UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Veiko Kääp

Java Virtual Machine multi-debugger proxy server

Bachelor's Thesis (9 ECTS)

Supervisor: Vesal Vojdani, PhD
Supervisor: Märt Bakhoff, MSc

Tartu 2018

Java Virtual Machine multi-debugger proxy server

Abstract: The Java platform provides not only a highly performant abstract computing machine, the Java Virtual Machine (JVM), but also contains sophisticated tools for interfacing with running applications. This functionality is specified in the Java Platform Debugger Architecture (JPDA). JPDA has a limitation of not being able to attach multiple debuggers to a single JVM which makes many debugging workflows and use cases impossible to accomplish. The purpose of this paper is to get rid of this limitation by creating a proxy server which would connect to the JVM and then allow multiple debuggers to connect to it. Such proxy server would allow the JVM to be debugged from multiple remote computers at the same time.

Keywords: Java Virtual Machine, Debugger

CERCS: P170, Computer science

Java virtuaalmasina mitme siluri puhverserver

Lühikokkuvõte: Java Platvorm ei koosne mitte ainult abstraktsest masinast, Java virtuaalmasinast (JVM), vaid sisaldab ka nutikaid tööriistu töötavate rakendustega suhtlemiseks. Seda funktsionaalsust käsitleb Java platvormi siluri arhitektuur (JPDA). JPDA kasulikkust piirab asjaolu, et ühe JVM-i külge on korraga võimalik ühendada ülimalt üks silur, mis teeb aga paljud silumise võtted võimatuks. Selle töö eesmärgiks on luua puhverserver, mis eemaldaks antud piirangu nii, et puhverserver ühenduks ise virtuaalmasinaga ning lubaks endaga ühendada mitu silurit. Selline puhverserver võimaldab ühte JVM-i siluda mitmest eri arvutist korraga.

Võtmesõnad: Java Virtuaalmasin, Silur

CERCS: P170, Arvutiteadus

Contents

1 Introduction

The Java platform provides not only a highly performant abstract computing machine, the Java Virtual Machine (JVM), but also contains sophisticated tools for interfacing with running applications. This functionality is specified in the Java Platform Debugger Architecture (JPDA), which is a widely used toolkit for monitoring applications on the JVM.

While the JPDA provides extensive tools for debugging any application running on the JVM, it has the limitation of only accepting a single connection between the debugger and the JVM. It is not currently possible to attach multiple debuggers to a single JVM and that makes many debugging workflows and use cases impossible to accomplish.

One important use case which is impossible to handle currently using JPDA is a situation where an application running on a single JVM is built from multiple modules developed on separate remote machines. When problems occur in the application, it is necessary to monitor and control the application execution for all the modules. Due to them being developed on separate machines, however, it is impossible to attach a debugger to the JVM from each machine at the same time. The only available workaround is to disconnect the connected debugger when another one needs to attach.

The purpose of this paper is to overcome this limitation by creating a proxy server which would connect to the JVM and then allow multiple debuggers to connect to it. Such proxy server would allow the JVM to be debugged from multiple remote computers at the same time. It would also enable the use of multiple different IDEs (Integrated Development Environment) while debugging.

Currently no alternative solution exists to solve this limitation. This is mainly due to the fact that the creation of such a proxy server is quite complex and requires special handling for most of the functionality the JPDA platform provides.

The development of the proxy server has two main requirements. First requirement is that if only a single debugger is connected to the proxy server, then the behaviour of the proxy server should be identical to a situation where the debugger is connected straight to the JVM.

The second requirement involves two or more debuggers being connected to the server. In such case, the proxy server will route and handle the data with the intention of creating an impression to every debugger of being connected directly to the JVM.

The main use cases when multiple debuggers are connected include setting, clear-

ing and hitting breakpoints in code, stepping to next lines and also resuming and suspending threads. These use cases will be covered and tested most extensively.

The following sections will introduce the Java Virtual Machine, the Java Platform Debugger Architecture and will walk through the creation of the proxy server which will provide a solution to the limitation of connecting only a single debugger to the JVM.

2 Overview of Debugging

One of the most time-consuming aspects of software development is ensuring the quality of the written program. That means ensuring that the program operates correctly with the given input, produces the desired output and doesn't cause any unwanted side effects. The development of the software will begin with the description of the problem followed by construction of an algorithm designed to solve that problem. Then the algorithm is implemented in a programming language.

Usually, the compiler detects basic syntactic and semantic problems in the program and ensures that the program is at least able to start. However, the compiler won't detect if there are any logic problems in the code which can cause the output to be unexpected or the program to halt execution early with an error message.

That means the developer needs to ensure that the algorithm chosen for the problem behaves correctly in all corner cases and that the algorithm was properly translated to the programming language. For this, tests are written which will run the program or small isolated parts of the program and ensure that the program terminates correctly and gives the expected output.

2.1 Debugging methods

When any of these tests have an unexpected outcome or a problem occurs in a production environment, the developer will need to investigate and understand which part of the program leads to the issue or caused the error. The developer has multiple options on how to determine the faulty part:

- Read and examine the code in order to find the problematic code.
- Change code in random places that seem connected to the issue and see if the outcome becomes correct.
- Insert logging statements in the code to print intermediate values, and after rerunning the program, see if and where the values differ from the expected.
- Use a debugger to interactively control and monitor the program execution.

In most cases, using an interactive debugger is the easiest and most efficient choice [1]. A debugger is a software that allows the developer to monitor and control the execution of the program. It gives the developer the functionality to suspend or resume the execution at any time and to evaluate and inspect the program state while the program is running.

Most debuggers work best when they have access to the source code of the program. Then the debugger can provide the developer with the functionality to change variable values in the program and set breakpoints at certain lines in the code. Breakpoints are markers that tell the debugger when to suspend the execution of the program and to give the control of the execution to the developer.

The main benefit of using a debugger instead of simply adding logging statements is the ability to start debugging immediately without the need for knowledge of where the fault might be. With logging statements, the developer might add them to one location but then discover that the problem is in another location. Then it is necessary to stop the program, add new logging to the other location and rerun the program. With an interactive debugger, it is possible to immediately go to the other location and start debugging there.

