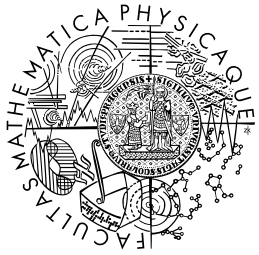

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vít Kabele

Integrated Server for Dynamic Program Analysis

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Ing. Lubomír Bulej, Ph.D.
Study programme:	Computer Science
Study branch:	IOI

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor Ing. Lubomír Bulej, PhD. for all the patience, help and advice he has given to me.

Title: Integrated Server for Dynamic Program Analysis

Author: Vít Kabele

Department: Department of Distributed and Dependable Systems

Supervisor: Ing. Lubomír Bulej, Ph.D., Department of Distributed and Dependable Systems

Abstract:

Dynamic analysis aids in many software development tasks, such as debugging, program comprehension, and performance optimization. However, implementing a new dynamic analysis tool is a non-trivial tasks.

To simplify development of dynamic analyses, researchers at Charles University and Università della Svizzera italiana in Lugano have jointly developed the DiSL and ShadowVM frameworks, which raise the level of abstraction for analysis tool developers and provide a convenient programming model both for bytecode instrumentation and for analysis execution.

Even though those frameworks were successfully used to develop many different dynamic analyses, it turned out that the internal design of the original implementation of both frameworks made further development of new features, such as support for instrumentation-time reflection, extremely difficult.

Both frameworks provide a client and server part and while they are designed to be used together, the design prevents sharing of information between the two client parts and the two server parts. This not only increases the amount of data that need to be exchanged over network, but also makes configuration of all parts more difficult.

In this work we propose and implement a new architecture of the analysis suite so that the functionality the functionality of the DiSL and ShadowVM frameworks can be hosted by a single server accommodating multiple clients.

Keywords: bytecode instrumentation dynamic program analysis

Contents

1. Introduction

Since the complexity of software solutions is constantly growing, developers that desire to develop large applications and aim to provide smoother experience to users are forced to use some kind of program analysis.

The analysis might be built upon an analysis of the source code and various suspicious pattern detection, including using undeclared variables, trying to predict race conditions or warn the programmer about unreachable code. This type of analysis is called static analysis and it is definitely a powerful tool. However its possibilities are limited to what can be discovered in the source code.

Sometimes programmers need to know what is going on in a running program, and here, static analysis cannot help. This includes observing execution times of methods, object lifetimes, real identities of objects hidden behind the facade of interfaces or abstract classes and so on.

Observing the execution time of methods can be surely done by manually adding a timestamp right after the method entry and printing the elapsed time before the method exit. This approach is shown in Figure 1.1. However, doing this manually is very time consuming for anything but a simple program with a few methods. Changing the logic of such a analysis is also very error prone as it means changing the source code on multiple places. Not to mention that one has to print the method's execution time before every possible method ending, which is not only the return statement, but also before any exception is raised. Furthermore the analysis code should not be present in the released software which means either to manually remove the appropriate logic or to deal with some type of preprocessor #ifdef statements.

It should be no surprise that no one in the developers community did not want to do this task manually and that an effort was made to automate it.

Generally it is all about inserting additional behaviour into the observed program — a process which is called *instrumentation*. In languages that are compiled directly into machine code, it is done in the binary machine code with tools like Valgrind [1] or PIN [2]. This instrumentation is called the *binary instrumentation* and it works well for low level programs written in C or C++, however it cannot be used when dealing with higher level managed languages such as Java.

The reason why we cannot use the machine code approach in Java programs is because they are compiled and distributed in the form of the so called bytecode. The bytecode is hardware independent and based on the specification of the Java Virtual Machine (JVM). When the program is executed using a JVM instance, the bytecode is translated into machine code right before it is needed by the

```
public void foo(int bar){
    long time = System.getNanoTime();
    // Original method logic here
    System.out.println(System.getNanoTime() - time);
    return result;
}
```

Figure 1.1: An exapmle of a naive approach to the method execution time profiler

program control flow. This process is called the Just In Time compilation (JIT). If we try to analyze the running Java process using let's say the Valgrind suite, we will end up analyzing the JVM instance also. Furthermore — for historical reasons — the bytecode is sometimes only interpreted and thus we have nothing to instrument.

To overcome this obstacle we have to go one step higher to the bytecode and perform modifications there. This is called the *bytecode instrumentation* and on the Java platfom it can be done using for example the ASM[1], BCEL[2] or Javassist[3] libraries. The inserted code is then executed along with the observed program no matter if it is interpreted or translated into machine code.

The common feature of PIN and ASM is the abstraction level of machine respectively bytecode instructions. Inserting a method call includes manually finding the place in the code where to insert the call, pushing the arguments onto the calling stack and calling the method using a call instruction (or its sibling in the appropriate instruction set) followed by the address of the desired method in the given program. This approach is hardly more elegant than manually modifying the source code.

Nonetheless the developers did not stop here and went further with a goal to invent a solution that will bring better experience than manually modifying the source codes as well as higher level of abstraction than modifying the bytecode.

The result is combination of both prevously described techniques. It allows the developer to define the inserted behaviour in the form similar to original language (e.g. methods and classes). To define points in the exectuion of analyzed program (e.g. regular expression to match method name) and to define an action that will be taken by the inserted code at a particular point (e.g. insert before or insert afer) and let the specialized program to weave the behaviours to the appropriate places.

This approach is known as the programming paradigm called Aspect Oriented Programming (AOP). It's representative in the world of Java is AspectJ[4]. In the terminology of AspectJ the inserted behaviour is called *aspect*, the point in program execution is called *joinpoint* and the action taken is called an *advice*. Finally the software used to weave the behaviours is called the *weaver*.

Nonetheless most of the analysis task are not as simple as only printing the execution time of the method. At least we want to print the method's name along with the time or maybe we want to know a value of some local variable or even a value of a method parameter. This is generally known as *context*.

We recognize two types of contexts. The first one, *static context* contains the information known at compile time. This includes method names, parameters names and types and so on. The second type is *dynamic context*. It stands for the set of information that is first known at runtime. For a method it can be real types of objects or values of method parameters and many more.

In the following text we describe the DiSL and the ShadowVM which are tools that help with development of dynamic analysis and summarize issues that are adressed in this work. At the end of this chapter we present the goals of this

[1] https://asm.ow2.io

[2] https://commons.apache.org/proper/commons-bcel/

[3] https://www.javassist.org

[4] https://www.eclipse.org/aspectj/

thesis.

1.1 DiSL

As the AOP seems to be ideal solution to perform dynamic analysis some tools were based on this paradigm. Let's mention the DJProf [3] profiler or the RacerAJ [4] data-race detector. Those tools are nevertheless highly specialized.

A developer that tends to create custom dynamic analysis using some of the existing AOP solutions will quickly face a fact that none of them was created having dynamic analysis in mind. Specifically they lack the flexibility while defining the joinpoints and they have a non-trivial performance overhead in common tasks.

Inspired by the AOP concept and aware of the weaknesses of current implementations the DiSL — Domain Specific Instrumentation Language — was created by Lukáš Marek et.al. [5]. It offers a higher level of abstraction than direct bytecode manipulation tools while providing significantly better performance than former AOP solutions. DiSL was inspired by Aspect Oriented Programming but allows greater extensibility when it comes to modifying the weaver behaviour.

The advices in DiSL are called *snippets* because they are standard java methods and they are used like templates, where the pre-defined variables are replaced by the weave-time pre-coputed constants of static context data or the inline code to obtain dynamic context data at runtime. This leads to boosting the performance in common analysis tasks.

In DiSL, the snippets are always inlined. Thanks to this, the developer has the possibility to exchange data between advices using local variables. To support this, DiSL introduced the concept of *synthetic local* variables. Those are inserted between the local variables of a given method and are used to share data between snippets of the same method.

