FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jakub Háva

Monitoring Tool for Distributed Java Applications

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D
Study programme: Computer Science
Study branch: Software Systems

Prague 2017

I declare that I carried out this master 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

Title: Monitoring Tool for Distributed Java Applications

Author: Bc. Jakub Háva

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D, Department of Distributed and Dependable Systems

Abstract: The main goal of this thesis is to create a monitoring platform and library that can be used to monitor distributed Java-based applications. This work is inspired by Google Dapper and shares a concept called "Span" with it. Spans represent a small specific part of the computation and are used to capture state among multiple communicating hosts. In order to be able to collect spans without recompiling the original application's code, instrumentation techniques are highly used in the thesis. The monitoring tool, which is called Distrace, consists of two parts: the native agent and instrumentation server. Users of this platform are supposed to extend the instrumentation server and specify the points in their application's code where new spans should be created and closed. In order to achieve high performance and affect the running application at least as possible, the instrumentation server is used for instrumenting the code. The tool is aimed to have a small foot-print on the monitored applications, should be easy to deploy and is transparent to target applications from the point of view of the final user.

Keywords: monitoring, cluster, instrumentation, distributed systems, performance

I would like to thank my thesis supervisor Dr. Pavel Parízek for leading me throughout the whole thesis and willing to help with any concerns I've ever had. I would also like to thank H2O.ai for being able to write this thesis under their coordination, particularly to Dr. Michal Malohlava for providing very useful technical advice.

Contents

1. Introduction

Lately, the volume of data applications need to handle is significantly increasing. In order to support this scaling trend, the applications are becoming distributed for reasons of scalability, stability, and availability. Not every task may be solved efficiently by distributed applications, however, when it comes to big data, the computational requirements may be higher than single physical node can fulfill. Such distributed applications may run on multiple physical or virtual machines in order to achieve the best performance and the ability to process data significantly large. For this, computation clusters are created where the user interacts with the application as it would be running locally and the cluster should handle the distributed computation internally.

However, with the growth of distributed applications, there is also increasing demand for monitoring and debugging of such applications. Analyzing applications in distributed environment is inherently more complex task comparing to single-node applications where well-known debugging and profiling techniques may be used. All information required for the analysis of single-node applications can usually be collected directly from the application itself. In the case of distributed applications, it is desired to collect the same information as on single-node applications, and additionally, the shared state between the communicating nodes of the distributed application. For instance, an error may occur on one of the computation nodes in the cluster and over time, more and more nodes can become affected. By collecting the relations between the nodes, the analysis tool may be able to use the collected data to determine where the error initially occurred and how it spread over the time.

Simple solution comes to mind to address this issue. Monitoring or debugging tools used in the case of single-node applications may be attached per each application node and collect the information from the nodes separately on each other. This solution does not require any additional tools, however, the state between the application nodes would not be preserved unless the monitored application is already designed to share the required information. However, most of the applications is not designed to transfer the information used for analysis of the application itself for several reasons. It may be hard or unwanted to design the application in a way that all the information required for analysis are transferred between communication nodes by default. Also, the introduction of a new analysis method could require new metrics to be collected, which would, in this case, mean recompiling and new deployment of the application.

For these reasons, several new monitoring and debugging tools have been developed. These tools are usually built on the code instrumentation technique, which is used to alter the code of the monitored application at run-time. The introduced code is usually responsible for collection the relevant information used for the application analysis. A significant advantage of this method is that the original application does not have to be changed when a new additional measurement needs to be performed. Usually, such tools use the instrumentation technique to add special information to the code that is later used to build a so-called distributed stack-trace. A stack-trace in single node application represents the call hierarchy of a method at the given moment. Distributed stack-trace is a very similar concept except that the dependencies between different nodes are preserved and can be seen on the collected stack-trace as well. Therefore, distributed stack-traces allow us to see the desired relations between the nodes of a distributed application. Google Dapper and Zipkin are the most significant available monitoring tools and are discussed later in the thesis.

1.1 Project Goals

This thesis introduces Distrace, a monitoring tool with the similar purpose, sharing some of the concepts mentioned above, however, the goals of this work are different. The Distrace tool should give the user an universal and high-performance solution to monitoring distributed applications. Main goals of the thesis are to create an open-source generic monitoring library with a small footprint on the monitored applications, whilst giving the user the possibility to use a high-level programming language to define the custom instrumentation points.

Main requirements for the Distrace tool are:

- **Small Footprint**
  The mentioned cluster monitoring tools can affect the application performance and memory consumption since they perform instrumentation in the same virtual machine as the monitored application. Distrace is required to have a minimal footprint on the monitored application. The instrumentation is needed in order to inject special information about spans to the application's code, where spans are structures used to collect the shared state between the application nodes. This information is later used for span collection and associating the relationships between spans.

- **Transparency and Universality**
  These two requirements are contradictory to each other. The universal monitoring tool for the majority of Java applications could collect only basic information shared between these applications and the application-specific observations would be missing. To be able to monitor also the specific behavior of some particular application, the developer of this application would be required to manually change the code to also collect specific information to this application. However, this leads to loose of the application-level transparency.

  The introduced tool by this thesis should do some trade-offs between the application-level transparency and the universality of the platform and should try to minimize the monitored application itself.

- **Easiness of Use**
  The application should use high-level programming language for the instrumentation and specification of additional information to be collected. The users of this tool are supposed to work with Java-based language and should not be required to have deeper knowledge about internal Java Virtual Machine structure.

- **Easiness of Deployment**
  The deployment complexity of this tool is also a significant aspect. In order

for developers and testers to use this tool frequently, its deployment and usage have to be relatively easy. This requirement has two sub-parts. Minimizing the configuration of the monitoring tool to the bare minimum and also minimizing the number of artifacts the users of this tool are required to use.

- **Modularity**
  The Distrace tool should be designed in a way that some parts of the whole tool may be substituted by user specific modules. For example, the users should be allowed to switch the default user interface to the user interface they prefer without the need of changing the code of the core tool.

The discussion of different approaches to meeting the requirements above is in Chapter 3.

1.2 Thesis Outline

The thesis starts with the background in Chapter 2. The purpose of this chapter is to give the reader overview of relevant tools to the thesis such as the list of several profiling tools, instrumentation and communication libraries. It also describes the relevant cluster monitoring tools like Google Dapper and Zipkin in more details. The following Chapter 3 contains analysis of the solution and discussion of how specific requirements of the thesis are met. It also mentions the weaknesses of the relevant cluster monitoring tools and describes how the thesis tries to overcome some of the issues. The following Chapter 4 contains the design of the Distrace tool. It starts with Section 4.1 describing overview of the whole tool and depicting the architecture of the whole system. This section is followed by Section 4.2 demonstrating a simple use case for this tool. Further, Chapter 4.1 contains sections for each important part of the application such as the explanation of spans in Section 4.3, the instrumentation server in Section 4.5, the native agent in Section 4.4 and the user interface in Section 4.6. The next Chapter 5 describes several implementation details in more depth and is followed by Chapter 6. The purpose of this chapter is to show a more complex example of how Distrace can be used, in this case on the H_2O machine learning platform. The task is to visualize the hierarchy of internal map-reduce calls inside the H_2O platform and also to see how long each map and reduce operation last. The last Chapter 7 mentions the current limitations of the tool, briefly describes some future plans and also gives the overhead measurements of the Distrace tool on the H_2O example.

2. Background

This chapter covers technologies relevant to the thesis. It starts with an overview of similar monitoring tools for cluster-based applications and follows by a short overview of tools used for debugging of large-scale applications. Different approaches to applications profiling are described in the following part.