2.2 Use cases of using multiple debuggers in parallel

This thesis focuses on how to use multiple debuggers in parallel on the JVM, but that kind of functionality can be useful for any programming language. Most modern software design patterns preach modularity as a simple way how to manage big software projects. So instead of having one huge project built into an application, often an application is built from many smaller projects. Depending on the IDE a developer is using, its interactive debugger might have a limitation of only allowing the debugger to use sources from a single active project.

For example, all IDEs available from the software company JetBrains have the feature of creating a separate window for each project. And its interactive debugger can only operate on projects in the current window. There are workarounds for this problem — a developer can import external projects as a submodule to an existing project window, but that requires time to set up properly.

Another use case is when one project is developed in one IDE and the other project uses a different IDE. Such a situation might happen due to legacy reasons — project layout and features are bound to a particular IDE — or because both IDEs have a different feature set which might benefit the projects differently. The problem is even more severe when the projects are developed on different machines.

Some debuggers have a completely different use case from other debuggers, which makes it useful to use them in parallel. For example, one debugger could be for monitoring the running threads and the other for looking into memory consumption. In such a case there is no workaround for using them at once other than allowing multiple debuggers to debug an application at the same time.

3 Debugging in the Java Virtual Machine

This section serves the following purpose:

- Introduce the Java Virtual Machine.
- Walk through the three layers of Java Platform Debug Architecture.
- Investigate the Java Debug Wire Protocol specification.

3.1 Java Virtual Machine

Java Virtual Machine (JVM) is an abstract computing machine which has an instruction set and which manipulates various memory areas at run time. It specifies the class file format which "contains Java Virtual Machine instructions (or *bytecodes*) and a symbol table, as well as other ancillary information" [2].

Despite the name of the virtual machine, JVM is not only limited to the Java language. The fact that the JVM only runs code represented by the class file format allows the virtual machine to be used by multiple different languages. If anybody wants to run a specific language on the JVM, all they have to do is create a compiler which translates that language to a class file.

There are many advantages to using the JVM instead of implementing your own virtual machine or platform:

Platform independence. The JVM is designed to adhere to the idea of writing once and running anywhere. All the language developer has to do is to compile the code to a class file and after that, the program will run in the same manner on all the platforms which the JVM supports.

Security. The JVM has built-in security features which prevent malicious software from compromising the Operating System (OS).

Optimization. The JVM can take unoptimized class files and optimize them to run faster and consume less memory. That means the language developer doesn't need to worry about optimization, but can instead leave that task to the JVM. This has the effect of making the compiler faster and less complex while at the same time avoiding bugs in the compiler.

Tooling support. There are many tools written for the JVM consisting of productivity tools, profilers, application performance management tools (APMs), application servers and libraries. Any language targeting the JVM will also be able to use and take advantage of these tools.

Debugging. Any debugger architecture or tool written for the JVM can be used to debug any JVM language.

For the present thesis, the last item is the most important — the problem being solved exists in the JVM and so affects all the JVM languages. The solution provided in this thesis will solve the problem for the JVM, so it's not only limited to the Java language but for any language targeting the JVM (e.g., Groovy, Kotlin, Scala).

3.2 Java Platform Debugger Architecture

"Java Platform Debugger Architecture (JPDA) is a multi-tiered debugging architecture that allows tools developers to easily create debugger applications which run portably across platforms, virtual machine (VM) implementations and JDK versions." [3]

Its three layers are Java Virtual Machine Tool Interface (JVM TI), Java Debug Wire Protocol (JDWP) and Java Debug Interface (JDI). A developer who intends to use JPDA can hook into JPDA on any of these layers [3].

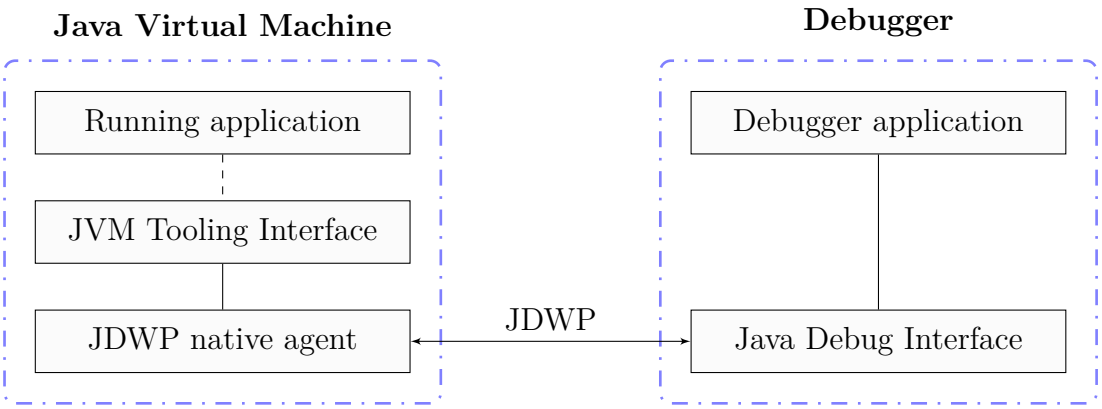

Figure 1: *Graphical representation of the Java Platform Debugger Architecture.*

Since JDI is the highest level and provides the best ease of use, then Oracle encourages developers to use that for simple Java language based debuggers [3]. While it is also possible to use the low level functionality provided by JDWP or JVM TI, they tend to require a lot more development hours depending on the task.

3.2.1 Java Virtual Machine Tool Interface

JVM TI is a native interface implemented by the Virtual Machine which serves as the back-end for JDWP. It is responsible for the services a VM must provide for debugging. That includes different requests for information about the JVM, but also actions the JVM should take and notifications sent to the debugger [4].

JVM TI is the lowest level layer in the JPDA. It provides the most direct access to the JVM, but that comes at the cost of portability — due to being a native interface, it requires to be built separately for each platform.

It also requires the developer to have access to the JVM startup arguments in order to attach the agent. It isn't possible to attach the agent remotely or after the JVM startup.

3.2.2 Java Debug Wire Protocol

Java Debug Wire Protocol (JDWP) is responsible for the format of information and transfer of requests between the JVM and the debugger. JDPW only specifies the format and layout of the protocol, but it does not set any restrictions to transport mechanism [5].

The main benefit of the JDWP layer is that the debugged program and debugger can run on different platforms and JVMs. It also creates the possibility of implementing the debugger in a language other than Java by creating a new front-end for JDWP.

