happened about two minutes before the error message generated by the system, and refers to the inactive state of the created instance. The system continued to execute the workload until it attempts to attach the volume to the created instance: it realizes that the instance is in an error state and returns an API error message.

An important consideration to make is that the fault was injected into *Nova Compute* component but the system fails during the attach of the volume, that it is a task related to *Cinder volume* component (spatial propagation). Furthermore, the time difference between the timestamp of the API error message and the timestamp of the fault activation (and of the assertion result) implies a temporal propagation: in fact, the created instance was left in an incorrect state, which then produced a failure during its usage (during the attach of the volume to the instance).

It is also useful to analyze system logs with severity *ERROR,* shown in Table 3.7. System

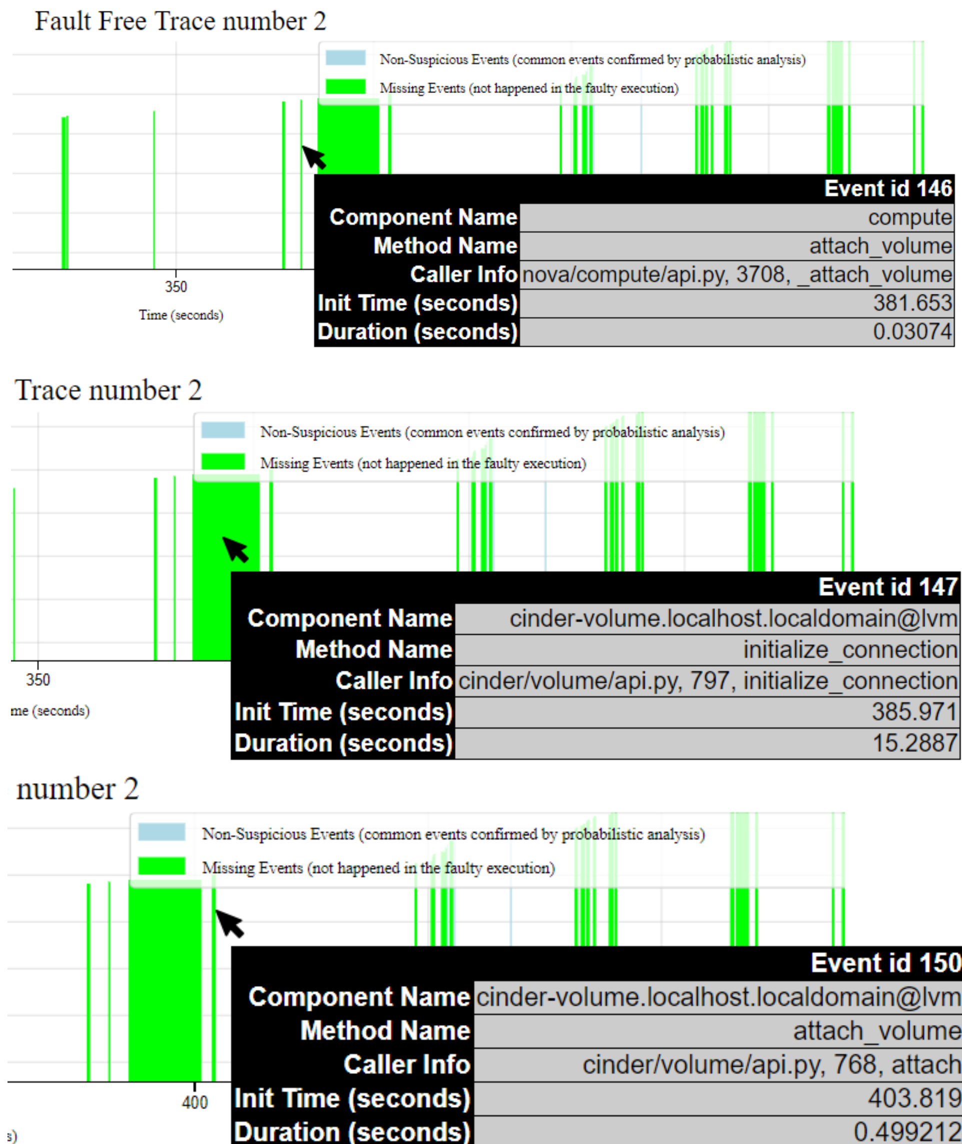


Figure 3.2.10: Experiment 2 - Missing events

logs suggest that the Python interpreter raised a *TypeError* exception (i.e., an operation or function is applied to an object of inappropriate type). The timestamp of the system log matches that of the fault activation, but the system has not interrupted the service and has not notified any anomaly to the user. Thus, it is necessary to propagate a notification of the problem to the other components of OpenStack to avoid inconsistent situations like this.

Therefore, also in this experiment, we can affirm that with PyF we are able to (i) understand the behavior non fail-stop of the target system, (ii) discover the effects of the

Table 3.6: Assertion Result and API error

	Timestamp	Message
Activation Fault	2018-06-26 17:37:04.891402	Injection activated
Assertion Result	2018-06-26 17:38:28.109708	FAILURE_INSTANCE_ACTIVE
API Error	2018-06-26 17:40:31.616653	openstack server add volume tempest-INSTANCE_SAMPLE-1530027170 tempest-VOLUME_SAMPLE-1530027170 --device /dev/vdb ; Impossibile 'attach_volume' l'istanza d1d15b96-62bc-4091-b833-60c795f64ca3 mentre si trova in vm_state error (HTTP 409) (Request-ID: req-1ee44de8-4de5-4bdc-81f9-76d8d7669960)

Table 3.7: Severity Error Log

Timestamp	Severity Log	Component	Message
2018-06-26 17:37:04.891	ERROR	nova.compute.manager	[instance: d1d15b96-62bc-4091-b833-60c795f64ca3] Unexpected build failure, not rescheduling build.: TypeError: notify_about_instance_usage() takes at least 4 arguments (7 given)

injected fault and (iii) identify a spatial and temporal propagation.

3.2.3 Experiment number 3 - Assertion Failure only

During this experiment, a fault was injected into sub-component *compute* of *Nova*. This injection involves a particular situation due to the fact that the execution of the workload did not generate any API error from the system, but there is an assertion failure in the log files. This means that the system has actually failed but that it has not noticed the failure. In fact, demonstrating this, the system did not notify the user of any failure through an API error message. The assertion failure logged is *FAILURE_VOLUME_ATTACHED,*

```
1  @wrap_exception()
2  @wrap_instance_fault
3  def attach_volume(self, context, instance, bdm):
4      'Attach a volume to an instance.'
5      driver_bdm = driver_block_device.convert_volume(bdm)
6      @utils.synchronized(instance.uuid)
7      def do_attach_volume(context, instance, driver_bdm):
8          try:
9              return self._attach_volume(context, instance, driver_bdm)
10         except Exception:
11             with excutils.save_and_reraise_exception():
12                 bdm.destroy()
13     do_attach_volume(context, instance, driver_bdm)
```

and refers to the failure to attach the created volume.

The Table 3.8 shows the main information related to the fault injection point.