In the next several sections the technologies considered to be used or used in the Distrace tool are introduced. The sections cover libraries for bytecode manipulation, communication, logging and also cover important relevant parts of Java libraries such as JNI and JVMTI. Docker is briefly described at the end of this chapter as it is used as the main distribution package of the whole platform.

2.1 Cluster Monitoring Tools

The most significant and relevant platforms on which this thesis is inspired are Google Dapper, Zipkin and HTrace. All three tools serve the same core purpose, which is to monitor large-scale Java-based distributed applications. Zipkin and HTrace are developed according to Google Dapper design and therefore these three platforms share a few similar concepts. The basic concept shared between the platforms is a concept called *Span*. Spans are structures used to collect the shared state between the application nodes and usually encapsulate a few calls between the neighboring nodes with well-defined start and end of the communication. They are formed in hierarchical structures called trace trees, in which the user can see how distributed calls related to each other. Spans are explained in more details later in the following sections.

The following three sections describe the basics of each of the mentioned platforms. Since Zipkin, HTrace and Google Dapper share some basic concepts, only those parts of each platform that are relevant and interesting to the thesis are described.

2.1.1 Google Dapper

Google Dapper [9] is a proprietary software used by Google. It is mainly used as a tool for monitoring large distributed systems and helps with debugging and reasoning about applications performance running on multiple hosts at the same time. Different parts of the monitored system do not have to be written in the same programming language. Google Dapper has three main pillars on which is built:

- **Low overhead**
  Google Dapper should share the same life-cycle as the monitored application itself to capture also the intermittent events and thus low overhead is one of the main design goals of the tool.

- **Application level transparency**
  The developers and users of the application should not need to know about the monitoring tool and are not supposed to change the way how they interact with the system when the monitoring tool is running. It can be

assumed from the paper that achieving application level transparency at Google was easier than it could be in more diverse environments since all the code produced by Google use the same libraries and share similar control flow practices.

- **Scalability**
  Such a system should perform well on data of significantly large scale.

Google Dapper collects the information from distributed applications as distributed traces. The origin of the distributed trace is the communication or task initiator and the trace spans across the nodes in the cluster also participating in the computation or communication.

There were two proposed approaches for obtaining the distributed traces when Google Dapper was developed: black-box and annotation-based monitoring approaches. The black-box approach assumes no additional knowledge about the application whereas the annotation-based approach can make use of additional information via annotations. Google Dapper is mainly using black-box monitoring scheme since most of the control flow and RPC (Remote Procedure Call) subsystems are shared among Google, however, support for custom annotations is provided via additional libraries build on top of the core system. This gives the developer of an application possibility to attach some additional information to spans, which are very application-specific.

In Google Dapper, distributed traces are represented as so-called trace trees, where tree nodes are basic units of work referred to as spans. Spans are related to other spans via dependency edges. These edges represent the relationship between parent span and children of this span. Usually, the edges represent a set of related RPC calls or similar kind of communication. Each span can be uniquely identified using its id. In order to reconstruct the whole trace tree, the monitoring tool needs to be able to identify the span where the computation started. Spans without parent id are called root spans and serve exactly this purpose. Spans can also contain information from multiple hosts, usually from direct neighbors of the span. The structure of a span in Google Dapper platform is described in Figure 2.1. The figure shows a single span encapsulating the client-server communication. It can be seen that events from both nodes are recorded within a single span.

Google Dapper is able to follow distributed control paths thanks to instrumentation of a few common shared libraries among Google developers. This instrumentation is not visible to the final users of the system and therefore Dapper has high-level of application transparency. Instrumentation in Dapper is achieved by tracing three main instrumentation points.

- Dapper attaches a so-called trace context as a thread-local variable to the current thread when the thread handles any kind of traced control path. Trace context is a small data structure containing mainly reference to a current and parent span via their ids.

- In distributed systems, the communication is often done via callbacks. A callback is a method which is usually executed when some execution in a different part of the application has finished. Dapper instruments the callback mechanism so when computation is deferred, traced callbacks still

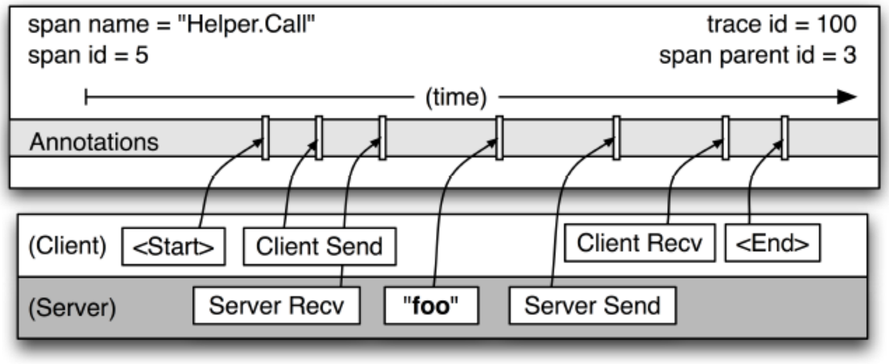

Figure 2.1: Example of a span in Google Dapper, taken from Google Dapper paper [9]

carry around the trace context of the creator and therefore also parent and current span ids.

- Most of the communication in Google is using single RPC framework with language bindings for different languages. This library is instrumented as well to achieve the desired application-level transparency.

The sampling of the captured data has also a positive effect on the low-level overhead of the whole application. As mentioned in the paper, the volume of data processed by Google is significant so only samples are taken at a time.

2.1.2 Zipkin

Zipkin[1] is an open-source distributed tracing system. It is based on Google Dapper technical paper and manages both the collection and lookup of the captured data. Zipkin uses instrumentation and annotations for capturing the data. Some information is recorded automatically, for example, time when a span was created, whereas some are optional. Zipkin has also support for custom application-specific annotations.

Zipkin architecture can bee seen in Figure 2.2. The instrumented application is responsible for creating valid traces. For that reason, Zipkin has set of pre-instrumented libraries ready to be used in order to work well with the whole Zipkin infrastructure. Spans are stored asynchronously in Zipkin to ensure lower overhead. Once a span is created by the application, it is sent to Zipkin collector. In General, Zipkin consists of four components:

- **Zipkin Collector**
  The collector is usually a daemon thread or process which stores, validates and indexes the data for future lookups.

- **Storage**
  Data in Zipkin can be stored in multiple ways which make this is a plug-

[1] More information about Zipkin tracing tool is available at `http://zipkin.io`.

