FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Debugging Microservices

João Pedro Gomes Silva

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Pedro F. Souto

Co-supervisor: Filipe F. Correia

July 22, 2019

Debugging Microservices

João Pedro Gomes Silva

Mestrado Integrado em Engenharia Informática e Computação

July 22, 2019

Abstract

The microservice architecture is a method of developing software that structures a system as a collection of distributed single-purpose, loosely-coupled services. This style enables the continuous delivery and testing of large and complex software applications. The services are often owned by different teams, which may be in different geographical locations. This trend has become increasingly popular in recent years, as enterprises increasingly adopt agile methods and DevOps. Unfortunately the tooling, specifically when it comes to debugging microservice-based systems, is still lacking.

Due to its distributed nature, systems based on a microservice architecture are harder to debug compared to monolithic systems. To some extent, this is often due to their heterogeneity (both software and hardware-wise) concurrent execution and for being subject to partial failure. Many approaches have been proposed to better solve this problem. The most popular ones are tracing, log analysis, visualization and record, and replay. Each has its own merits, but also their own drawbacks.

This document describes a solution for debugging individual microservices, by trying to isolate its interactions with other services, recording them, and allowing to replay them in future executions. The approach proposed in this dissertation focuses on the instrumentation of network communication, independently of the protocol used, and random number generation in programs executing in the Java Virtual Machine in order to deterministically replay a previous execution of any of the services.

A prototype following the proposed approach was developed and used to conduct a case study, which showed that deterministic replay of executions can be obtained without introducing large execution time overhead in the original program, and with a high level of accuracy.

Resumo

A arquitetura de microserviços é um método de desenvolvimento de software que estrutura o sistema como uma coleção de serviços idependentes, cada um com um único propósito bem definido. Este estilo facilita a integração contínua e o teste de aplicações complexas. Frequentemente, os serviços são desenvolvidos por equipas diferentes, que podem até encontrar-se em localizações geográficas distintas. Esta tendência tem vindo a tornar-se cada vez mais popular recentemente, à medida que as empresas adoptam cada vez mais métodos ágeis de desenvolvimento e *DevOps*. Infelizmente, as ferramentas para fazer a depuração de sistemas baseados em microserviços não acompanharam este desenvolvimento.

Devido à sua natureza distribuída, sistemas baseados em microserviços são mais complicados de depurar quando comparados com sistemas monolíticos. De certa forma, isto é devido à sua heterogeneidade (quer a nível de software como de hardware), execução altamente concorrente, e o facto de que estão sujeitos a falhas parciais. Muitas abordagens foram propostas para resolver este problema. As mais populares são *tracing*, análise de *logs*, visualização, e *record & replay*. Cada uma tem as suas vantagens, mas também as suas desvantagens.

Este documento descreve uma solução para depurar microserviços, produrando isolar as suas interações com outros serviços, registando-as e permitindo a sua reprodução em execuções futuras. A abordagem proposta nesta dissertação foca-se na instrumentação de comunicação através da rede, idependentemente do protocolo utilizado, assim como na geração de números aleatórios em programs executando na *Java Virtual Machine*, com o objectivo de replicar deterministicamente uma execução prévia de um serviço.

Um protótipo que segue a abordagem proposta foi desenvolvido e utilizado para elaborar um caso de estudo, que mostra que replicação deterministica de uma execução pode ser obtida sem introduzir um *overhead* grande no tempo de execução do programa original, e com um nível alto de fidelidade.

Acknowledgements

I would like to express my sincere gratitude to my advisors Prof. Pedro F. Souto and Prof. Filipe F. Correia for their help and support throughout the development of this work. I would also like to thank Blip for the opportunity to conduct this work with them, and in particular to Diogo Monteiro and Rui Borges for their assistance during this project.

João Pedro G. Silva

"Ow! My brains!"

Douglas Noel Adams

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

API	Application Programming Interface
ECA	Event Condition Action
GDB	GNU Debugger
GUI	Graphical User Interface
HTTP	Hyper-Text Transfer Protocol
IP	Internet Protocol
JDK	Java Development Kit
JRE	Java Runtime Environment
JSON	Javascript Object Notation
JVM	Java Virtual Machine
KB	Kilobyte
MB	Megabyte
OS	Operating System
RNG	Random Number Generation
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
VM	Virtual Machine

Chapter 1

Introduction

Debugging is a core activity in the process of creating software systems. Developers report spending around half of their time in debugging tasks [LVD06]. The reason for this is, of course, that despite their creators' best efforts no non-trivial computer program is bug-free.

Microservices, a way of architecting systems as a complex collection of single-function, highly distributed, and loosely coupled services have become mainstream in the development world [Res18]. Its advocates claim that it increases maintainability and testability in large, complex systems. While this may well be true, the distributed nature of this architecture also introduces new challenges in the debugging process.

Despite the widespread adoption of distributed, microservice-based architectures, tools to help debug them are still in its infancy. As is shown in Chapter 3, many companies have developed custom solutions, that work only for their specific architecture. It's the purpose of this thesis to propose a more general approach to help solve this issue.

1.1 Motivation and Objectives

Traditionally, the debugging process has encompassed a few well-defined steps and made use of a set of basic generally applicable techniques. This section makes the argument that those steps and techniques are ill-suited to debug large and complex distributed systems such as microservice-based systems, and that novel ways to debug these kinds of systems are necessary.

A software bug is usually manifested through some unexpected behavior in the system. Once the behavior has been identified, the process finding and eliminating the problem which is producing it, also known as debugging, begins.

The first step in the debugging process is trying to replicate the problem. In other words, finding a set of input values and execution environment for which the unexpected behavior manifests. In complex distributed systems, this often turns out to be a non-trivial problem, either because the input and context are unknown or hard to replicate or because the problems are sporadic, due to race conditions or issues in synchronization between processes. There may be cases where it is impractical, if not impossible, to reproduce the problem.

The irreproducibility of problems in a complex distributed system is a problem for two main reasons. First, it is substantially harder to analyze a problem of which only one example exists. It greatly increases the search space, as each input or environment attribute is a possible cause for the unexpected behavior. Second, and perhaps more dramatic, is that if the circumstances under which some bug manifests itself are not known, then there is no hope of ensuring that a potential fix works. Once a fix has been devised and is being tested, if the unexpected behavior is not present there is no way of knowing if that is because the root cause of the issue has been found and fixed, or if it is just pure chance.

Even if one manages to reproduce the problem, there is still the need to find the root cause of unexpected behavior. Over the years a number of techniques [CBM90] have emerged to analyze the execution of a program and identify the root cause of a problem:

- *Output debugging*: Is the simplest technique available in the debugging toolbox. In output debugging a developer inserts output statements at selected places in the program that is being debugged. The developer then attempts to understand the behavior of the program based on the outputs of those statements.

  This technique is very simple and allows a developer to focus only on the section of the code that they hypothesize might be causing unexpected behavior. However, it requires modifications to the source code, and those modifications must be done in specific areas. The job of selecting the sections of code where to insert the output statements is not easy and often requires in-depth knowledge of the code base. In distributed systems, the difficulties are exacerbated because the developer must observe the output of the program in many different processors. If the number of processors is big enough this becomes unfeasible, since transient bugs may manifest themselves in some processors but not in others.

- *Tracing*: Tracing is similar to output debugging in that it relies on logging and outputting information in order to trace the execution of a program. Often, it relies on tracing functionality built into either the operating system or the platform. Unlike output debugging though, it can be turned on and off on demand and spans the entirety of the program, as opposed to just the sections of interest.

  Its advantages are that it is not necessary to change the program, avoiding the risk of introducing new bugs in the process, and that there is no need for the developer to decide where to place the output statements in the program's source code, avoiding cases where misplaced output statements offer no information, or erroneous information, about the execution. Unfortunately, it is still not very adequate to large scale distributed micro-services, since it produces vast quantities of data that are very difficult to analyze manually.

- *Breakpoints*: Breakpoints are points in the program where the developer has chosen to interrupt normal execution and output the complete state of the process at that point. When a breakpoint is hit, the developer can not only inspect the state of the process but also modify it. Furthermore, it is also possible to continue execution by one instruction at a time and inspecting or interacting with the state at each step.

This technique offers some clear advantages. In the first place, there is no need to modify the source code, once again avoiding the possibility of introducing new unexpected behavior when inserting the debugging probes. Secondly, the full state of the process can be inspected at each breakpoint, which means that the developer is not responsible for choosing the state variables that may or may not be useful to inspect. And finally, breakpoints can be used to analyze just the sections of code that the developer hypothesizes that might be responsible for the observed unexpected behavior. On the other hand, this places on the developer the burden of selecting the correct places on the source code to place the breakpoints such that meaningful and interesting information can be obtained that will be useful in the debugging process.

This is a very powerful technique and widely used to debug single processes. However, there are some difficulties when applying this technique to distributed environments. For instance, what should happen when a breakpoint is hit in a distributed system? Should all the processes in the system be halted, or just the process in which the breakpoint was defined? Furthermore, the state of a process is fairly easy to define and output, but the entire global state of a distributed system at some point is not so easy to define and would probably require synchronized clocks between all the processors in which parts of the system are executing on. Even the semantics of executing a single instruction is more complicated in a distributed environment. It is not clear if a single instruction should be executed in all the processes in the system or just in the process in which the breakpoint was defined. It is also not clear whether single instruction execution would be helpful at all: it is usually used to detect faulty logic, but not faulty interactions between processes.

Of course, breakpoints (as well as all the other traditional debugging techniques) remain useful to debug unexpected behavior that is confined to a single process, even if it is part of a larger distributed system.

- *Assertions*: Assertions define invariants in the source code that are checked at runtime. Assertions can behave like breakpoints in the sense that execution can be halted and the process state recorded when an assertion is violated or instead the execution can proceed as normal, but the state is saved somewhere for later analysis.

  Assertions require modifications to the source code, which means that there is the possibility of mistakenly introducing new bugs in the program. They also require some amount of expertise from the programmer for both knowing which invariants should hold at each time, and to place the assertions in the correct places. However, assertions offer something that the other methods of debugging analyzed so far don't: the possibility of collecting large amounts of data when, and only when the system does not behave as expected (and the invariant doesn't hold). This data has a higher likelihood of being relevant, as it was collected in a case where the process was misbehaving. It can then be analyzed to understand and identify the causes of the error.

But when it comes to applying assertions to a distributed system, once again the problem of obtaining a correct and synchronized global state at the precise moment that the assertion was violated does not have a simple solution. And without data on the global state of the system it becomes very difficult to detect bugs resulting from the interaction between different processes in the system.

- *Controlled Execution*: In controlled execution, a developer is may change the order of interactions between processes. This is especially useful to devise tests in which interactions are executed in different orders or at different speeds in order to identify problems that exist because of race conditions or other concurrency issues. This is a method often used to debug multi-threaded or multi-process programs. Thus, it is inherently more adequate to use in distributed systems, due to their similarities.

  However, this method has some disadvantages. Firstly, it requires the creation of a framework or system to support the controlled execution with a high enough degree of granularity to be useful. And secondly, it means extra work for the developers that would have to write the appropriate tests in order to identify unexpected behavour.

- *Replay*: Replay is a way to record information about the execution of a distributed environment and trigger a new execution with the same set of inputs as the first one. This technique can be used in combination with others such as assertions that identify and collect information when the behavior of the system is unexpected.

  The main advantage of this technique is that it allows for easier reproducibility of the erroneous behavior. However, this would necessitate great precision in the recording of significant events. Not only would the data transmitted between processes in the system need to be the same, but more crucially timings would need to be the exact same, or otherwise it is possible that the unexpected behavior would not manifest itself. This makes this technique very difficult to implement in practice. Not only that, but the replay is also an invasive technique that would require many changes in the source code, with the risk of introducing new bugs in the process.

- *Monitoring*: One of the most common ways of debugging a system is capturing as much information as possible about all relevant events that occur, and then going through the information once the system behaves unexpectedly. This technique is called monitoring and is one of the most popular techniques not only for debugging distributed systems, but also detecting issues such as performance drops, service availability and resource usage.

  The advantage of this technique is that if enough information is recorded then that information can be used to diagnose unexpected behavior. Furthermore, the information is useful even if the system is behaving as expected, for reasons such as improving the efficiency of the system. Many out-of-the-box solutions for monitoring distributed systems exist, and effort to integrate one into an existing system is small when taking into account the gain. However, it requires large amounts of space to store all the information that the system maintainers might need to debug, and the developers still have to filter through great amounts of information in order to find the relevant data that they might need to detect and resolve a bug.

Monitoring, while far from perfect, is the current preferred way to debug distributed systems. It was designed with distributed architectures in mind, and provides more benefits than just detecting and fixing unexpected behavior.

As shown in this section, traditional debugging techniques are mostly ill-suited to debug complex distributed systems, such as the ones based in the micro-service architecture. Many suffer from the problem of having been designed to debug single sequential processes and thus don't easily generalize to distributed environments. Others, such as monitoring, while easily applicable to distributed environments are still expensive and time-consuming. This dissertation argues that a better framework for debugging distributed systems can be devised based on the traditional techniques presented in this section.

A good framework for distributed debugging shall meet the following requirements:

- *Inexpensive*: A good debugging framework should minimize the costs with storage by storing only the required information to diagnose instances of unexpected behavior. Furthermore, it should strive to make the effort required to filter the collected data minimal, so that the debugging process can be made shorter freeing the developers time for other tasks and potentially reducing expensive downtime.
- *Non intrusive*: Every modification to source code has the possibility of introducing new bugs. This is obviously not ideal in a debugging framework, where the primary objective is to reduce the number of bugs in the system. As such, modifications to the source code should be kept to a minimum. This also has the benefit of facilitating the adoption of the framework.
- *Gradually adoptable*: In order to encourage adoption, gradual integration into existing systems must be possible. This also opens up the possibility of gradually adjusting the implementation according to empirical evidence for each specific system.
- *Generalizable*: A distributed framework should be able to detect and offer assistance in resolving bugs resulting from the distributed nature of the system, but also ones that are confined to one single process.

This dissertation builds on the basic techniques exposed in this section to create an approach that complies with the requirements exposed for a good framework for debugging distributed systems. Specifically it describes a way of recording an execution of a program so that it can be replayed in the future under controlled conditions, such as in the programmer's development machine and attached to an interactive debugger.

1.2 Dissertation Structure

This chapter introduces the context around the topic debugging in distributed microservice-based environments and makes the argument for the necessity of a new, general-purpose framework for distributed debugging.

The concepts and background information that are necessary to better understand the content of this document are explained in Chapter 2. Chapter 3 describes the state of the art and presents a survey of previous research regarding the topic of distributed debugging and a comparison between existing approaches and solutions. An overview of the problem and the challenges that the approach proposed in this document aims to address is detailed in Chapter 4. Chapter 5 describes one of the main contributions of this dissertation, which is a novel approach to debug microservices based on the recording and replay of past executions of a program. An implementation of a prototype tool according to the approach outlined in Chapter 5 is explained in detail in Chapter 6. A case study has been conducted and is explained in Chapter 7. The conclusions of the work developed in this document are described in Chapter 8.

Chapter 2

Background

The purpose of this chapter is to introduce concepts that are necessary to be aware of in order to understand the ideas outlaid in this document. The approach to debugging microservices described in Chapter 5 and the prototype implementation of that approach found in Chapter 6 make heavy use of the concepts explained in the remaining sections of this chapter.

2.1 SOA and Microservices

Service-oriented architecture, or SOA, is a style of software design where each component in a system provides a service to other components through a communication protocol, such as HTTP or RPC, over a network. According to the definition of SOA found in [Gro14], a service must have the following properties:

- *It logically represents a business activity with a specified outcome.*
- *It is self-contained.*
- *It is a black box for its consumers.*
- *It may consist of other underlying services.*

Service orientation promotes loose coupling between services. In effect, this improves modularity as it means that component services may dynamically bound to other services, and services can be replaced without introducing downtime or otherwise affecting the other systems in the system. SOA encourages the establishment of clearly defined interfaces between services which offer the sole point of access. These interfaces along with the communication protocol establish the only contract between services in the system.