Table 3.8: Experiment 3 - Fault Injection Point Info

FAULT TYPE	MISSING_FUNCTION_CALL
TARGET COMPONENT	/nova/compute/manager.py
TARGET CLASS	ComputeManager
TARGET FUNCTION DEF	do_attach_volume+4920;8
FAULT POINT	do_attach_volume(context, instance, driver_bdm)

In particular, the method *do_attach_volume* of the *ComputeManager* class defined in the file *manager.py* was modified. The following program listing shows the original portion of code before the fault injection:

The method *do_attach_volume* is defined in the method *attach_volume* and invoked at the end of the latter. The injected fault type is *missing function call*, i.e. the call to the target method is replaced with a *null* operation, which is obtained in Python with the *pass* statement. The method after the described fault injection is shown below:

The Figure 3.2.11 shows the bar chart produced by PyF.

As we can see, the faulty trace (Test number 1) does not contain anomalous events (i.e, red colored bars): therefore the injection did not cause different calls from what we can define a normal execution of the workload. However, after to the dashed vertical line indicating the injection activation time, we notice some rather large empty spaces that

```
1  @wrap_exception()
2  @wrap_instance_fault
3  def attach_volume(self, context, instance, bdm):
4      'Attach a volume to an instance.'
5      driver_bdm = driver_block_device.convert_volume(bdm)
6
7      @utils.synchronized(instance.uuid)
8      def do_attach_volume(context, instance, driver_bdm):
9          try:
10             return self._attach_volume(context, instance, driver_bdm)
11         except Exception:
12             with excutils.save_and_reraise_exception():
13                 bdm.destroy()
14     pass
```

Figure 3.2.11: PyF Visualization - Experiment 3

indicate the absence of calls. In order to better investigate, it is necessary to look at the subplot relative to the execution of the fault-free number 9, that is, the one that was found to be more similar to the faulty. It is easy to identify in the fault-free execution two sets of events colored in green, which denote a series of missing events in the faulty trace. And it is on these events that we have to focus on to identify the failure.

Moving the mouse pointer over one of the bars of the first set of missing events as shown in the Figure 3.2.12, we note that these events correspond to a series of *get_ports* calls made by *q-plugin,* thus we can not focus on them since they are considered as events identified as suspicious due the non-deterministic behavior of the system.

For this reason we focus on the second set of missing events of the Figure 3.2.11 (the set of

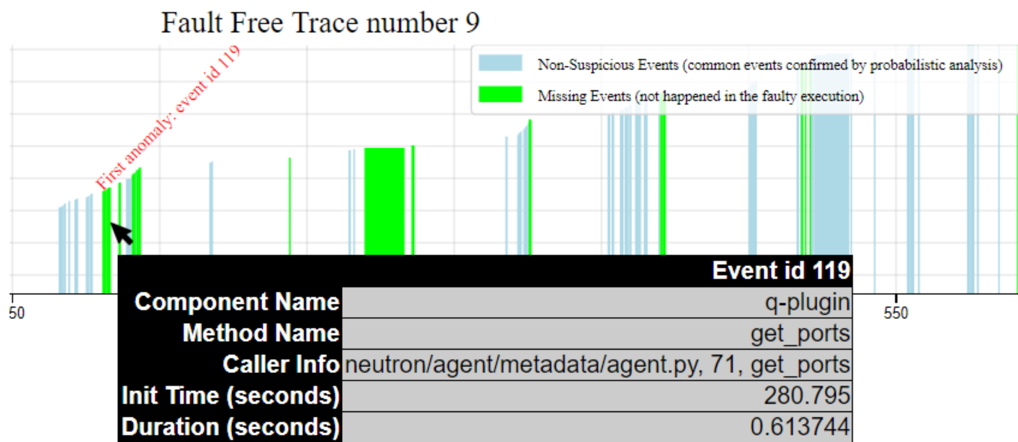


Figure 3.2.12: Experiment 3 - Missing event - 1

consecutive green bars preceding the time instant of 400 seconds in the second subplot), which start with a large green block corresponding to a long duration call if compared to the calls of the others events. This event, as shown in the Figure 3.2.13, corresponds to the method *initialize_connection* invoked by the component *cinder-volume.* From the official API documentation of OpenStack, the method has two input parameters:

- *volume*: The volume to be attached.
- *connector*: Dictionary containing information about what is being connected to.

It allows connection to connector and return a dictionary of connection information. Thus, this call is closely linked to the attach of a volume. Continuing to deepen this area, we analyze the next missing event, as shown in 3.2.14.

This event corresponds to the call *attach_volume* always invoked by *cinder_volume.* This is a callback for volume attached to instance or host. The last events colored in green are always calls made by Neutron / q-plugin for which applies what is written above.

The result of the analysis of the graph makes it clear that the injected fault involves the failure to attach the volume, thus confirming the assertion failure present in the log file. The fact that the workload continued after the injection gives indications that the system has not found any failures, as confirmed by the absence of an API error message. The faulty trace, after the injection and the consequent missing events related to the attach of

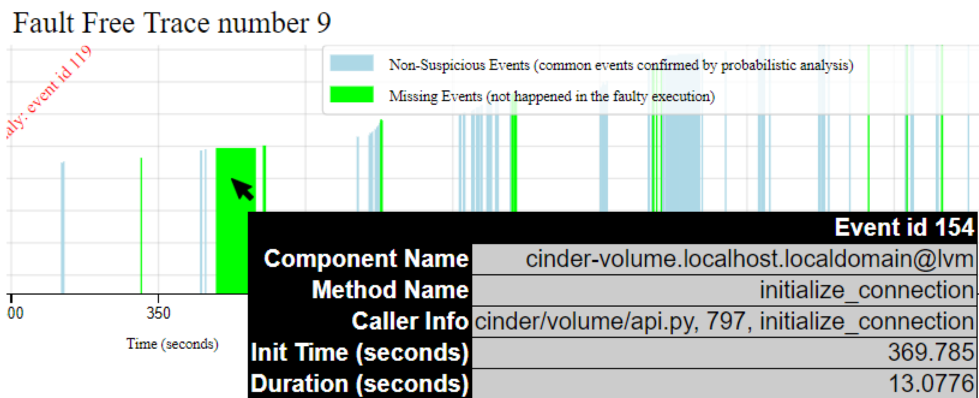


Figure 3.2.13: Experiment 3 - Missing event- 3

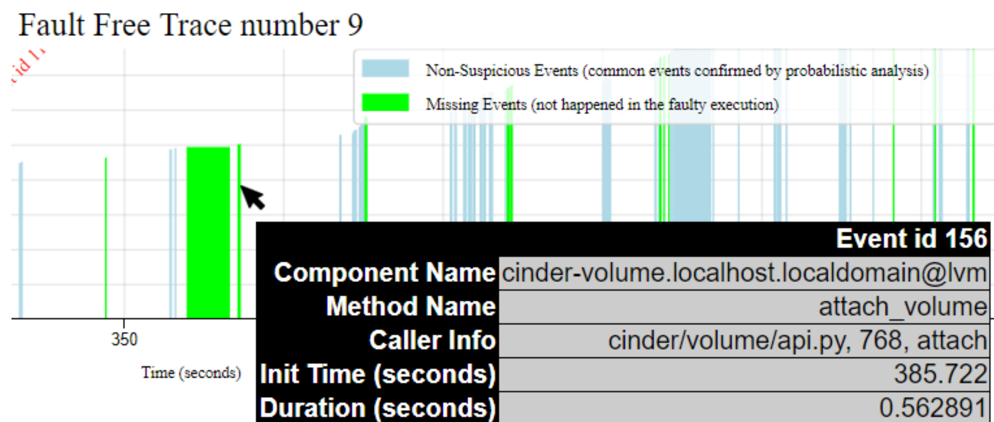


Figure 3.2.14: Experiment 3 - Missing event - 3

the volume, does not present any other suspicious events since the subsequent events are aligned with the fault-free trace.