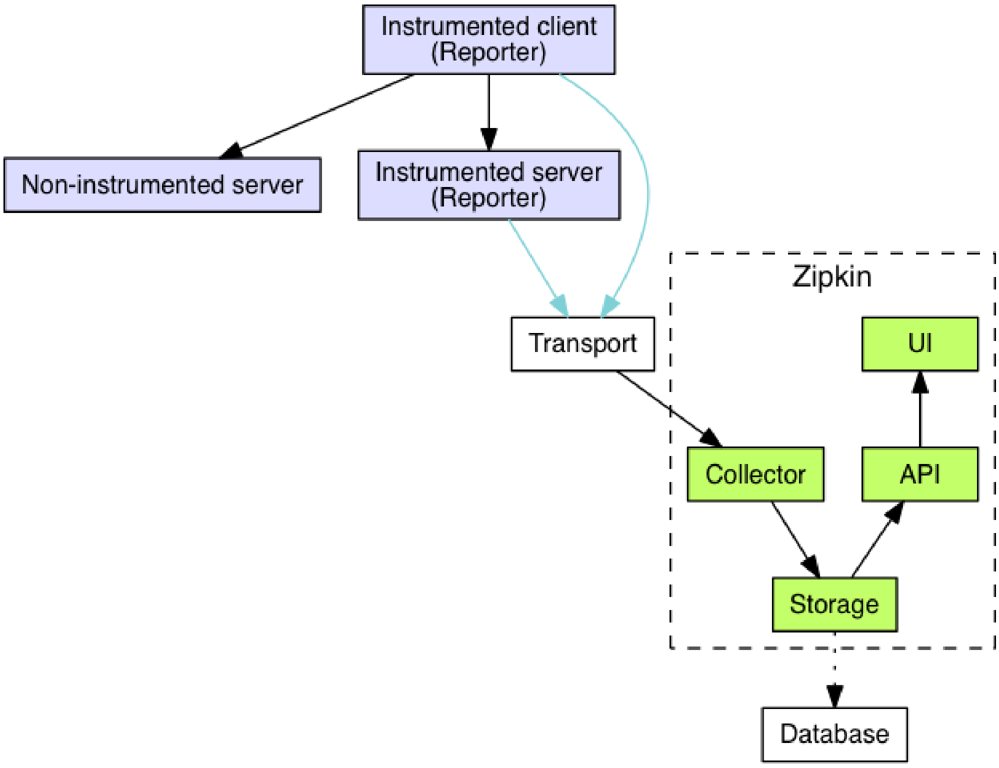

Figure 2.2: Zipkin architecture, from Zipkin Architecture [1]

gable component. For example, data can be stored in Apache Cassandra[2], MySQL[3] or can be send to Zipkin user interface right away without any intermediate storage. The last option is good for handling a small amount of data since the user interface is not supposed to handle storing data of big size.

- **Zipkin Query Service**
  This component acts as a query daemon allowing the user to query various information about spans using simple JSON (Javascript Object Notation)[4] API.

- **Web user interface**
  A basic, but very useful user interface, which allows the user to see whole trace trees and all spans with dependencies between them. The user interface accepts the spans in JSON format. By default, it is available on port 9411.

The Zipkin Web user interface is used as a front-end for the monitoring tool developed in the scope of this thesis. More information on how the user interface is used in Distrace tool is described in more details in Section 4.6.

[2]Apache Cassandra is a free and open-source distributed NoSQL database. It is designed to handle large amount of data and provide high-availability

[3]MySQL is an open-source relational database system.

[4]JSON is a lightweight data-interchangeable format based on object notation used in Javascript programming language

2.2 HTrace

HTrace[5] is a tracing framework created by Cloudera used for monitoring distributed systems written in Java. It is based on Google Dapper as well and shares the same concepts such as spans and trace trees. In order to allow tracing of the application, the users need to manually attach a span identifiers to desired RPC calls (Remote Procedure Calls). These identifiers are then used to create relationships between spans collected on different nodes. HTrace stores the span and trace information in thread-local storage and the user is responsible for making sure this state is transferred to a new thread or node. HTrace has also support for custom spans annotations and thus allows the user to collect application specific information as part of the spans.

The disadvantage of this tool is the need for instrumenting the monitored application in order to allow the tracing and spans collection.

2.3 Tools for Large-Scale Debugging

Standard techniques and tools can be used for debugging distributed applications, however, the main purpose of these tools is to debug single node applications and therefore when applying them on nodes in the distributed application the information about dependencies between different nodes in the cluster is not available. Many tools for large-scale debugging exist, but this section just points out basic ideas behind two different approaches - discovering scaling bugs and behavior based debugging.

2.3.1 Discovering Scaling Bugs

The scalability is one of the most important aspects of distributed systems. It is desirable to know how the platform scales when it process significantly large data and what is the expected scalability trend. It can happen that the platform can run significantly slower on big data than expected after testing on smaller data. This issue is usually called a scaling bug. Tools which can be used to help to discover scaling bugs are for example Vrishna and WuKong [6]. Both of the mention tools are based on the same idea. They build a scaling trend based on data batches of smaller size and the observed scaling trend acts as a boundary. The scaling bug becomes observable when the scaling trend is violated. The first tool, Vrisha, is not able to distinguish which part of the program violated the scaling trend. This is, however, possible in the second tool, WuKong. In comparison to Vrisha, WuKong does not build one scaling trend of the whole application but creates smaller models, each per some control flow structure in desired programming language. All these smaller models represent together the whole scaling trend. When the application observes the scaling bug, WuKong is able to locate the developer in the place in the code where the trend is probably violated.

[5]The project is available at `https://github.com/cloudera/htrace`.

2.3.2 Behavior-based Analysis

The second category of tools used for debugging large scale applications is based on behavior analysis. The basic idea behind these tools is creation of classes of equivalence from different runs and processes of the application. Using this approach, the amount of data used for further inspection is reduced down. These tools are especially helpful when discovering anomalies between different observed application runs. For example, STAT - Stack Trace Analysis Tool [6], is a lightweight and scalable debugging tool used for identifying errors on massive high-performance computing platforms. It gathers stack traces from all parallel executions, merges together stack traces from different processes that have the same calling sequence and based on that creates equivalence classes which make it easier for debugging highly parallel applications.

The other tool used as an example in this category is AutomaDed [6]. This tool creates several models from an application run and can compare them using clustering algorithm with (dis)-similarity metric to discover anomalous behavior. It can also point to specific code region which may be causing the anomaly.

2.4 Profiling Tools

Profiling is a form of dynamic code analysis. It may be used for example for determining how long execution of each part of the system takes compared to the time of whole application run or for example to determine which part of the application uses the most memory. Profiling tools can be divided into two categories:

- **Sampling Profilers**
  Sampling profilers take statistical samples of an application at well-defined points such as method invocations. The points where the application should take samples have to be inserted at the compilation time by the compiler. For example, these profiles are good to collect information about time how long a method run, caller of the method or for example the complete stack trace. However, they are not able to collect any application specific information.

- **Instrumentation Profilers**
  The instrumentation profilers are based on the instrumentation of the application's source code. They record the same kind of information as the sampling profilers and usually give the developer the ability to specify extra points in the code where the application-specific data are recorded.

Sampling profilers usually have less overhead compared to instrumentation profilers, but on the other hand, instrumentation profilers allow to monitor application-specific parts of the application.

Profilers can be also looked at from different point of view and categorized based on the level on which they operate and are able to record the information.

- **System Profilers**
  System profilers operate on operating system level. They can show system

code paths but are not able to capture method calls for example in Java application.

- **Application Specific Profilers**
Generally, application specific profilers are able to collect method calls within the application. For example, JVM profilers can show Java stack traces but are not able to show the further call sequence on the operating system level.

The ideal solution for monitoring purposes of Java applications would be to have information from both kind of profilers, however combining outputs of these profiler types is not straightforward. The profilers used for collecting traces from both operating system and JVM level are usually called mixed-mode profilers. JDK8u60 comes with the solution in a form of extra JVM argument - *XX:+PreserveFramePointer* [4]. The operating system is usually using this field to point to the most recent call on the stack frame and system profilers make use of this field. In case of Java, compilers and virtual machines don't need to use this field since they are able to calculate the offset of the latest stack frame from the stack pointer. This leaves this register available for various kind of JVM optimizations. The *-XX:+PreserveFramePointer* option ensures that JVM abides the frame pointer register and will not use it as a general purpose register. Therefore, both system and JVM stack frames can appear in single call hierarchy. Using the JVM mixed-mode profilers, it is possible to collect stack traces leading to:

- **Page Faults** - page faults are useful to show which JVM code triggered main memory to grow.

- **Context Switches** - context switches are used to determine code paths that often lead to CPU switches.

- **Disk I/O Requests** - capturing Input/Output information allow to see code paths leading for example to blocking seek operation on the hard-drive.