3.2.3 Java Debug Interface

Java Debug Interface (JDI) is a Java interface which defines information and requests at a user code level. It serves as the front-end to JDWP. This interface is the most common way to debug a JVM and most of the Integrated Development Environments (IDEs) also use it for implementing their graphical user interface for the debugger.

Its biggest disadvantage is that the interface is available only for JVM languages. If a developer wished to create a debugger for the JVM in some other language, then it would be necessary to implement a new debug interface using the JDWP in that particular language. For that reason, JDI might be considered the least important layer of the JPDA since it is the easiest to exchange it for another interface and in many cases, it is even mandatory to do just that.

3.3 Java Debug Wire Protocol Specification

The JDWP startup consists of connection establishment and the following handshake between the virtual machine (VM) and the debugger. Handshake consists of debugger sending 14 bytes of ASCII characters of the string "JDWP-Handshake" to the VM and the VM replying with the same 14 bytes [6].

JDWP is packet based and is not stateful. In JDWP there exist two types of packets: command packets and reply packets. The JDWP is asynchronous so it is possible to send multiple command packets before a reply is received for the first command.

Both the target VM and the debugger can send command packets. For the debugger, they're used to "request information from the target VM, or to control program execution." The target VM sends them to "notify the debugger of some event in the target VM such as a breakpoint or exception." [6]

A reply packet is sent as a response to the command packet and provides information whether the command was a success or failure. The reply packet can also return a value or data requested by the command packet. In the current version of the protocol, events that are sent from the target VM do not require a response or a reply packet from the debugger.[6]

Headers of command and reply packets are equal in size and always 11 bytes. The layout of the command packet is the following:

- Header
  - length (4 bytes)
  - id (4 bytes)
  - flags (1 byte)
  - command set (1 byte)
  - command (1 byte)
- Data (variable size)

The layout of the reply packet is the following:

- Header
  - length (4 bytes)
  - id (4 bytes)
  - flags (1 byte)
  - error code (2 byte)
- Data (variable size)

The length field in the packet represents the size of the entire packet in bytes.

Since the header size is always 11 bytes, then no packet can have a size less than 11 bytes. A packet which contains no data has a size of 11 bytes. The id field of the packet is used to "uniquely identify each packet command/reply pair." [6] It is required for the reply packet to have the same id as the command packet to which it replies. Furthermore, the values of the id field must be unique for all command packets sent from a single source. The flags field is currently only used to mark whether a packet is a reply packet or a command packet.

The command set field in command packets is used for grouping commands. Command sets with a value from 0 to 63 are used for command packets sent to the target VM and command sets with a value from 64 to 127 is used for command packets sent to the debugger. Rest of the possible values from 128 to 255 are left for vendor-defined commands and extensions. [6] The command field combined with the command set field is used to identify how the command packet should be handled. Reply packets don't need command set and command fields since they are paired with a command packet which already contains this information. Error code field is only present in reply packets and it is used to show whether the command packet was processed successfully or an error has occurred during its processing.

The data field is unique for each specific command and it also differs between the command and reply packet pairs, so a command packet can have a different data field value from its reply. The data field of a packet is usually abstracted to a group of multiple subfields that define the packet data. The subfields are encoded in big-endian (Java) format. For the abstracted subfields, the protocol uses some of Java primitive types like byte, boolean, int, long, but it also defines some custom types. Most of these custom types can have a variable size in bytes depending on the JVM. In order to find the size of these types, the protocol has an `idSizes` command which replies to the debugger with the sizes of different types. [6] All the types used in the data field are listed in the following Oracle documentation: `https://docs.oracle.com/javase/10/docs/specs/jdwp/jdwp-spec.html#detailed-command-information`

4 Multi-debugger connections for JVM

This section will first describe why connecting multiple debuggers to a single JVM is a needed feature for the Java Platform and how the JPDA is limited without it. Then it will discuss how to approach solving this limitation and which approach is the best solution.

4.1 The need for connecting multiple debuggers to JVM

This section will describe the need for a proxy server for JDWP. More general use cases for using multiple debuggers in parallel were described in Section 2.2, but this section will focus in more detail on specific use cases unique for JPDA.

The JVM has support for tools called Java agents which use application programming interfaces (APIs) provided by the JVM to instrument programs running on the JVM. These APIs give JVM tool creators extensive functionality for creating simple to use agents which can be used for solving many problems. Agents can be used for monitoring performance, exceptions, logging or even to add previously unavailable features to the JVM, like reloading code at runtime.

However, since the instrumentation APIs provide so many possibilities, it is also quite easy to misuse the functionality and create hard to understand cases where the program doesn't function as expected. For such cases, interactively debugging the agent and the program running at the same time is one of the few possible ways to locate the cause of the problem. It does not, however, make much sense to have the running application and the agent in the same IDE workspace since their functionality and purpose are completely different. In such case, being able to write both projects in different workspaces but to debug both of them at the same time is crucially important.

Similarly, any application framework which allows the creation of plugins can benefit from being able to attach multiple debuggers. For example, most IDEs are designed so that all the supported languages and special features are not part of the core code base, but instead available as plugins. Such approach has many benefits including making the main code base smaller and easier to maintain. It also creates a clear separation between what the IDE should do and what each plugin does and makes it easier to later add new functionality by implementing a new plugin. Since separation is so important in such cases, then it makes sense to develop all plugins in separate workspaces. There again debugging them at once makes sense since at runtime the IDE with all the plugins functions as a single application creating a need for attaching multiple debuggers to the JVM.

4.2 Ways to solve the multi-debugger problem

Since there is no support from the JVM to connect multiple debuggers simultaneously, then writing a custom solution is required. JPDA is multi-tiered, as described in Section 3.2, so this thesis will consider all three layers of the architecture when solving the problem. For each layer, there is a different approach on how to solve the limitation.

Starting with the JVM TI, at first, this might seem like the most promising layer for allowing multiple debuggers since it is in this layer where the connection with the debugger is done. Working in this layer, however, would mean creating a new native agent which would duplicate all the debugging functionality of the built-in JDWP back-end agent, but at the same time add the support for multiple debuggers. Simply taking the existing native agent and modifying it isn't an option, since this agent might differ between different JVM vendors and might change with each JVM release. Also, since it's a native agent, then it would need to be built for each supported platform making its distribution and usage more difficult.