In this experiment it is not possible to make considerations from the point of view of temporal propagation of the fault since the system has not noticed the failure but, precisely for this reason, it is possible to make even more important considerations, that is, to provide suggestions on how to improve the fault mechanisms tolerance of the target system. By analyzing the injection point in the program listing it is possible to state that the system does not log any error because there is not a “redundant check” on the operation performed (the attach of the volume). So what could be done to improve the fault tolerance mechanism is to insert this check in the API method *attach_volume* (which calls *do_attach_volume* method) in order to improve the reliability with respect to effects

```
1 def detail(self, req, resp_obj):
2         context = req.environ['cinder.context']
3         if authorize(context):
4             for vol in list(resp_obj.obj['volumes']):
5                 self._add_volume_tenant_attribute(req, vol)
```

caused by the injected fault.

A further consideration we can make is that of the spatial propagation of the fault. The injected component is *Nova compute*, but the error occurred in *Cinder volume*. In fact, *Nova compute* during the attach of the volume calls some volume API which is the way *Nova* interacts with cinder volumes.

It is also important to underline that in the system logs there is no message with severity error.

3.2.4 Experiment number 4 - No Assertion Failure and No API error

The last case we show is probably the least interesting from the point of view of failure mode analysis, but it is still analyzed specifically to illustrate the results of PyF when there are no failures during the execution of the workload.

During this experiment, a fault was injected into component *Cinder*. The Table 3.9 shows the main information related to the fault injection point:

Table 3.9: Experiment 4 - Fault Injection Point Info

FAULT TYPE	MISSING_FUNCTION_CALL
TARGET COMPONENT	/cinder/api/contrib/volume_tenant_attribute.py
TARGET CLASS	VolumeTenantAttributeController
TARGET FUNCTION DEF	detail+41;16
FAULT POINT	self._add_volume_tenant_attribute(req, vol)

In the following program listing is shown the original portion of code before the fault injection:

The injected fault type is *missing function call*, i.e. the call to the target method is replaced with the *pass* statement. The method after the described fault injection is shown below:

```
1 def detail(self, req, resp_obj):
2         context = req.environ['cinder.context']
3         if authorize(context):
4             for vol in list(resp_obj.obj['volumes']):
5                 pass
```

The Figure 3.2.15 shows the graph produced by PyF.

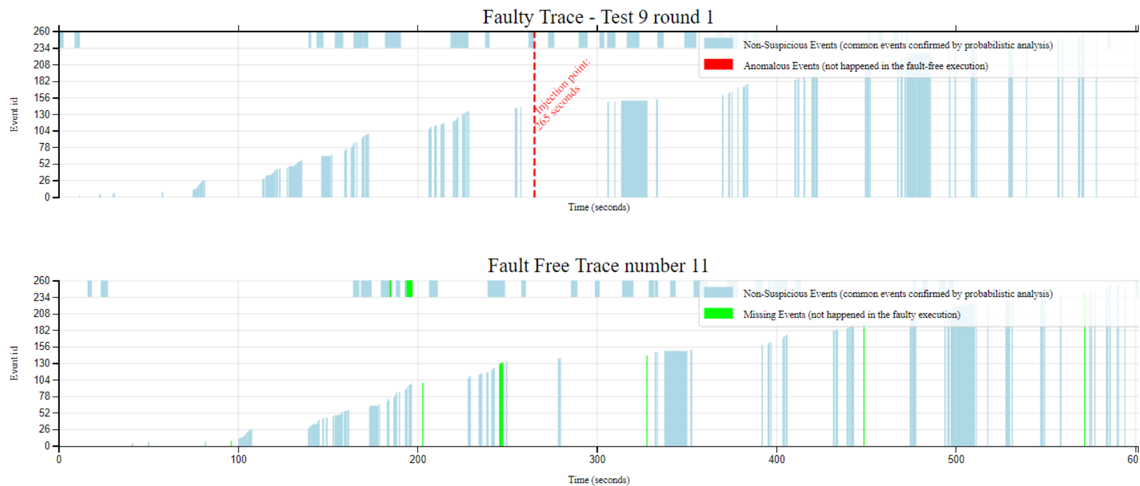


Figure 3.2.15: PyF Visualization - Experiment 4

As we can easily note, the faulty trace (Test number 9) and the fault-free trace are very similar. There are no anomalous events in the faulty one, and there are very few missing events highlighted in the second subplot. All this suggests that in the observation time there are no obvious effects due to the injection.

Moving the mouse pointer over one of the bars of the few missing events as shown in the Figure 3.2.16, we note that these events correspond to a series of *get_ports* calls made by *q-plugin*, for which the same considerations made above are valid.

For completeness, in this last case we show in the Figure 3.2.17 also the graph in debug mode. The dark blue events in the first subplot are those events not aligned during the computation of the LCS, but with a high value of likelihood to be in that position of the trace. So these events are not confirmed to be suspicious after the PPM-C usage. The application of LCS with PPM-C would have cataloged these events as anomalous, representing them as red vertical bars. Analyzing these events, we see that they correspond to

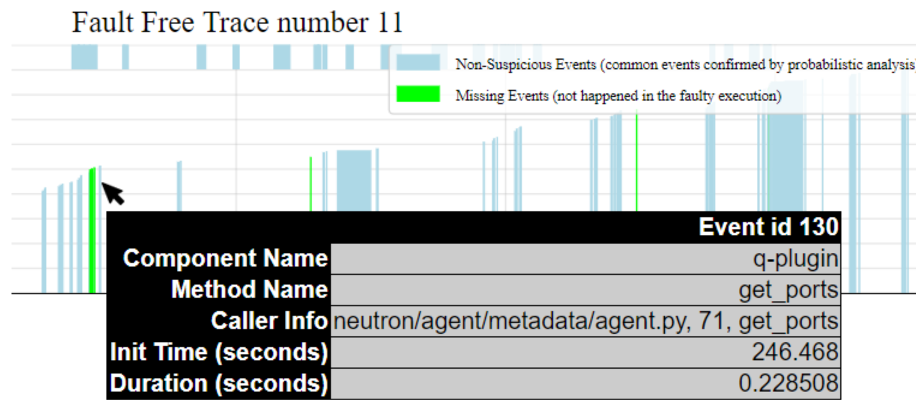


Figure 3.2.16: Experiment 4 - Missing Event

the above discussed *get_ports* calls made by *q-plugin* or to *sync_instace_info* calls made by the *scheduler*, i.e. to events that are not always possible to align after the computation of the LCS due the above discussed not-deterministic nature of the system. Therefore it is necessary to apply a probabilistic model in order to limit false positives.