- **TCP Events** - these traces show code paths going from high-level Java code to low-level system methods such as `connect` or `accept`. They can be used to reason about performance and good design of network communication in much more better detail.

- **CPU Cache Misses** - information about cache misses can be used to optimize Java code to make better use of the existing cache hierarchy.

All the information above can be described on special graphs called Flame graphs.

Flame Graphs

Flame Graphs are special graphs introduced by developer Brendan Gregg. Flame graphs are visualization for sampled stack traces, which allows the hot paths in the code to be identified quickly. The output of sampling or instrumentation profiler can be significantly big and therefore visualizing can help to reason about

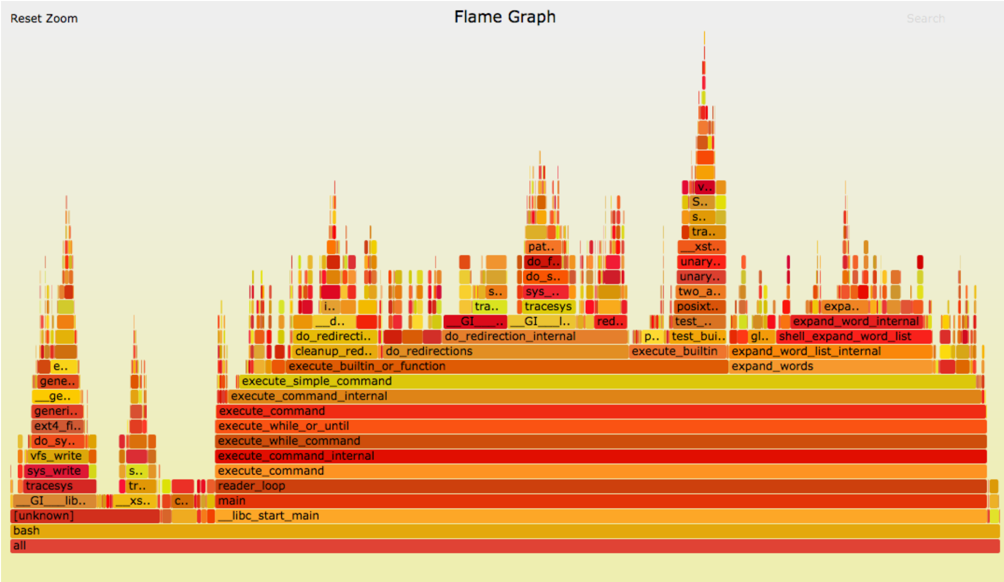

Figure 2.3: Flame Graph example

performance in more comfortable way. The example flame graph is shown in Figure 2.3.

The flame graph is a graph where:

- Each box represents a function call in the stack.
- The **y-axis** shows stack frame depth. The top function is the function which was at the moment of capturing this flame chart on the CPU. All functions underneath of it are its ancestors.
- The **x-axis** shows the population of traces. It does not represent passage of time. The function calls are usually sorted alphabetically.
- The width of each box represents the time of how long the function spent on CPU.
- The colors are not significant, they are just used to visually separate different function calls.

Flame charts can be created in a few simple steps, but it depends on the type of profiler the user wants to use. The three steps are:

1. Capture stack traces. For this step, the profiler of custom choice may be used.
2. Fold stacks. The stacks need to be prepared so Flame graphs can be created out of them. Scripts for most of the major profilers exist and may be used to prepare the folded stack trace. The scripts are available on the official page for the Flame Graphs.
3. Generate the flame graph itself again using the helper script.

The purpose of this really short section is just to introduce the idea of Flame graphs because it's one of the future plans to add support for flame graphs into the Distrace monitoring tool developed in the scope of this thesis. For more information about the flame charts please visit the Brendan Gregg's blog[6].

2.5 Byte Code Manipulation Libraries

Distrace highly depends on the Java bytecode instrumentation and this section gives an overview of four bytecode manipulation libraries considered to be used at the thesis: Javassist, Byte Buddy, CGLib, and ASM. Since it's a core feature of the whole platform and affects both the performance and the usability of the tool, the library was thoroughly reviewed before selected. Byte Buddy library was selected and is therefore described in more detail. However, the reasons for its selection can be found in the following Chapter 3.

2.5.1 ASM

ASM[7] is a low-level high-performance Java bytecode manipulation framework. It can be used to dynamically create new classes or to redefine already existing classes. It works on the bytecode level so the user of this library is expected to understand the JVM bytecode in detail. ASM also operates on event-driven model as it makes use of Visitor design pattern[8] to walk through complex bytecode structures. ASM defines some default visitors such as `FieldVisitor`, `MethodVisitor` or `ClassVisitor`. The ASM project can be a great fit for project requiring a full control over the bytecode creation or inspection since it's low-level nature.

2.5.2 Javassist

Javassist[9] is a well-known bytecode manipulation library built on top of ASM. It allows the Java programs to define new classes at run-time and also to modify class files prior the JVM loads them. It works on higher level of abstraction compared to ASM so the user of this library is not required to work with the low-level bytecode. The advantage of Javassist is that the injected code does not depend on the Javassist library at all. The code to be injected to the existing bytecode is expressed as instances of Java `String` class. The disadvantage of this approach is that the code to be injected is not subject to code inspection in most of the current IDEs. The strings representing the code are compiled at run-time by special Javassist compiler. This run-time compilation works well for most of the common programming structures but for example, auto-boxing and generics are not supported by the compiler [3]. Also, it is important to mention that Javassist does not have support for the code injection itself. Therefore, it can be used for specifying the code which alters the original code but external tool needs to be used to inject the code itself.

[6] The blog is available at `http://www.brendangregg.com/flamegraphs.html`.

[7] The library is hosted at `http://asm.ow2.org`.

[8] The visitor pattern is a way to separate data and the operations, which can be performed on the data.

[9] Javassist library is hosted at `http://jboss-javassist.github.io/javassist/`.

2.5.3 CGLib

CGLib[10] as another byte-code manipulation library built on top of ASM. The main concepts are built around `Enhancer` class, which is used to create proxies by dynamically extending classes at run-time. The proxified class is then used to intercept method calls and field access as is defined by the developer. However, CGLib lacks comprehensive documentation making it harder to even understand the basics.

2.5.4 Byte Buddy

Byte Buddy[11] is fairly new, light-weight and high-level bytecode manipulation library. The library depends only on visitor API of the ASM library which does not further have any other dependencies. It does not require from the user to understand format of Java bytecode, but despite this, it gives the users full flexibility to redefine the bytecode according to their specific needs. Despite it's high-level approach, it still offers great performance [11] and is used at frameworks such as Mockito[12] or Hibernate[13]. Byte Buddy can be used for both code generation and transformation of existing code.

Code Generation

Code generation is done by specifying from which class a new class should be subclassing. In the most generic case, class can be created based on the `Object` class. The newly created class can introduce new methods or intercept methods from its super class. In order to intercept existing methods (change their behavior and return value), the method to be intercepted has to be identified using instances of the `ElementMatchers` class. These matchers allow the developer to identify methods using, for example, their names, number of arguments, return types or associated annotations. The whole list of matchers and also examples how code can be generated is greatly described in the documentation of the Byte Buddy library.

The power behind Byte Buddy is also that it can be used to redefine classes at run-time. This is achieved by several concepts, mainly via Transformers, Interceptors and Advice API.

Code Transformation

In order to tell Byte Buddy what method or field to intercept, the place in code which triggers the interception has to be identified. First, a class containing the desired method for instrumentation needs to be located. It can be done by simply

[10] CGLib library is hosted at `https://github.com/cglib/cglib`.