Another possibility is to use JDI, but that currently only supports being the front-end of JDWP and so acts as the debugger. That means all of its logic is written for connecting and communicating with the JVM, but it has no code for being the back-end for another debugger. Also, since it's meant to be the most accessible layer of JPDA, then it hides quite a lot of the connection details and packets to make the usage simpler. While it is possible to access and use the implementation code which is aware of all these details, it's made a lot more difficult and in some cases even impossible by the introduction of the Module system in Java 9. Since the solution should be backwards compatible with newer Java versions, then supporting the Module system is important and its limitations must be taken into account.

The last layer to look at for the solution is the transport protocol JDWP which is used for communication between the front-end and back-end of the JPDA. This layer doesn't have any code to reuse since it's simply a description of how the other layers of JPDA should communicate, but it is the simplest place where to solve the problem for numerous reasons. First of all, a solution in transport layer wouldn't set any restrictions on how the front-end and back-end of the debugger platform are implemented. The other layers can remain completely oblivious to the fact that there is something going on between them. This would solve the problem of having to support multiple JVM vendors and platforms or letting a developer use a front end other than JDI. Secondly, JDWP is backwards compatible, so that the solution created will keep working even with newer versions of JVM.

The idea is to create a proxy server for the JDWP which would connect to the JVM as a debugger and then let multiple debuggers connect to it. It would exchange packets with the JVM and debuggers and would pass them on and manipulate them as needed in order to maintain all the features of the debuggers as specified by the JPDA. What makes this solution simpler from the other two mentioned above is the fact that most packets don't require any special handling and can be passed directly from the sender to the receiver. With other layers, it is still required to implement the handling of such packets, but with JDWP proxy server it is possible to simply pass these packets on to where they were sent with no extra logic needed. Thanks to that, most of the development time for the proxy server can be focused on the packets and actions which do need special behaviour.

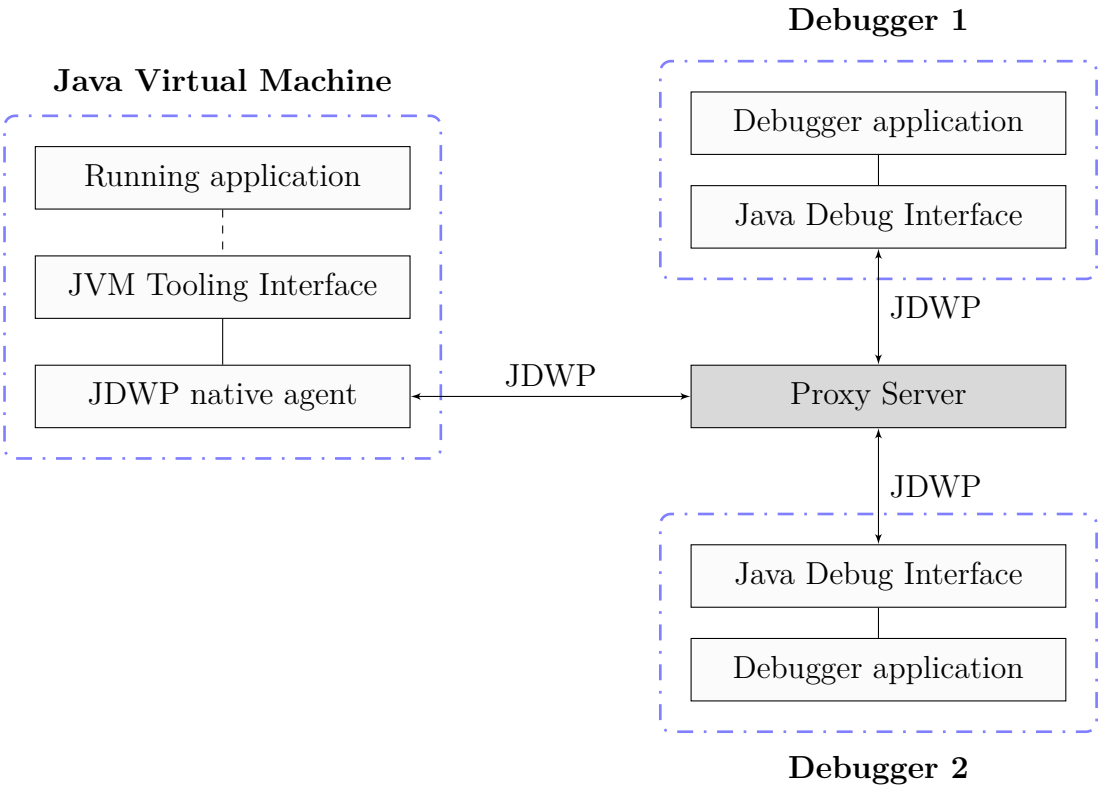

Figure 2: *Graphical representation of the Java Platform Debugger Architecture with JDWP proxy server.*

Figure 2 illustrates how the JPDA should function with a JDWP proxy server in use. The proxy server will intercept the connection between the JVM and the debuggers and start passing and modifying sent packets. In the following section, we'll walk through the implementing, running and testing of the JDWP proxy server.

5 Implementation of JDWP proxy server

This section will describe in detail how the JDWP proxy server was implemented, what problems were encountered and if it solved the limitation worded in the previous section. It will also discuss the ways the program is tested and what features it's still lacking. The following subsections will describe the implementation in the order in which they were written, starting from establishing a connection and ending with testing.

The proxy server was written in Java 8. Java was used because it is a JVM language and so can be used on any machine where debugging Java can be used. Version 8 was used because that's the latest publicly available Java version with long-term support. The integration tests also use Kotlin since it allows one to write shorter, more concise and easy to read tests.

5.1 Connection with the JVM and debuggers

The connection between the proxy server and the JVM and between the proxy server and debuggers is similar, but not identical. JPDA specifies that after a connection is established between JVM and a debugger, the debugger sends a handshake consisting of ASCII string "JDWP-Handshake" to the JVM and the JVM sends back the same string. After the connection is established and the handshake is finished, then JVM stops allowing new connections to that socket until the debugger is disconnected.

The proxy server behaves in a similar fashion, first connecting to the JVM and sending to it the handshake bytes and waiting for the bytes to be sent back. Then it starts to wait for connections from debuggers and will reply to their handshakes once they establish a connection. The proxy server will connect only to a single JVM, but it will allow an unlimited number of debuggers to establish a connection with itself.