Mainstream AOP languages provides three main types of advices. These are the *Before*, *After* and *Around* advices. While the two former are used to insert the snippet (or advice more generally) before or after the joinpoint, the Around advice is often used to wrap the whole joinpoint into a conditional branch and change the original program control flow or to introduce a variable that will be shared in code before and after the joinpoint.

As oposed to the mainstream AOP languages, DiSL was not meant to change the control flow of the base program. Specially, DiSL refuses any instrumentation that tries to do so. This and the fact that all the snippets are inlined leads to the absence of *around* advice execution.

1.1.1 Usage of DiSL

As well as the other instrumentation tools DiSL cannot be used from whithin the same process as the observed application. DiSL itself is written in Java language and thus it requires the full support of a running JVM instance in order to work. However, we need to instrument classes even before the whole class library is loaded so the process of instrumentation is performed in a separate JVM.

All classes of a observed program are sent to the process running DiSL using a socket even before they are loaded into the JVM runtime and once they are instrumented, they are delivered using the same socket back to the client. The

remote process is called the DiSL server and the one who sends and receives the classes is called DiSL client.

The instrumentation source has a form of one or more Java classes, each containing one or more static methods. Each method can be a snippet. If the method is a snippet, it is marked with an appropriate annotation that represents a joinpoint and the advice. An example is provided in Figure 1.2.

The sources are compiled with a standard java compiler and packed into a jar file, along with the *MANIFEST.MF* file describing the instrumentation structure. The packed jar file is then used to initialise the DiSL library.

1.1.2 Introductory example

To help the reader gain deeper insight into the software described in this thesis, a short introductory example is provided below to illustrate the common way in which DiSL is used. The following example will not show all the required steps to successfully run the instrumentation, as it is only meant to introduce the reader to the context. For the in-depth guide on how to write and run a complete instrumentation, please see Chapter 4.

```
public class IntroductoryExample {
    @Before(marker=BodyMarker.class, scope="TargetClass.main")
    public static void onMethodEntry() {
        System.out.println("inst: Before method Main");
    }

    @After(marker=BodyMarker.class, scope="TargetClass.main")
    public static void onMethodExit() {
        System.out.println("inst: After method Main");
    }
}
```

Figure 1.2: Example instrumentation

In Figure 1.2 two example snippets are shown. Each snippet is marked with an annotation which describes the place in the instrumented program where the code should be inserted.

Each annotation has a pair of arguments. The first argument, called the marker, is used to specify the joinpoint in the application code. The one that is used here — BodyMarker.class — matches the whole method body. Another option can be e.g. the BBMarker.class, matching the bytecode basic block[5]. The second argument, called the scope, limits the application of a snippet only to bytecode regions matching the pattern. This parameter accepts wildcards, so for example the "TargetClass.*" will match all methods in the TargetClass class.

Finally, the type of annotation itself is used to tell the weaver, whether to insert a snippet @Before or @After the specified joinpoint. It is worth mentioning that the names of snippet methods doesn't matter at all, as the weaver is instructed by the annotation only.

[5]A basic block is the sequence of instructions, where either all or none of them is executed. This term is used in the theory of compilers

```
class TargetClass{
    public static void main(String[] args){
        System.out.println("app: Inside method main");
    }
}
```

Figure 1.3: Target class of instrumentation

Figure 1.3 shows a sample application to which the instrumentation is applied, corresponding to the scope in the example above. After correct compilation and execution, the output shown in Fig. 1.4 will be printed out by the application.

```
> inst: Before method Main
> app: Inside method Main
> inst: After method Main
```

Figure 1.4: Output of instrumented application

The instrumentation might be extended in many ways, starting with creating custom (synthetic) local variables in methods to share the data between snippets, ending with writing custom marker classes.

1.2 ShadowVM

So far we have demonstrated how to insert a trivial analysis code into the analyzed application. In this section we will take a closer look on the analysis code itself. The analysis is often not that easy as simply printing out some data. At least some aggregation of produced data might be required by an analysis developer.

As the inserted code grows in the terms of complexity and the amount of dependencies, it will soon happen that the analysis wants to use a class that is also being analyzed. We can try to avoid such a dependency clash by not instrumenting the core classes when analyzing the application code, but this effort cannot succeed once we want to analyze the Java class library too.

Existing dynamic analysis solutions based on instrumentation suffer from a low isolation problem. This typically means, that the analysis code executed in the inserted snippets shares the application's state and classes with the observed program [6]. Even with considerable effort, it is impossible to avoid the interference completely, since we must rely at least on some parts of JVM runtime such as I/O.

Ignoring this would have significant consequences to both the analysis results as well as its stability and the stability of the whole program. The results would be corrupted because the analysis would most likely end up analysing itself. The stability would suffer by blind insertions of snippets into wrong places which make it easy to create some kind of deadlock. [7]

The solution offered by other analysis/instrumentation tools is to exclude the predefined groups of classes and/or packages from instrumentation. A typical exclusion list contains classes from packages providing basic functionality of the JVM runtime, including *java*, *javax* or *com.sun*. This straightforward solution

definitely works, but then the analysis cannot be applied to the excluded classes, which poses a problem as in those classes, something with nontrivial impact on the analysis results can happen.

To avoid the need of exclusions, it is essential to perform as much of the analysis logic as possible outside of the observed environment. We are able to do this thanks to the Java Native Interface (JNI) and its ability to allow implementation of Java functions in a native code. Offloading the analysis to the native subsystem is a great step forward, since we are no longer interacting with the observed environment. However this step forces the analysis to be written in native code, meaning either C or C++. This can be very limiting, because many developers are not able to write code in those language fast enough. Those who are will quickly face the problem of missing essential support features required by the analysis, like reflection.

ShadowVM was created to allow writing analysis that will take advantage of both isolation and higher abstraction level provided by the Java language. It introduces the concept of *remote analysis*. The real computation is performed in a separate JVM, with code written in Java language with full support of the JVM environment, to which the events from the observed application are delivered by the socket. Thus, the amount of inserted code is reduced to minimum, by only calling the native methods that generate remote procedure calls. The remote analysis processor is called the ShadowVM server and the application that is being analysed is called the ShadowVM client.

This approach performs well, until we need to work with a reflection mentioned above. To use reflective information, the classes loaded to the observed program have to be also loaded into the analysis process. To do so, ShadowVM client also sends all the classes to the server as they are loaded.

When dealing with classes in Java it is always needed to care about the classloader. Multiple classes with the same name can be loaded into one JVM instance and their's scope is defined by their respective classloader. This feature is used in some modular applications. To support analysis of such applications, classes are sent to the remote analysis server along with an identifier of their respective classloader.

1.2.1 Introductory example

As in the case of the DiSL, we provide short example to demonstrate the practical application of ShadowVM when writing a dynamic analysis.

The following easy analysis is written directly into the source code of an observed app in order to make things easy and not to obfuscate the instrumentation. Not all technical details are presented here. For the complete guide how to write and execute analysis, please see the Chapter 4.

As an example, we show an analysis that will mirror the output of the observed application. In Fig. 1.5 an application is shown and in Fig. 1.6 the remotely executed code is illustrated.

Notice that in the code of example analysis the instances of strings are printed out, but instead of calling the `System.out.println` method directly, we call it through a wrapper with additional functionality consisting of invoking three static methods on REDispatch class. Such an invocation is also present in the initial-

```
package cz.uk.mff.kabelevi;
import ch.usi.dag.dislre.REDispatch;

class TargetClass{

    private static short anlMethodId = REDispatch.registerMethod(
        "cz.uk.mff.kabelevi.Analysis.printString"
    );

    public static void Main(void){
        for(Integer i = 0; i < 5; ++i) {
            printWithAnalytics(i.toString());
        }
    }

    public static void printWithAnalytics(object o){
        REDispatch.analysisStart(anlMethodId);
        REDispatch.sendObjectWithData(o);
        REDispatch.analysisEnd();
        System.out.println(o);
    }
}
```

Figure 1.5: Analyzed app

isation of a static field anlMethodId. Those methods are the entry points to the analysis's native code.