Service orientation also enables the concept of location transparency. This means that the consumer of the service has no information about where the service actually resides, meaning that it can be in the same machine, in a server accessed through the internet or even in multiple remote machines. The actual physical location of the provider service is hidden behind a point of access that is published by a service broker, whose main responsibility is to maintain and provide information about the services to any potential consumers.

Another benefit of SOA is re-usability. Because services are self-contained, there is nothing stopping one service being part of multiple applications. In fact, from the point of view of the producer service, there is no distinction between the consumers; it does not know whether they are part of the same application or not because there is no concept of application at the service level.

In an enterprise setting, SOA also provides an organizational benefit: Services can be independently developed and tested by different individuals or teams. This increases parallelism in development, and thus development speed. In addition, it promotes good testability as each team relies on the services maintained by others to behave in accord with specifications.

Lastly, service-orientation also facilitates availability and scalability of a system by clustering services, therefore, introducing redundancy in the system. Additionally, clustering might be done only in the services that represent a bottleneck in the overall system throughput, further increasing efficiency.

Microservices are a variant of SOA. The main difference lies in service granularity, or in other words, in how big a service must be. There is no official prescription regarding this, but the consensus in the industry seems to be that a service in the microservice architecture must be feasibly maintained by a team of 6 to 8 developers. Another differentiation between microservices and SOA is that microservices are usually more loosely-coupled. As with any development style, SOA and microservices are subject to some criticisms [fow14].

- It can be hard to establish service boundaries. When splitting a monolithic application into microservices, it can be hard to define where exactly the split should be made. If the split is made into too few services, it can be hard for one small team to maintain and the benefits gained from the transition are limited. On the other hand, if the split is made into too many services, then this may cause large dependencies between services to the point that development in one service may be blocked by the development on another. Additionally, this might also cause developers to lose sight of the overarching system goals and focus too much on the goals of the service itself.

- Communication between services over a network has a higher time cost because of network latency and message processing than in-process calls in a traditional monolithic system. Depending on the complexity of the system, the number of messages exchanged, and the purpose of the application the added overhead might be prohibitive.

- While testing individual services may become simpler with SOA and microservices, testing the overall application might become more complicated, because of the great number of dependencies between services. For the same reason, deployment can become more complicated.

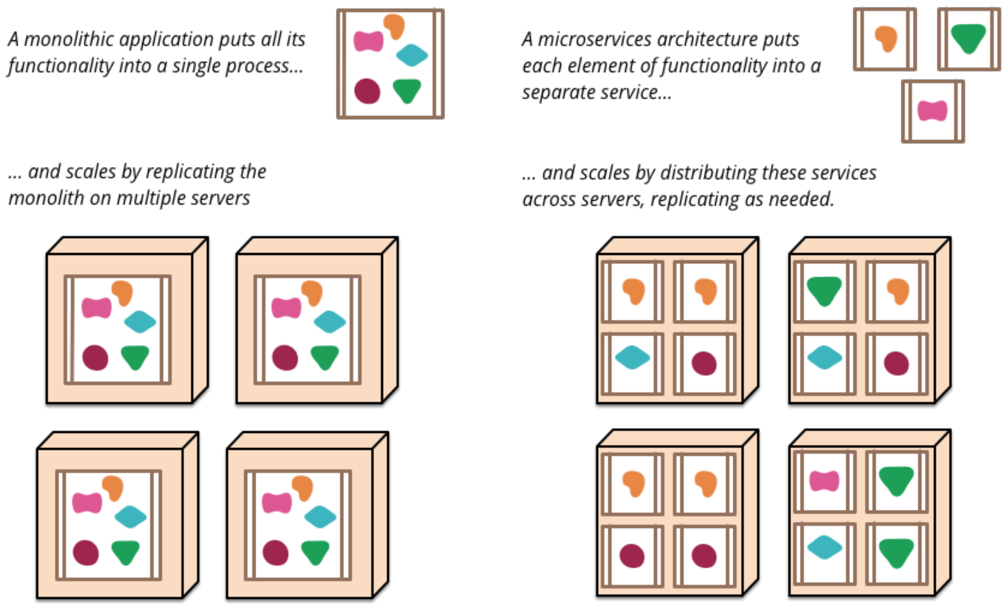

A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

Figure 2.1: Differences between monolithic applications and microservices [Fow]

- If there is not enough reusability of the services, the benefits of moving to SOA or microservices are greatly diminished and the same effect can be achieved with internal modularization while maintaining a simpler design.
- Data consistency is significantly harder to guarantee in microservice-based architectures. For example, if each microservice maintains its own database as opposed to one centralized application for the whole application, then a programmer can no longer rely on the database to ensure data consistency. Additionally, protocols such as 2-phase commits are inadequate for microservices since they increase coupling between services.
- Microservices usually increase the cognitive load for programmers. The architecture introduces additional complexity and new problems such as network latency, message serialization, load balancing, and fault tolerance. New failure modes are also introduced which are difficult to reason about. Debugging becomes increasingly complicated as the sources of problems are potentially more.

Despite all the challenges, microservices have been rising in popularity in the last few years. Its advocates claim that this architectural style further reinforces a modular structure for the system, facilitates independent development and deployment of each service and increases developer freedom to choose whichever languages and technologies are best suited for the job at hand [Mic].

2.2 The Java Instrumentation API

The Java instrumentation framework provides services that allow Java agents to instrument programs running on the Java Virtual Machine, through the mechanism of byte-code manipulation.

A Java instrumentation agent can be started at VM startup, through a command line switch as such:

```
-javaagent:jarpath[=options]
```

Where the *jarpath* is the path to the agent's JAR file and *options* is the options passed to the agent — a string whose parsing and interpretation is left to the agent's implementation. An agent can also be loaded after the VM has been started, through an implementation-specific mechanism. More than one agent can be loaded into one application and multiple agents can share the same *jarpath*.

According to the Java Instrumentation API documentation [Ins], an agent is a JAR file crafted according to the following specifications:

- The manifest of the agent JAR file must contain the attribute Premain-Class, which is the name of the agent class. The agent class must implement a public static premain method similar in principle to the main application entry point. After the Java Virtual Machine has initialized, each premain method will be called in the order the agents were specified, then the real application main method will be called. Each premain method must return in order for the startup sequence to proceed.

  The premain method may have one of two signatures. The JVM will first attempt to invoke the method with the following signature:

  ```
  public static void premain(String agentArgs, Instrumentation inst);
  ```

  If this method is not implemented, the JVM will instead try to invoke the method with the following signature instead:

  ```
  public static void premain(String agentArgs);
  ```

- In addition to a *premain* method, the agent class may also implement an *agentmain* method that is invoked when the agent is started after the Java Virtual Machine has started. This method is not invoked when the agent is started at JVM startup through a command line switch.

  The *agentmain*'s method signature is the same as the signature of the *premain* method.

- If the agent is meant to be loaded after the JVM has been started it must, in addition of implementing the *agentmain* method, contain the attribute *Agent-Class*, containing the name of the agent class, in the manifest of its JAR.

In addition to the *Premain-Class* and *Agent-Class* attributes mentioned so far, the agent's JAR manifest file may also define the following attributes:

- *Boot-Class-Path*: A list of paths to be searched by the bootstrap class loader. This attribute is optional.

- *Can-Redefine-Classes*: A Boolean value that determines whether or not the agent can redefine classes. This value is optional, and is particularly useful if the agent wishes to modify JVM classes.

- *Can-Retransform-Classes*: A Boolean value that determines whether or not the agent can re-transform classes. As with *Can-Redefine-Classes* this value is optional and useful if the agent wishes to modify JVM classes.

- *Can-Set-Native-Method-Prefix*: A Boolean value that determines whether or not the agent can set native method prefix. This value is also optional.

The agent class is loaded by the system class loader. This class loader is typically responsible for loading the class that contains the *main* method. The agent options are passed to the agent through the *agentArgs* parameter, and if the agent cannot be resolved the JVM will abort. The second argument passed to the agent's *premain* or *agentmain* methods is an instance of the Instrumentation interface. This interface provides the services needed to instrument Java code. The instrumentation capabilities offered by this interface are purely additive, meaning that they do not modify the application state or behavior. An instrumentation instance is created when the JVM is launched with a Java agent via a command line switch, or when a JVM provides a mechanism to load agents after the JVM is started.

Once an agent acquires an instance of the *Instrumentation* interface, it can use it to modify the classes that are loaded. The *Instrumentation* instance allows the registration of instances of the *ClassFileTransformer* interface. A *ClassFileTransformer* instance implements a single method named *transform*.

Once a transformer has been registered with *Instrumentation.addTransformer*, the transformer will be called for every class definition and redefinition. The *transform* method takes an array of bytes that contains the bytecode of the class that is being loaded. If no transformations are needed, then the method must return *null*. Otherwise, it should create a new byte array, copy the input bytes into it, make the desired transformations and then return the new array. The input byte array must not be modified. This technique is called bytecode manipulation and is very powerful.

Obviously, manually altering every byte in the byte array in order to modify the class is not very practical. As such some libraries have emerged that aim to facilitate this process. Some of the most popular are:

- **ASM** [ASM]: A Java bytecode manipulation framework that provides a set of common bytecode transformations and analysis algorithms that can serve as a base to build more complex tools. It is designed for performance, aiming to be as small and as fast as possible.

- **Javassist** [Jav]: A library to modify Java bytecode. It offers two levels of API — The bytecode-level API allows the modification of classes using Java bytecode, while the source-level API allows the transformation of classes using only Java source text, requiring no knowledge of Java bytecode.

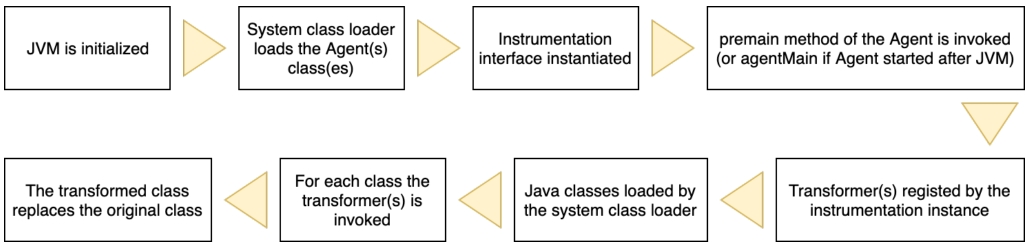

JVM is initialized
System class loader loads the Agent(s) class(es)
Instrumentation interface instantiated
premain method of the Agent is invoked (or agentMain if Agent started after JVM)
Transformer(s) registed by the instrumentation instance
Java classes loaded by the system class loader
For each class the transformer(s) is invoked
The transformed class replaces the original class

Figure 2.2: The instrumentation process of a Java program

- **Byteman** [Byt]: A higher-level framework that allows the injection of Java code into application methods. It makes bytecode transformations easier by implementing an abstraction layer in front of the transformer. The places in the code where the code is to be injected are defined in rule files written in a specific format.

It's worth mentioning that bytecode transformations can be applied not only to user-defined classes but also to core JDK classes. While most of the core classes will be loaded before the agent's *premain* method is called, and thus will not be transformed, it is possible to re-transform them. This requires the attribute *Can-Retransform-Classes* to be true in the agent's JAR manifest. Then the *Instrumentation.retransformClasses* method can be used to re-transform most of the core classes of the JDK. This is obviously extremely powerful and disruptive, and if it is not used with care might break programs in all sorts of unexpected ways.

Common uses for the instrumentation API are program profiling and analysis without modifying the original program source code. Other uses include class generation and code optimization and obfuscation.

2.2.1 Byteman

Of the bytecode manipulation frameworks mentioned in this section, *Byteman* is the one used extensively in the implementation described in Chapter 6. Byteman is a bytecode manipulation tool that enables the modification of previously compiled Java code. Byteman uses a system of rules to specify when and wherein the program the code should be modified. Rules are written in the Event Condition Action, or ECA [byt18], rule language. Figure 2.1 shows the basic skeleton of a Byteman rule. The most common keywords are the following:

- `RULE`: This keyword defines the beginning of a rule and is followed by a text string that defines the name of the rule. The name can be any string, as long as there is no other rule with the same name. The rule's name is used merely for identification in the source code, for example, in the case of compilation errors in the rule.

- `CLASS`: This is followed by the fully qualified name of the Java class that is to be instrumented. The name of the Java class can be preceded by the "^" character. In such cases, the rule instruments all classes that are sub-classes of this one.

- `METHOD`: This keyword is followed by the name of the method of the class that is to be instrumented. It can contain just the name of the method, in which case it will instrument all the methods with that name, or the name of the method followed by a list of type parameters, in which case it will only instrument the methods whose signature matches the number and type of the parameters defined.

  From this point on, the arguments passed to the method are available to the next statements. The following are some of the most relevant:

  – `$0`: Holds the instance of the instrumented class where the method has been invoked.
  – `$1, $2, ..., $n`: Hold the arguments that are passed to the instrumented method.
  – `$!`: Holds the return value of the instrumented method.

- `AT`: This keyword determines the point at which the method should be instrumented. Some of the valid values that this can take are:

  – `AT ENTRY`: The code in the `DO` block is inserted right before the first instruction of the method.
  – `AT EXIT`: The code in the `DO` block is inserted just before the return statement of the method.

- `BIND`: This keyword permits the binding of the values available at this point, such as the arguments passed to the rules or the instance of the object that is being instrumented, to other variable names.

  In Chapter 6 binds are used extensively in order to improve the readability and facilitate the comprehension of the listings.

- `IF`: This keyword is followed by the conditional statement that determines whether or not the rule is valid for a specific instance. It is checked every time before execution of the code that is defined after the `DO` keyword.

- `DO`: This is the keyword that defines the code that must be executed in the defined point in the instrumented method. This keyword is followed by the Java code that is to be executed. It can include multiple statements separated by a semi-colon.

- `ENDRULE`: This keyword defines the end of the rule.

There is another important keyword, which is `HELPER`. This keyword is followed by the fully qualified name of a Java class whose methods can be used in the `IF` and `DO` blocks, instead of writing the code directly in the rule definition.

```
# rule skeleton
RULE <rule name>
CLASS <class name>
METHOD <method name>
BIND <bindings>
IF <condition>
DO <actions>
ENDRULE
```

Listing 2.1: The skeleton of a Byteman rule [byt18].

2.3 Protocol Buffers

Protocol buffers [Pro] are a mechanism to serialize and de-serialize data. It has been developed by Google. In protocol buffers, the format and structure of the data is defined in special files with the `.proto` file extension. These files are then used to generate source code in different languages that allow reading and writing the data to a variety of formats. Furthermore, protocol buffers allow updating the structure of the data without breaking programs that make use of a previous version of that structure.

Chapter 3

Literature Review

This chapter presents an analysis of the state of the art of the work done in the area of debugging large highly distributed systems, including those based in the micro-service architecture.

3.1 Introduction

The field of distributed programming is still fairly recent. It is a field of active research and many approaches have been proposed.

Proposed approaches for debugging distributed systems differ in many ways. One of the differences in the phase of the process in which they are applicable. Debugging can be thought of as a process consisting of five steps [CPP]:

1. **Recognize that a bug exists.** Bugs manifest themselves through unexpected program behavior. Some are particularly serious and cause the program to terminate abnormally. Those are easy to spot. Others manifest themselves in more subtle ways and can go unnoticed for a long time. The best way to find bugs early is through comprehensive testing.

2. **Isolate the bug.** Bugs are contained to a section of the source code. Isolating the bug is identifying the section of code in which the error lay, thus restricting the search space for the next step in the process.

3. **Identify the cause of the bug.** Knowing the general vicinity of the bug is not enough. The next step is identifying what is actually causing it. The previous step was concerned with finding the *where*. This step aims to find the *why*.

4. **Determine a fix for the bug.** Having understood *where* the bug is and *why* it the program behaves as it does, the programmer is then equipped to determine how the problem can be mitigated..

5. **Apply the fix and test it.** The last step is to validate whether or not the devised solution fixes the bug. Furthermore, it also aims to ensure that no other bugs have been introduced.

The distributed debugging approaches exposed for the remainder of this chapter aim to improve either step 2, step 3, or both steps from the aforementioned process. The reason is that those are generally the most time-consuming and thus benefit the most from any increased efficiency.