[11] Byte Buddy library is developed by Rafael Winterhalter and is freely available at `https://github.com/raphw/byte-buddy`. The page contains also a full API documentation and code examples.

[12] Mockito is a mocking framework with a clean API allowing developers to write readable tests. More information is available at `http://mockito.org`.

[13] Hibernate is an open source Java persistence framework project and provides object-relational mapping in the Java programming language. More information is available at `http://hibernate.org`.

specifying the class name or using more complex structures. For example, the element matchers may be used to only consider all classes `A` extending class `B` whilst implementing interface `C` at the same time.

The next step is to define the `Transformer` class itself. Transformers are used to identify methods in the class, which should be instrumented, and they also specify the class responsible for the instrumentation itself. This class may be either Interceptor or Advice and their description is given in more detail in the following section.

The methods to be instrumented can be specified in the transformer using the element matchers. In more detail, `AgentBuilder.Transformer` interface has a method `transform` which takes `DynamicType.Builder` as it's argument. This builder is used to create a single transformer wrapping all the transformers for all classes in the code so the result of this builder can be thought of as a dispatcher of the instrumentation for complete application.

There are two ways how to instrument a class in Byte Buddy. Either via Interceptors or via Advice API.

Interceptors

Interceptor is a class defining the new or changed desired behavior for the method to be instrumented. The demonstration how Byte Buddy uses interceptors is shown on a small example. Let's assume the class `Foo` is the original unchanged class:

```
class Foo {
        String bar() {
                return "bar";
        }
}
```

Let's also assume that the Interceptor is of type `Qux`. The interception of the class `Foo` using the defined interceptor looks like this in schematic code:

```
class Foo {
        // Requires your interceptor class to be known
        static Qux $interceptor;
        String bar() {
                return $interceptor.intercept ();
        }
        static {
                // Requires knowing the framework
                $interceptor = ByteBuddyFramework.defineField(Foo.class);
        }
}
```

It can, therefore, be seen that in the case of interceptors, Byte Buddy does not inline the bytecode to the `Foo` class but requires the interceptor class to be available on the machine where the instrumentation takes place. Also the interceptor field, in this case, `$interceptor`, needs to be initialized before it is used for the first time. This is handled automatically by Byte Buddy framework using the special helper class called `LoadedTypeInitializer`. When the framework discovers that the instrumented class is about to be used for the first time, it initializes the static field to correct interceptor.

However, Byte Buddy also provides ways how to handle setting interceptors field manually. It is a more technical task but required for example in cases when the instrumentation is happening on a different machine where Byte Buddy framework is not available.

1. In Byte Buddy, the initialization strategy can be modified accordingly to the specific needs. The **no-op** strategy can be used, which has the effect of Byte Buddy not trying to initialize the interceptor fields. In this case, the user needs to handle the initialization. `LoadedTypeInitializer` instance can be recorded right before the class is about to be instrumented and this observed initializer can be used later for the initialization. This is especially useful in case the instrumentation happens in a different machine than in which the application is actually running. The initializer can be serialized together with the `Qux Interceptor` class to a machine with the application and a hook in the native agent can be created to call this initializer when the instrumented class is used for the first time.

2. Instead of referring to `Qux` as an instance, it can be delegated to as a static `Qux` class. In this case, no initializers have to be used since no interceptor field is required and the interception is performed via static methods. However, the interceptor class still needs to be known at run-time. This is demonstrated in a simple code bellow.

   ```
   class Foo {
   String bar() {
   // no need to have interceptor field
   return Qux.intercept();
   }
   }
   ```

3. Instead of using Interceptors API, advice API which in-lines the code to the bytecode of the class itself may be used.

Advice API

Advice API is another approach how code can be instrumented in Byte Buddy. This approach is more limited compared to the interceptors, but in cases where it's possible to use it, the code is in-lined into the bytecode of the original class and therefore no other dependencies are required. It is also stated in Byte Buddy documentation that performance of Advice API is better compared to the performance if interceptors. However, the instrumentation using Advice API is only allowed before or after the matched method. This is achieved using the `Advice.onMethodEnter` and `Advice.onMethodExit` annotations.

For example, the following method may be used as an advice:

```
class CustomAdvice {
  @Advice.OnMethodExit
  public static void exit(@Advice.This Callback callback) {
      TraceContext tc = TraceContext.getFromObject(callback);
```

```
        System.out.println("Method finished")
        tc.closeCurrentSpan();
    }
}
```

The method within a class on which the advice should operate needs to be defined. This is achieved by creating an instance of `Transformer` class which specifies the method to be instrumented.

```
class CustomTransformer implements AgentBuilder.Transformer {

  @Override
  public DynamicType.Builder<?> transform(
                              DynamicType.Builder<?> builder,
                              TypeDescription typeDescription,
                              ClassLoader classLoader,
                              JavaModule module) {
    return builder.visit(Advice.to(CustomAdvice.class)
    .on(ElementMatchers.named("run")));
  }
}
```

Lastly, the class on which the transformer should operate needs to be defined. This is done on the main Byte Buddy instrumentation builder object in the following way:

```
.type(is(Task.class))
.transform(new CustomTransformer())
```

Therefore, in this case, the `CustomAdvice` is applied on `Task` in the `run` method.

2.6 Communication Middleware

This thesis consists of several parts written in different languages that need to be able to communicate together. In order to arrange communication in such a heterogeneous environment, following libraries have been inspected.

2.6.1 Raw Sockets

It this case raw sockets is not a library but it is referred to as using raw sockets on their low-level API. Using raw sockets has several advantages and disadvantages. It gives the user full flexibility and the highest possible performance since there isn't any additional layer between the application data and the socket itself. However, integrating different platforms and different languages can be time-consuming. Several frameworks have already been created to hide the implementation details of specific platforms so the user does not need to know about the language or underlying platforms differences.

2.6.2 ZeroMQ

ZeroMQ[14] is a communication library built on top of raw sockets. The core of the library is written in C++, however binding into different languages exist. The library is able to transport messages inside a single process, between different processes on the same node or transfer messages over the network using TCP or also using multicast communication. The library also allows the user to create topologies using one of the many supported communication patterns. For example, publisher-subscriber or request-reply patterns are supported. The library has several benefits compared to raw sockets:

- Hiding the differences between underlying operating systems.
- Message framing - delivering whole messages instead of a stream of bytes.
- Automatic message queuing. The internals takes care of ensuring the messages are sent and received in correct order. The user can send the messages without knowing whether there are other messages in the queue or not.
- Mappings to different languages.
- Ability to create different topologies. An example of a topology can be one-to-many communication pattern, where one socket can be connected to using multiple endpoints.
- Automatic TCP reconnection.
- Support for zero-copy processing of messages.

Zero-copy in ZeroMQ

The library tries to apply a concept called zero-copy whenever possible. When high-performance is expected from a system or network, copying of data is usually considered harmful and should be minimized as possible. The technique of avoiding copies of data is known as zero-copy.

An example of data copying can be transferring data from a memory to a network interface or from a user application to an underlying kernel. It can be seen that zero-copy can't be implemented at all layers. For instance, without copying the data from the kernel to the network interface, no data could be actually exchanged. However, ZeroMQ can achieve zero-copy at least on the application message level so the users can create ZeroMQ messages from their data without any copying which is a significant performance benefit.

2.6.3 Nanomsg

Nanomsg[15] is a socket library shadowing the differences between the underlying operation systems. It offers several communication patterns, is implemented in C and does not have any other dependencies. Generally, it offers very similar features to ZeroMQ since it's heavily based on it.

[14] More information about the library is available at `http://zeromq.org`.

[15] More information about the library is available at `http://nanomsg.org/`.