Firstly, the registerMethod method generates a unique id for the method whose name is provided in the parameter and sends the signature-id pair to the analysis server. Then, each invocation of the printing method starts a remote procedure call of the method identified with the previously obtained ID. It packs the argument of the method, and ends the RPC. The remote call is then buffered inside the native code and sent to the analysis backend.

```
package cz.uk.mff.kabelevi;

class Analysis {
    public void printString(ShadowObject o){
        System.out.println(o);
    }
}
```

Figure 1.6: Remote analysis code

In Fig. 1.6 the referenced method from the previous class is displayed. It has to be in the classpath of the analysis server and its arguments must exactly match the arguments as shown in the previous figure between the analysisStart and analysisEnd methods. With this much code we achieved the desired result. The analysis will now print the same strings as the original application.

We can also notice here, that instead of using a method wrapper, we can take advantage of the instrumentation engine described in the previous section.

1.3 Goals

In this chapter we described how both DiSL and ShadowVM clients send the classes from an observed virtual machine to their respective servers. DiSL to instrument them and ShadowVM to use their reflective information in analysis.

As both tools were used in practice it turns out that they were almost never used without one another. Therefore the classes were in most cases sent twice even though it would be enough to sent them once. This only stressed the observed application for no reason.

The goal of this thesis is to design a new server and protocol to provide both DiSL and ShadowVM functionalities packed together and spare the client side the unnecessary work.

2. Overview of DiSL and ShadowVM

In this chapter we describe the state of the real implementation with emphasis on problems, that are solved by this thesis. This chapter contains analysis of problems that we solved but we chose different name since the term *analysis* could be overloaded in current context.

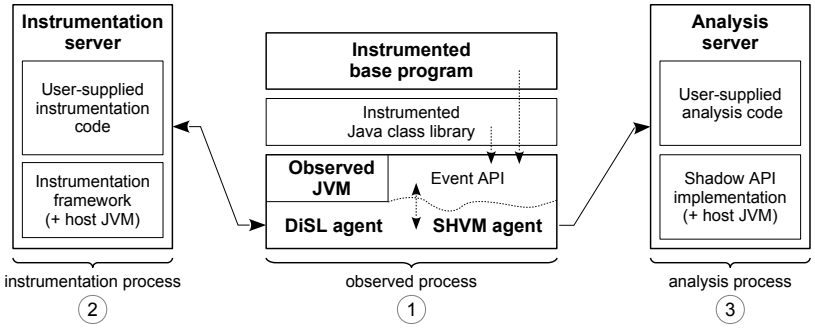

Figure 2.1: Original architecture as described in [7]

Figure 2.1 shows the architecture of DiSL and ShadowVM. The observed JVM running the analysed and instrumented application with plugged JVMTI agents is in the middle.

Number two on the left site, depicts the instrumentation server connected by a socket to the observed JVM. On the other side, with number three, there is the analysis server, again connected by a socket to the main process.

2.1 General implementation properties

This section outlines the general properties of the implementation as they appeared before the start of this work.

The whole DiSL software suite is maintained in one git repository hosted on the ObjectWeb consortium GitLab instance[1]. The most visible thing worth mentioning is the source code organisation. All the source codes were located in subfolders of the root of the repository and everything was built using one huge all encompassing *build.xml* ant[2] script. There is also the bin directory containing a launcher script written in python and examples directory where example applications of DiSL and ShadowVM are present as complete projects.

The most important directories and files of the repository are depicted in Figure 2.2. This solution was impractical. It caused mixing of dependencies of different parts of the suite, which either means the dependencies of the executables and libraries, or the build logic of C libraries and Java programs.

[1] http://disl.ow2.org
[2] https://ant.apache.org

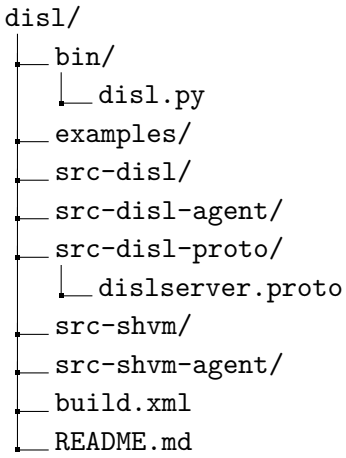

Figure 2.2: Original implementation repository layout (simplified)

JVM properties

Another concept present across the whole repository was the configuration by JVM properties. These are set on the command line when the JVM starts by the **-D** parameter. The properties can be obtained in the code by calling a static method getProperty on a System class in Java or by calling the GetSystemProperty method on a jvmti* struct in C. This approach was chosen because it lets the developer access the parameter values without parsing the command line which means a lower start-up costs.

This is a great way of passing arguments to native agents as the other option here is to pass them concatenated in one string and then parse in C code. The C language does not provide convenient methods for direct strings manipulation and therefore parsing the arguments would be potentially insecure.

Unfortunately this approach was used also in the configurations of both servers and — even worse — in configuration of both the DiSL and ShadowVM libraries. It tempts the developer to misuse the ability to read the option anywhere in the application by accessing the global state. In the original implementation, methods deep in the library code were configuring themselves by reading the property set on the command line.

To make things even more confusing, the properties recognised by the application were not documented anywhere. The properties are also used by the JVM itself and an application has no possibility to warn the user about a misspelled property name, as it assumes, that the option is meant for configuring the Java runtime.

Packages

The whole Java codebase suffered from the misuse of the Java package concept. To give an example, the DiSL library had more than 20 packages. They were used to organise the source code in a way similar to the namespaces that are used in C#, i.e. to introduce the missing hierarchy into the source code organisation. The packages in Java, however were not meant to be used in this way, as they should rather represent a unit of reuse. As a consequence, almost all methods and classes were marked as public and exposed to the user as a public API.

Event name	Event description
VMStart	The VM stopped booting
VMInit	The VM is initialized. All JVMTI events can be called
VMDeath	VM was terminated.
ClassFileLoad	The *.class file is being loaded into JVM
ObjectFree	Triggered by Garbage collector. Notify about object death.

Figure 2.3: Some important JVMTI events

Global variables

The former implementation was written to only allow analysis/instrumentation of one process at the same time. As a consequence of such an approach mainly static methods and properties — which are i.e. global variables in Java — were used. As some internal data structures including caches were static, it was not possible to use multiple library instances in one process. This was very limitng for future improvements.

2.1.1 JVMTI

The Java Virtual Machine Tool Interface[3] is an interface provided by the JVM to developers to allow them to extend the functionality of the JVM by writing native agents in the C/C++ programming language. Some of the actions exposed by the interface allow registering a handler for internal JVM events or allow registering JNI-compliant native-code implementations of Java methods.

Both DiSL and ShadowVM agents are using capabilities of the JVMTI interface to register event handlers. Moreover the ShadowVM client also provides native implementation of the analysis related functions.

Some events that are importat for us are shown in Figure 2.3. The event handlers have the form of standard C functions. Their parameters must correspond with the JVMTI specification of the concrete handler. The handler gets all the event context in the parameters. For example the objectFree handler has two parameters — the pointer to JVMTI environment and the identifier of object that is being freed.

2.2 DiSL

In the original implementation, both agent and server were written in a way that allowed them to work only in a one-to-one scenario.