3.2 General Approaches For Debugging Distributed Systems

Several tools and frameworks for debugging distributed systems have been proposed. These usually take one of the following approaches to the debugging process [BWBE16]:

- **Tracing**: Collecting information about a request throughout its lifetime in the system, including cross-process boundaries.
- **Log analysis**: Manually or automatically analyzing system logs in order to detect and diagnose anomalies.
- **Visualization**: Transforming execution data in a high-level visualization of the system, in order to make it more understandable for a programmer.
- **Record and Replay**: Capturing information about an execution of the program so that it can be deterministically replayed as many times as necessary.

Each outlined approach has its advantages and disadvantages. The following sections discuss in further detail each approach and analyze particular implementations of each.

3.3 Distributed Tracing

Distributed tracing is a method to monitor distributed systems. It consists of instrumenting the source code of a program in order to collect and record metrics regarding its functionality. It is commonly used to track requests, such as user requests in a web service, and collect important metrics such as latency and failure rate. It helps the programmer identify where failures happen and understand what is causing worse than expected performance.

In microservices systems, requests often span more than one service. Distributed tracing can follow requests through service boundaries, provided that each service is adequately instrumented. The information recorded for each individual step that a request takes through a distributed system is referred to as a *span*. *Spans* reference other *spans* and together they form a complete *trace*, an end-to-end visualization of the lifetime of a request in the system. This is achieved by assigning a unique request ID to each incoming request and passing that ID to all services involved in handling the request. In that way, each service in the system knows to which *trace* a certain *span* belongs to, therefore binding the collected data to the request that originated it.

Distributed tracing shines when used in conjunction with tools to aggregate, search and visualize the data. Such tools would, for example, aggregate request tracing data according to the ID of the request. Some alert programmers when a metric exceeds a pre-defined threshold, helping maintain high service availability and discover latent bugs.

Distributed tracing is one of the most used approaches to debugging in large-scale, complex distributed environments. Despite its popularity, there are some difficulties:

- **Metric overload.** It often is the case that distributed tracing produces overwhelming amounts of data. Once the instrumentation is in place, it is usually easy to track more and more metrics. In the end, it's up to the programmer to exercise its best judgment in deciding how many and which metrics to keep track of without obscuring potentially important information.

- **Vendor lock-in.** There are few standards when it comes to the structure of *traces*. Therefore, each distributed tracing tools rolls their own implementation, making it hard to switch between tools. Efforts have been made to develop vendor-neutral APIs, namely through *OpenTracing*[Opeb].

- **Runtime overhead.** Distributed tracing tools and frameworks must aim to maintain runtime overhead to a minimum. Failure to do so affects the distributed system negatively and can defeat the purpose of adding the distributed tracing infrastructure in the first place.

- **Storage requirements.** *Traces* are often collected locally in each node in a distributed system, and transferred later for centralized processing. This may make it unsuitable for services with high throughput.

Many distributed tracing tools have been proposed, such as Magpie [BIN03], Pinpoint [CKF+02] and X-Trace [FPKS07]. Each has made new important contributions to the design of distributed tracing tools. But perhaps none is as influential has Dapper [SBB+10], a distributed tracing framework developed at Google. The following section analyzes Dapper and its contributions to the state of the art.

3.3.1 Dapper

Dapper is a low-level tracing framework to trace infrastructure services developed at Google. The stated goals of Dapper are low overhead, application-level transparency, and scalability. It records a collection of timestamped messages and events that trace all the work that is done in the system, starting in a single initiator.

The framework traces the work done on the system starting from a single initiator. It uses an annotation-based [BIN03] monitoring scheme, meaning that every record is tagged with an identifier that binds the records together and with the original request. This approach has the drawback of requiring the instrumentation of programs. Dapper takes advantage of the homogeneous infrastructure at Google and restricts instrumentation to only a small set of libraries shared by all applications. Causal relationships between *spans* are preserved by representing a trace as a tree structure, in which each node is a *span* and each edge indicates a causal relationship. Thus, the root *span* is causally related to all the other *spans* in the three, while a leaf *span* has only a causal relationship with its parent.

One of the new contributions introduced by Dapper is the usage of sampling to maintain low overhead, which is especially useful in services that are highly sensitive to latency variations. The Dapper developers use a uniform sampling probability to select which requests to record. They found that recording on average 1 out of each 1024 requests is likely to still capture data of interest in high throughput services. However, in services with lower traffic, those sampling rates are not high enough to ensure that important information is not missed.

3.3.2 Conclusion

Many of the benefits provided by Dapper are only possible due to the uniformity of Google's distributed environment. Most applications share some common libraries and use the same programming languages. Unfortunately, this does not translate very well to microservice architectures. Microservices are inherently heterogeneous. The work required for instrumentation would be very high and so would the effort required to maintain interoperability. However, some of the ideas introduced by Dapper have proven to be useful and are widely used today.

Furthermore, while distributed tracing may provide great insights into the workings of a distributed application, and does offer important performance information, it is not very useful for debugging past the point of recognizing the existence of the bug. It may, in certain cases, provide the programmer with useful information to isolate the bug, but when it comes the time to identify its cause, the programmer is left to its own devices.

3.4 Log Analysis

Debugging through log analysis consists of examining the free text logs generated and collected by applications in order to recognize and isolate a bug. It is perhaps the most primitive form of distributed debugging, but it offers something that all other approaches do not: it works even if the system being debugged cannot be modified. In an age where third-party cloud services are ever more commonly part of the architecture of most distributed systems, log analysis is indeed often the only kind of debugging approach that is applicable.

This approach relies on relevant and informative log messages being present throughout the code, and in specific in the section that contains the problem. Adequately placing log statements requires experience and familiarity with the code base. It is essential to strike a balance between having enough information to assist in the debugging process and cluttering log files and obscuring relevant information with irrelevant messages.

Production services tend to produce large amounts of logs. So many, in fact, that trying to find relevant information is akin to trying to find a needle in a haystack. This problem is amplified in distributed systems since the logs may be scattered in the different machines on which the services run. A programmer, if left unassisted, has little hope of finding the logs that he needs to help identify or isolate a bug.

Efforts [XHF+10] have been made to automate the task of filtering and organizing the information recorded by logs. The big difficulty in extracting information from log messages is their unstructured, free text nature. One of these attempts is analyzed in the following section.

3.4.1 Mining Console Logs

Researchers at Berkeley [XHF+10] have attempted to detect bugs in programs by automatically analyzing the application logs. They devised an approach to transform logs into more structured data with a higher signal to noise ratio. The process is described as having three steps:

1. **Log Parsing.** The aim of this step is to extract information from logs such as *message types* and *message variables*. *Message types* are messages with the same structure while *message variables* are the variable parts of each message.

   Static source code analysis is used to extract all possible log messages and build an index of message templates to which logs are matched to at runtime.

2. **Feature Creation.** Many problems can only be detected in a sequence of messages. This step focuses on grouping related messages in preparation for problem detection.

   Messages are grouped in two different ways. Firstly, based on state variables, which enumerate the possible states of an object in a program. Secondly, based on the number of times an identifier shows up in the logs. Different features lead to different types of problem detection.

3. **Machine-Learning.** Finally, statistical methods are utilized to identify common patterns and therefore discover anomalies in the data, which may indicate the presence of a bug.

This approach has proven successful, but with some faults. Namely, it does not detect certain classes of bugs such as those that arise in distributed environments because of the contention for resources and race conditions.

3.4.2 Conclusion

The unstructured nature of log messages makes it so that the potentially useful information that can be automatically extracted is not enough to reliably detect many classes of bugs. For that reason, log analysis remains of limited usefulness for developers and is best used when the quantity of log messages is small or there is no other option.

In addition, log analysis aims at identifying and recognizing that a bug exists in the program. It offers little help in isolating the bug, and none at all in the further steps of the debugging process.

3.5 Visualization

Distributed systems based on microservices can be highly complex and thus hard to understand for the human brain. It is that complexity that inspired work on visualization tools that aim at making distributed systems more transparent to the programmer. They offer a high-level visual abstraction over the underlying system that can be used to spot anomalies and understand the overall behavior of the system.

The downside of such approaches is that the underlying structure of the system is hidden and important details may be omitted. Such details could otherwise be used by programmers to define or refine their hypothesis about the root cause of problems in the system.

The following section analyzes a visualization tool called Theia [GKT$^+$12] that takes advantage of application-specific knowledge about Hadoop [Had] in order to help developers identify problems in their clusters.

3.5.1 Theia

Theia [GKT$^+$12] is a visualization tool that analyzes application-level logs for large Hadoop clusters, generating visual signatures of each job's performance. The idea behind Theia, as in most other visualization tools, is to provide a high-level overview that allows developers to explore the large amounts of data collected by the monitoring systems and narrow down the root cause of the issues.

Theia exploits application-specific knowledge about Hadoop to provide relevant insights to the developer and distinguish application from infrastructure errors with some amount of success.

3.5.2 Conclusion

Visualization tools appear to be most useful when used in a well-known and controlled environment, about which assumptions can be made that allow the tool to create relevant and insightful visualizations. This is not the case in large scale highly distributed systems based on microservices, which are mostly heterogeneous. Assumptions made about services would likely not hold for others. This limits the usefulness of visualization tools for providing generic performance information about the system.

3.6 Record and Replay

Record and replay consists of capturing an execution such that it can be replayed deterministically any number of times that a programmer might need to identify and resolve a problem.

This section goes over three debugging frameworks based on the record and replay approach: GoReplay [GoR], D3S [LGW$^+$08] and Friday [GAM$^+$07].

3.6.1 GoReplay

GoReplay [GoR] is a tool which allows the recording of production traffic and replaying that traffic during testing. It provides the ability to replay the traffic at different speeds and to re-write requests. Furthermore, it performs the analysis and recording of network traffic without modifying or otherwise affecting the applications that use it.

GoReplay supports a large but limited number of network communication protocols, including binary protocols. Furthermore, it is aimed at replaying only network traffic, and not necessarily the execution of a program. If part of the execution of the program is non-deterministic, because it depends on random number generation or for any other reason, then GoReplay does not ensure that the program executes in the same way. This somewhat limits the application of this tool.

3.6.2 D3S

D3S [LGW+08] is a framework that models the execution of a distributed system as a state machine in which states are snapshots of the system with different global timestamps. It allows developers to write functions that check invariants in the system. During runtime, global snapshots of the system are recorded. When an invariant is violated the checker functions signals it and the developer is able to inspect the system snapshots that lead to the faulty state, effectively replaying the execution.

D3S can be fairly expensive to implement. Firstly it requires large amounts of data to be collected, though this can be mitigated by using buffers to store the system snapshots, that is only persisted in case an invariant is violated. Secondly, it introduces overhead in the system that can go up to 8%. Whether or not this is significant in a system is up to the maintainers to decide.

While D3S does not require changes to the existing source code of a program, it is not as non-invasive as it might appear at first sight. This is because it rewrites the binary modules of the process that it is attached to. Assuming D3S is well tested this should not have any nefarious consequences, but as any modification to the source code, it carries the added risk of introducing bugs in the system.

Furthermore, using D3S effectively requires some level of familiarity and expertise with the system being debugged, since developers have to write functions to check function invariants. This also adds the risk of false positives due to bugs in the checker functions.

In the other hand, D3S presents some significant advantages. It can be adopted gradually, in the sense that developers can write new checker functions incrementally in order to cover larger and larger sections of code. Additionally, it has also been shown empirically to work in some real-life systems to detect and diagnose some classes of bugs.

Ultimately, the greatest problem with D3S is that it cannot diagnose bugs if they are not covered by the invariants checked by the checker functions. This requires extra development effort in designing and implementing comprehensive checker functions and even then some bugs are bound to remain undetected and thus unsolved.

3.6.3 Friday

Friday is a system that is based on another tool by the same authors called *liblog* [GASS06]. *Liblog* itself provides the record and replay functionality, while Friday manages the replay in the context of an interactive debugger, such as *GDB* [GDB]. Furthermore, it extends debugger functionality with the addition of distributed breakpoints and watchpoints that allow a programmer to inspect the state of the system during the replay.

Liblog is the core of Friday. It works by intercepting system calls and recording information that is non-deterministic, such as random number generation or the sending and receiving of messages so that the execution can be replayed deterministically.

The limitations of Friday have much to do with the limitations of the underlying *liblog* library. *Liblog* produces vast amounts of data, which makes it unsuitable for systems with high throughput. Furthermore, it also introduces some amount of network overhead, since it embeds a Lamport clock [Lam78] in each message in order to preserve causal relationships.

Friday provides something that most other solutions do not, and that is the ability to assist developers for all steps in the debugging process, up until testing and validating that any devised solution works as expected.

3.6.4 Conclusion

Record and replay based approaches tend to be more complex than others. They tend to introduce a large runtime and network overhead and require large amounts of storage in order to record enough information to accurately replay executions.

However, as demonstrated by Friday, such approaches have the potential of being of great use throughout the entirety of the debugging process, as opposed to just helping to identify and isolating a bug. With record and replay systems a programmer has access to the entirety of the state of the application at each and any moment and can inspect it as they would in any other sequential single process application. In fact, record and replay tools have the possibility of making distributed debugging more similar to the debugging of single process applications.

3.7 Conclusions

Different approaches to debugging distributed environments, and in specific microservices, each have their own merits and drawbacks. Most can be used complementary, for the areas in which one excel is usually the ones other lack. It is common, for example, for monitoring systems to take advantage of distributed tracing to collect data and visualization tools to present it to the programmer. Table 3.1 presents a summary of the analysis done in this chapter.

Unfortunately, most approaches focus on helping the programmer identify a problem. When it comes to identifying the root cause of a bug the developer is left fending for themselves. This is where the record and replay approach outshines the others. While it is much more complex, and most likely requires some familiarization with debugging tools such as interactive debuggers, it assists developers in identifying the cause of the problem by allowing them to inspect the state of the program at each step in the execution.

Table 3.1: Summary and comparison between the approaches to debugging a distributed micro-service based system.

Approach	Tracing	Log Analysis	Visualization	Record and Replay
Target	Helps to identify the cause of a bug	Helps to identify the cause of a bug	Helps to identify the existence of a bug, as well as isolating it and identifying its cause	Helps with all steps of the debugging process, since identifying the existence of a bug to determining and applying a fix
Cost	Low to medium overhead cost, but possibly high storage cost	Low overhead cost, possibly high storage cost	Low overhead cost, possible high costs of storage and maintenance of the visualization service or tool	Higher overhead and high cost of storage
Invasiveness	Somewhat invasive, may require modification of source code	Not very invasive	Not very invasive	Can be invasive and require modifications to the source code
Gradual adoption	Possible, at the cost of missing information	Possible	Possible, although at the cost of usefulness	Possible

Chapter 4

Problem Statement

The overarching problem addressed in this document, as described in Chapter 1 is of how to best debug microservices. This chapter breaks that overarching problem into more specific issues, that are addressed throughout the remainder of this thesis.

4.1 Main Issues

Part of what makes distributed programs harder to debug comes down to their increased complexity and high concurrency. Developers generally find that reasoning about concurrent events is more challenging than sequential ones. Furthermore, the fact that distributed programs run simultaneously on multiple processors, each of which maintaining their own time reference, further complicates the debugging process.

Cheung [CBM90] identified the following difficulties in debugging distributed programs:

- *Maintaining precise global states*. Local state on each machine is readily available, but combining them into a global state requires that all processors clocks be synchronized, which is difficult to achieve.

- *Large state space*. The state space includes each machine's own state as well as the records of the interactions between all machines in the system. Large amounts of data are produced, from which the interesting parts must be selected to be analyzed. This is often non-trivial and requires manual effort by the developer or system operator. When the system grows to a certain size, this task becomes cumbersome.

- *Interaction between multiple asynchronous processes*. Bugs that result from the interaction between processes are often sporadic and hard to reproduce, resulting from synchronization issues or race conditions.

- *Communication limitations*. Communication delays and limited bandwidth may render some traditional debugging techniques impractical.

- *Error latency*. The time interval between the occurrence of the error and it's detection is usually larger in a distributed program. By the time the error is detected, it may have propagated to other nodes in the system. This may make it harder to find the root cause of the issue.

A great number of issues in distributed programs arise from troubles in the communication between the nodes in the system. Microservices are even worse in this aspect. Because they are made of smaller, more focused components, then this means that there are usually more nodes in the system when compare to a distributed system comprised of more monolithic components. This larger number of nodes has two consequences:

- Each individual node has a larger number of dependencies.
- A larger number of messages are exchanged in the system.