Even though JDWP doesn't specify over which communication channel the connection should be made, the most often used way is to use TCP/IP based transport. In such a case the JVM is made to listen for connections from a debugger on a specific port specified in JVM startup arguments. In its current state, the proxy server only supports TCP/IP based transport. Because the JVM has already specified a port on which to connect to the debugger, then in case the proxy server is running on the same machine, then it can't use the same port and must instead start listening for debugger connections on a different port.

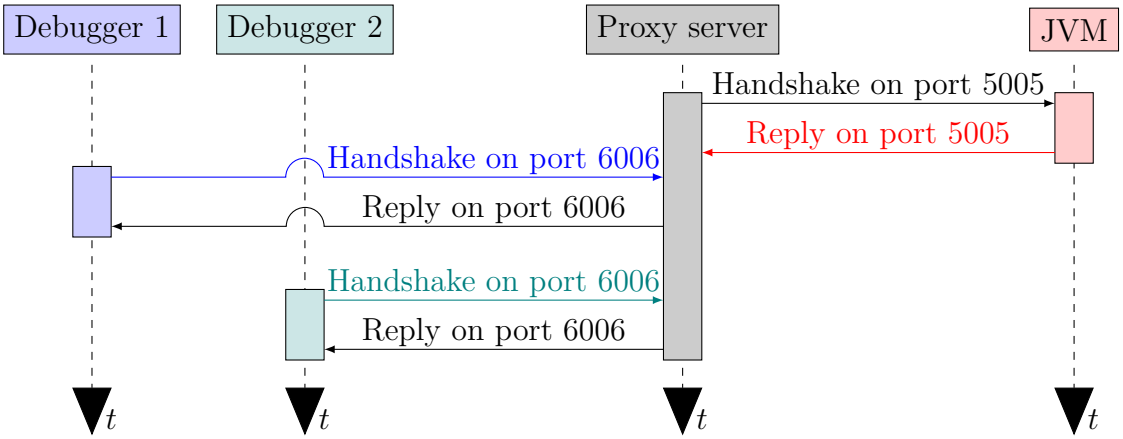

Figure 3: *Illustration of handshakes between JVM, debuggers and the proxy server.*

Figure 3 illustrates how the connection works on different ports for the debuggers and the JVM. In that figure the JVM is using port 5005 for listening to debugger connections, but the proxy server exposes port 6006 for other debuggers to connect to. The debuggers themselves have to connect to the proxy port since the JVM port won't accept any more connections due to the proxy server already being connected to it. The graphic shows clearly how for the JVM, the proxy server starts the handshake, but for the debuggers, it simply replies to the handshake.

5.2 Sending and receiving packets

For handling the packets read from the JVM and the debuggers and the packets written to them, it was decided to use non-blocking IO with message passing. A single thread is handling all the reads and writes regardless of the number of debuggers connected. That thread waits until any stream from debugger or JVM is ready for writing or reading and then checks whether that stream has something to read from or write to. If there is something to be read from the stream, then the packet bytes are read and then the packet is added to a read queue. For writing to the streams, there is a write queue for each stream and if there is anything in that queue, then it is written to the stream. These queues are used for avoiding concurrency problems with other threads. Other threads don't have a direct access to the streams, but must instead get all the read packets from the read queue and if they wish to write something to a stream, then they'll add it to the write queue.

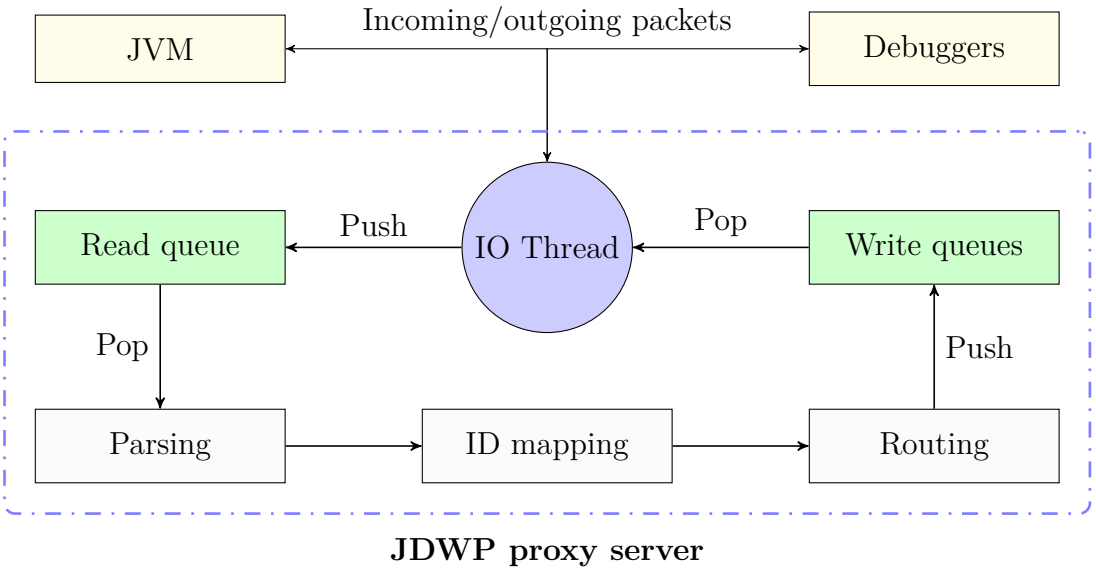

Figure 4: *Graphical representation of the JDWP proxy server logic.*

Figure 4 shows how using the separate thread and queues for transport keeps the reads and writes separate from rest of the proxy server logic and guarantees that only one packet is read from or written to a stream at a time.

5.3 Parsing command and reply packets

Since the packets received from the JVM or debuggers are just sequences of bytes, then it is necessary to parse them before they can be processed further. The header of a packet is quite easy to parse since it's always 11 bytes and can only be either a reply or a command packet. The data bytes of a packet, however, are different for each command and reply packet.

First, it is important to understand what are the values of the command set and command fields of the packet. For a command packet, the fields are available in the packet header. Reply packet doesn't contain these fields, but based on the packet id it's possible to find the command packet and use its command set field and the command field. Due to this, the command packet field values must be stored until a reply has been received.