The original implementation consisted of four basic modules, although as mentioned in Section 2.1, they were not really modules in the common sense, as they shared a lot of code and dependencies. The three main modules were: native JVM agent, instrumentation library, server and public API and the DiSL bypass package.

[3]https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

2.2.1 Server

The server and the instrumentation library were both placed in the src-disl directory as a single code base. The disadvantage of this arrangement was that the library and the server did not have their responsibilities clearly divided. Part of the network handling system was in the instrumentation library and the server did some non-trivial manipulation with the requests.

The main packages in src-disl directory were the ch.usi.dag.disl containing the source codes and other packages related to instrumentation itself and the ch.usi.dag.dislserver with the server logic.

The instrumentation **.jar* files were specified in the classpath of server process. The DiSL library loaded the classes using the system classloader.

2.2.2 Agent

The DiSL agent was written in the C language and placed in the src-disl-agent directory. Although it was possible to develop it in the C++ language, C was chosen because it is easy to use. The agent takes advantage of features provided by JVMTI 2.1.1.

This agent is passed to the observed program JVM instance by the -agentpath option and is then notified about the events in the Java runtime.

Below a more in-depth description of the events handled by DiSL agent is provided.

Events description

- **JVM Init** Update the instrumentation options to reflect that the JVM stopped booting.
- **JVM Start** Just write that the JVM has started and any related function can be executed safely.
- **JVM Death** Stop the connections, stop the threadpool.
- **ClassFileLoadHook** The essential hook that allows DiSL to work. This event is triggered when the JVM loads a file with a new class. As an parameter, the hook gets right the loaded bytecode and a pointer where to save the redefined bytecode. The original class is then sent to the server where is instrumented, then is received and placed to the appropriate position in the memory. The whole process is done synchronously and thus it is viable to make it as fast as possible.

2.2.3 Network communication

The DiSL agent, as written before the beginning of this work, used the concept of thread pool to manage the opened connections to the server. Every application thread that wanted to instrument some class took a connection from the pool, and put it back when the request was completed. This was done lazily so that a new connection was not created before it was really necessary. Specifically, single

Message	Use case
InstrumentClassRequest	Send the class to the server for instrumentation
InstrumentClassResponse	Send the instrumented class back to the client

Figure 2.4: Types of messages used in instrumentation

threaded application creates only one connection to the server. This construction was called *connection pool*.

To handle the requests efficiently on the server side, each connection was also managed by a single thread which requires the inner structures of the instrumentation library to be thread-safe.

Protocol Buffers

The data transfer itself was not written manually but instead the Protocol buffers[4] library was used.

The Protocol buffers is a software suite that aims to facilitate the process of developing message based network protocols. The suite is developed and maintained by the Google company. It allows the developer to describe the structure of the messages using a special syntax in a file with the **.proto* extension and then to use a dedicated compiler to generate the source codes for desired programming language.

In our case, it is the output to the Java language using the official Google compiler. The C language output is generated by the community driven opensource implementation, because the official one provides only a C++ output.

The generated code creates a higher level of abstraction to the raw network transferred data and provides a native interface in a given programming language. Java generates classes with properties that correspond to its message fields. In C a similar approach is used with structs instead of classes.

When using the protocol buffers, one must realise that the suite only helps to serialise and deserialise the messages. It does not affect the way of how messages are transferred over the network. To successfully send and receive messages it is crucial to send the message's length first, and then on the other side read the proper amount of data. Without sending the length, the library is not able to parse the data directly from a stream.

Also, the protocol cannot effectively recognise the type of a message that was delivered. Therefore it is required to pack all the client messages in one client supermessage and all the server messages in one server supermessage with two fields. One for the data itself and one for the type identifier.

The usage of the protocol buffers has many advantages that simplify development. Mainly it guarantees, that both sides will use the same version of protocol and gets rid of the programmer's need to hassle with a low level abstractions, such as the network byte ordering. The only disadvantage worth of mentioning is another dependency added to the repository.

[4]https://developers.google.com/protocol-buffers

Method name	Method purpose
analysisStart	Initiate new RPC to the server
sendObject	Sends the object instance to the analysis
sendLong	Sends one long to the analysis.
analysisEnd	Ends the RPC and sends the data

Figure 2.5: Some of the native methods provided by ShadowVM agent

2.3 ShadowVM

ShadowVM was built using similar concepts as the DiSL suite. Some of the code was even shared between them. Many of the classes still have the RE prefix, which is abbreviation of the legacy name — Remote Evaluation. The sources were located in src-shvm directory.

The suite consists of a native agent, an analysis server providing the shdow object programming model and the REDispatch package with API exposed to the analysed virtual machine.

At its core, ShadowVM just transfers the analysis method invocations to the remote process using the Remote Procedure Call (RPC).

2.3.1 Server

As in the DiSL, here we also encountered the problem of misusing the java packages as a replacement for namespaces. Moreover, the server part and the shadow object programming model were not separated so for example the network data unmarshalling was performed inside the object model library. This is definitely violation of the single responsibility principle. It leads to duplicite code and complicates any future modification of the library as it forces the developer to hassle with problems unrelated to analysis itself.

The server part itself was really simple, having only about three different classes.

2.3.2 Agent

The ShadowVM agent was a bit more complicated than its DiSL sibling. It was caused mainly by a higher number of managed events and by the need of object tagging.

The networking model of the ShadowVM is different from that of DiSL, presented in Section 2.2.3. It does not use a connection pool to manage connections to the server, but instead it involves one dedicated sender thread that takes care of sending the data out. It was a sufficient solution, as this agent was not required to receive the responses and thus the data were written into the buffer queue and the thread continued. This also implies, that only one connection with the server was initiated.

The agent also exposes a set of native functions. These are implemented in native code and when the agent is loaded they are registered in the JVM and can be called from Java programs. These methods are members of the REDispatch and some of them are illustrated in Figure 2.5.

Methods similar to sendLong are available for all primitive types of Java language.

Object tagging

Since we need to send instances of objects to the server (including the instances of type java.lang.class), we need to uniquely identify the instances. When an agent is invoked by the JVMTI event (or when sending the object in the native RE-Dispatch.sendObject method) the reference is obtained as an identifier. However, since Java is a managed language, its core component is the garbage collector and with garbage collection involved, reference validity is limited by it's next run.

JVMTI guarantees that the reference that we get as an argument in an event handler will not be invalidated before the handler returns. Unfortunately, that is not enough to effectively recognise the instances between different entries of the native code. To solve this issue the JVM provides a feature called *ObjectTagging*. By using this feature the agent can tag an instance of an object with a user-defined 64bit long number.

To make the things more complicated, reading and writing the tags is internally done synchronously. This is definitely something that the event handler should avoid, as the observed program execution could be stopped as handler waits for the lock which might be a non-trivial time in some parallel applications.

To avoid this, the agent only writes down the objects that need to be tagged and enqueues this information to a queue along with the data that is being sent to the server. The thread that dequeues the item then performs the tagging. But here the reference is leaving the scope of the event handler, so the JVM does not guarantee the reference's validity. Fortunately, the JVMTI has an API call that allows the agent to request validity for a longer time. This is possible, but again it will hurt the JVM performance when holding it for too long, as it means effectively disabling garbage collection. And in this case it could take a long time if the dequeueing thread is a sending thread that is waiting for some I/O action.

In order to perform the essential tagging and minimise the impact on the performance of the Java runtime, a compromise was chosen. Another dedicated thread was created to be the consumer of the tagging queue events and the producer for the sender thread queue.

2.3.3 Network communication

The network communication was done in the most straightforward way packing the data to the buffer, starting with the message length, followed by the message identifier and data itself. Such a buffer was written to the socket and received and unmarshalled on the other side.

No library was used and both the server and agent had to be kept in sync manually in order to communicate properly. Furthermore, the precise message format was exposed to the developer at bit-level, forcing the programmer to hassle with the details such as network byte ordering on a significantly lower level of abstraction than the rest of the code.