This makes this kind of issues even more common. The problem with all these issues is that they are mostly transient, and transient errors are very hard to reproduce. Often programmers are left scavenging through logs to find any information that might help root cause the problem, with no other recourse. Sometimes the problems are even left unsolved due to lack of logging and the development team must wait until the next time the problem occurs to try and find the root of the issue.

4.2 Objectives

The goal of this work is to define a conceptual framework for debugging microservices and will focus on one main research question: **"Can a specific execution of a system be faithfully replayed in a developer's local machine?"**

An execution is a subset of all the behaviors encoded in a programs source code that is manifested every time a program executed. Some executions may be anomalous meaning that the behavior of the program is different from the expected behavior. Anomalous executions are especially interesting in the scope of this document as they represent bugs in a program. While some instances of anomalous behavior might be automatically detected, others require knowledge of the business rules and the purpose of the program and thus require manual inspection of the results. That is not the scope of the framework, but in order to use the framework, it is necessary to have a system in place to detect anomalous executions that might be of interest to be replayed.

It is also important to define what it means to faithfully reproduce an execution. A naive approach to determine whether or not a replayed execution is faithful to a reference execution would be to check if the output produced is the same. However, that would not be enough since it is possible that the replayed execution produces the same results in a different way. That is not ideal since the purpose of the framework is to allow a programmer to inspect the internal state of the system. Evidently, this is only useful if the state of the system during the replay is the same as the state of the system in the point in the original execution.

The approach described in this document to reproduce an execution requires first recording said execution so that it can then be replayed. As such, to answer the main research question it is necessary to answer other questions as well:

- Can enough information be recorded in an execution without introducing overhead in execution time that impacts the viability of the system?
- Is the amount of data generated during the execution small enough to be viably stored in either the machines executing the instrumented program or in centralized storage?
- Can the recording and replaying of executions be performed without modifying the source code of the program?

The solution proposed in this document attempts to achieve replay-ability of executions of programs running in the Java Virtual Machine without modifying the source code of the program, and relying purely on bytecode manipulation. This is unlike all the other solutions analyzed in Chapter 3.

4.3 Methodology

In order to answer the research questions enunciated in this chapter, a prototype tool was developed based on the record and replay approach described in Chapter 5. Details regarding the implementation of the prototype can be found in Chapter 6.

This prototype has been used to conduct a case study in a Java service that is part of a larger system based on the microservice architecture. The description of that case study can be found in Chapter 7. The purpose of this case study is to collect and analyze data in order to answer the research questions.

Chapter 5

Approach

The aim of this chapter is to describe the approach taken to the development of the prototype tool, which is subsequently used to help answer the research questions raised in Chapter 4. It begins by enunciating the assumptions and core tenets on which the solution lies, the needs that a tool based on this approach must meet and the reasoning that lead to the proposal. It then presents a detailed description of the approach, including some of the main challenges to its implementation.

5.1 Assumptions and Core Tenets

The solution described in this chapter and the next is based on recording and accurately replaying executions of instrumented programs. This approach was chosen instead of others because it provides the greatest insight into the behavior of the program being debugged since the programmer has access to all of the internal state of the program as it happened during a previous execution. Furthermore, this approach offers easy reproduce-ability of errors since if an error is recorded even once, then it can be replayed and thus reproduced. Its goal is to assist programmers throughout all of the steps in the process of debugging distributed systems thus making it faster and more efficient. The solution was developed with the following assumptions and core tenets in mind:

- The source code of the programs to be debugged must not be modified, and it is acceptable to sacrifice accuracy to do so. This is the most important assumption and the one that guides the implementation of the solution. Some sacrifices are made in terms of accuracy of the replayed execution for certain types of programs and user experience in order to maintain the source code of the programs unmodified.

- The solution must be designed for and work on production environments and with production workloads. This restricts the amount of information that can be recorded, as well as the way in which such information must be stored.

- The functionality that is instrumented by the solution to be replayed is concentrated in a relatively small set of libraries that are widely used by most programs in each language. These libraries are usually part of the standard library for a programming language.

5.2 Designing an Approach for Record and Replay

In order to be useful, the solution presented in the next chapter must meet the following needs:

- **Low Overhead**. In order to maximize its utility, the framework must record information continuously in the service. In order for this to be possible in latency-sensitive applications, the framework must minimize its overhead. Furthermore, if the information is recorded for each external request to the system, it is possible that instrumented nodes may run out of storage in applications with high throughput. So the framework must also use as little storage as possible to record information while still allowing for accurate replays.

- **Incremental adoption**. To allow gradual adoption of the framework in production systems, the framework should allow for interoperability between systems and services in the system that are instrumented and those who are not. Unfortunately, this might result in information being lost and interfering with the accuracy of the replay executions. But it is a necessary trade-off in order to encourage its utilization in systems that are already in place and would benefit from it.

- **Accurate Replay**. The usefulness of the framework is related to the accuracy of the replays. If the replay execution is not faithful to the original, then bugs that have manifested in the original execution might not in the replay. The internal state of the system in the replay execution must match the original state so that developers are not misled in their debugging efforts.

- **Cross-platform compatibility.** Often the development and the deployment of programs happen in different platforms and operating systems. In order for a tool like this to be useful, it must support the largest possible number of platforms.

There are many challenges in implementing such a framework. The following are some of the most prevalent:

- **Minimizing storage requirements.** Compression can only go so far when it comes to minimizing the size of the data that needs to be stored so that enough information is present to allow accurate replays.

  A possible approach to this issue is to frequently copy the collected information to centralized storage, freeing the nodes in the system from the burden of storing it for longer periods of time.

- **Minimizing network overhead.** The *liblog* library adds 16 bytes to each message [GASS06] in order to establish a partial ordering between the events in the system. Unfortunately, this might make it unfit for certain classes of applications. The framework must avoid this issue by striving to keep the network overhead to the bare minimum possible.

- **Replaying multi-threaded applications.** It is impossible to control context switches from userland. Other mechanisms must be used in order to guarantee accurate and deterministic replays of multi-threaded applications.

Solving the aforementioned challenges is crucial to develop a solution that can be used without constraints in distributed systems based in the microservice architecture.

5.3 Overview of the Approach

A program can be thought of as a function that yields an execution from a set on inputs. An input to a program is any information that is supplied to it and originates outside it. An execution of a program is a subset of expressed behaviors from all the behaviors that have been encoded in its source code. The subset of expressed behaviors that determine an execution is, then, the result of the execution of the program with a set of inputs. If the same program is executed with the same inputs, the result will be the same execution, or in other words, the same subset of expressed behaviors.

Therefore, creating a reproduced execution from a reference execution entails executing the original program in such a way that the subset of behaviors expressed in both is the same. This means executing the program that yielded the reference execution with the inputs that were provided to the program to generate it. The first challenge is, then, to determine and register these inputs so that they can be provided to the program again.

There are many different types of inputs such as program arguments, user interfaces, program communication, among many others. However, all inputs have in common the fact that they are exogenous elements that are supplied to the program in different ways but that, in one way or another, depend on making a system call. System calls are the way in which a program can request a service from the kernel. The kernel provides the following services through system calls [opea]:

- **Process control**. This includes operations such as loading and executing programs, creating and terminating processes and event allocating and freeing memory.

- **File management**. Operations such as creating and deleting files, opening and closing them and even reading and writing files.

- **Device management**. Among others, this includes such operations as requesting and releasing devices, or attaching and detaching them.

- **Information maintenance**. Operations that handle system information such as getting and setting time and date, and managing other system data.
- **Communication**. This includes operations such as creating and deleting connections over the network and transmitting and receiving messages.

The result of the invocation of a system call is determined by the state of the operating system and the environment. Considering a program that wishes to open and read the contents of a file: in order to do so, it must first invoke the appropriate system call in order to open the file and get its file descriptor. The result of the system call, which can be either the file descriptor or some sort of error, can be seen as an input to the program. This means that the same system call may result in different inputs when under different contexts. Thus in order to replay an execution, the inputs that yielded it must be determined. That is done by recording the result of the system calls made during that execution.

However, the solution described in this document does not intercept and record the result of system calls. It is designed to work specifically for Java and the Java Virtual Machine, which provides an abstraction layer over the system calls provided by each platform. The solution makes use of that abstraction layer instead of directly intercepting system calls. In fact, it goes a few steps further and instead intercepts and records the results of some key methods in core classes of the Java Runtime Environment that provide functionality such as network communication and random number generation. This has the obvious disadvantage that it does not work for programs written in other programming languages, but it also brings some advantages:

- **Portability**. If the JVM supports a platform, then the solution presented here also works on that platform. This ensures that the tool works on a large number of platforms as enunciated in the previous section.
- **Effort**. Instrumenting core classes of the JRE instead of system calls for all platforms requires less programming effort and knowledge of each platform. Furthermore, it permits the usage of tooling developed specifically for Java programs.

Having determined the inputs that must be supplied to the program, in order to reproduce a reference execution it is necessary to ensure that those inputs are supplied to the program for a second time. Considering the nature of the inputs explored in this section this is not always trivial. Evidently, attempting to send or receive a message over the network may yield radically different results depending on the state of the world and the OS at any given time. Replaying an execution, however, requires that all attempts yield the same result. There are a few possible ways of ensuring this:

- **Reproduce the full state of the environment**. This is the optimal approach, but evidently impossible to achieve. Of course, in practice, it wouldn't be necessary to guarantee the same state of the whole environment, just for the services relevant for the execution. Even that, however, is hardly trivial. It is very difficult, for example, to guarantee that a service will always be available over the network and that it will always remain unchanged. Transient failures are hard to account for, and latency may vary wildly.

In some cases, however, this may be a viable option. It is comparatively easier for example to guarantee that a file will be available for reading, or that a user will always click the same button.

- **Intercept and replace the result of key method calls**. This approach foregoes the burden of trying to ensure the same state of the world, at the cost of decreased reliability. It may be the only option however to deal with cases such as network configuration in which it would be difficult to guarantee the exact same conditions between executions.

  The difficulty of this approach is that it requires identifying these key methods and instrumenting the source code in such a way that their behavior and return values can be modified.

- **Hybrid approach**. Alternatively, it is possible to mix both approaches. Easily reproducible inputs are supplied to the program by ensuring the same state of the world in both executions. On the other hand, inputs that are hard or impossible to replicate that way are supplied to the program by intercepting and replacing the behavior and result of key method calls.

  This approach is a compromise between reliability, which is best achieved through reproducing the state of the environment, and the relative ease of ensuring the same results by intercepting and replaying key method calls.

The solution described in this document takes the hybrid approach to solve this problem. It assumes that inputs that are easily reproducible between executions remain the same, and deals with the others by modifying the appropriate method calls. Specifically, the solution instruments the following input sources:

- **Communication**. A fundamental part of microservices. Systems based on the microservice architecture exchange a large number of messages between component services, so ensuring that messages can be accurately replayed is a major concern in the proposed solution.

- **Random number generation**. Another input source that is common and fundamentally non-deterministic. If a reliable replay is to be achieved, the random number generation must be controlled.

5.4 Instrumenting Network Communication

Communication over the network can be done using a great variety of protocols. In order to encompass a greater number of use cases, it is important to carefully select the point of instrumentation so that the maximum number of protocols are supported. The Internet protocol suite defines four distinct layers [Nie94], each one with a different number of protocols. These are the

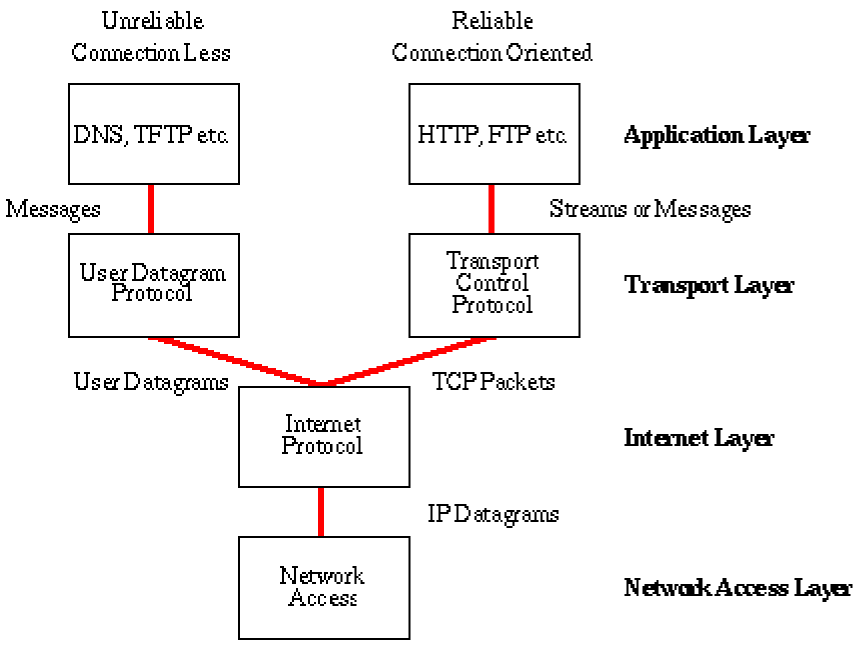


Figure 5.1: The Internet protocol suite [int]

application layer, the transport layer, the Internet layer, and the network access layer. The foundational protocol of the internet is the Internet Protocol, or IP, and stands on the Internet layer of the Internet protocol suite. That layer is the narrowest in terms of the number of protocols. However, there are not too many applications that use IP directly. Instead, most use the transport layer protocols, namely the Transmission Control Protocol, or TCP, and the User Datagram Protocol, or UDP. For that reason, the solution in this document instruments communication that happens at the transport layer. This strikes a compromise between the number of protocols that must be instrumented, which would be too many in the application layer, and ease of implementation that would be hampered had the instrumentation been done at the Internet layer.

There is another issue with instrumenting network communication at the transport layer. The lower the layer in the Internet protocol suite, the less information is available. This means that in exchange for simplicity in the implementation since only two protocols are handled as opposed to many more, some information is given up. This might affect the accuracy of the replay to some degree, although it is unlikely because it is possible to determine the right time to replay a message without knowing the structure of its contents.

5.5 Instrumenting Random Number Generation

Random number generation is comparatively simple to instrument. Unlike the case of network communication, there are not many ways to generate random numbers. Usually, random numbers are generated either through hardware random number generators, that generate genuinely random numbers, or mathematical pseudo-random number generators. Regardless of the underlying mechanisms, random number generators are typically abstracted behind a few interfaces that provide the service. That is certainly the case for Java and the JRE. These interfaces are promising points of instrumentation.

Chapter 6

A Prototype for Record and Replay

This chapter provides a detailed description of the implementation of a prototype of the solution proposed in Chapter 5. This implementation is used to validate the conceived approach in a case study described in Chapter 7-

6.1 Overview

The solution proposed in this document is a record and replay agent for Java programs according to the approach described in the previous chapter. The solution takes the form of a Java Instrumentation API agent, distributed as a JAR file with all necessary dependencies. The agent has two different but complementary modes of operation:

- **Record mode**. If in record mode, the agent records the inputs to the program that are necessary to replicate the execution in the future.
- **Replay mode**. In replay mode the agent uses the inputs recorded during the recording phase and supplies them to the program, effectively reproducing the original execution.

The structure of the system is different based on the mode of operation. In record mode, the main objective is recording the inputs to the program to ensure that the behavior will be the same during the replay. However, it must also strive to maintain low overhead and, importantly, ensure that the program being instrumented will not fail due to the agent. This is essential in order to allow the agent to be attached to programs running in production environments, in which any downtime may affect many users and be very costly. The architecture of the system consisting of the program under test and the agent in record mode can be observed in Figure 6.1, and the individual components of the system are described in Table 6.1.