Using the command set and command field values, it is possible to identify the correct command and the field types contained in its data part. Each command requires writing a custom parser for it to understand the specific command data format.

```
@AutoValue
public abstract class BreakPointEvent extends VirtualMachineEvent {
  public abstract int getRequestId();
  public abstract ThreadId getThread();
  public abstract Location getLocation();

  public EventKind getEventKind() {
    return EventKind.BREAKPOINT;
  }

  public static BreakPointEvent create(int requestId, ThreadId thread, Location location) {
    return new AutoValue_BreakPointEvent(requestId, thread, location);
  }

  public static BreakPointEvent read(DataReader reader) {
    return create(
        reader.readInt(),
        ThreadId.read(reader),
        Location.read(reader)
    );
  }

  public void write(DataWriter writer) {
    writer.writeType(getEventKind());
    writer.writeInt(getRequestId());
    writer.writeType(getThread());
    writer.writeType(getLocation());
  }
}
```

Listing 1: *Breakpoint event representation as a class with read and write methods for reading and writing the bytes for the packet data field.*

The code listing 1 shows how the bytes are parsed for reading for a breakpoint event in the method `read` and how the parsed data is processed back into bytes in method `write`. Helper classes `DataReader` and `DataWriter` are used to make parsing and writing simpler and less error-prone. The meaning of fields for parsing is taken from Oracle documentation on JDWP: `https://docs.oracle.com/javase/10/docs/specs/jdwp/jdwp-protocol.html`

One detail that makes parsing the data field more difficult is the fact that the subfields used in the data field can have variable size depending on the JVM. For example, a ThreadId type of field might take 8 bytes on one JVM, but 5 bytes on another. It is only specified in the JDWP specification that the upper limit for such types is 8 bytes. So in order to find the correct size of these types, the proxy server will catch the `IdSizes` reply packet and read the lengths of the subfields

from there.

5.4 Avoiding id collision

As described in section 3.3, all command packets coming from a single source must have a unique id. This can cause problems for the proxy server since multiple debuggers can send packets with identical ids and the JVM requires each id to be unique. That means the ids must be changed for the incoming packets in order to ensure their uniqueness. Another reason to do that is that it might be necessary for the proxy server to artificially create new packets to be sent to the JVM or debuggers and for these packets, also unique ids are needed. If the id generation is left to the debuggers and the JVM, then the created packet id might conflict with the original received packets.

The logic for changing the ids of debugger command packets is that a new unique id is generated and the original id is changed to the generated one. Then the old id and the generated id are cached as a pair so if later a reply packet is received then it can be mapped back to the original id which the debugger expects to receive. The generated ids are global for all the debuggers, so the possible number of id values for a single debugger is twice smaller when two debuggers are connected. When more debuggers are connected, then the possible number of values will decrease even further.

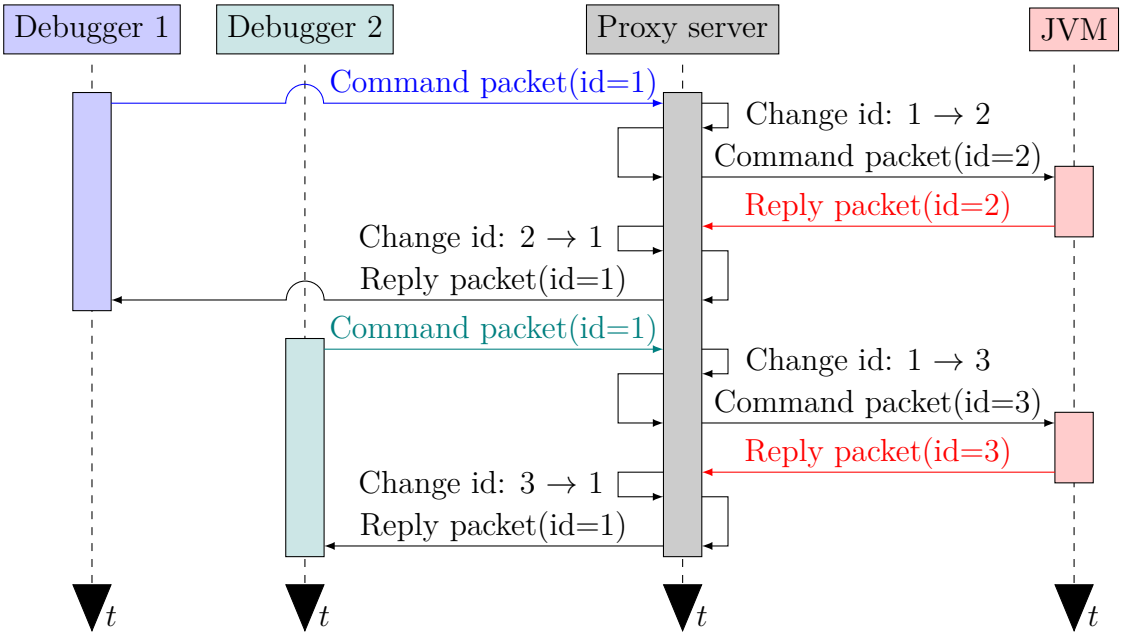

Figure 5: *Illustration of the proxy server changing packet ids for debuggers.*

As can be seen from figure 5, both debuggers send out a command packet with an identical id and later receive a reply with the same id, but the JVM will receive two different ids and regard them as completely different packets.

```
public Packet visit(CommandPacket packet) {
  // Generate new id to ensure every packet reaching vm has a unique id
  // This also makes it easy to later connect reply and command packets and avoid collision
  int newId = getNewId(packet.getId());
  return new CommandPacket(
    newId,
    packet.getCommandSetId(),
    packet.getCommandId(),
    packet.getData(),
    packet.getSource()
  );
}

public Packet visit(ReplyPacket packet) {
  // Restore original id
  int originalId = getOriginalId(packet.getId());
  return new ReplyPacket(
    originalId,
    packet.getErrorCode(),
    packet.getData(),
    packet.getSource()
  );
}
```

Listing 2: *Visitor methods for processing the ids of received and written debugger packets.*

The `visit` methods from the code listing 2 are responsible for creating a changed id for received command packet and restoring the original id of the reply packet.

5.5 Routing and manipulating the packets

Most of the commands sent by the debuggers are mostly stateless and don't manipulate or change the state of the JVM. Such commands usually ask for some information and receive it in the reply without the JVM having to alter its behaviour. No special handling of such commands is needed apart from changing the packet id due to reasons discussed in section 5.4.