It is obvious, that it was very error prone to maintain this codebase and was very time consuming to add a new feature.

In Figure 2.6 the types of messages sent during the instrumentation are shown. When compared to the DiSL messages, the important difference is that they do not have qualified names.

Message
Register analysis method on the server
Send the class bytecode to the server
Send the class information to the server
Notify the server about the object death
Send string information to the server
Send thread information to the server
Notify the server about the thread end
Invoke the analysis method on the server

Figure 2.6: Analysis related messages

2.4 Specific goals

At the end of Chapter 1 we described general goals of this thesis. In this section we describe the specific goals according to the analysis of DiSL and ShadowVM tools.

- **Session support** We want to use multiple instances of DiSL and ShadowVM libraries in single process.
- **Traffic optimization** Both DiSL and ShadowVM transfer the classes to their respective servers. This however only stresses the observed JVM and the main goal is to optimize this behaviour. The only way how to do this is merging both servers into one process.
  - **Protocol** We want to migrate the communication protocol of ShadowVM to Protocol Buffers and possibly merge into the DiSL protocol. This is also essential to support communication with merged server as the messages will be transferred using common socket.
- **Refactoring and responsibility segregation** We want to update the directory structure to better reflect the semantics of the code. This includes refactoring of the build system to increase modularity of the whole repository as well as flattening the package structure. We also want to strictly define the responsibilities of the modules and to create clear interfaces that will allow reusing them.
- **Ease of use** Since we are going to change the original implementation in a non-trivial way, we have a lot of opportunities to introduce the changes in the way of how the suite is used. We want to use these opportunities to improve the usability for DiSL users — the analysis developers.

3. New implementation

In this chapter we will discuss our new implementation, we will solve our goals and discuss the obstacles we had to overcome to achieve them.

3.1 Session support

In the context of this work sessions represent the feature of using a single server instance to analyze multiple clients. This includes analyzing multiple different clients simultaneously, a single client multiple times or combination of both.

To do so we had to rewrite the internal representation of some data structures in DiSL and ShadowVM libraries to be non-static. The task was really complicated, because when global variables were used, there was no need to pass any instances to method calls. Therefore we had to reimplement almost half of the methods in both libraries to pass references to data-structure instances. Some methods are called through many intermediate methods and the reference must be passed to all of them.

For instrumentation this also meant to change the way how **.jar* files with instrumentation are passed to the server. The original implementation loaded them using the system classloader from its classpath. This approach is however not possible when multiple different clients with multiple different instrumentations are connected. We decided to change the process of loading instrumentation so that the path to it is now provided to the agent and the agent sends the file to server.

It was also required to store the data on the server in some way and thus to identify the session with a unique key. But if the agent generates its own key and sends it to the server, there might be a possible clash. To prevent such collision we implemented the two way handshake. The client requests the session key from the server, the server respond with a key or possibly with an error. Once the client has the session key assigned, it sends the files with instrumentation to the server. The session specific messages are shown in Figure 3.2 in section **General messages**. The instrumentation related messages are in section **DiSL messages**.

The server loads the instrumentation files using a session-specific classloader so that the loaded classes can be thrown away when the session terminates, but the native reflection support can be used to parse the instrumentation for weaver purposes. Using the session specific classloader also prevents name clashes in case that multiple clients are using same names for instrumentation classes[1].

For analysis the classes are still passed to the server in the classpath and loaded using the system classloader. It doesn't prevent name clashes, but we decided so just because the instrumentation could rely on some external libraries. When the analysis is packed together with all its dependencies, the resulting **.jar* file can be too large to send it over the socket to the server. It would also mean to rewrite the analysis library to work with session specific classloader in the same way as the DiSL library does. Nonetheless such a modification will maybe require

[1]That is a very common case. Instrumentation classes are often named just DiSL-Class

rewriting the library from scratch.

3.2 Traffic optimization

As explained in Chapter 1, both DiSL and ShadowVM transfer the loaded classes to their respective servers. DiSL to instrument them and ShadowVM to build its reflective information tree for analysis purposes. In both cases the transferred data are the same.

As the tools were used over the time, it turned out that they are almost never used on their own but 99% of the time they are used together. This finding draws attention to the fact that that in most cases, sending the classes twice only burdens the observed virtual machine. Also we need to share the reflection information to make it available in the ShadowVM immediately after the class is instrumented. In this context we decided to merge both servers into one so that they are able to share the received class data.

We decided to obsolete both former servers and write a new one, integrating both DiSL and ShadowVM backends. The new architecture is shown in Figure 3.1. The server is placed at the right side with number two, combining the functionalities of both obsoleted servers that were described in Chapter 2 and shown in Figure 2.1

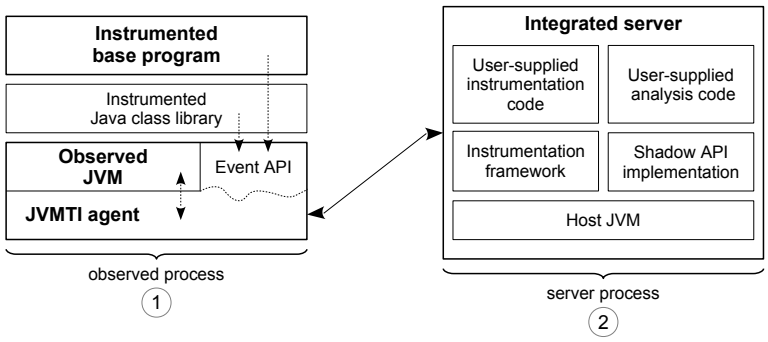

Figure 3.1: New architecture

With the new server, there were three possible ways of how to realise the data sharing between both backends. The most convenient one was to modify both libraries to use the same internal data representation to store the received classes and then simply provide them with the reference to the shared storage. Although this would be the best solution, it involves modifying both libraries in a non-trivial way or maybe even rewriting them from scratch, which is outside the scope of this thesis.

The following two solutions are almost equivalet in terms of complexity. One of them means receiving the classbytes in the DiSL InstrumentationRequest message, putting them in a cache and sending the ShadowVM new class message with the name and classloader tag only and then retrieve the classbytes from the cache. The second solution just lets the InstrumentationRequest dispatch in the ShadowVM library instead of the new class message, as they carry the same data. In this case the new class message is not even set.

Independently of the chosen method we need to transfer the DiSL message before the ShadowVM message. As reported in Chapter 2, each message was sent in the ClassFileLoadHook event handler of a different agent. The crucial thing here is that JVMTI does not offer an API to ensure the event handling order. It does make a sense, from the point of the JVMTI since many agents from different developers can be loaded inside, but to our need it is very limiting.

To solve this issue, we decided to merge the agents together and ensure the proper event ordering by simply calling the instrumentation agent first. This brought us a few benefits. It reduced the amount of total C lines of code as a lot of functionality was duplicated in both agents and we got the seession support in ShadowVM almost for free. The sessions now identify not only the instrumentation context, but provide also the identification of the analyzed VM.

On the other hand, merging the agents also revealed some obstacles. As a consequence the unified agent contained two different networking models and was connected to the server using not one, not even two, but potentially a even higher number of connections, each using a different protocol. This permitted using a single agent with multiple servers, but it was found irrelevant and in order to tidy the codebase we decided to pass all the messages through a single connection. To achieve this it was necessary to create a single protocol and because the ShadowVM still used the manually defined data format it was also necessary to migrate the ShadowVM to protocol buffers.

In the ShadowVM library the network unmarshalling was moved away from library code and the library was closed behind the minimalistic API.