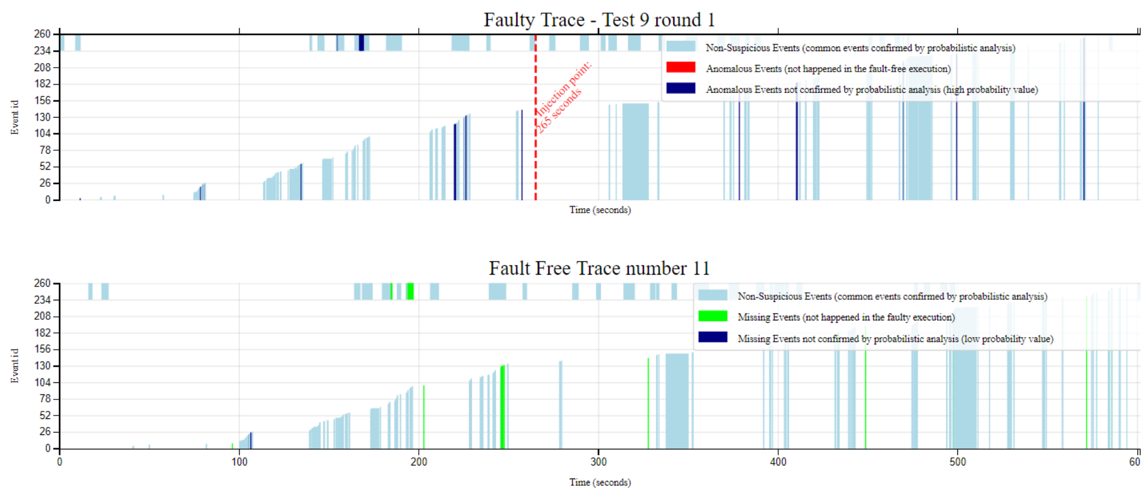


Figure 3.2.17: PyF Visualization - Experiment 4 - Debug mode

From the results obtained by PyF, the conclusion we can deduce is simply that the injected fault did not affect the system during the observation time. But to confirm this, it is necessary to analyze the workload logs and those of the system. The log relative to the workload does not contain an assertion result or an API error message, while the logs of the target system do not contain messages with severity error, thus confirming the results of the approach.

3.3 Performance evaluation of the approach

In this section the accuracy and the computational time of the approach are evaluated in order to do some important considerations on the performances of PyF.

3.3.1 Accuracy

A good metric to evaluate the performance of the approach in terms of accuracy of the results is to identify the cases of *false positives* and *false negatives*. False positives are the non-suspicious events labeled as suspicious by the approach. To find these events, we built a pool of N fault-free traces and we compared each of them with others M fault-free traces using PyF, to vary by the number N. In order to make the detection of false positives even more precise, we limit the analysis to the RPC messages, excluding the REST APIs, since the latter are client-side request and therefore are not subject to the non-deterministic nature of the system. The Table 3.10 shows the false positives obtained and their occurrences when $N = 10$. As easily hypothesized, the false positives with the greatest number of occurrences are related to the events of *q-plugin_get_ports* and *scheduler_sync_instance_info*, already discussed above, due the non-deterministic nature of the target system.

The Table 3.11 shows the distribution of the false positives in each comparison. From the data of the table, the average of false positives in each comparison made through the PyF approach is 12.9, while the median is 12. Out of a total of 1960 events, PyF reported in its analysis 129 false positives, corresponding to 6.6% of the total events.

To evaluate the false negatives, i.e. those suspicious events that are labeled as non-suspicious by the approach, we can remove from the analysis the events shown in the Table 3.10 similarly to how we do for the noisy events. If the approach continues to highlight suspicious events, then it has identified a true anomaly. If, however, the approach does not detect any suspicious events, then it is a case of false negative. The Figure 3.3.1 shows a summary of the steps performed in order to find false negatives and, thus, to

Table 3.10: False positives

TargetName_InvokedMethod	Occurrences
q-plugin_get_ports	25
scheduler_sync_instance_info	25
compute_external_instance_event	18
neutron-vo-Subnet-1.0_push	9
q-plugin_dhcp_ready_on_ports	9
q-plugin_update_device_list	6
dhcp_agent_network_delete_end	5
neutron-vo-Port-1.1_push	5
q-l3-plugin_sync_routers	5
q-plugin_get_active_networks_info	5
neutron-vo-Network-1.0_push	4
q-l3-plugin_update_floatingip_statuses	4
l3_agent_routers_updated	3
q-agent-notifier-security_group-update_security_groups_member_updated	2
q-plugin_release_dhcp_port	2
q-agent-notifier-port-delete_port_delete	1
q-agent-notifier-security_group-update_security_groups_provider_updated	1

evaluate the accuracy of the approach.

Figure 3.3.1: Steps to identify False Negatives

The false negatives detected in the campaigns on Nova and Cinder subsystem are shown in Table 3.12 and in the Table 3.13, respectively. It is evident that the approach is very accurate in the cases where the system notify a failure through the generation of an API error message: in fact, both in the Nova campaign and in the Cinder campaign, there are zero cases of false negatives. The approach becomes less accurate when the system does not notify a failure, that is in the case presented in point 3 of the Section 3.1. In these cases, a false positive is reported in 28 experiments on 52 in the campaign on Nova subsystem, while no false positive is detected in the only experiment belonging to this case in the Cinder campaign.

The Table 3.14 shows a summary of the statistics on the campaigns. Out of a total of 952

Table 3.11: Distribution of the false positives

Comparison	False Positives	Events	Percentage %
Number 1	12	185	6,5 %
Number 2	11	194	5,7 %
Number 3	17	194	8,8 %
Number 4	9	195	4,6 %
Number 5	23	192	12 %
Number 6	15	212	7 %
Number 7	13	201	6,5 %
Number 8	12	189	6,3 %
Number 9	8	203	3,9 %
Number 10	9	195	4,6 %
	129	**1960**	**6,6 %**

Table 3.12: Campaign Nova - False Negatives

Case	Number of Experiments	False Negatives	Percentage
API error only	93	0	0 %
Assertion Failure and API error	460	0	0 %
Assertion Failure only	52	28	53,8 %
	605	**28**	**4,6 %**

experiments with failure, only in 28 cases (2,9%) PyF reported a false negative as result. These cases are all belonging to the worst case, i.e. when the failure is unnoticed by the system, which is the most complicated case to analyze (52,8 % of false negatives in the case *Assertion Failure only*).

An interesting analysis is to understand how accuracy increases as the number of learning traces (i.e, the fault-free traces) increases. The Table 3.15 shows the number of false negatives as the number of fault-free traces grows. We note that the number of false positives is about half when the number of learning traces increases from 1 to 5, after which the improvement of the accuracy is attenuated. Also in this case, the false negatives are all belonging to the case *Assertion Failure only* (53 total cases). The Figure 3.3.2 shows how the number of false negatives decreases as the number of learning traces grows, resulting in an improvement in accuracy.