In replay mode, the focus of the agent is on ensuring the accuracy of the replay, and not necessarily on minimizing overhead or storage space since it is assumed that the replay happens in

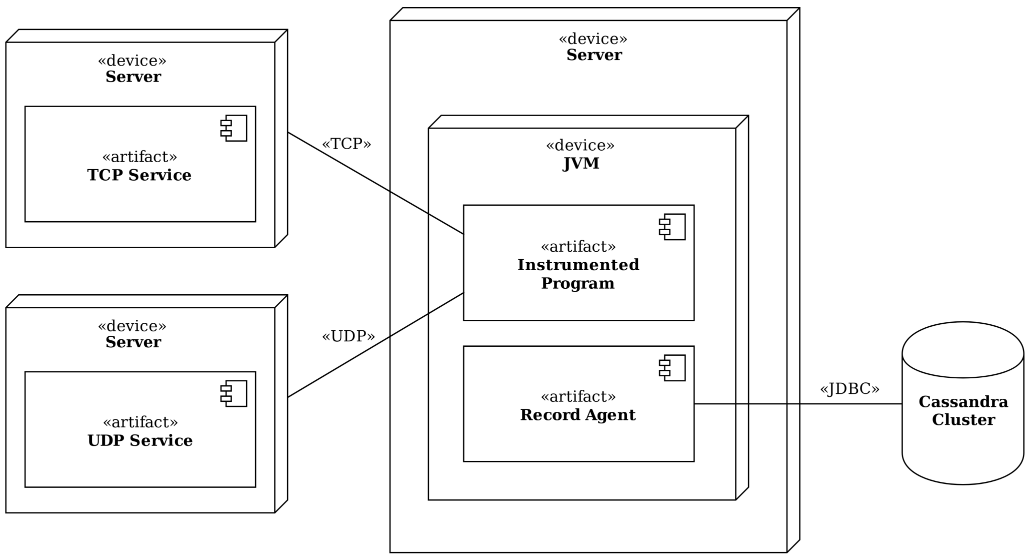


Figure 6.1: The architecture of the system in record mode.

Table 6.1: Description of the components of the system in record mode.

Component Name	**Description of Component**
Server	A generic hardware device in which one or more services are deployed. Can be a machine that is available over the local network or over the internet, for example in a data center.
JVM	The Java Virtual Machine that the Java record agent is instrumenting. Classes loaded by this virtual machine will be transformed by the agent.
Instrumented Program	The program under scrutiny that is running in the instrumented JVM. This is the program whose execution the programmer wishes to record and, eventually, replay.
Record Agent	The Java instrumentation agent that transforms the classes loaded into the JVM in order to record messages exchanged over the network and random number generation.
TCP Service	A generic service that communicates with the instrumented program via a TCP connection.
UDP Service	A generic service that communicates with the instrumented program via UDP datagrams.
Cassandra Cluster	The centralized storage in which the records generated during the recording phase of the agent are stored. This is an Apache Cassandra cluster.

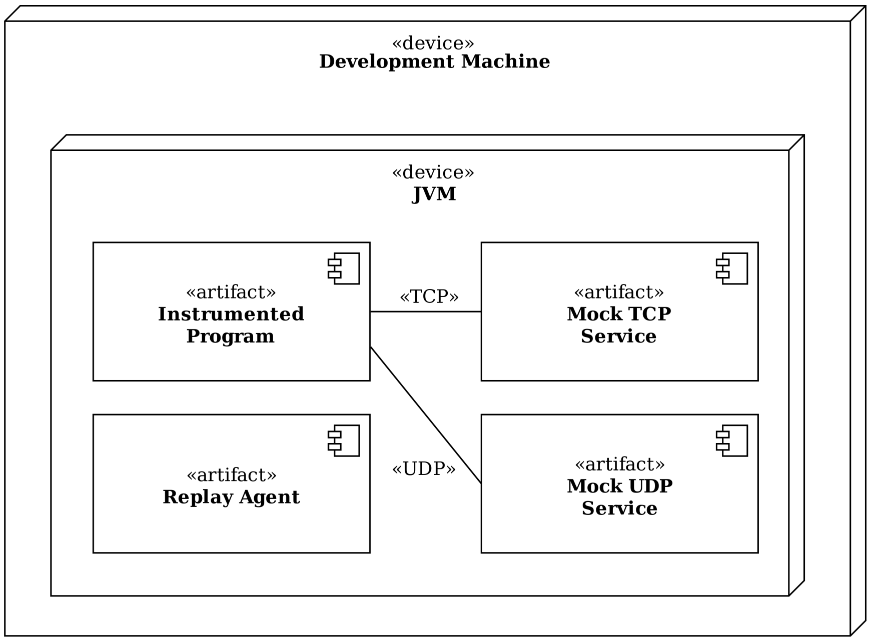


Figure 6.2: The architecture of the system in replay mode.

a controlled environment such as the programmer's development machine. Furthermore, while not ideal, it is acceptable for the program under test to crash due to a problem in the replay agent. The architecture of the system in record mode is shown in Figure 6.2. Table 6.2 contains a description of the components in the system.

Each mode of operation is implemented separately, but they share a common interface: the record object. The record object contains information about one input to the program, and all necessary metadata to match the record to the place in which it was recorded.

6.2 The Record Object

As explored in the previous chapter, the framework focuses on collecting two types of inputs that are difficult to reproduce. These inputs are messages exchanged during communication and random number generation. As such there are two types of records:

- **Communication records**. Communication records hold information about messages exchanged in the system.

  The information stored in communication records is the following:

  - *Request payload*: A collection of bytes written to a socket by the instrumented program in the case of a request originating from the instrumented program, or a collection of bytes read from a socket by the instrumented program, in the case of a request originating externally.

Table 6.2: Description of the components of the system in replay mode.

Component Name	**Description of Component**
Development Machine	The programmers development machine. In reality, it can be any environment, but the agent is most useful in replay mode when executed under a controlled environment.
JVM	The Java Virtual Machine that the Java replay agent is instrumenting. Classes loaded by this virtual machine will be transformed by the agent. This JVM may be attached to a debugger in order to inspect the internal state of the instrumented program.
Instrumented Program	The program under scrutiny that is running in the instrumented JVM. This is the program whose execution the programmer wishes to replay.
Replay Agent	The Java instrumentation agent that transforms the classes loaded into the JVM in order to replay the inputs that have been generated during the recording phase.
Mock TCP Service	A mock service that simulates the original services that communicated with the program via TCP connections. Necessary because the original dependencies of the program may not be available during the replay.
Mock UDP Service	A mock service that simulates the original services that communicated with the program via UDP datagrams. Necessary because the original dependencies of the program may not be available during the replay.

- *Response(s) payload(s)*: A list of collections of bytes. These bytes have either been read from a socket by the instrumented program in the case of a response to a request that it has initiated, or written to a socket by the instrumented program in the case of a response to a request that originated externally.

  A list is used instead of just a collection of bytes to account for the cases in which a request has more than one response or the cases in which the response comes in fragments, and thus requires more than one read or write to a socket.
- *Response(s) delay(s)*: A list of 64-bit integers, each representing the time difference in milliseconds between the moment in which the interaction was initiated and the moment in which the response was received. The order of the values in this list corresponds to the order of the responses in the response payloads list, such that the first delay value corresponds to the first response payload and so on.
- *Hostname*: A string value that represents the name of the host to which the message was addressed.
- *Port*: A 32-bit integer that represents the port in the host to which the message was addressed.
- *Type*: An enumeration that can take the values of *Inbound* or *Outbound*. *Inbound* records represent an interaction that was initiated by an external source. *Outbound* records represent an interaction that was initiated by the instrumented program.

  - *Protocol*: The transport layer protocol that was used to transmit the message.

- **RNG records**. RNG records hold information about random number generation and the order in which the number has been generated in the program.

  The information stored in RNG records is the following:

  - *Next*: The randomly generated number.

Furthermore, both record types share the following metadata fields:

- **Execution ID**: A randomly generated UUID that uniquely identifies an execution.
- **Execution name**: A human-readable name that is used to quickly and intuitively identify an execution by the programmers using the tool.
- **Timestamp**: A timestamp of the exact moment that the record has been generated.

Instances of the record object are created when the agent is running in record mode. When the agent is running in replay mode it takes a list of record objects and uses them to supply the program with the necessary inputs in order to replay the original execution.

The record objects are defined as Protocol Buffer [Pro] messages. The Protocol Buffer library is used to generate the Java classes for the record objects as well as to serialize and de-serialize the objects into JSON and the custom Protocol Buffer format.

6.2.1 Storing Records

Depending on the number of messages exchanged in the system, a large number of records may be generated. Because of that, it is not possible to store records locally in each machine that produce them, because that may cause the machine to run out of disk space and fail unexpectedly. Instead, records must be offloaded to centralized storage designed to handle large amounts of data. Having the records in a centralized space also makes them easier to access.

Given these restrictions, the records are stored in an Apache Cassandra cluster which is a robust database designed specifically to hold large amounts of data. In addition, Cassandra also provides fault-tolerance, high scalability, and high availability.

6.3 The Record Phase

Logically, in order to replay an execution, the first step is to record its inputs. In order to do that, the agent must be executed in record mode. The purpose of the agent in record mode is to generate and store record objects that represent the inputs to the program. To this end, the agent instruments key points in the application where these inputs are supplied to the program. The way the agent instruments these key points is through bytecode manipulation, functionality offered by the Java instrumentation API and the Byteman [Byt] framework.

As in the previous section, there are two kinds of records that the agent generates: Communication records and RNG records. For each type of record, there are some core JDK classes that must be instrumented.

6.3.1 Generating Communication Records

Although communication records are grouped together and represented by the same record object, there are actually two different types of communication records, based on the protocol used to transmit the message: TCP and UDP records. These records have the same fields and are distinguished only by the value of the *protocol* field.

Each communication record stores both the request and the response of an interaction. This can be either a request from an external source to the program and the subsequent response or a request from the program to an external source and the subsequent response. Of course, messages do not always elicit a response: In these cases, a record is only generated if the message was sent from an external source and addressed to the program; otherwise, it is not. The reasoning behind it is that if the message is sent from the instrumented program and has received no response, then during replay when the program sends the message again no response will need to be supplied by the agent, because there was no response in the original execution.

There is the need, then, to match the requests to the corresponding responses so that they can be stored under the same record. This is challenging for two reasons:

- The exchange of messages is asynchronous so the order of events cannot be used to determine the relations between messages. The response to a message can arrive earlier than the response to a message that was sent before it. Furthermore, some messages elicit no response further complicating the situation.
- There is no knowledge of the application level protocols, so the payload of the messages cannot be used to determine whether or not they are a new request or just a response to a previous message. In fact, the payload is, from the point of view of the agent, just a meaningless string of bytes that carries no valuable information for its purpose.

  This precludes one of the most common strategies to associate requests and responses which is embedding in the request a unique per-execution value that is echoed in the response. This is the strategy used, for example, by the Apace Kafka protocol [kaf], in which this value is referred to as correlation id.

Because of these restrictions and the lack of information available at the points of instrumentation the agent solves the problem imperfectly: The relation between messages are established per socket by the order that the messages are sent and received. The way this works is the following:

1. A map is maintained that associates socket object and record objects. Two maps are maintained by the record agent, an inbound map and an outbound map. These maps associate

socket objects and record objects. One map is used exclusively for outbound records, meaning records that represent requests that have been initiated by the instrumented program. The other map is used exclusively for inbound records. Both maps are empty in the beginning.

2. When a message is received, or in other words, when a byte string is read from a socket, then the maps are updated: First the outbound map is queried to see if there is already an entry for the socket from which the bytes have been read. If there is, then that means that the message is a response to a previous request. That entry, a record object, is then updated with the bytes read from the socket. If, on the other hand, there is no entry in the outbound, then that means that the message is a request originating from an external source. In these cases a record object is created and added to the inbound map in association with the socket.

3. Similarly, when a message is sent, or in other words, when a byte string is written to a socket, the maps are updated in much the same way. If there is an entry in the inbound map for that socket, then it means that the message is a response to a previous request from an external source, and the byte string is registered in the record object. On the other hand, if there is not an entry in the inbound map then that means that it is a request being initiated by the instrumented program. In such cases, the record object is created and added to the outbound map in association with the socket.

The instrumentation points are different depending on whether the protocol used to transmit the message is TCP or UDP.

6.3.1.1 TCP Instrumentation

There are a few classes in the JDK that are used by most programs to handle TCP traffic, either directly or indirectly through higher-level libraries and frameworks. Specifically these classes belong to the `java.net` and `java.nio` packages. The agent modifies these classes, in order to correctly instrument the largest number of programs possible. Of course, it would most likely be possible to achieve TCP communication without using these classes and in such cases, the agent will be of little help. However, the assumption was made that that is not the typical approach when developing Java programs, and this assumption seems to be justified as can be observed by the large number of protocols correctly instrumented in Chapter 7.

In order to collect enough information to replay the execution, the following operations must be intercepted:

- **Writing to a socket.** At this point, the information available should be the bytes being written to the socket and the address of the socket.
- **Reading from a socket.** At this point, the information available should be the bytes being read from the socket, as well as its address.

To obtain this information the `java.net.Socket` and `java.nio.channels.SockatChannel` classes are instrumented.

```
RULE Instrument java.net.Socket reads
CLASS java.net.SocketInputStream
METHOD read(byte[], int, int, int)
HELPER rr.helpers.RecordHelper
AT EXIT
BIND socket = $0.socket;
     bytes  = $1;
     offset = $2;
     length = $!;
IF true
DO recordTcpInboundMessage(socket, bytes, offset, length)
ENDRULE
```

Listing 6.1: Byteman rule used to instrument reads for instances of `java.net.Socket`.

The first class instrumented by the agent in order to intercept TCP communication is the `java.net.Socket` class. Objects of this class provide input and output streams to respectively read and write messages to the socket. These streams are returned by the `getInputStream` and `getOutputStream` methods of the `java.net.Socket` class, which return instances of `java.io.InputStream` and `java.io.OutputStream` respectively. However, the actual objects that are returned by these methods are instances of the `java.net.SocketInputStream` and `java.net.SocketOutputStream` classes, which are internal classes and also sub-classes of the `java.io.InputStream` and `java.net.OutputStream` classes. In order to intercept reads and writes to sockets the agent must, then, detect reads and writes to input and output streams, but only those that belong to sockets. The way it does so is by intercepting reads and writes to streams that are instances of these internal classes.

The class `java.net.SocketInputStream` offers a `read` method to read from the socket. This is the method instrumented by the agent in order to intercept messages read from objects of the class `java.net.Socket`. The agent modifies the `read` method such that just before it returns, a piece of code is executed that takes the bytes that were read from the socket and generates the appropriate record object. The Byteman rule to modify the bytecode of this method is shown in Listing 6.1 (the syntax of Byteman rules is explained in Chapter 2). The class `java.net.SocketOutputStream` offers a `write` method to write to the socket. This method is instrumented by the agent in order to intercept messages written to objects of the class `java.net.Socket`. The agent modifies the `write` method such that at entry, before the first instruction of the method is executed the appropriate record object is created, using the bytes that are being written to the socket, as well as information from the socket itself, such as the destination address. The Byteman rule used to intercept this method is shown in Listing 6.2.

Another class that is commonly used to achieve TCP communication in Java programs is `java.nio.channels.SocketChannel`. Unlike `java.net.Socket`, this class provides methods to read and write messages, without having to use input and output streams. These methods are aptly called `read` and `write`, and are the methods that the agent modifies in order to generate

```
RULE Instrument java.net.Socket writes
CLASS java.net.SocketOutputStream
METHOD write(byte[], int, int)
HELPER rr.helpers.RecordHelper
AT ENTRY
BIND socket = $0.socket;
     bytes  = $1;
     offset = $2;
     length = $3;
IF true
DO recordTcpOutboundMessage(socket, bytes, offset, length);
ENDRULE
```

Listing 6.2: Byteman rule used to instrument writes for instances of `java.net.Socket`.

the appropriate record objects. The `read` method is modified such that just before it returns the bytes that were read from the socket, as well as information from the socket itself, are used to generate a record object. On the other hand, the `write` method is modified in much the same way, except that the record is generated just before the first instruction of the method is executed and uses the bytes that are being written to the socket. The Byteman rules used to instrument these methods are shown in Listing 6.3.

6.3.1.2 UDP Instrumentation

In Java programs, UDP communication is most commonly achieved through the usage of the `java.net.DatagramSocket` class. This class offers two methods, `send` and `receive` to send and receive datagram packets, which are instances of the class `java.net.DatagramPacket` class. These methods are the points of instrumentation for the recording of UDP messages. The Byteman rules used can be found in Listing 6.4. The `send` method is instrumented in such a way that just before the first instruction of the method is executed, the bytes that are being sent, as well as information from the packet such as the destination address, are used to generate a record object. The `receive` method is modified in much the same way, except that the record object is generated just before the method returns using the bytes read from the socket.