Unlike ZeroMQ, Nanomsg matches the full POSIX[16] compliance. The author of the library states that since it's implemented in C, the number of memory allocations is drastically reduced compared to C++, where, for example, C++ STL containers are used Also compared to ZeroMQ, objects are not tightly bound to particular threads. This gives the user flexibility to create their custom threading models without significant limitations. Nanomsg also implements zero-copy technique at additional layers which again leads to performance benefits compared to ZeroMQ [10].

As in ZeroMQ, Nanomsq supports the following transport mechanisms:

- **INPROC**
  In-process communication is used for transporting messages within a single process, for example between different threads. In-process address is an arbitrary case-sensitive string starting with `inproc://`.

- **IPC**
  Inter-process communication allows several processes to communicate on the same node. The implementation uses native IPC mechanism available on the target platform.

  On Unix-like systems, IPC addresses are references to files, where both absolute and relative path can be used. The application has to have appropriate rights to read and write from the IPC file in order to allow the communication.

  On Windows, the named pipes are used. The address can be an arbitrary case-sensitive string containing any character except the backslash. On both mentioned platforms, the address has to start with the `ipc://` prefix.

- **TCP**
  TCP is used to transport messages in a reliable manner to a single recipient in a reachable network. The address in format `tcp://interface:port` has to be used when connecting to a node. When binding a node to a specific address, the address in format `tcp://*:port` has to be used.

Nanomsg can be used via the core C library, but several language mappings for different languages exist as well, which makes working with the library easier.

C++11 Mapping

Nanomsgxx[17] a C++11 mapping for Nanomsg library. It is a small layer built on top of the core library making the API more C++11 friendly. Especially, there is no need to explicitly tell when to release resources, since it's handled automatically in the class descriptors. The `nnxx::message` abstraction over Nanomsg `nn::message` automatically manages buffers for zero-copy and also errors are reported using the exceptions which are sub-classes from `std::system_error`.

[16] POSIX is a family of standards used to maintain compatibility between Unix-like operating systems by specifying a well-known interface for system methods [5].

[17] More information about Nanomsgxx mapping is available at `https://github.com/achille-roussel/nanomsgxx`.

Java Mapping

Several Java bindings of Nanomsg library exists, but only jnanomsg library[18] is described here. This language binding is built on top of JNA (Java Native Access)[19] library. It offers all the functionalities offered by the core library, but also introduces non-blocking sockets exposed via a callback interface.

2.7 Java Libraries

This section describes some fundamental Java-related libraries and technologies on which this thesis heavily depends. Firstly, Java Virtual Machine Tool Interface (JVMTI) is described, followed by the basic introduction to the Java Native Interface.

2.7.1 JVMTI

The JVM Tool Interface is an interface used by development and monitoring tools for communication with JVM. It allows the user to monitor and control the application running in Java virtual machine. An application communicating with the JVM using JVMTI is usually called an agent. Agents are notified via events occurring inside JVM and can react upon them. Agents run in the same process as the application itself, which reduces the delay in the communication between the application and the agent. Since JVMTI is an interface written in C, agents can be written in C or C++.

JVMTI supports two modes of how an agent can be started. It can be started either in **OnLoad** phase or in **Live** phase. In the **OnLoad** phase, the client is started together with the application and agent location can be specified using 2 arguments:

- `-agentlib:<agent-lib-name>=<options>`
  In this case, the agent library name to load is specified and it is loaded using platform specific manner.

- `-agentpath:<path-to-agent>=<options>`
  In this case, the path to a location of the agent library is specified and the agent is loaded from there.

In the **Live** phase, the agent is dynamically attached to a running application. This approach is more flexible since it is not required to attach the agent library to the monitored application in advance. However, it brings several limitations as well.

The goal of this section is not to describe full JVMTI functionality, but just give the reader a brief introduction to the interface. For more details about

[18] More information about jnanomsg mapping for Nanomsg library is available at `http://niwinz.github.io/jnanomsg/latest/`.

[19] Java Native Library is a library allowing to use native shared libraries from Java without using JNI or native code. For more information about the library, please see `https://github.com/java-native-access/jna`.

JVMTI, please visit the official documentation[20]. The following sections try to briefly describe the important parts of JVMTI relevant to the thesis.

JVMTI Agent Initialization

When an agent is started, the following JVMTI method is always called: `Agent_OnLoad(JavaVM *jvm, char *options, void *reserved)`. This method should contain the agent initialization specific for the application. Usually, the agent initialization process consists of several phases:

1. Optionally, parse arguments passed to the JVMTI agent.
2. Initialize JVMTI environment in order to be able to communicate with the observed application. JVMTI does not handle threads switches automatically, so proper locking and thread management fully depend on the user code.
3. Register capabilities of the JVMTI agent. The capabilities specify what are the operations the JVMTI agent can perform. The agent can be, for example, allowed to re-transform classes or react to different class hook events.
4. Register events the agent should react to. JVMTI does not inform the agent about all events by default, these events have to be manually defined.
5. Register callbacks for the events the agent is interested in. Even though the JVMTI supports more events, the interesting events are:
    - `cbClassLoad`
    - `cbClassPrepare`
    - `cbClassFileLoadHook`
    - `callbackVMInit`
    - `callbackVMDeath`
6. Optionally, initializing phase is also good for creating locks which may be later used for synchronization between different JVMTI threads.

The user of JVMTI is also required to manually implement queuing and locking when processing multiple JVMTI events at the same time since the framework is not designed to handle these cases. [7].

JVMTI basic callbacks

As mentioned above, there are several events sent from the observed Java application. When instrumenting the code of the applications, the following callbacks are used to capture the mentioned events.

- `cbClassLoad` - triggered when a class has been loaded by target JVM.

[20]The official JVM Tool Interface documentation describes the full API and is available at `https://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

- `cbClassPrepare` - triggered when a class has been prepared by target JVM. All static fields, methods, and implemented interfaces are available at this point, but no code has been executed at this phase.
- `cbClassFileLoadHook` - triggered when the virtual machine obtains class file data, but before the class is loaded. Usually, class instrumentation is based on this hook, since the callback contains output argument for the instrumented bytecode.
- `callbackVMInit` - triggered when virtual machine is initialized.
- `callbackVMDeath` - triggered when the virtual machine has been closed. This event is triggered in both planned and forcible stop.

2.7.2 JNI

Java Native Interface is a framework allowing Java code running in Java Virtual Machine to call native applications (usually written in C or C++). It also allows native applications to access and call Java methods. All JNI operations require an instance of class `JNIEnv` to be available. This environment is always bound to a specific thread and manages the connection to the Java virtual machine. When calling a Java method from the native application, the correct method has to be first found. This is achieved by specifying the types and method signature of the method.

Java Types Mapping

For each Java primitive type, there is a corresponding native type in JNI. Native types always start with the `j` as the prefix, for example, `boolean` is a Java type whereas `jboolean` as a native type. All other JNI reference types are referred to via `jobject` class. This means that Java arrays are accessed as `jobject` as well since at this level they are referred to as Java objects. The most important question is how the types in method signatures can be specified. There is a mapping assigning each type a signature, which is used exactly for this purpose. The following Table 2.1 is based on the JNI documentation[21]. and describes the mapping in detail:

Type Signature	Java Type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
L fully-qualified-class ;	fully-qualified-class
[type	type[]
(arg-types) ret-type	method type

[21] The original and more complete documentation for JNI types mapping is available at `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html`.

Table 2.1: Mapping type signatures to Java types.

For example, the method:
`xx.yy.Person foo(int n; boolean[] arr, String s);` has the following signature: `(I[ZLjava/lang/String;])Lxx/yy/Person;`

Note that in JNI, the path elements in the fully qualified class name are separated by slashes instead of dots.