The ShadowVM model with dedicated sender thread was however designed to only send messages to server but not to receive responses. In DiSL this was allowed by using multiple connections. When migrated to a single connection, this was now essentially impossible. The instrumentation needs to wait to the response, but we have only sender thread and the sender thread cannot wait for because it needs to send data as fast as possible.

Therefore we implemented a new receiver thread. This thread only waits for incomminng data on the connection that we already have estabilished to our server. When done so, we can receive the data, but we have no option to pass them back to the original thread that initiated the communication and we don't have an option to wake the thread up. To solve this issue, we introduced *asynchronous network tasks*.

The asynchronous network task is initiated by calling the function send_and_receive. This function has a blocking semantics. It takes the request as a parameter and returns the response. Internally it creates a struct with conditional variable, pointer to request and pointer to response. The struct is then enqueued to the sender queue and the function hangs on the conditional variable. Once the sender thread receives this struct, it generates unique id for the request and sends the data to the server along with the id. The struct is then saved to the hashtable with the id as a key. When the receiver thread obtains a response, it takes the struct from hashtable, writes the correct address to the response pointer and wakes up the source thread using the conditional variable from struct.

To support this we also needed to update the protocol messages to carry the request and response id.

3.2.1 Protocol

As was stated in the Subsection 2.2.3 Protocol Buffers provide many advantages for the developement of the network applications and therefore migrating the ShadowVM to use this library was only a matter of time. Due to the clean separation of network related code in the previous implementation of ShadowVM agent, the migration only meant creating a new protocol. New types of messages are illustrated in Fig. 3.2 in section **ShadowVM messages**.

General messages	Meaning
SessionInitRequest	Ask the server about new session ID
SessionInitResponse	Send the session id or error to the client
ConnectionClose	Notify server that the JVM stopped.
DiSL messages	Meaning
InstrDelivery	Send the instrumentation jars to the server
InstrAcceptConf	Notify the client, if instrumentation was ok
InstrClassRequest	Send the class to the server for instrumentation
InstrClassResponse	Sends the instrumented class back to client
ShadowVM messages	Meaning
Analyze	Remote Procedure Call
ClassInfo	Send class info to the server
NewClass	Send new class bytes to the server
ObjectFree	Notify server about object death
RegisterAnalysis	Register method id on the server
StringInfo	Send the string content to the server
ThreadInfo	Send the thread data to the server
ThreadEnd	Notify the server about thread end

Figure 3.2: The complete new protocol

Unfortunately not all types of messages could be fully migrated to the Protocol Buffers. Fully migrated means completely avoiding manual manipulation with data in buffers. Fortunately the only message which was impossible to describe using the language provided by the protocol buffers is the analyse message. Not only that it is not possible because the message format is dependent on the particular analysis and thus is not known at a compile time but also because of the limitations of the object tagging mechanism.

To tag the objects the source thread packs them into a buffer along with a structure that claims the position of an object in the buffer and the tagging thread then reads the claims and puts the tags onto the appropriate place in the buffer. This is however not possible when using the protocol buffers since the structure of packed message is unknown. We could pack the uncompressed message, tag the objects and then pack them into the buffer, but this approach involves copying the data once again and we do not want to task the already busy hot path.

Fig. 3.2, shows the complete overview of the new, merged protocol.

3.3 Refactoring and responsibility segregation

3.3.1 Packages hierarchy and responsibilities

As described in Section 2.1 the packages were misused in the codebase. In our work we tried to flatten the structure and reduce the total amount of present packages. We needed to refactor almost every file in the project as the imports on their beginning became invalid. We merged the most packages except those containing the public API exposed to the analysis developer. By doing so, we kept the backwards compatibility with previously developed analyses.

In order to clean up the codebase, we defined interfaces for both DiSL and ShadowVM libraries and marked all classes and methods not used in the interface or in the public API as package private. All classes and methods shared between multiple modules were extracted to the separate package ch.usi.dag.util and placed in the directory util.

3.3.2 Build system and directory structure

As mentioned in Section 2.1, the whole project including all the logic was built using a single *build.xml* file in the root of the repository. This was not practical, since every small change to the build process lead to changes in the thousand-line-long main build script. This was confusing, since navigation in such a long file is difficult, and editing is very error-prone. Not only that all the modules were built using one shared logic, they also shared dependencies, and a common build output. It was really challenging to edit one project without breaking another.

To fix that, the repository was split into subprojects, each containing its own build logic, build output directory and well defined dependencies. The master *build.xml* was kept in place, but modified, so that the subproject dependent tasks were delegated to the child build scripts and only the needed artifacts, such as the final packed **.jar*, or the linked native library, are copied to common output location.

By splitting the repository like this, it was also possible to create a better environment for running unit tests for each module independently and to offer the possibility of modifying only one module at a time without being worried about breaking some functionality of the others.

The updated repository layout is depicted in Figure 3.3.

3.3.3 Tests and CI

The repository was equipped with a good testing suite. But it was not really a unit testing suite, although a few of them were present. The vast majority had a form of the black-box tests, as they consisted of running DiSL server, running ShadowVM server, running the client and evaluating the standard output of those processes. Although they were not unit tests, their assistance was absolutely priceless while developing the new server and modifying both agents. However, the higher test coverage, the better, so each of the newly created modules has its own unit testing suite and a possibility to run the tests independently.

Once the tests are written, it is desired to run them automatically while pushing to the remote repository. This functionality is referred to as Continuous

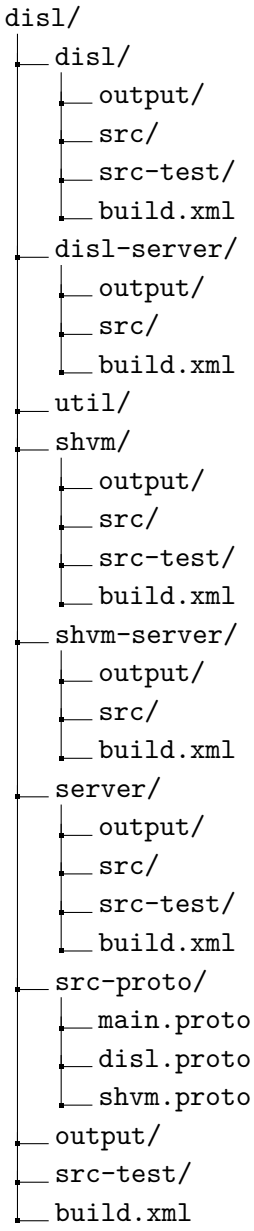

Figure 3.3: New layout of the repository (simplified)

Integration or CI in short. Since the repository of the project was moved to the git versioning system instead of the old SVN, and hosted on the GitLab[2] instance, configuring the CI was a simple, but very useful step. Not only that now every push to that repository is verified by running the tests in a well defined environment, but by separating the tests into the submodules, it is now possible to run all the four sets of tests in parallel.

3.4 Ease of use

All of the changes described above improved the usability in some way. Merging the servers implies that the user of the analysis does not need to run three

[2]https://about.gitlab.com

processes to perform the instrumentation based analysis, but only two.

By using the merged client, the user can use less parameters when executing the analysed JVM, which is definitely a benefit, since running the analysis properly can still involve dozens of configuration options.

To make the user experience even better, the new server is no longer configured via the JVM properties, but uses the GNU/POSIX style compliant command line options. This now allows the user to make simple execution scripts and then run the server from the command line in a way that every macOS/linux user is familiar with. The new server also supports the `–help` option that prints out short description of all the recognized parameters.

Although there is now a corresponding support, only a few of the original options were migrated to the new format. This is mainly caused by the fact that both libraries are still configuring itself from the global state of application and to deliver the option to the proper place we would need to pass the option through the whole call stack of intermediate method calls.