Table 3.13: Campaign Cinder - False Negatives

Case	Number of Experiments	False Negatives	Percentage
API error only	3	0	0 %
Assertion Failure and API error	343	0	0 %
Assertion Failure only	1	0	0 %
	347	0	**0 %**

Table 3.14: False Negatives Statistic

Case	Number of Experiments	False Negatives	Percentage
API error only	96	0	0 %
Assertion Failure and API error	803	0	0 %
Assertion Failure only	53	28	52,8 %
	952	28	**2,9 %**

3.3.2 Computational Time

As a further analysis of the performance of PyF, it is possible to evaluate the scalability, measuring how the computational time to provide the results of the analysis increases as the number of fault-free or the number of events of each fault-free trace increase. An important parameter that affects the timing of the analysis is the size of the context used during the probabilistic analysis. A high context size results in greater accuracy, but it implies an increase of the time spent on analysis. We have chosen to maintain a high context size (i.e., equal to 50), favoring the accuracy of the results.

The Table 3.16 shows the time taken to produce results as the number of fault-free traces increases.

A further analysis to do is evaluate how the computational time grows as the number of events in each trace increases. To do this type of considerations we used a pool of 10 fault-free traces and we re-modeled the traces at each PyF run. We performed 6 measurements, in which the traces are divided by 4, halved, unmodified, doubled, quadrupled and multiplied by 8, respectively. It is important to note that dividing a trace by four (or halving it) does not necessarily mean having a quarter (or the half) of the number of events of the original trace, because the phase of noise removal is performed successively. Obviously,

Table 3.15: Improvement of the Accuracy

Number of Fault-Free Traces	Number of Experiments	False Negatives	Percentage
1	952	50	5,35 %
5	952	26	2,73 %
10	952	25	2,63 %
15	952	24	2,52 %
20	952	24	2,52 %

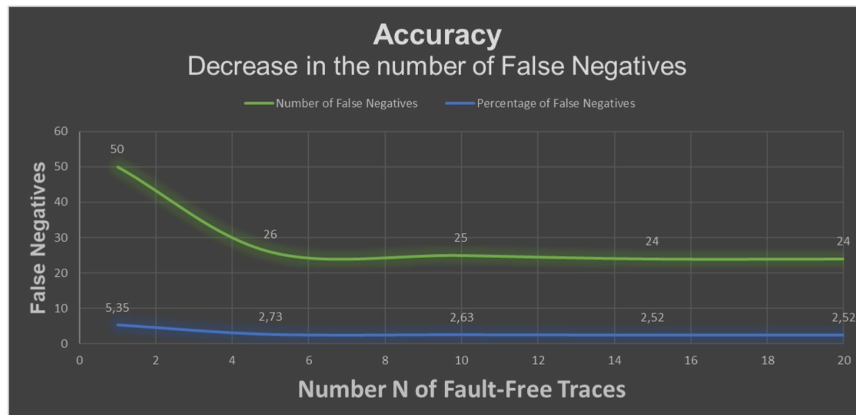


Figure 3.3.2: Improvement of the Accuracy

the same does not apply to the original trace doubled, quadrupled and multiplied by eight because, in these cases, the increase in the trace size is obtained by concatenating n time (with n equals to 1, 2 and 4 respectively) the original trace to itself. The Table 3.17 shows the collected measurements.

The Figure 3.3.3 and the Figure 3.3.4 show the graphs of the computational times of PyF when the number of traces increases and when the size of traces (i.e., the number of events in each trace) grows, respectively. In the first case, the running time is *linearithmic time* (i.e., time complexity is $O(n \log n)$), therefore the approach it is not very scalable when the number of fault-free traces increase, due the number of applications of LCS (in order to find the most similar fault-free trace) that increases with the number of fault-free traces. However, this is not a key factor because the number of fault-free traces is relatively small if compared to the number of fault injection experiments. The important result we deduce from the data of the Table 3.16 is that, for a high number of learning traces (in fact 20 traces are more than enough to have good accuracy), the computational times are very

Table 3.16: Computational Time as the number of traces increases

Number of Fault-Free Traces	Computational Time (seconds)
1	8,2
2	12,7
3	19,7
4	28,1
5	37,6
10	107,7
15	212,9
20	336,8

Table 3.17: Computational Time as the number of traces increases

Dimension of the Fault-Free Traces	Average number of events in the Fault-Free Traces	Computational Time (seconds)
Divided by 4	53	14
Divided by 2	90	22,8
No Operation (Original Traces)	196	58,9
Multiplied by 2	393	150,6
Multiplied by 4	786	281,6
Multiplied by 8	1573	523,85

limited (336,8 seconds) allowing us to state that the approach is fast in producing the results. The Figure 3.3.4 shows that, as the number of events increases, the computational time of the approach grows linearly, so the time complexity is $O(n)$. Therefore, we can affirm that PyF is also scalable when the dimension of traces increases.

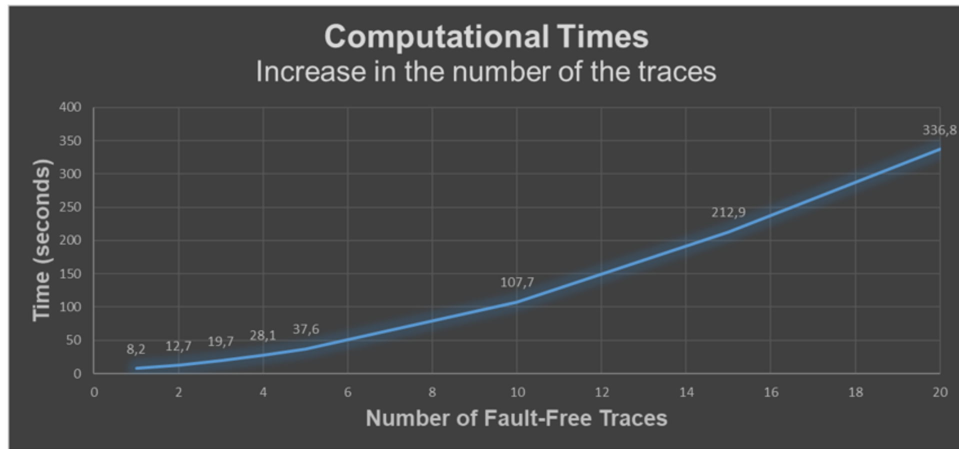


Figure 3.3.3: Graph of the computational times when the number of traces increases

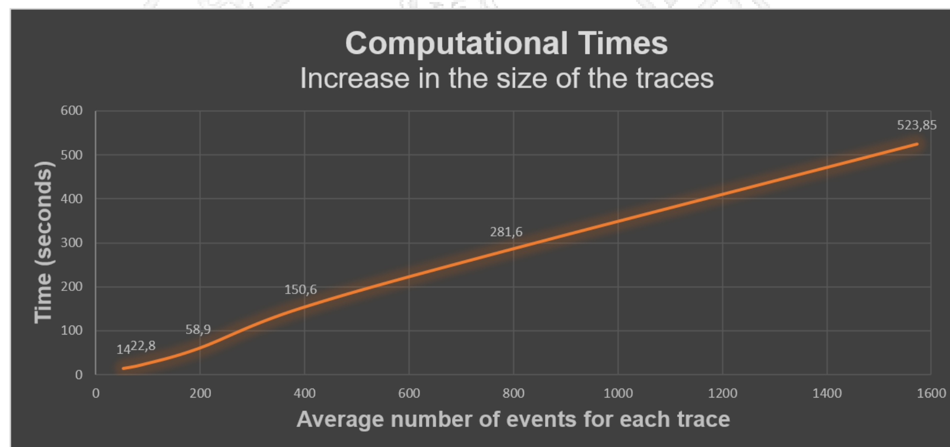


Figure 3.3.4: Graph of the computational times when the size of the traces increases

Conclusion

Like any other distributed system, cloud management stacks are susceptible to faults whose root cause is often hard to diagnose. Therefore, it is fundamental to have a technique of fault injection accurate in its diagnosis and in identifying of the fault. PyF is a portable, accurate, fast and scalable approach to failure mode analysis.