Example JNI Method Call

The method bellow demonstrates how JNI can be used to call a Java method `getClassLoader` from the native environment.

```
jobject getClassLoaderForClass(JNIEnv *jni, jclass clazz){
// Get the class object's class descriptor
// (jclass inherits from jobject)
 jclass clsClazz = jni->GetObjectClass(clazz);
// Find the getClassLoader() method in the class object
jmethodID methodId = jni->GetMethodID(
                clsClazz,
                "getClassLoader", // name of the Java method
                "()Ljava/lang/ClassLoader;"); // method signature
return (jobject) jni->CallObjectMethod(clazz, methodId);
}
```

It can be seen that the reference to the method needs to be obtained at first. This reference is used later for the method invocation itself. For the performance reasons, it is a good practice to cache the references to Java methods or objects which are accessed from JNI often, since creating the reference has some initial overhead.

2.7.3 Relevant Aspects of the Java Language

This section covers selected areas of the Java programming language and run-time platform relevant to the thesis. It briefly describes the class loading process for dynamically loaded classes. This is followed by the explanation of two important class loaders relevant to the thesis and lastly, `ServiceLoader` class is shortly described.

Class Loading Process

Java allows programs to load classes dynamically at run-time. This is achieved by the following process:

1. **Loading** - Load the bytecode from a class file.

2. **Linking** - Linking is the process of incorporating a new class to the run-time state of the JVM. This phase consists of three sub-phases:

   (a) **Verification** - Ensure that type in the binary format is correct and respects JVM restrictions.

(b) **Preparation** - This phase mainly allocates the memory for fields inside the loaded type.

(c) **Resolution** - This phase is optional and depends on JVM implementation. Resolution is the process of transformation symbolic references in the type's constant pool into direct references. The implementation may decide to behave in a lazy way and delay resolution for the time when the type is accessed for the first time or behave in an eager way and resolve all types in advance. Constant pool contains all references to variables and methods, found during the compilation time, in the class file.

3. **Linking Phase** - class (static) variables are initialized to initial values.

Relevant Class Loaders

There are several class loaders used natively in Java. However, this section describes only two, which are referenced later in the thesis.

- **Bootstrap class loader**
  This class loader is used to load system classes. When using the native agent, even classes loaded by bootstrap class loader can be instrumented and thus behavior of standard Java classes can be changed.

- **sun.reflect.DelegatingClassLoader**
  This class loader is used on the Oracle JVM as the effect of a mechanism called *inflation* [2]. Usually, reflective access to a method or a field is initially performed via JNI calls. When Oracle JVM determines that there is a repetition in calling the same method or the same field via JNI (reflection), it creates a synthetic class[22], which is used to perform this call without the JNI. This has initial speed overhead, but at the end, it speeds up the reflection calls. The classes created for this purpose are loaded and managed by exactly this class loader.

ServiceLoader Class

`ServiceLoader` class is used to locate and load service providers. The service provider is an implementation of a service that is usually defined as a set of methods inside an abstract class or interface.

Service loader is used to load specific service providers at run-time. The group of service providers to be loaded can be specified via the service type, usually type of an interface or abstract class. The available service providers have to be defined in the META-INF folder of the application's JAR (Java Archive) distribution. For example, imagine there are service A and two implementations, `Impl1` and `Impl2`. In that case, META-INF folder should contain text file named A containing lines:

```
Impl1
Impl1
```

[22] Class created dynamically at run-time

As a result, service loaders can be used to extend the application capabilities without changing the source code. When a new implementation of a service should be supported, it just needs to be registered inside the META-INF folder and the application will automatically use the new service provider together with the rest of the service providers defined earlier.

2.8 Logging Libraries

Logging can have a negative effect on the performance of the application, but sometimes it's necessary to have information from various application runs to be able to locate bugs or discover the wrong configuration. Since one of the thesis's requirements is low-overhead, the selection of logging library is important for the performance of the Distrace as well.

Spdlog[23] is a fast, header only logging library written in C++11 on which this project is based on. It allows both synchronous and asynchronous logging and custom message formatting.

2.9 Docker

Docker[24] is an open source project used to pack, ship and run any application as a lightweight container. It is used to package applications in prepared environments so the user does not need to worry about configuration and downloading the correct dependencies for the application.

Docker Compose is an extension built on top of Docker allowing the user to specify multi-container startup script. This script can define dependencies between different containers which leads to a simple and automated way how to start a group of related applications in separated environments using one single call.

[23] More information about Spdlog is available at `https://github.com/gabime/spdlog`.
[24] More information about Docker is available at `https://www.docker.com`.

3. Analysis

This chapter discusses different approaches to fulfill the desired requirements. It also provides the arguments for the selection of some specific libraries later used in the Distrace tool. It starts with the section summarizing the limitations of the similar monitoring solutions and is followed by several sections where each of them is dedicated for a single requirement. There we discuss in more detail what solution is the best for meeting the desired requirement. This chapter ends with the summary of desired and unwanted features.

3.1 Limitations of Similar Solutions

In this thesis, we try to overcome some of the limitations of the relevant monitoring tools and give users the alternative solution. One goal of the Distrace tool is to be an open-source solution, which is in contrast to the proprietary Google Dapper described in Section 2.1.1. Google Dapper is also targeted to the only specific type of applications sharing the same code structure and libraries used at Google. Therefore, it can be said that there is a certain lack of the universality at Google Dapper.

Zipkin, more described in Section 2.1.2, is stable open-source tracing system. Zipkin creates several pre-instrumented libraries which may be used at the monitored application to communicate with the Zipkin backend. These libraries try to minimize a number of code changes in application sources, however, the user is still required to add custom annotations or change the default tracing mechanism. The similar holds for very similar HTrace tool which is described in more details in Section 2.2. HTrace also performs the instrumentation in the same JVM (Java Virtual Machine) where the application is running which can have a performance impact on the application itself.

The Distrace tool proposed by this thesis attempts to address these problems and find solutions how applications can be instrumented without the need of changing the source code and have the minimal performance impact on the running application. It, however, shares the same concepts as *Span* and *Distributed traces* with the platforms already mentioned.

3.2 Small Footprint and High Performance

The mentioned cluster monitoring tools, such as Google Dapper and Zipkin, can affect the application performance and memory consumption since they perform instrumentation in the same virtual machine as the monitored application. One of the thesis requirements is to have minimal footprint on the monitored application.

Since the instrumentation is the core functionality of the system, it directly affects the performance of the application. Different instrumentation methods are discussed in the following few paragraphs to give the reader insights into how each method can affect the application's performance. Two standard ways of instrumentation exist - either using the agent or Java agent. The advantages and disadvantages of these two approaches are discussed here together with the arguments for the final solution which is actually a compromise of both techniques.

3.2.1 Java Agent

Java agents are used for instrumenting the applications on the Java language level, where the user does not need to worry about the JVM internals. For example, Byte Buddy, Javassist or CGLib may be used for this purpose. Usually, the programmer extends and creates custom class file transformers and the agent internals takes care of applying the code when required.

The advantage of this approach is obvious - the ability to write the instrumentation in the high-level language without the knowledge of the underlying bytecode. The distributions of Java Agents is also platform independent since they are packaged inside JAR (Java ARchive) files as the rest of Java classes.

The disadvantages of this approach are usually the performance and the flexibility of the agent. Objects created by Java agents are affected by garbage collection of the monitored application and thus can have a negative impact on the application itself. Also, the objects created from the agent are put on the application's heap and therefore consume the memory of the application. For the reasons above, it can happen that the observed information via Java Agent can be influenced by the monitoring process itself. Java agents also can not be used to respond to internal JVM events. It is important to note that Java system classes can not be instrumented when using this method.