6.3.2 Generating RNG Records

Records of random number generation are simpler to generate than communication records, both because there is less information required, as can be observed in the structure of the RNG record object, but also because the process is completely synchronous. In Java, random numbers are typically generated using the `java.util.Random` class. This class has an internal method called `next` that is responsible for the generation of random numbers. The agent uses this method as a point of instrumentation for random number generation. The Byteman rule used is shown in Listing 6.5. The `next` method is modified so that after the number is generated and just before the method returns, that number is used to create the appropriate record object.

```
RULE Intercept java.nio.channels.SocketChannel reads
CLASS ^java.nio.channels.SocketChannel
METHOD read(java.nio.ByteBuffer)
HELPER rr.helpers.RecordHelper
AT EXIT
BIND socket    = $0.socket();
     buffer    = $1;
     bytesRead = $!;
IF TRUE
DO recordTcpInboundMessage(socket, buffer, bytesRead)
ENDRULE

RULE Intercept java.nio.channels.SocketChannel writes
CLASS java.nio.channels.SocketChannel
METHOD write(java.nio.ByteBuffer[])
HELPER rr.helpers.RecordHelper
AT ENTRY
BIND socket  = $0.socket();
     buffers = $1;
IF TRUE
DO recordTcpOutboundMessage(socket, buffers)
ENDRULE

RULE Intercept java.nio.channels.SocketChannel writes (1)
CLASS ^java.nio.channels.SocketChannel
METHOD write(java.nio.ByteBuffer)
HELPER rr.helpers.RecordHelper
AT ENTRY
BIND socket = $0.socket();
     buffer = $1;
IF TRUE
DO recordTcpOutboundMessage(socket, buffer)
ENDRULE
```

Listing 6.3: Byteman rules used to instrument `java.nio.channels.SocketChannel`.

```
RULE Instrument java.net.DatagramSocket send
CLASS java.net.DatagramSocket
METHOD send(java.net.DatagramPacket)
HELPER rr.helpers.RecordHelper
AT ENTRY
BIND socket = $0;
     packet = $1;
IF true
DO recordUdpOutboundMessage(socket, packet)
ENDRULE

RULE Instrument java.net.DatagramSocket receive
CLASS java.net.DatagramSocket
METHOD receive(java.net.DatagramPacket)
HELPER rr.helpers.RecordHelper
AT EXIT
BIND socket = $0;
     packet = $1;
IF true
DO recordUdpInboundMessage(socket, packet)
ENDRULE
```

Listing 6.4: Byteman rules used to intercept sending and receiving of UDP packets.

```
RULE Instrument java.util.Random next
CLASS java.util.Random
METHOD next(int)
HELPER rr.helpers.RecordHelper
AT EXIT
BIND next = $!;
IF true
DO recordRNG(next)
ENDRULE
```

Listing 6.5: Byteman rule used to intercept random number generation.

6.4 The Replay Phase

After the execution has been recorded, it is possible to replay it. This is done by running the agent in replay mode. The purpose of the agent in replay mode is to supply the program under test with the inputs that were collected during the recording phase so that an execution can be accurately replayed. Similarly to what happens during the recording phase, the agent makes use of the Java Instrumentation API and byte-code manipulation to instrument strategic points of the program.

During the recording phase, a large number of records are created corresponding to the whole execution of the program. However, sometimes it is enough to replay just a small part of the execution. As an example, requests from clients to servers are often independent in the sense that their result is not related to the result of previous or subsequent requests. In those cases, it might be interesting to replay the execution of just one request from the client, if that request has produced unexpected behavior while the others have not. As such, there must be a way to choose among all the records generated during the recording phase the ones that should be replayed. The chosen records can then be supplied to the agent in replay mode in the form of a replay script, which is used to replay a certain portion of an execution.

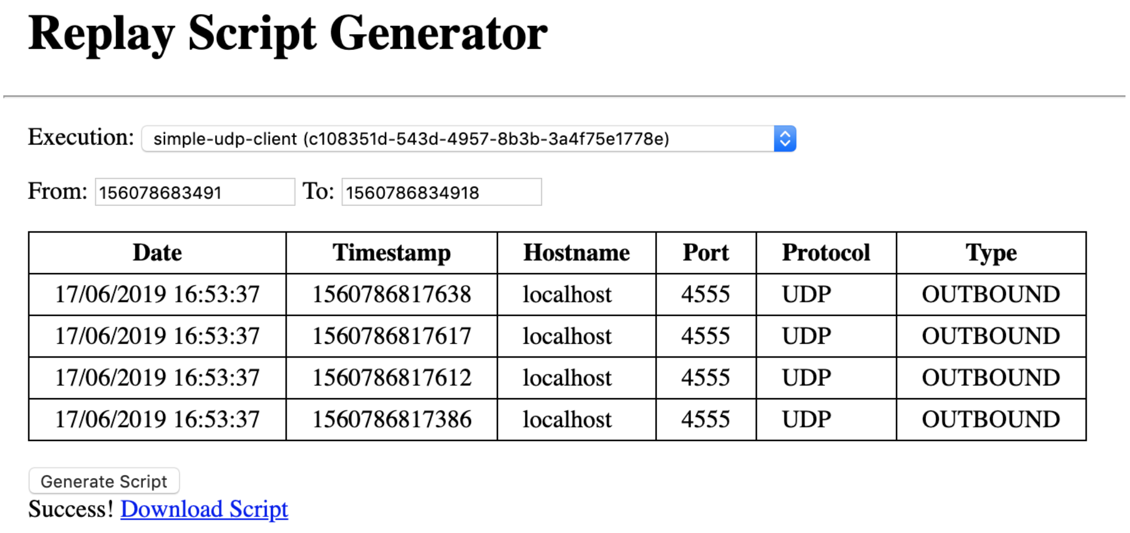
6.4.1 Generating a Replay Script

A replay script is a file that contains a sequence of records that are meant to be used as inputs in the replay of an execution. Replay scripts are generated using the Replay Script Generator, a web application that reads the records generated during the recording phase from the centralized data store, allows a user to select which ones they are interested in, creates the replay script and returns a download link to transfer the file. The replay script generator also allows the filtering of records per timestamp and per execution, using the execution name and id to identify it.

The interface of the replay script generator is shown in Figure 6.3. The list of records shows some useful information that may help in identifying which records the programmer wants to reproduce, but the information is very limited because the agent that generated the records has no knowledge of the specifics of the program. Instead, this tool is meant to be used in conjunction with other monitoring tools in the system. After identifying an anomaly using the monitoring tools in the system, the replay script generator could be used to generate a script that would replay the execution that happened at the time that the unexpected behavior occurred.

6.4.2 Replaying the Execution

Once the replay script has been generated and downloaded, the program can be initiated with the agent in replay mode. When the agent is started in replay mode it shows a very simple GUI that prompts the user to load the replay script. The execution of the program is halted until the replay script has been loaded. Once the replay script has been loaded, the agent proceeds with the startup process.



Replay Script Generator

Execution: simple-udp-client (c108351d-543d-4957-8b3b-3a4f75e1778e)

From: 156078683491 To: 1560786834918

Date	Timestamp	Hostname	Port	Protocol	Type
17/06/2019 16:53:37	1560786817638	localhost	4555	UDP	OUTBOUND
17/06/2019 16:53:37	1560786817617	localhost	4555	UDP	OUTBOUND
17/06/2019 16:53:37	1560786817612	localhost	4555	UDP	OUTBOUND
17/06/2019 16:53:37	1560786817386	localhost	4555	UDP	OUTBOUND

Success! Download Script

Figure 6.3: The replay script generator web application

6.4.2.1 Replaying Network Communication

The main challenge in replaying network communication is to match the messages sent and received by the program to the records that were generated by recording a previous execution. In order to create this correspondence, it is necessary to use a heuristic based on the information that is available both at the time of recording and at the time of the replay:

- **The remote socket address of the request**. This can be either the destination address in the case of a transmission or the source in case of reception. The address of the request is the same during the recording phase and the replay phase, otherwise, it is probably not the same request. The address is composed of both the host address and the port.

- **The payload of the request**. The payload of the request must also be the same during the recording phase and during the replay phase in order to establish the correspondence.

- **The protocol**. The protocol, either TCP or UDP, used to transmit the message should also be the same in the recording phase as in the replay phase.

Using that information, and the order in which messages are sent and received, it is possible to create the correspondence between the messages sent and received by the program in the replay phase and the records collected during the recording phase. The process by which this correspondence is made is the following:

1. The replay script is loaded, and the records are stored in a map of unbound records. This map creates a correspondence between record keys, which are tuples consisting of the address and payload of the message that originated the record, and a chronologically ordered queue

of unbound record objects. Unbound records are records that have not yet been associated with any socket, or, in other words, that have not yet been used during the replay.

In addition to the map of unbound records, the agent maintains a map of bound records, which are record objects that have been associated with a socket during the replay. The map of bound records is empty.

2. When a message is transmitted, the map of bound records is first queried to check whether or not there is an entry for the socket used for the transmission. If there is, then that means that this message is the response to a previous request that originated from an external source. In such cases, nothing needs to be done by the agent.

   If, on the other hand, there is no entry for the socket then it means that this message is a request from the program under test to an external service, and it is necessary to find the record that corresponds to this request. To do so, a record key is created using both the remote address of the socket, in this case, the destination address, and the request payload. That record key is then used to find the record in the map of unbound records that matches the request. If a record is found then that record is inserted in the bound records map in association to the socket.

3. When a message is received, the map of bound records is queried to check whether or not there is an entry for the socket through which the message was received. If there is not, then that means that this message is a request originating from an external source. In such cases, the agent does not need to do anything, since the program will most likely handle the request in the same way that it did in the original execution.

   On the other hand, if there is an entry in the bound records map for the socket, then it means that the message is a response to a previous request from the program under test. In such cases, the agent uses the record found in the map to replay the correct response.

There is yet another challenge encountered when replaying network communication, that is that if the services that the program under test depends are not available over the network, then the socket connections will fail and the replay agent will subsequently fail to reproduce the original execution. In order to avoid this problem, the replay agent runs two mock servers, one that accepts TCP connections and another that accepts UDP packets. The replay agent also ensures that every request from the program under test to one of its dependencies is redirected so that it instead arrives at the appropriate mock server. In this way, the socket connections are always established without error, and the replay agent may proceed with the replayed execution.

This, however, raises another problem in the way that records are matched to the messages transmitted by the program. The reason is that, as explained before, the records are matched to requests using the remote address of the socket and the payload of the message. If the remote address of the socket is modified by the agent, so that it becomes instead the address of the mock server, then it becomes impossible to find the record that corresponds to the message since the remote address will no longer be the same as it was during the recording phase.

```
RULE Instrument java.net.Socket reads
CLASS java.net.SocketInputStream
METHOD read(byte[])
HELPER rr.helpers.ReplayHelper
AT EXIT
BIND socket = $0.socket;
     buffer = $1;
IF true
DO RETURN interceptTcpInboundMessage(socket, buffer)
ENDRULE
```

Listing 6.6: Byteman rule to intercept reads for `java.net.Socket`

The agent addresses this issue by maintaining a map between sockets and their original addresses. This map is originally empty and is updated every time the program transmits a message and just before the message is redirected to the correct mock server. This map is then queried every time that the original remote address for a socket is required, namely to generate the correct record keys.

Replaying TCP Communication

In order to replay TCP communication, the replay agent instruments the program in many of the same places that the record agent does in order to record TCP communication.

The first classes instrumented by the agent are, then, the ones corresponding to the input and output streams of the `java.net.Socket` class. These are `java.net.SocketInputStream` and `java.net.SocketOutputStream`, respectively. The `read` method of the input stream is instrumented in such a way that just before it returns, the contents of the buffer that holds the bytes that have been read from the socket are replaced with the contents of the bytes that have been saved in the corresponding record object, and the return value of the method is also replaced by the length of the replayed message. The Byteman rule used to intercept this method is shown in Listing 6.6. The write method of the input stream is modified such that, just before its first instruction, the record that matches the transmitted message is found and subsequently bound to the socket in the bound records map. The Byteman rule used to achieve this modification is shown in Listing 6.7.

The other class that is instrumented by the replay agent is java.nio.channels.SocketChannel. As observed in previous sections, this class provides methods to read and write to the socket, called `read` and `write` respectively. These methods are instrumented by the replay agent in order to inject the appropriate messages in the socket. The `write` method is instrumented by the agent so that before the first instruction in the method, the socket is associated with the appropriate record in the bound records map. The read method, on the other hand, is instrumented in such a way that just before it returns the message in the buffer is replaced with the message that is found in

```
RULE Instrument java.net.Socket writes
CLASS java.net.SocketOutputStream
METHOD write(byte[], int, int)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket = $0.socket;
     buffer = $1;
     offset = $2;
     length = $2;
IF true
DO interceptTcpOutboundMessage(socket, buffer, offset, length)
ENDRULE
```

Listing 6.7: Byteman rule to intercept writes for `java.net.Socket`

the corresponding record and the return value is modified to be the length of the injected message. The Byteman rules used to do it are shown in Listing 6.8.

Furthermore, it is also necessary to instrument the point at which the sockets are connected in order to both save the original address and to redirect the socket connection to the mock server. This is, however, not done for all the addresses. It is possible to configure the replay agent so that some addresses are not redirected and thus messages exchanged with services available at that address happen as normal.

For the `java.net.Socket` objects, the connection is established either by invoking the constructor or by invoking the `connect` method. The constructor is instrumented so that before any instruction in it is executed the original address is saved and the remote address for the socket that has been passed as the argument to the constructor is modified to point at the mock server. Some of the Byteman rules used to achieve this can be found in Listing 6.9. For objects of the class `java.net.Socket` that are created unconnected, the `connect` method is instrumented so that the original address is saved and the remote address for the socket that has been passed has an argument to the method is modified. The Byteman rule that intercepts this call is shown in Listing 6.10.

In the case of the `java.net.SocketChannel` class, the `connect` method is modified by the replay agent in such a way that before the first instruction is executed the original address is saved and the remote address for the socket is modified. Listing 6.11 contains the Byteman rule to intercept and modify this method.

Replaying UDP Communication

In the case of UDP communication, and similarly to what is done in the record phase, the `send` and `receive` methods of the `java.net.DatagramPacket` are instrumented. It is not necessary to instrument any other point in order to obtain and save the original address of the

```
RULE Intercept java.nio.channels.SocketChannel read
CLASS ^java.nio.channels.SocketChannel
METHOD read(java.nio.ByteBuffer)
HELPER rr.helpers.ReplayHelper
AT EXIT
BIND socket = $0.socket();
     buffer = $1;
IF true
DO RETURN interceptTcpInboundMessage(socket, buffer);
ENDRULE

RULE Intercept java.nio.channels.SocketChannel write
CLASS ^java.nio.channels.SocketChannel
METHOD write(java.nio.ByteBuffer[], int, int)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket  = $0.socket();
     buffers = $1;
     offset  = $2;
     length  = $3;
IF true
DO interceptTcpOutboundMessage(socket, buffers, offset, length)
ENDRULE

RULE Intercept java.nio.channels.SocketChannel write (1)
CLASS ^java.nio.channels.SocketChannel
METHOD write(java.nio.ByteBuffer)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket = $0.socket();
     buffer = $1;
IF true
DO interceptTcpOutboundMessage(socket, buffer)
ENDRULE
```

Listing 6.8: Byteman rules to instrument `java.nio.channels.SocketChannel`.

```
RULE Instrument java.net.Socket constructor
CLASS java.net.Socket
METHOD <init>(java.lang.String, int)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket  = $0;
     host    = $1;
     port    = $2;
     address = new java.net.InetSocketAddress(getTcpMockServerHost(),
         getTcpMockServerPort());
IF addressNotExempt(address)
DO saveOriginalAddress(socket);
   host = getTcpMockServerHost();
   port = getTcpMockServerPort();
ENDRULE
```

Listing 6.9: Byteman rule to intercept constructor for `java.net.Socket`

```
RULE Instrument java.net.Socket connect
CLASS ^java.net.Socket
METHOD connect(java.net.SocketAddress, int)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket  = $0;
     address = $1;
IF addressNotExempt(address)
DO saveOriginalAddress(socket, address);
   address = new java.net.InetSocketAddress(getTcpMockServerHost(),
       getTcpMockServerPort());
ENDRULE
```

Listing 6.10: Byteman rule to intercept connect for `java.net.Socket`

```
RULE Instrument java.nio.channels.SocketChannel connect
CLASS ^java.nio.channels.SocketChannel
METHOD connect(java.net.SocketAddress)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket  = $0.socket()
     address = $1;
IF addressNotExempt(address)
DO saveOriginalAddress(socket, address);
   address = new java.net.InetSocketAddress(getTcpMockServerHost(),
       getTcpMockServerPort());
ENDRULE
```

Listing 6.11: Byteman rule to intercept connect for `java.nio.channels.SocketChannel`

```
RULE Instrument java.net.DatagramSocket send
CLASS java.net.DatagramSocket
METHOD send(java.net.DatagramPacket)
HELPER rr.helpers.ReplayHelper
AT ENTRY
BIND socket = $0;
     packet = $1;
IF true
DO interceptUdpOutboundMessage(socket, packet)
ENDRULE

RULE Instrument java.net.DatagramSocket receive
CLASS java.net.DatagramSocket
METHOD receive(java.net.DatagramPacket)
HELPER rr.helpers.ReplayHelper
AT EXIT
BIND socket = $0;
     packet = $1;
IF true
DO interceptUdpInboundMessage(socket, packet)
ENDRULE
```

Listing 6.12: Byteman rules to intercept and replay exchange of UDP messages.

packet because the address is carried together with the packet. The Byteman rules are shown in Listing 6.12.