The portability is guaranteed through the use both of a powerful and flexible data collection tool such as Zipkin and a data analysis tool that does not rely on any particular system. PyF guarantees also a high level of non-intrusiveness, through the addition of very few changes only in the data collection phase.

The accuracy of the approach depends very much on the goodness of the training model used for probabilistic analysis. With a sufficiently high number of fault-free traces (e.g., more than 10), it is possible to build an good training model able to detect the false positives that are inevitably obtained with the calculation of the Longest Common Subsequence. It is not possible to limit to the use of the only LCS due the non-deterministic nature of the complex distributed systems. It would also have been possible to use only VMMs, excluding the analysis of LCS. However, this would mean applying a probabilistic model to hundreds of events, increasing the risk of both getting false positives (i.e., non-suspicious events labeled as suspicious) and false negatives (i.e., suspicious events labeled as non-suspicious). Moreover, applying only VVMs would require a very high number of fault-free traces in order to obtain a model capable of analyzing a sequence composed of a large number of events. However, this involves a substantial increase in time both in the data collection phase and in the usage phase of PPM-C, therefore not

guaranteeing the practicability, an important merit of SFI. Thus, the application of LCS is fundamental because it allows limiting probabilistic analysis to the only differences highlighted by the calculation of the longest common subsequence.

It has been shown that the proposed approach is very accurate in all the cases of the analyzed experiments. In fact, it identifies whether the system clearly notify the presence of the failure, by assuming a fail stop behavior, or if it notifies a failure not immediately, detecting the spatial and temporal propagation. The approach woks well also when the failure is unnoticed. It was also able to provide indications on the location of the fault and its effects, identifying the behavior assumed by the target system and providing, where applicable, indications on the spatial and temporal propagation of the errors. This is also very useful for providing guidance to system developers on how to improve the fault tolerance mechanisms of the system. Furthermore, PyF can be used to identify the faults that did not cause any impact on the software, i.e. the faults which activation generate the so named *latent errors.* The accuracy of the approach decreases in the most complicated case for the analysis, i.e. when the system does not notify a failure to the user, providing a significant percentage of false negatives, but it's possible to limit this problematic increasing the number of fault-free traces. Furthermore PyF is fast in producing results and it is scalable, in fact when the number of events increases, the computational time grows linearity.

The weak point of PyF is related to the failure to detect the false positives in the most similar fault-free trace. An event happened both in the faulty trace and in the fault-free is marked as difference from the LCS (therefore as an anomalous event in the faulty trace and event missing in the fault-free) when it is not possible to align it, or more generally when it is not part of the path for the construction of the longest common subsequence. If the anomalous event of the faulty trace is corrected by probabilistic analysis (i.e., high likelihood value), it is marked as non-suspicious events, thus detecting the false positive. But what does not happen is the non-correction of the event in the fault-free trace because there is no mechanism able to provide a perfect correspondence between an event of the

faulty trace with one of the fault-free trace (there is no temporal link between events of two different traces and the information on the method, on the target and on the caller do not provide a unique correspondence). Therefore, in the fault-free trace, the event continues to be a missing event, increasing the number of the false positives and. therefore, decreasing the accuracy of the result.

Regarding the future developments, a possible improvement could be to separate the analysis of REST requests from the RPC messages. In fact, since RPC messages are often interspersed with REST calls (depending on the running workload), it might be useful to train the model considering only the RPC messages. In this way, it would be easier for PPM-C to detect an anomalous system call, providing more precise probability values. Regarding REST calls, it may be useful not to apply the PyF algorithm but simply to identify those requests that return a HTTP error status code, representing them with a different color bar in the report.

As previously described, there is no mechanism capable of associating an event of the faulty trace with one of the fault-free trace. Thus, when a suspicious event present both in the faulty trace (as an anomalous event) and in the fault-free trace (as missing event) is corrected by probabilistic analysis in one of the two traces, then it is not corrected in the other trace. This problem involves a situation of inconsistency. One possible solution is to use a heuristic approach, based on identifying two events that have the same *Target name*, the same *Method name*, the same *Caller Info*, a similar duration (i.e., the temporal duration of the two events differ by a value below a given threshold) and having an initial time that is in the same temporal range. So, for example, when probabilistic analysis corrects an event happened in the faulty trace, a check is made through the illustrated mechanism to understand if the same event (labeled as missing event) exists in the fault-free trace and, if it is found , corrected in non-suspicious event.

Appendix A

Zipkin

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data.

Applications are instrumented to report timing data to Zipkin. The Zipkin UI also presents a Dependency diagram showing how many traced requests went through each application. In order to troubleshoot latency problems or errors, it's possible to filter or sort all traces based on the application, length of trace, annotation, or timestamp. Once a trace has been selected, it's possible to see the percentage of the total trace time each span takes which allows to identify the problem application.

Tracers live in our applications and record timing and metadata about operations that took place. They often instrument libraries, so that their use is transparent to users. For example, an instrumented web server records when it received a request and when it sent a response. The trace data collected is called a *Span*.

Instrumentation is written to be safe in production and have little overhead. For this reason, they only propagate IDs in-band, to tell the receiver there's a trace in progress. Completed spans are reported to Zipkin out-of-band, similar to how applications report metrics asynchronously.