4. Example analysis

So far we have seen DiSL and ShadowVM tools and we mentioned that they can help us to develop dynamic analysis in a fast and elegant way. Although both of them can be used separately, their power is in cooperation. The previous chapters were given to description of their teoretical properties and now the time is up to show a practical use.

In this chapter a step-by-step tutorial to writing the dynamic analysis using these tools is provided and some special attention will be given to the details that can cause obstacles to the project newcomers.

This tutorial is not meant to be a comprehensive summary of all options and possibilities that DiSL and ShadowVM offer. Such a documentation can be found in the original paper referring about DiSL [5], respective ShadowVM [7]. This guide will only present the usage in order to show how it is different from previous version.

The so called developer documentation cannot be provided simply because this job consisted mainly from modifications of existing software. Therefore this work doesn't have a public iterface except the command line api of new server, that is described in this section also and its documentation can be obtained by running with `–help` option.

The following analysis is a simple method execution time profiler. The observed program will be instrumented to record the wallclock time of each method and send the value to the remote analysis server afterwards, along with the method identifier. Finally, when the observed program stops, the analysis will print out the summary for each method.

This guide is divided into four sections. The first one is focused on downloading the DiSL suite, in the second and third one, the instrumentation, respectively the analysis will be written and finally, the analysis will be presented on an example application.

4.1 Getting the DiSL suite

To successfully run the suite, the computer must meet the following criteria:

- Apache Ant installed
- An C11 compliant compiler (cc, gcc, clang)
- macOS/linux operating system
- Java8 installed

The stable builds of DiSL are released once a time. However until the next stable release, some more work has to be done in the repository and the version described in this work must be build from source codes. We download the software by cloning the `devel` branch to your computer using git.

```
> git clone --branch DEVEL https://gitlab.ow2.org/disl/disl.git
```

This will clone the remote repository locally, into a directory called disl. Now when all the sources are available and all requirements are met, the suite can be build using the following command:

```
> ant build
```

This will download the dependencies and build the project executables into the directory output/lib in the root of the cloned repository.

4.2 Writing the instrumentation

The time has come, to create the instrumentation. The first version will only print the elapsed wall clock time of each method to the stdout, thus it will not rely on the ShadowVM features.

```
import ch.usi.dag.disl.annotation.After;
import ch.usi.dag.disl.annotation.Before;
import ch.usi.dag.disl.annotation.SyntheticLocal;
import ch.usi.dag.disl.marker.BodyMarker;

class Instr {

    @SyntheticLocal
    public static long wallTime;

    @Before(marker=BodyMarker.class)
    public static void onMethodEntry(){
        wallTime = System.nanoTime();
    }

    @After(marker=BodyMarker.class)
    public static void onMethodExit(){
        System.out.println(System.nanoTime() - wallTime);
    }
}
```

Figure 4.1: file Instr.java

In the Figure 4.1 the basic instrumentation is shown. We create the directory demo in the root of the repository and save the file here, with the name *Instr.java*.

Now create a file named *DemoApp.java*. This is the demo application on which the example will be shown, however you can pick your own Java app and it should also work. The code of the *DemoApp.java* is in the Fig. 4.2.

Figure 4.2: DemoApp.java

```
package cz.uk.mff.disl;

class DemoApp {
    public static void main(String[] args){
        System.out.println("Hello from method main");
    }
}
```

Figure 4.3: MANIFEST.MF

```
Manifest-Version: 1.0
DiSL-Classes: Instr
```

By executing the following command, the instrumentation is compiled into the Instr.class file, located also in the demo directory. Note that we need to compile the instrumentation with the classpath to the compiled classes of DiSL instrumentation library, because of the annotations and the marker.

```
> javac -classpath../disl/output/build/disl Instr.java
```

The instrumentation is now compiled, but we need to tell the DiSL, in which classes the instrumentation can be found. As mentioned in Subsection 1.1.1, this is done via the *MANIFEST.MF* file of the resulting jar file. The manifest file for our example is the Figure 4.3.

Now, we have the instrumentation and the appropriate *MANIFEST.MF* and we can create the resulting jar file. We do so by running the following command.

```
> jar cmf MANIFEST.MF Instr.jar Instr.class
```

This will create the jar file containing the *Instr.class* file in the root and a directory META-INF, containing the manifest file. The structure can be viewed by running jar tf Instr.jar.

Now the instrumentation should theoretically work. All methods of all classes will be instrumented and will print out the elapsed time. However, if we run the instrumented application now, it crashes. This is caused by instrumenting some core classes. As we mentioned in the first chapter, the power of DiSL and ShadowVM is in the ability to instrument and analyze even those, however some special care is required to optimise the snippets, which is not in the scope of this manual. Such a limitation is not a weakness of the DiSL, but rather the JVM itself. To avoid crashes here, we need to exclude all the methods on classes in the java.lang package. To do so, create a file called exclusions with one line java.lang.*.*.

Finally, the instrumentation can be executed by running following two commands:

```
> java -jar../output/lib/server.jar --disl --exclusion-list exclusions
and
> java -noverify \
    -agentpath=../output/lib/libdislreagent.jnilib \
```

```
-Ddisl.allow=true \
-Ddisl.instrumentation=Instr.jar \
-Xbootclasspath/a=../output/lib/disl-bypass.jar \
-classpath build \
cz.uk.mff.disl.DemoApp
```

The first one will run the server. The `--disl` option tells the server that it should expose the DiSL functionality to the clients and the `--exclution-list` points the server to our exclusion list file created in the previous step. The complete list of supported CLI options can be displayed by the `--help` option and is also available in the server/README.md file.

The second command executes the client. The `-Ddisl.allow` option allows it to act as the DiSL agent (both the ShadowVM and the DiSL functionalities are disabled by default, which makes the commands more self describing) and the `-Ddisl.instrumentation` option points to the jar file with instrumentation we created previously. The agentpath option tells the JVM from where to load the agent library. Note that when are running linux and not macOS, it is need to use the *.so extension instead of *.jnilib. The `-Xbootclasspath/a` option allows dynamic bypass redefinition. This feature was not discussed in this work, but a description can be found in [7]. Then the classpath to our demo application is specified along with the name of the entry point class of the app.

As both programs needs to communicate and run simultaneously, the server must be started first and the client after the server initialisation is finished.

The program, if executed now, will print several pages of numbers. This does not have a big informational value, so in the next step we will develop a remote analysis based on ShadowVM to help us aggregate the data.

4.3 Writing the analysis

To successfully run the ShadowVM based analysis, we need to create two classes. The first one, called CodeExecuted is loaded into the observed JVM and its functions are called from the instrumented code. The second one, called CodeExecutedRE is loaded into the analysis machine and its methods are executed by the remote procedure calls from CodeExecuted class. In our simple example, only one type of RPC is performed.

The CodeExecuted class is shown in Figure 4.4. The performRPC method does exactly what its name says, using the native methods provided by the agent library. The static analysisId field describes the mapping between the remote method name and its identifier for the purposes of the Remote Procedure Call.

On the server side, the received information is aggregated so that the time for each method is summed up and finally, when the analysis finishes all methods are printed out along with their durations.

To aggregate the data, we use a simple hash table. The methodName string is the key and the time spent is the value in nanoseconds. In the receiveRPC method the hash table is updated with a newly received time.

Note the important part: The REDispatch.sendObjectPlusData is mapped onto the first argument of the type ShadowObject of the receiveRPC method and the REDispatch.sendLong is mapped onto the second argument of type long.

Figure 4.4: CodeExecuted.java

```
import ch.usi.dag.dislre.REDispatch;

public class CodeExecuted {

    public static short analysisId = REDispatch.registerMethod(
        "CodeExecutedRE.receiveRPC"
    );

    public static void preformRPC(String methodName, long time) {
        REDispatch.analysisStart(analysisId);
        REDispatch.sendObjectPlusData(methodName);
        REDispatch.sendLong(time);
        REDispatch.analysisEnd();
    }
}
```

The CodeAnalysisRE class must extend the RemoteAnalysis class. The RemoteAnalysis has abstract methods objectFree and atExit that must be overridden in order to handle the object's free events and to specify what should be done, when the server obtains the notification about the client's death.