There are however commands which either control the execution of the program in JVM or request for notifications about certain events in the future. Some of the most widely used commands which alter the execution of the debugged program

are resume and suspend commands. These tell the JVM to suspend or resume all execution in a single thread or for all threads. The proxy server must have special logic to handle these commands since sending these commands directly to the JVM can cause unpredictable behaviour.

For example, if two debuggers send a suspend command to the proxy server, then the proxy server needs to remember that both debuggers want the JVM to be suspended, but won't suspend the JVM twice. When one of these debuggers later sends a resume command to the proxy server, then it won't be sent on to the JVM because the other debugger still needs the JVM to stay suspended. The resume command is only sent on when both debuggers have sent it. Otherwise, the second debugger would think the JVM is still suspended, but actually, it has been resumed by the first one.

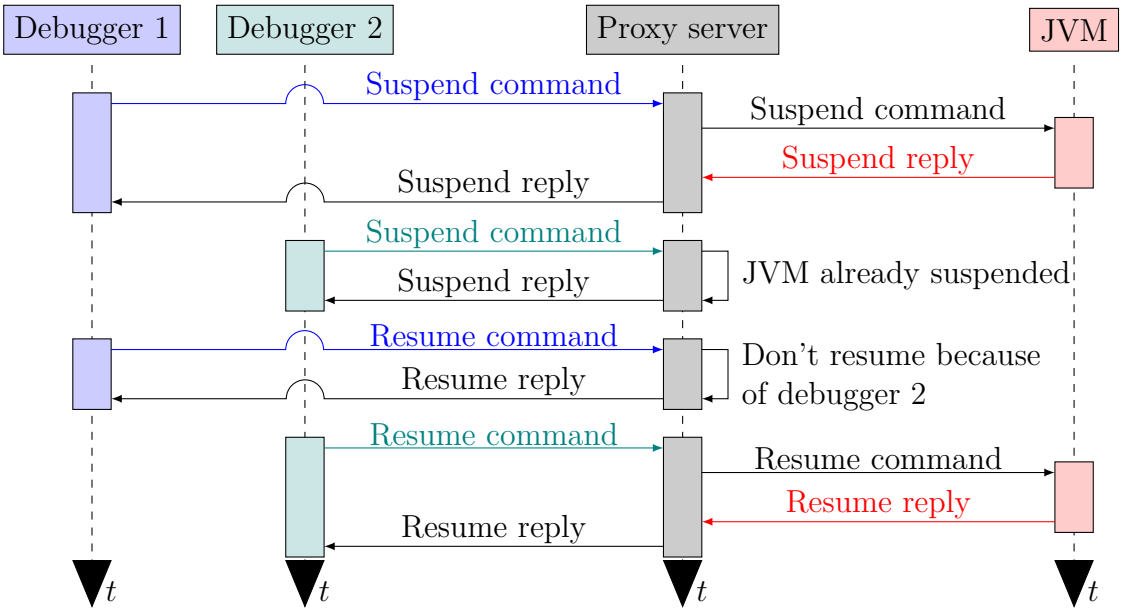

Figure 6: *Illustration of proxy server routing suspend and resume commands.*

As can be seen in figure 6, the first suspend command is forwarded to the JVM, but the second one is not. With resume command it's the opposite: the first resume command is not forwarded, but the second one is. It is also important to mention that even though the command is not always forwarded to the JVM, the proxy server still has to reply to the debugger since each command packet sent from a debugger expects to receive a reply packet from the JVM.

Similar problems arise with events and event requests. A debugger can send event requests to the JVM which will specify for which kind of events the debugger wishes to get notifications. Later when these events occur then the JVM will

send event command packets to the debugger informing what event happened and other details about the event. These events can be about classes loaded or threads started or certain code being invoked in the application.

Events and event requests are tricky for the proxy server to route because each debugger can ask for different kind of events, but the JVM later sends the events that happened at the same time, together. Event requests can also differ on what to do with the event when it occurs. One debugger can specify that the whole JVM should suspend when a breakpoint is hit, but other debugger says it should just suspend a single thread.

Due to these difficulties, each event request is parsed by the proxy server and cached to later know which debugger requested which events. When JVM sends events to the proxy server, then these events are split accordingly and each debugger receives only the events it requested.

```
private void sendEventsToSource(PacketSource source, CompositeEventCommand command) {
  List<VirtualMachineEvent> events = command.getEvents().stream()
      .filter(event -> isEventRequestedBySource(source, event))
      .collect(Collectors.toList());
  if (events.isEmpty()) {
    return;
  }

  proxyCommandStream.write(
      source,
      CompositeEventCommand.create(
        proxyCommandStream.getVmSource().createNewOutputId(),
        command.getSuspendPolicy(),
        events, vmInformation
      )
  );
}

private boolean isEventRequestedBySource(PacketSource source, VirtualMachineEvent event){
  if (event instanceof VmStartEvent || event instanceof VmDeathEvent) {
    return true;
  }
  else {
    RequestIdentifier identifier = new RequestIdentifier(
      event.getEventKind(), event.getRequestId()
    );
    List<PacketSource> sources = eventRequestIdSourceMap.get(identifier);
    return sources.contains(source);
  }
}
```

Listing 3: *Methods for sending only the requested events to a debugger.*

In the code listing 3, the `CompositeEventCommand` object has been parsed from a command packet received from the JVM. The command is then split into a separate command for each debugger with only the events which it has requested. The JVM start and death events are always sent to each debugger as according to the JDWP specification [6].

5.6 Building and running the proxy server

The proxy server is built using the Maven build tool which needs to be installed in order to build the application. Java 8 is also required for building and running the proxy server. Maven handles the compilation and the dependencies of the project and runs all the tests when building the application. In order to build the proxy server, navigate to application source root and invoke command `mvn clean package`.

After that, to run it, navigate to directory `server-logic/target` and invoke the command `java -jar server-logic-0.1-SNAPSHOT.jar jvm.host=<hostname> jvm.port=<port> proxy.port=<port>`. If the `jvm.host` argument is not specified, then it defaults to address "127.0.0.1".