For example, when an operation is being traced and it needs to make an outgoing http

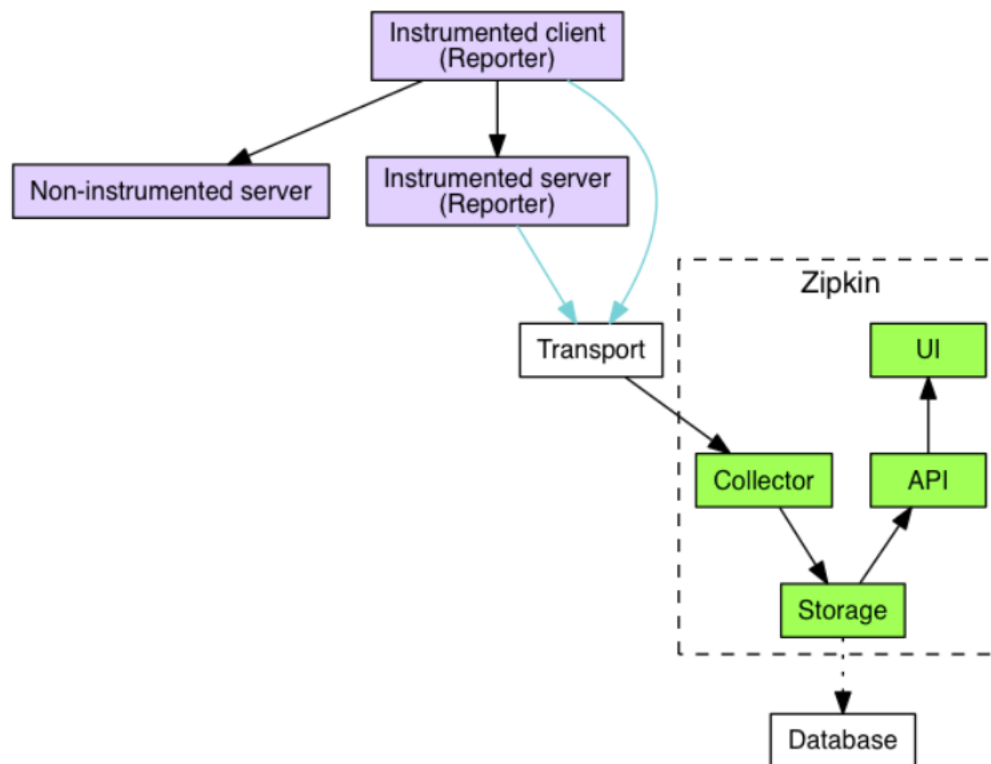


Figure A.0.1: Zipkin flow diagram

request, a few headers are added to propagate IDs. Headers are not used to send details such as the operation name.

The component in an instrumented app that sends data to Zipkin is called a *Reporter*. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. Later, storage is queried by the API to provide data to the UI. The Figure A.0.1 describes this flow:

As already mentioned, identifiers are sent in-band and details are sent out-of-band to Zipkin. In both cases, trace instrumentation is responsible for creating valid traces and rendering them properly. For example, a tracer ensures parity between the data it sends in-band (downstream) and out-of-band (async to Zipkin).

The Figure A.0.2 shows an example sequence of http tracing where user code calls the resource /foo. This results in a single span, sent asynchronously to Zipkin after user code receives the http response.

Trace instrumentation report spans asynchronously to prevent delays or failures relating

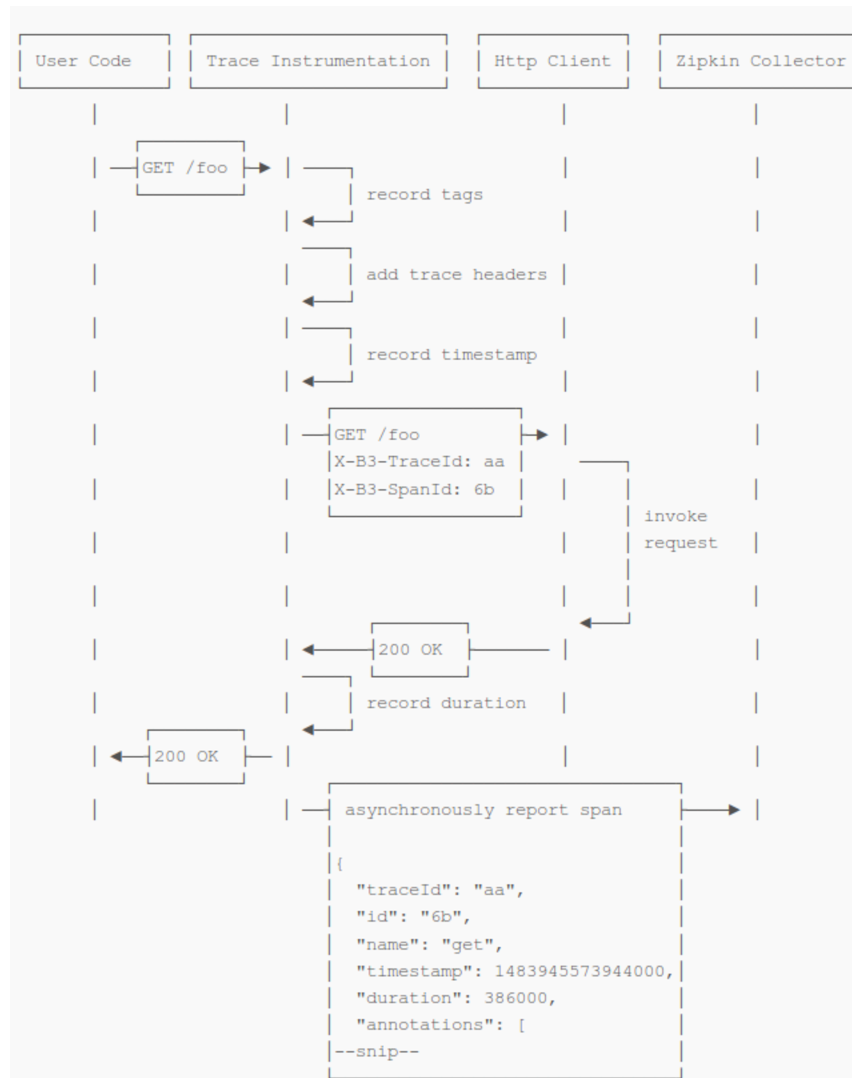


Figure A.0.2: Example sequence of http tracing

to the tracing system from delaying or breaking user code.

Spans sent by the instrumented library must be transported from the services being traced to Zipkin collectors. There are three primary transports: HTTP, Kafka and Scribe.

There are 4 components that make up Zipkin:

- *collector*
- *storage*
- *search*
- *web UI*

Zipkin Collector Once the trace data arrives at the Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

Storage Zipkin was initially built to store data on Cassandra since Cassandra is scalable, has a flexible schema, and is heavily used within Twitter. However, this component has been made pluggable. In addition to Cassandra, there is a natively support to ElasticSearch and MySQL. Other back-ends might be offered as third party extensions.

Zipkin Query Service Once the data is stored and indexed, it's needed a way to extract it. The query daemon provides a simple JSON API for finding and retrieving traces. The primary consumer of this API is the Web UI.

Web UI The GUI presents a nice interface for viewing traces. The web UI provides a method for viewing traces based on service, time, and annotations. Note: there is no built-in authentication in the UI.

Bibliography

[1] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[2] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

[3] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48. IEEE, 2000.

[4] Suratna Budalakoti, Ashok N Srivastava, and Matthew E Otey. Anomaly detection and diagnosis algorithms for discrete symbol sequences with applications to airline safety. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 39(1):101–113, 2009.

[5] Jörgen Christmansson and Ram Chillarege. Generation of an error set that emulates software faults based on field data. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 304–313. IEEE, 1996.

[6] Domenico Cotroneo and Henrique Madeira. Introduction to software fault injection. In *Innovative Technologies for Dependable OTS-Based Critical Systems*, pages 1–15. Springer, 2013.