6.4.2.2 Replaying Random Number Generation

The replay of the generation of random numbers is simpler than the replay of network communication. When the replay script is loaded, the RNG record objects are stored in a queue in chronological order, from the ones that happened first to the ones that happened last. Then, whenever the method `next` of the `java.util.Random` class is invoked, the next record is taken from the queue and that number is injected as the return value. The Byteman rule used to perform the interception can be found in Listing 6.13.

```
RULE Instrument java.util.Random next
CLASS java.util.Random
METHOD next(int)
HELPER rr.helpers.ReplayHelper
AT EXIT
BIND value = $!;
IF true
DO RETURN replayRNG(value)
ENDRULE
```

Listing 6.13: Byteman rule to intercept and replay generation of random numbers.

6.5 Usage

The agent is distributed as a JAR file along with the necessary dependencies. The most important dependency is Byteman, to perform bytecode injection and the Byteman scripts that define the rules. The structure of the directory that is distributed is:

```
dist
├── libs
├── scripts
│   ├── record.btm
│   └── replay.btm
└── rr-agents.jar
```

The entry point to the agent is the *rr-agents.jar* JAR file. The *libs* directory holds the JAR files of all the dependencies of the program, including the Byteman JAR. The *scripts* directory holds the Byteman scripts used by the agent to intercept method calls. There are two scripts, one used during the recording phase and another used during the replay phase.

The agent can be started from the command line using the *javaagent* command line switch. Generically the *javaagent* switch is used in the following way:

```
java Main -javaagent:<agent>=<options>
```

Where *agent* is the path to the agent's JAR file and *options* is a string passed to the agent. The options string must be parsed by the agent. The format of the options string for this particular agent is a comma-separated list of properties, where each property takes the form of `<key>:<value>`. The supported list of properties are:

- **mode:** String value that represents the execution mode of the agent. Possible values are *record* and *replay*. This parameter is required. If absent, the agent will throw an exception.
- **name:** String value that represents the execution name. The execution name is a human-friendly name intended to allow users to quickly identify the execution in the replay script generator. This can take any value and is a required parameter if running the agent in record mode. If absent, the agent will throw an exception.
- **tcp:** Integer value that specifies the port that the mock TCP server should use in replay mode. Meant to avoid conflicts with programs that are using the default port 9998. This is an optional parameter. If absent, the default value for the port will be used.
- **udp:** Integer value that specifies the port that the mock UDP server should use in replay mode. Meant to avoid conflicts with programs that are using the default port 9999. This is an optional parameter. If absent, the default value for the port will be used.

- **exclude:** String value representing an address to be excluded for instrumentation during replay. This parameter is optional, and may appear more than once.
- **verbose:** Meant for debugging purposes. This parameter does not require any value. This is an optional parameter. If absent, the default level of logging will be used.

For example, if the program is the main class is called `Application`, this is how the application can be started with the agent in record mode:

```
java Main -javaagent:rr-agents.jar=mode:record,name=my-app
```

Starting the program with the agent in replay mode is very similar, except that the value of the *mode* parameter is different and there is no need to specify the *name* parameter:

```
java Main -javaagent:rr-agents.jar=mode:replay
```

The same JAR file of the agent is also used to start the replay script generator application. The command line parameters required are:

- **node**: A string value, representing the hostname of the Cassandra cluster used to store the record objects generated in the recording phase of the agent. This parameter is required.
- **port**: An integer value, representing the port of the Cassandra cluster used to store the record objects generated in the recording phase of the agent. This parameter is required.

Assuming that the Cassandra cluster is available on the address *localhost:9042*, the replay script generator can be started from the command line in the following way:

```
java -jar rr.agents.jar localhost 9042
```

Chapter 7

Case Study

Throughout the development, the solution described in this document has been tested with simple programs that isolate a single aspect of the desired functionality. Some of these programs, for example, only communicate via TCP connections. Others communicate exclusively via UDP datagrams. Still others generate random numbers and displayed them in the console. Multi-threaded variations of these simple programs were also used to test the limitations of the solution.

However, in order to gain stronger confidence on whether or not the solution meets the criteria defined in Chapter 5 a case study was designed. The remainder of this chapter will describe this case study.

7.1 Description of the Service

The case study was conducted on one service of a microservice-based system. The system in question is highly distributed, is composed of hundreds of services and large numbers of messages are exchanged between them. The service under analysis in this chapter is an Apache Storm [1] topology. A topology in Apache Storm is essentially a graph that determines the way in which the data is computed. Each node in the topology has processing logic, and the topology determines the path that the data takes through the nodes. There are two types of nodes in a topology:

- **Spouts:** The source of data streams. Spouts read from an external data source and emit them to the next nodes as defined in the topology. Commonly used data sources are message queues or web APIs.
- **Bolts:** Bolts consume streams, perform computations on the data and may emit new streams. A bolt can consume one or more streams and can be followed by one or more bolts. A topology may have many bolts.

[1] Apache Storm is a distributed real-time computation system designed to process endless streams of data. Apache Storm ensures scalability and fault-tolerance making it easier for programmers to create real-time processing systems.

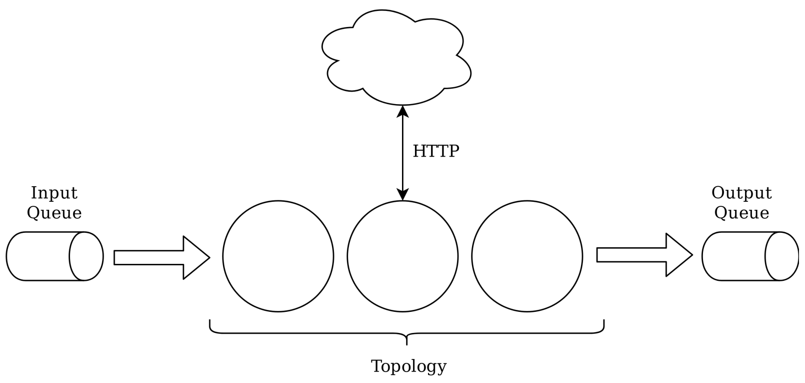


Figure 7.1: The structure of the locally deployed topology.

The Apache Storm topology used in this case study consists of a single spout and three bolts, connected sequentially. The data source for the bolt is a Kafka message queue. Each bolt does some computation on the stream. One of the bolts exchanges messages with an external web server. The last bolt in the topology writes the result of the computations to another Kafka message queue. Figure 7.1 shows the structure of the topology.

The topology used for this case study is interesting because it features different communication protocols. The spout reads from Kafka using the Kafka protocol, a custom protocol over TCP, and one of the bolts uses HTTP to communicate with a web server.

7.2 Setup

Typically, Apache Storm topologies are deployed on a cluster to become available over the Internet. This means that different bolts may execute in different nodes in the cluster. However, for practical reasons, the topology has been set up locally for this case study. The Apache Storm Java library provides this functionality through the `org.apache.storm.LocalCluster` class, which emulates a cluster in a process. It is important to note that this is not the same way that the Apache Storm executes in production. However, it is the way the topology is tested before deployment and should accurately represent its behavior in the production environment.

The Kafka message queues were made available through a Docker container. The container uses the `spotify/kafka` docker image. This image provides a Kafka cluster installation that is made available by default at port 9092 of the container. The appropriate input and output topics have been created before executing the topology. These message queues are not written to by any service. Therefore, in order to provide data for the topology a number of messages have been extracted from previous executions in production and a simple Java program has been created to inject these messages when desired. It is important to note that while the way in which the messages arrive in the input message queue is different from what it would be in production, this does not affect the execution of the topology in any way.

The web server that exchanges messages with one of the bolts in the Apache Storm topology is available over the Internet and as such required no extra setup.

There is one extra setup step required in order to be able to run the agent in record mode attached to this topology, which is setting up the data store. As described in the previous chapter, when running in record mode the agent stores the generated records in an Apache Cassandra cluster. Similarly to what is done for the Kafka message queues, the Cassandra cluster is made available through a Docker container, in this case using the `bitnami/cassandra` image.

7.3 Methodology

In order to validate the solution by the criteria defined in Chapter 5 the following metrics are measured:

- **Execution time**, with and without the agent running alongside the topology. This is intended to measure the overhead introduced by the agent in the component, and thus in the overall system.

  To this end the program was modified to measure the time between the instant that a message is read from the input Kafka queue to the moment that the message is written to the output Kafka queue.

- **Disk space** used by the generated records. It is important to measure this in order to gauge the cost of maintaining the records in storage. Depending on the results, it may or not be necessary to decrease the period of storage of each record.

  This measurement is taken by querying the Cassandra data store to obtain the amount of disk space occupied by the cluster before and after the recording has been obtained, and dividing the difference between these values by the number of records generated in order to determine the average size per record.

- **Accuracy** of the replay, or in other words, the difference between the output of the original execution and that of the replayed execution. This is important in order to determine whether or not the agent works as intended.

  This is measured by comparing the contents of the output Kafka queue between the original execution and the replay execution as well as the output logs between both executions.

The topology has been executed in three different configurations. For the first set of executions, the topology has been executed without the agent attached. Measurements taken during this executions have been used as a reference to understand the impact of attaching the agent to the program in both record and replay modes. The command used to start the topology in this configuration is:

```
java Topology
```

For the second set of executions, the topology has been executed with the agent attached in record mode in order to collect enough information to do a replay. The command used to run the topology in this configuration is the following:

```
   java Topology -javaagent:rr-agent.jar=mode:record, name:case-study,
excluded=localhost:2000
```

In record mode, the *excluded* option allows configuring the agent to not record communication with remote addresses, so as to minimize the number of records collected. In this case, `localhost:2000` excludes cluster management messages that should not meaningfully impact the execution.

The final set of executions of the topology is a replay of one of the executions of the second set. Since the address `localhost:2000` has been excluded during the recording phase it has also been excluded from the replaying phase. This means that the agent will not intercept communication with the remote address `localhost:2000` and therefore in the replay execution cluster management traffic may be different from that in the recording execution. However, this difference should have no meaningful impact. The command used to execute the program in this configuration is the following.

```
   java Topology -javaagent:rr-agent.jar=mode:replay,
excluded=localhost:2000
```

The results of these executions were recorded in files for later analysis.

7.3.1 Measuring Execution Time

The spout of the topology reads messages from the input queue and turns them into tuples which are then processed by the topology. Each message corresponds to one tuple. The execution time is measured by recording the time at the moment the tuple enters the system and subtracting it to the time at the instant the tuple is emitted by the last bolt in the topology. The measurement is made in milliseconds.

7.3.2 Measuring Storage Space Usage

The total number of bytes in use in the data store to store the records generated in an execution is recorded in bytes. The total number of record objects generated is also stored in order to calculate the average storage space used per record.

7.3.3 Computing Replay Accuracy

The logs of the topology contain the payload of the messages transmitted and received from the web server, as well as the result of the processing of each tuple by each bolt. The logs of both the reference and the replayed executions have been used to calculate the accuracy of the replays, since if the replay is faithful to the original execution, then the logs will be as well. To that end, during each execution, the logs are written to a text file. Then, the contents of the files containing the logs of each replayed execution are compared to the logs of the original execution in order to assess:

- **Differing lines:** The number of lines in the log file of the replayed execution that are different from the corresponding line in the log file of the original execution.
- **Lines out of position:** The number of lines that while not matching the corresponding line in the log file of the original execution, do match another line in that file. In practice, this usually means that the lines were logged in both executions, but at different relative times.
- **Accuracy:** A percentage value that represents the ratio between the number of lines that are present in the log file of the replayed execution but not on the original execution and the total number of lines in the log file of the replayed execution.

However, there are some parts of the generated logs that are always different between executions, even if they are an accurate replay of another. These include values such as timestamps, thread IDs, and local socket addresses, among others. This means that in order to be able to use the logs to calculate the accuracy, some post-processing is required to substitute these values.

The execution logs are stored in plain text files after each execution. The post-processing of these files is done using the stream editing tool `sed`. Transformations to be applied to the file are defined in a set of rules as described in Appendix A, and `sed` does one pass over the contents and applies them. The transformations applied to the raw log files are:

- **Removing cluster management logs**. Logs that pertain to cluster management are not meaningful for the purpose of determining whether or not the replay is accurate since messages regarding cluster management have been explicitly ignored during the recording phase.
- **Replacing local socket addresses**. Sockets are bound to different local ports in each execution. This has no impact in the accuracy of the replay, and as such all addresses in the logs are made the same.
- **Replacing object addresses**. Object addresses are also different for each execution. Again this has no impact on the accuracy of the replay so all addresses are replaced during post-processing so as to be the same.
- **Removing timing information and timestamps**. The logging framework used by the topology under study in this chapter outputs structured messages. One of the fields in these

messages is the time in milliseconds since the beginning of the execution. Because the replay agent does not guarantee that things happen at the same time during the replay as they happened during the original execution, these timestamps must be removed as they have no impact on determining the accuracy of the replay.

It is important to note that the information lost in these transformations does not meaningfully impact the measurement of the accuracy of the program.

7.4 Results and Analysis

This section contains the results obtained from the measurements described in the previous sections of this chapter.

7.4.1 Execution time

Tables 7.1, 7.2 and 7.3 contain the measurements of execution times for the reference, record and replay execution modes respectively. For each execution mode ten executions have been measured, each one processing ten tuples and thus ten messages. Table 7.4 contains a comparison between the execution time measured in each execution mode and the reference executions.

As shown in Table 7.4, it is evident that for this case study the agent in record mode introduces very little overhead, around 1.5%. This value is negligible for the execution times registered in this case study, which are of around 626 milliseconds in the reference execution, meaning that a 1.5% increase corresponds to around 6 milliseconds. This is an acceptable value and shows that the agent in recording mode introduces little overhead. In replay mode, this is not the case, as the overhead introduced is of around 74% to 84% of the original execution time. However, as explained previously in this document, the agent only executes in replay mode in controlled environments such as a programmers development machine in which the introduced overhead has no impact on the users. A replay overhead of less than 100% is acceptable for the purpose of debugging, considering that the tool is meant to be used with interactive debuggers that allow the introduction of breakpoints which make the overhead introduced negligible. Overall, the agent meets the criteria defined in Chapter 5.

7.4.2 Storage Space Usage

The storage space used by the records generated during the recording executions can be observed in Table 7.5. Storage space has not been measured for the other execution modes because they do not generate records and thus do not use any storage space.