5.7 Testing the proxy server

For testing the proxy server, both unit and integration tests were used. The unit tests were written in Java, but for the integration tests, a simple test framework was written in Kotlin and Java. The test framework will start a simple application in a forked JVM which has debugging enabled. It will also start the proxy server which is attached to the JVM and starts one or two debuggers which connect to the proxy server. After that, the test framework will allow the developer to add breakpoints, run code when breakpoints are hit and suspend/resume the JVM through both debuggers. The test written using this framework can assert that the proxy server will behave correctly when manipulating the execution of the JVM with two concurrent debuggers.

```
val testClass = SimpleBreakpointClass::class.java
val firstLocation = BreakpointUtil.findBreakLocation(testClass, 0)
val secondLocation = BreakpointUtil.findBreakLocation(testClass, 1)

@Test fun `test single breakpoint with 2 debuggers`() =
  runTest(testClass) { jvm, firstDebugger, secondDebugger ->
    val firstBreak = firstDebugger.breakAt(firstLocation) {
      jvm.outputDeque.assertAddedOutput("Before breakpoints")
```

```
    } thenResume {}

    val secondBreak = secondDebugger.breakAt(firstLocation) {
      firstBreak.joinAndTest(4, TimeUnit.SECONDS)
      assertTrue(jvm.outputDeque.isEmpty())
    } thenResume {}

    firstDebugger.allBreakpointSet()
    secondBreak.joinAndTest()
    jvm.waitForExit()
    jvm.outputDeque.assertAddedOutput("After breakpoint 0", "After breakpoint 1")
}
```

Listing 4: *Example of an integration test adding breakpoints and asserting JVM output.*

```
public class SimpleBreakpointClass {
  public static void main(String[] args) {
    System.out.println("Before breakpoints");
    BreakpointUtil.mark(0);
    System.out.println("After breakpoint 0");
    BreakpointUtil.mark(1);
    System.out.println("After breakpoint 1");
  }
}
```

Listing 5: *Test class being executed on the forked JVM.*

The code listing 4 shows how the breakpoints are set and output asserted using the test framework and the listing 5 shows the class that is executed by the JVM.

Below, in listing 6, is an example of a unit test from the proxy server.

```
public class FieldAccessEventTest extends EventTestBase {
  @Test
  public void testReadAndWriteEqualsOriginalEvent() throws ReflectiveOperationException {
    assertWrittenEventEqualsReadEvent(EventKind.FIELD_ACCESS, FieldAccessEvent.
        create(
            randomInt(),
            randomThreadId(),
            randomLocation(),
            randomByte(),
            randomReferenceTypeId(),
            randomFieldId(),
            randomTaggedObjectId()
    ));
  }
  protected <T extends VirtualMachineEvent> void assertWrittenEventEqualsReadEvent(
      EventKind expectedEventKind, T originalEvent) throws ReflectiveOperationException {
```

```
    ByteBuffer buffer = ByteBuffer.allocate(4096);

    writeEvent(buffer, originalEvent);
    EventKind readEventKind = EventKind.read(createReader(buffer));
    T readEvent = readEvent(buffer, (Class<T>) originalEvent.getClass());

    assertEquals(expectedEventKind, readEventKind);
    assertEquals(originalEvent, readEvent);
  }
}
```

Listing 6: *A unit test testing whether reading and writing a data field of a particular JVM event returns identical bytes to the original event.*

The test coverage of the proxy server per lines of code is 72%.

5.8 Known problems and limitations

The implemented proxy server solves most of the use cases for using multiple debuggers concurrently described in section 4.1. There are however small problems and corner cases where it might not function as expected. Not all implemented use cases are covered by the tests yet so these use cases cannot be guaranteed to work. What also needs to be improved is how to handle debuggers connecting and disconnecting while the JVM is suspended or when some packets from the disconnected debugger are still being processed.

The JPDA also has very many different events and filters for the event requests. All of these different events should be tested to make sure there is no undefined behaviour caused by the proxy server.

The data given to the JVM with event requests specifies a suspend policy for the triggered event. It specifies whether the JVM should suspend only the thread in which the event occurred or all threads. It is possible also to suspend no threads and to just notify the debugger that the event occurred. The proxy server currently doesn't support the case where one debugger creates events with global suspend policy and another creates events with single thread suspend policy.

6 Conclusion

This thesis introduced different ways of debugging programs and described how debugging works for the Java Platform. It then detailed why using multiple debuggers concurrently is a needed feature and what is impossible to accomplish without it. Multiple choices were proposed how to fix the problem and why most possibilities were lacking.

In the last section, the implementation of Java Debug Wire Protocol (JDWP) proxy server was described. It brought out how to build and run the proxy server, how it was tested and why and how it really works.

As described in section 5.8, the proxy server has its share of limitations which can be improved upon in the future. One of the most important aspects is to increase test coverage by creating more integration tests for testing wider set of use cases. Also, error management can be improved to make sure the proxy server behaves predictively if one debugger or the JVM were to suddenly disconnect or start sending corrupted packets.

The most important lacking feature for the proxy server functionality is the failure to specially handle breakpoints with a single thread suspend policy. Currently, the proxy server assumes breakpoints are created which suspend all the JVM threads, but it is also possible to create breakpoints and other event requests which suspend only the thread where the event occurred. Such use cases work in some cases, but are currently mostly untested and might cause weird behaviour.

References

[1] Tomáš Martinec. Evaluation of Usefulness of Debugging Tools. Master's thesis, Charles University in Prague, Prague, 2015.

[2] Java Virtual Machine introduction. https://docs.oracle.com/javase/specs/jvms/se10/html/jvms-1.html#jvms-1.2.

[3] Java Platform Debugger Architecture specification. https://docs.oracle.com/javase/10/docs/specs/jpda/architecture.html.

[4] Debugger interface: Java Virtual Machine Tooling Interface. https://docs.oracle.com/javase/10/docs/specs/jpda/architecture.html#jvmti.

[5] Debugger interface: Java Debug Wire Protocol. https://docs.oracle.com/javase/10/docs/specs/jpda/architecture.html#jdwp.

[6] Java Debug Wire Protocol Specification. https://docs.oracle.com/javase/10/docs/specs/jdwp/jdwp-spec.html.

Appendices

A Source code

The source code for the proxy server is available on GitHub at the following link: `https://github.com/veikokaap/jvm-multi-debugger-proxy`. The instructions for building and running the proxy server are in section 5.6.

Non-exclusive licence to reproduce thesis and make thesis public

I, Veiko Kääp (date of birth: 4th of January 1996),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Java Virtual Machine
multi-debugger proxy server

supervised by Vesal Vojdani and Märt Bakhoff

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, May 14, 2018