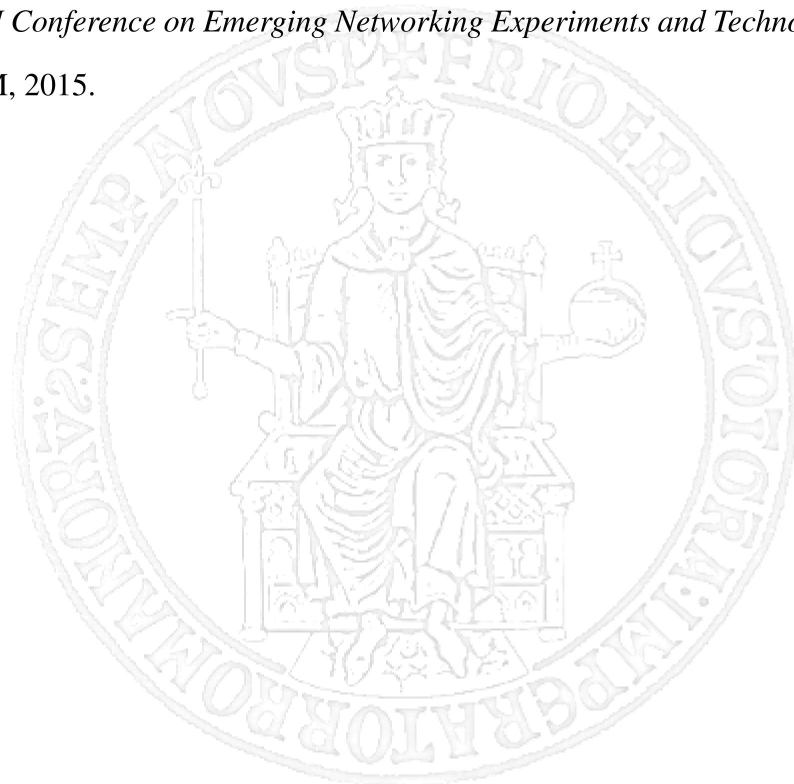
[7] Luigi De Simone. Dependability benchmarking of network function virtualization. 2017.

[8] Ayush Goel, Sukrit Kalra, and Mohan Dhawan. Gretel: Lightweight fault localization for openstack. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 413–426. ACM, 2016.

[9] Seungjae Han, K. G. Shin, and H. A. Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213, Apr 1995. doi: 10.1109/IPDS.1995.395831.

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

[11] Dhruv Sharma, Rishabh Poddar, Kshiteej Mahajan, Mohan Dhawan, and Vijay Mann. H ansel: diagnosing faults in openstack. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 23. ACM, 2015.



Ringraziamenti

È stato un percorso lungo e duro, durante il quale ho dovuto fare i conti con le difficoltà legate al mondo accademico ma soprattutto con gli ostacoli che la vita ci pone. L'impegno e la forza di volontà non sono mai mancati, ma è grazie all'apporto di tantissime persone, che ho avuto la fortuna di conoscere, che sono riuscito ad andare avanti.

Ringrazio anzitutto il Professore Domenico Cotroneo per i suoi numerosi consigli e suggerimenti durante questo periodo di formazione e per avermi dato l'opportunità di vivere una bellissima esperienza, stando a stretto contatto con un gruppo di persone fantastiche e preparatissime. Un ringraziamento particolare al Professore Roberto Natella, per me un punto di riferimento da quando ero un giovane studente. Essere formato da una persona come lui è per me un motivo di orgoglio e vanto. Ringrazio l'Ingegnere Luigi De Simone che mi ha sempre seguito, non negandomi mai un aiuto in qualsiasi giorno ed orario. Mi ha insegnato tantissime cose, e ricorderò con piacere la fiducia che ha sempre riposto in me oltre al bel rapporto che si è venuto a creare.

Un ringraziamento speciale i miei due amici e colleghi Alfonso e Salvatore. Abbiamo affrontato un percorso impegnativo assieme, tra mille difficoltà che abbiamo superato sempre brillantemente. Reputo il nostro gruppo completo ma soprattutto unito, cosa che ci ha permesso sia di raggiungere i nostri obiettivi e sia di migliorarci a vicenda. Le cose che ho imparato stando a stretto contatto con loro due sono state più importanti e formative di qualsiasi nozione appresa durante i nostri esami. Spero che il nostro rapporto resti vivo anche nei giorni a venire.

Un immenso grazie ai miei genitori. Gli innumerevoli sacrifici fatti per garantirmi serenità

e stabilità in un percorso di studio lungo e difficile sono il motivo principale di questo felice esito. Riuscire a ripagare i vostri innumerevoli sforzi sarebbe per me la più grande soddisfazione. Spero di rendervi fieri di me. Da oggi sono un ingegnere grazie a voi.

Non ci sono parole per poter esprimere il mio ringraziamento a mia sorella Luana, per i suoi continui e costanti pensieri verso di me. Il voler ricambiare in qualche modo i suoi gesti spontanei sono stati per me una motivazione costante ad andare avanti senza mai fermarmi.

Un sincero grazie a Chiara, l'ultima arrivata nella mia vita. Abbiamo condiviso solo l'ultima parte di questo mio percorso di vita, ma il tuo sostegno e la tua considerazione nei miei confronti mi ha dato la giusta forza e determinazione per non scoraggiarmi nei momenti di difficoltà.

Ringrazio mio cugino Fabio, per me un fratello. I tuoi consigli sono stati fondamentali per superare periodi bui della mia vita. Grazie.

Grazie anche ad Oliver, anche se non leggerà mai queste parole, la sua compagnia e le emozioni che mi ha regalato sono state fondamentali nella mia vita, e spero lo saranno per molto altro tempo ancora.

Ringrazio tutti i miei amici, una componente fondamentale della mia vita. Non posso non iniziare da Nello, un amico che mi è stato sempre vicino dall'età di 14 anni, vivendo assieme tanti momenti belli e dandoci forza in quelli brutti. Un forte ringraziamento a Luigi e Mario, con cui si è creata un'amicizia solida e duratura, che mi hanno sempre sostenuto nel momento del bisogno, credendo fortemente in me. Un sincero ringraziamento alla mia amica Luisa che ha scelto di essere presente nei miei momenti di difficoltà, aiutandomi a prendere decisioni importanti per il mio percorso. Ringrazio il mio amico Alessandro che mi ha aiutato tanto nel raggiungimento di questo obiettivo, e Manolo, che mi ha sostenuto nei momenti di difficoltà.

Ringrazio inoltre il Professore Luigi Panariello, che mi ha fatto scoprire la passione per le materie scientifiche, dandomi la spinta per diventare un ingegnere. I suoi insegnamenti sono stati necessari per il superamento di tante difficoltà.

Non posso citare tutti le persone, ma il ringraziamento va a davvero a tutti voi. Ringrazio anche chi ha fatto parte della mia vita, per poi prendere strade diverse. Il vostro apporto è stato indiscutibilmente fondamentale per me.

Concludo ringraziando i miei nonni Domenico e Rina. Devo tutto ciò che sono come persona a voi due e ai miei genitori, e per questo non smetterà mai di dirvi grazie. Siete sempre vivi nei miei pensieri. La dedica di questo lavoro va a voi due, ai miei genitori e a mia sorella.