Each record generated during recordings for this case study takes on average 12395 bytes in storage space, as can be observed in Table 7.5. This is a very significant value, considering the large number of records generated for the small number of messages processed in each execution, and the duration of each recorded execution which was around 30 seconds. In reality, most records are much smaller than that. However, some are also much bigger, resulting in a high average value.

Table 7.1: Execution times per tuple and per execution without the agent running, in milliseconds.

Execution	1	2	3	4	5	6	7	8	9	10
#1	454	513	568	634	686	741	797	854	960	1015
#2	374	424	480	534	586	644	700	757	810	866
#3	390	439	492	545	596	651	707	766	820	873
#4	337	399	463	528	589	652	717	782	845	907
#5	71	141	207	238	303	367	433	501	564	629
#6	381	442	506	570	632	697	762	828	892	956
#7	341	402	469	532	595	658	725	790	852	915
#8	458	508	565	621	674	730	787	846	900	956
#9	472	524	576	627	653	686	739	797	854	885
#10	389	438	492	549	620	672	727	784	836	890
Mean (ms)	367	423	482	538	593	650	709	771	833	889
Median (ms)	381	438	492	545	596	658	725	784	845	890

Table 7.2: Execution times per tuple and execution with the agent in record mode, in milliseconds.

Execution	1	2	3	4	5	6	7	8	9	10
#1	500	553	610	666	716	772	829	893	949	1004
#2	363	415	470	572	624	679	734	791	845	900
#3	402	454	509	563	614	669	729	786	841	891
#4	332	381	438	491	547	601	655	723	776	831
#5	359	408	463	666	742	796	850	950	1006	1060
#6	402	454	508	575	627	679	735	794	847	901
#7	304	354	408	463	515	572	630	693	748	805
#8	321	372	431	490	545	601	656	713	771	824
#9	365	416	472	528	580	635	691	749	802	868
#10	373	422	479	532	583	641	696	755	811	863
Mean (ms)	372	423	479	555	609	665	721	785	840	895
Median (ms)	365	416	472	555	609	665	721	785	840	891

Table 7.3: Execution times per tuple and execution with the agent in replay mode, in milliseconds.

Execution	1	2	3	4	5	6	7	8	9	10
#1	538	787	975	1147	1295	1469	1626	1829	1998	2164
#2	434	614	796	940	1124	1306	1456	1616	1808	2000
#3	505	661	839	974	1105	1246	1384	1518	1662	1801
#4	463	609	761	896	1030	1164	1299	1446	1630	1781
#5	434	561	704	835	965	1133	1271	1433	1599	1751
#6	377	528	694	849	993	1180	1320	1454	1597	1751
#7	449	579	725	852	980	1129	1264	1427	1597	1754
#8	420	550	691	820	943	1086	1220	1368	1548	1695
#9	614	794	955	1098	1228	1372	1530	1726	1920	2087
#10	423	551	698	840	970	1119	1259	1417	1555	1680
Mean (ms)	466	623	784	925	1063	1220	1363	1523	1691	1846
Median (ms)	442	594	743	874	1012	1172	13190	1450	1615	1768

In the end, the amount of space required per record may be a limitation for usage of the tool in production environments, and special care might have to be taken in order to limit the number of records generated or the amount of time during which the records are stored.

7.4.3 Replay Accuracy

Table 7.7 shows the comparison between the log files produced during the replayed executions and the log file produced during one of the ten recorded executions. Accuracy is the percentage of lines that show up in both log files, regardless of the relative position in which they appear.

The number of lines in the post-processed log files is just slightly lower than the number of lines in the corresponding raw log files, as can be observed in Table 7.6. This reinforces the claim that little meaningful information is lost in these transformations.

Accuracy of the replay for this case study is close to perfect, as observed in Table 7.7, meaning that the same operations that occurred in the original execution will happen again in the replay. Due to the highly concurrent nature of the program under analysis in this case study, some of the messages will be written to the log files in different positions in relation to other messages, even though it is usually very close. Furthermore, once one message appears out of position, most of the following messages will appear one line before or after the original. This explains the sometimes

Table 7.4: Comparison of execution times between different modes of execution.

	Reference	**Record**	**Replay**
Mean Execution Time per Tuple (ms)	625	634	1151
Median Execution Time per Tuple (ms)	627	637	1092
Difference of Mean to Reference (%)	0%	+1.44%	+84.16%
Difference of Median to Reference (%)	0%	+1.59%	+74.16%

Table 7.5: Storage space used per execution in record mode.

Execution	Space Used (Bytes)	Number of Records	Space Used per Record (Bytes)
#1	403768	67	6026
#2	399814	53	7543
#3	392696	28	14024
#4	392138	26	15082
#5	393280	30	13109
#6	392980	29	13551
#7	392435	27	14534
#8	391854	25	15674
#9	393274	30	13109
#10	395697	35	11305
Mean	394793	35.0	12395
Median	393127	29.5	13330

large number of lines that appear on both the original log file and the replayed log file in different positions. Regardless, the replayed execution will be very faithful to the original, thus fulfilling the main objective of the replay agent.

7.5 Conclusions

There is at this point enough information to be able to answer the research questions enunciated in Chapter 4. These answers are based on a single case study. While the program was chosen because it is a good representation of the kind of programs that this tool is intended to help debug, further case studies would need to be conducted in order to obtain more definitive answers:

- *Can enough information be recorded in an execution without introducing overhead in execution time that impacts the viability of the system?*

  The results obtained are very encouraging. As reported in this section the overhead introduced by the agent in record mode is about 1.5%, so small enough as to be negligible. However, in order to give a definitive answer to this question more services would need to be analyzed.

- *Is the amount of data generated during the execution small enough to be viably stored in either the machines executing the instrumented program or in centralized storage?*

Table 7.6: Number of lines in the log files before and after post-processing

Orignal File Name	Original Line Count	Post-Processed Line Count	Difference
record-logs.txt	1899	1781	-118
replay-logs.txt	1899	1781	-118

Table 7.7: Comparison between log files of the replayed executions and the log file of the recorded execution.

Replay	Differing Lines	Lines Out of Position	Accuracy
#1	48	48	100.00%
#2	43	43	100.00%
#3	42	42	100.00%
#4	122	122	100.00%
#5	46	45	99.94%
#6	103	102	99.94%
#7	1677	1676	99.94%
#8	44	44	100.00%
#9	37	37	100.00%
#10	1665	1665	100.00%

This looks to be the biggest problem with the approach described in this document. Due to the amount of information needed to replay the execution, the records use too much space to be viably stored. The Apache Storm topology analyzed in this case study required about 390 KB for 30 seconds of execution, which would be around 46 MB per hour of recording. This does not look like much, except that the number of messages (just 10) used for this case study is not necessarily representative. In reality, some services would have much higher throughput and would need much more storage space for all the records generated. This limits the application of this tool to services with low throughput. It is possible to work around this limitation, such as limiting the time that the records are stored and limiting the network communication that generates records.

- *Can the recording and replaying of executions be performed without modifying the source code of the program?*

  The answer to this question is that yes, it can be done. in this case study the source code of the program has only been modified in order to collect the metrics necessary to perform the analysis.

- *Can a specific execution of a system be faithfully replayed in a developer's local machine?*

  This is the most important research question and the one that this case study aimed to answer. The answer seems to be that yes, it can as seen by the replay accuracy values in Table 7.7, which are around 100%. This means that for this case study every replay has been a faithful reproduction of the original execution.

7.6 Validation Threats

Some issues have been identified in the way this case study has been developed that raise questions about the validity of the solution to other programs and environments. These are:

- For this case study both the recording phase and the replaying phase have been executed in the same machine. While it should work the same if the recording phase happened in a different machine, this case study does not do so and therefore cannot be used in support of that claim.
- Determining the accuracy of the replay using the output logs of the original and replayed executions might be inaccurate for the cases in which the output is the same but the behavior has been different.
- The post-processing of output logs, while necessary, may unintentionally omit some information that stems from different behavior of the program. Some effort has been spent to ensure that this is not the case, but this is not guaranteed.
- This case study is focused on a specific application, so the results cannot be generalized. In order to do so, it would be necessary to conduct other case studies with different applications, including ones using different network communication protocols.

Chapter 8

Conclusions

This document demonstrates that it is possible to faithfully replay executions that have been previously recorded. An approach has been proposed to do so that, unlike other solutions analyzed in Chapter 3, does not require any modifications to the source code, introduces minimal overhead, and guarantees accurate replay not only for network communication but for all sources of non-determinism. In order to demonstrate this, a prototype tool has been developed and a case study has been conducted on a service that is part of a large microservice-based system. The case study shows that the replay of executions can be done with high levels of accuracy, to the point that the replayed execution is indistinguishable from the original when it comes to the behavior of the program. Furthermore, it demonstrates that it can be done without modifying the source code of the program and introducing minimal overhead when it comes to the execution time of the instrumented program. On the other hand, a large amount of data must be generated in order to replay the executions accurately. This may be a limitation to the system and might make it unusable for some types of programs.

8.1 Contributions

The main contributions from this document are:

- A novel **approach** to debug microservices that run in the Java Virtual Machine using the Java Runtime Environment. This approach is described in Chapter 5 and outlines a way to record and replay programs without modifying their source code in any way by making use of the Java instrumentation API and bytecode manipulation.
- A **prototype** that implements the described approach. The implementation of this prototype is described in Chapter 6 and focuses on some of the more common causes of non-determinism in executions, in specific communication over the network and random number generation. This prototype is intended to help validate the proposed approach.

- A **case study** designed to showcase and verify the validity of the approach proposed in this dissertation. The description of the case study can be found in Chapter 7. The case study instruments a service that is part of a complex system of microservices and is used by a company in production. It puts the prototype to the test with a real-world example of the kind of programs that it is meant to help debug.

8.2 Future Work

The work described in this document is meant to answer the research questions outlined in Chapter 4. Further work would be necessary in order to make the tool useful and viable in the real-world and with programs in production environments serving large numbers of users. Some of the issues that would require further work are:

Improved correspondence of records between the recording phase and replaying phase. Using the Java instrumentation API it is possible to obtain the place in the source code where messages are sent and received. That information could be added to record objects in order to facilitate the correspondence of records in the replaying phase.

Better integration of the tool in the debugging process. Using this tool in the debugging process implies cross-referencing a bug report with the registered records for the execution in order to find the relevant records to be replayed. At this moment, the timestamp of the anomalous event must be used to find the records that correspond to that moment. Furthermore, the tool has no way of figuring out which records are part of the interaction that the programmer is interested in and which records are not. The better judgment of the programmer is required to make that distinction.

More thorough instrumentation of relevant JRE classes and methods. As it is described in Chapter 6 the implementation of the prototype focused on a specific set of classes and methods that are widely used to achieve network communication and random number generation. However, there are other classes that can and are used to do the same thing. An improved implementation of the approach would instrument these alternative classes and methods in order to ensure that the largest possible number of use-cases are covered.

Further empirical validation. While the case study that was conducted attempts to validate the proposed approach, it is just one example of the many types of services that this approach aims to help debug. In order to increase confidence in the validation, it would be necessary to conduct case studies with other programs and ideally programs of different kinds, different architectures and that use different communication protocols. It would also be of interest to conduct some studies in production environments.

Rethink record generation and storage. The biggest limitation of the approach described in this dissertation is the amount of disk space required to store the number of records necessary to accurately replay executions. It is necessary to think of better strategies to minimize the number of generated records, the amount of data needed per record and the way in which said records are stored.

References

[ASM] ASM. https://asm.ow2.io/. Accessed: 2019-02-06.

[BIN03] Paul Barham, Rebecca Isaacs, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 85–90. USENIX, May 2003.

[BWBE16] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. Debugging distributed systems. *Queue*, 14(2):50:91–50:110, March 2016.

[Byt] Byteman. https://byteman.jboss.org/. Accessed: 2019-02-06.

[byt18] Byteman programmer's guide. https://downloads.jboss.org/byteman/4.0.5/byteman-programmers-guide.pdf, 2018.

[CBM90] W.H. Cheung, J.P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, pages 106–115, January 1990.

[CKF+02] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604, June 2002.

[CPP] Computer Programming Principles maintaining/debugging. https://en.wikibooks.org/w/index.php?title=Computer_Programming_Principles/Maintaining/Debugging&oldid=3449890. Accessed: 2019-02-06.

[Fow] Microservices. https://martinfowler.com/articles/microservices.html.

[fow14] Microservices. https://martinfowler.com/articles/microservices.html, 2014.

[FPKS07] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, 2007. USENIX Association.

[GAM+07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th Usenix Conference on Networked Systems Design and Implementation*, 2007.

[GASS06] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. pages 27–27, 05 2006.

[GDB] Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/. Accessed: 2019-02-06.

[GKT+12] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Presented as part of the 26th Large Installation System Administration Conference (LISA 12)*, pages 33–42, San Diego, CA, 2012. USENIX.

[GoR] GoReplay. https://goreplay.org.

[Gro14] Open Group. Service-oriented architecture ontology, version 2.0. Technical report, April 2014.

[Had] Apache hadoop. http://hadoop.apache.org/. Accessed: 2019-02-06.

[Ins] java.lang.Instrument. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html. Accessed: 2019-02-06.

[int] The Internet Protocol Stack. https://www.w3.org/People/Frystyk/thesis/TcpIp.html. Accessed: 2019-02-06.

[Jav] Javassist. https://www.javassist.org/. Accessed: 2019-02-06.

[kaf] Kafka Protocol Guide. https://kafka.apache.org/protocol. Accessed: 2019-06-06.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[LGW+08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, Mariëlle Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In *NSDI*, 2008.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings - International Conference on Software Engineering*, pages 492–501, January 2006.

[Mic] Microservice Trade-Offs. https://martinfowler.com/articles/microservice-trade-offs.html.

[Nie94] Henrik Frystyk Nielsen. The HTTP Protocol in the World Wide Web Library of Common Code. Master's thesis, Aalborg University, Denmark, August 1994.

[opea] *Operating System Concepts*. Wiley.

[Opeb] The OpenTracing Project. https://opentracing.io/. Accessed: 2019-02-06.

[Pro] Protocol buffers | google developers. https://developers.google.com/protocol-buffers/. Accessed: 2019-06-29.

[Res18] Dimensional Research. Global microservices trends: A survey of development professionals. Available at https://go.lightstep.com/global-microservices-trends-report-2018.html, April 2018.

[SBB$^+$10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[XHF$^+$10] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Experience mining google's production console logs. In *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

REFERENCES

Appendix A

Script for Post-Processing of Logs

This appendix contains the `sed` script used to perform the post-processing of the log files collected during the record and replay phases of the executions analyzed in the case-study. The script removes information that is always different in each execution such as memory addresses and timestamps, but also logs that are irrelevant to the execution and pertain to cluster management or other tasks.

```
sed -e ’1d’ \
    -e ’s/[0-9][0-9]* \(\[\)/X \1/’ \
    -e ’s/Thread-[0-9][0-9]*/Thread-X/g’ \
    -e ’s/@[a-z0-9][a-z0-9]*/@XXXXXXXX/g’ \
    -e ’s/[a-z0-9]\{8\}-[a-z0-9]\{4\}-[a-z0-9]\{4\}-[a-z0-9]\{4\}-[a-z0-9]\{12\}(:[0-9]*)*/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXX/g’ \
    -e ’s/\[SLOT_[0-9][0-9]*\]/\[SLOT_X\]/’ \
    -e ’/\[NIOServerCxn.Factory/d’ \
    -e ’/\[SyncThread/d’ \
    -e ’/\[SLOT_/d’ \
    -e ’s/time [0-9][0-9]*/time X/g’ \
    -e ’s/0x[a-z0-9]\{15\}/0xXXXXXXXXXXXXXXX/’ \
    -e ’s/([a-z0-9]\.)*:[0-9]*/<old_address>/’ \
    -e ’s/Topology_[^ ]*Topology_<topology_ID>/g’ \
    -e ’/\[timer/d’ \
    -e ’s/[0-9\.][0-9\.:]*:[0-9][0-9]*/<address>/g’ \
    -e ’s/\/var\/folders\/[^ ]*/<tmp_dir>/g’ \
    -e ’s/[a-z0-9]\{8\}-[a-z0-9]\{4\}-[a-z0-9]\{4\}-[a-z0-9]\{4\}-[a-z0-9]*/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXX/g’
```