Next we need to modify the instrumentation, so that the elapsed time is not printed out, but sent to the analysis instead. To do so, we need to modify the OnMethodExit method in *Instr.java* to the following and also import the MethodStaticContext and ClassStaticContext classes from the ch.usi.dag.disl.staticcontext package.

```
public static void AfterMethod(
    MethodStaticContext msc, ClassStaticContext csc
){
    CodeExecuted.performRPC(
        String.format("%s.%s", csc.getName(), msc.thisMethodName()),
        System.nanoTime() - entryTime);
}
```

We are almost ready to run our analysis. The last step is to recompile and rebuild the instrumentation and compile and run the analysis. Because we changed the dependencies, we also need to change the compilation process. The required commands are in Figure 4.6

Now the analysis can be executed as shown in the Fig. 4.7.

The observed app will print out only the Hello from method main and the analysis server's output will look similar to the Figure 4.8.

With not so much effort we wrote the complete analysis that can help us for example with detecting hot-paths in our application, that can analyze even the core methods and that gives a correct results. The analysis developement was also much easier when compared to the methods using ASM or AspectJ. The detailed comparison can be found in [8].To gain deeper insight into the program runtime, we can extend the analysis so that it will keep track of calling context, but this is not in the scope of this thesis.

Figure 4.5: CodeExecutedRE.java

```
import ch.usi.dag.dislreserver.remoteanalysis.RemoteAnalysis;
import ch.usi.dag.dislreserver.shadow.ShadowObject;
import java.util.HashMap;
import java.util.Map;

public class CodeExecutedRE extends RemoteAnalysis {

    private final HashMap<String,Long> execTimes = new HashMap<>();

    public void receiveRPC(ShadowObject methodName, long time){
        execTimes.compute(name.toString(),
            (key, val) -> {
                Long v = (val==null) ? 0L : val;
                return v + time;
            }
        );
    }

    @Override
    public void objectFree(ShadowObject o){
        // Not handling the object free events
    }

    @Override
    public void atExit(){
        for(Map.Entry m : execTimes.entrySet()){
            System.out.println(
                String.format("%s - %d", m.getKey(), m.getValue())
            );
        }
    }
}
```

```
# Build the remote analysis
# The classpath contains the definition of RemoteAnalysis class
javac -cp ../output/lib/dislre-server.jar CodeExecutedRE.java
# The classpath contains the definition of REDispatch class
javac -cp ../output/lib/dislre-dispatch.jar CodeExecuted.java

# Pack the analysis
jar cf Analysis.jar CodeExecutedRE.class CodeExecuted.class

# Build the instrumentation
# The classpath contains the definition of CodeExecuted class
javac -cp ../disl/output/build/disl:Analysis.jar Instr.java

# Pack the analysis
jar cmf MANIFEST.MF Instr.jar Instr.class
```

Figure 4.6: The complete compilation process

```
# Start the server
java -cp Analysis.jar:../output/lib/server.jar \
    ch.usi.dag.server.Main --disl --shvm --exclusion-list exclusions

# Start the client
java -noverify \
    -agentpath:../output/lib/libdislreagent.jnilib \
    -Ddisl.allow=true \
    -Ddisl.instrumentation=Instr.jar \
    -Dsvm-agent.allow=true \
    -Xbootclasspath/a:../output/lib/disl-bypass.jar:Analysis.jar \
    -Xbootclasspath/a:../output/lib/dislre-dispatch.jar \
    -classpath:build \
    cz.uk.mff.disl.DemoApp
```

Figure 4.7: Running the analysis

```
...
java.nio.CharBuffer.<init> - 238003
sun.usagetracker.UsageTrackerClient.<clinit> - 8651894
cz.uk.mff.disl.DemoApp.main - 11482488
java.nio.DirectLongBufferU.put - 482466
java.net.URL.getUserInfo - 148780
java.io.UnixFileSystem.getBooleanAttributes - 3422434
...
```

Figure 4.8: Output

Conclusion

Although the goals of this thesis or maybe even the steps described in Chapter 3 might seem as a straighforward modification of the DiSL suite the complexity of this task was given mainly by dealing with already existing code with poor documentation and by dealing with the internals of the JVM. The previous implementation was written in a way that was not ready for any future extension nor modification and while working with JVM one has to be extremly careful as it is easy to introduce some kind of deadlock or cause a huge performance penalty.

Regardless of this we have reached the declared results while keeping as much of the original API as possible.

By introducing sessions the analysis server can be executed once and used many times. It's possible to run a single instance of analysis server as a daemon for a very long time.

Merging of servers and merging of agents along with the new protocol brought the required traffic optimalization and reduced the total amount of duplicite code in repository.

The modularization and overall splitting of project into more independent pieces is beneficient especially for the developement and testability.

We can thus conclude that this work satisfies the declared goals.

Summary of changes

The following list is an overview of the changes that we introduced by this work into the DiSL project. All introduced changes were in range of about 200 commits.

- Refactoring
  - We separated the modules (including the build logic)
  - We extracted the shared Java code into the separate package
  - We added a lot of comments
  - We introduced CI
  - We flattened the package structure
- Architecture updates
  - We created new integrated analysis server
  - We designed a new protocol
  - We migrated the ShadowVM to Protocol Buffers
  - We modified both DiSL and ShadowVM libraries to be more modular
  - We introduced asynchronous network tasks in the JVMTI agent

Except the required results, this work has also highligted a need of some additional future improvements. The most notable of them are mentioned below.

ShadowVM client/server separation

Although the client and server are now almost independent, there is still a connection between in the form of the running server instance and the analysed instance. This connection is the *.jar file with a remote evaluation code, that has to be passed to the server's classpath. It would be nice if the client was able to send the jar to the server in the same way as the DiSL agent does. However this feature implies that the whole server side of the analysis has to be rewritten to work with session specific classloaders in order to prevent the interference between remote evaluation codes of multiple different instances. Such a redesign was not in the scope of this work.

Recent Java versions

Currently the whole suite is developed and tested only on Java version 8. Although teoretically it should be not a problem to run it on newer releases, various bugs might occur and some extra care will be required to provide the same level of experience as on the currently supported version.

Distribution

Currently the software is available only in the form of source codes. Everyone who wants to use the suite is forced to build it on its own. From the next release on, we would like to distribute the sofware as a pre-built package for either Java platform in the form of let's say maven package of in the form of distribution dependent packages like `*.deb` for debian and so on.

Bibliography

[1] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. SIGPLAN Not., 42(6):89–100, June 2007.

[2] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Acm sigplan notices, volume 40, pages 190–200. ACM, 2005.

[3] David Pearce, Matthew Webster, Robert Berry, and Paul Kelly. Profiling with aspectj. Software Practice and Experience, 37:747–777, 06 2007.

[4] E. Bodden and K. Havelund. Aspect-oriented race detection in java. IEEE Transactions on Software Engineering, 36(4):509–527, July 2010.

[5] Lukáš Marek, Alex Villazon, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development, 03 2012.

[6] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The jvm is not observable enough (and what to do about it). In Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages, VMIL '12, pages 33–38, New York, NY, USA, 2012. ACM.

[7] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. Shadowvm: Robust and comprehensive dynamic program analysis for the java platform. SIGPLAN Not., 49(3):105–114, October 2013.

[8] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tuma, and Z. Qi. Productive development of dynamic program analysis tools with disl. In 2013 22nd Australian Software Engineering Conference, pages 11–19, June 2013.