3.2.2 Native Agent

Native agents are used for monitoring and instrumenting the applications in the low-level programming language (C, C++) using JVMTI and JNI. Native agents are written as native libraries for specific platforms and therefore the packaging is not platform independent.

The disadvantage of this method can be that the agent has to be written in C or C++, but on the other hand, this approach gives the developer the full flexibility in the instrumentation and monitoring of the JVM state. For example, even the System classes can be instrumented using this approach and callbacks may be created to respond to several JVM internal events such as start or end of the garbage collection process, creation of a new instance of a specific class or thread switches. The native agent is running as part of the Java process and therefore any resource-demanding computation can have a negative performance impact on the application's performance as well as in the previous agent approach. However, objects created in the native agent are not subject to garbage collection and are not created on the heap, except when they are created using JNI in the target Java application.

The significant technical disadvantage of this approach is that it does not provide any helper methods to help with the code instrumentation and generally, there is a lack of stable instrumentation libraries written in C++ or C that could be used inside the agent. The developer of the native agent has to write all the required methods for extracting the relevant parts of the bytecode and the instrumentation itself.

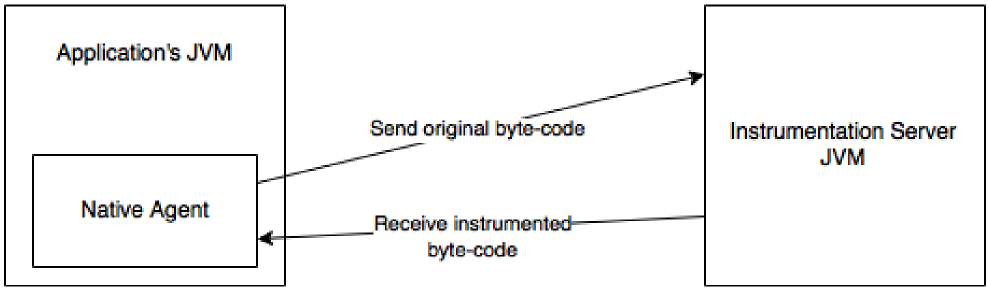

Figure 3.1: Sketch of the chosen approach.

3.2.3 Chosen Method

The desired solution should be able to instrument the code without affecting the performance and memory of the monitored application whilst still having access to internal JVM state and allowing the developer to use a high-level programming language to write the instrumentation code. For these reasons, the compromise between the proposed approaches has been chosen, together with the introduction of the special Java process used specifically for the instrumentation. The solution can be seen in Figure 3.1.

In more detail, the native agent is used to communicate with the application being monitored. When a class needs to be instrumented, the native agent sends the class's bytecode to a special instrumentation server JVM. This machine handles the instrumentation and sends the instrumented bytecode back to the native agent. Therefore the native agent is only used for collecting important internal JVM information, sending classes for instrumentation and receiving the instrumented classes back. The instrumentation does not happen in the same JVM as the monitored application, which allows the tool to have a minimal performance impact on the application. Also, since the instrumentation is not done in the native agent, but in the instrumentation server based on Java, this machine can use any of the available bytecode manipulation tools which operate on Java language level and therefore, there is no need to implement the bytecode manipulation library completely from scratch in the native agent. Another advantage of this solution is that even the application's Java system classes may be instrumented in Java language on the instrumentation server.

Byte Buddy library was selected for the bytecode manipulation within the instrumentation server as it allows to write the instrumentation in Java without the deep knowledge about the Java bytecode, which is necessary when using ASM library. Also, the library is supposed to have a really good performance results based on the benchmarks conducted by the author of the library [11]. Compared to Javassist, the code is not written inside Java strings, which has the effect that the instrumentation code can be validated in today's IDE (Integrated Development Environment) during compilation time and bugs in the instrumentation code can be found easier. Compared to CGLib, the API of the library is well-documented and the library is under active development. Byte Buddy is also a highly configurable library, which was also the significant reason for choosing it as the tool for instrumenting classes inside different JVM than where they are actually used. Achieving the instrumentation in the secondary JVM turned out to be challenging part of the thesis and the technical aspects of the solutions are described later in the thesis.

The disadvantage of the chosen approach is that the native agent has to send the bytecode to the instrumentation server and wait for the instrumented bytecode. However several optimizations have been implemented to minimize this delay as much as possible. More information about these optimizations can be found in Section 4.5

3.2.4 Alternative solutions

Two alternative instrumentation solutions were analyzed but rejected at the end. The first alternative solution was to perform the instrumentation right in the agent, even for the price of affecting the application's performance by instrumenting in the same JVM. However this solution required to write the instrumentation from scratch in C++ or C language, since there are no stable libraries for this purpose. Even though this would be possible, it would take a significant amount of time and also, it was not the goal of the thesis to create such a library. The performance impact on the application's was also the important reason for rejecting this method.

The other alternative solution was based on the idea of running multiple Java Virtual Machines inside one native process. In particular, that would mean running the application and the instrumentation server inside the same process. This would have the same negative performance impacts as the solution above, however it would allow the developer to perform the instrumentation in Java programming language compared to C++ or C. Also all the communication between the machines would be only inside one single process compared to the chosen solution where the communication needs to be handled over the network or between different processes. However, as of JDK/JRE 1.2, creation of multiple virtual machines in a single process is not supported [8].

3.2.5 Chosen Communication Layer

The selection of the tool used for the communication between the native agent and the instrumentation server was also important decision and we choose Nanomsg for the tool purposes. Comparing to raw sockets approach, Nanomsg hides the platform specific aspects of the socket communication. It has also several performance benefits and general improvements over the well-known ZeroMQ library, such as better threading and more sophisticated implementation of the zero-copy technique. The mappings of this library into Java and C++ languages mentioned in Section 2.6.3 were also perfect fit into the tool.

3.3 Transparency and Universality

One of the most important goals of the thesis is to achieve high level of application transparency while ensuring the tool universality. Each of these two requirements directly affects the second one and therefore we needed to find compromise between these two.

The rejected approach was to create an universal monitoring tool similar to Google Dapper, but allow to monitor mostly shared aspects of distributed appli-

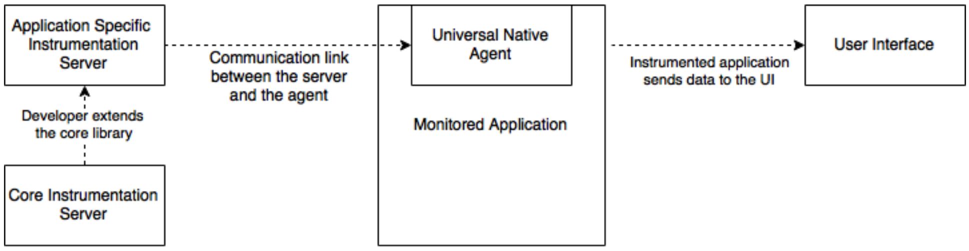

Figure 3.2: .

cations. This solution would give the user great flexibility, universality and would also ensure that users don't need to extend the library. However, implementing such a solution was decided as not a feasible task. Every platform or application is different in its architecture or in the way how it communicates and therefore identifying the shared parts between all distributed applications is an extremely challenging task. Also, such instrumentation tool could only instrument very basic information about Java-based programs.

The architecture of the chosen approach can be seen in Figure 3.2. The chosen approach for the monitoring tool was to design it as an general extendable instrumentation library with two kinds of users - developer and end user. The tool should be available as a library with well-defined methods for defining the instrumentation points and specifying the custom annotations. The developer of the application is responsible for extending this library and based on it, create application-specific monitoring tool. Therefore, the developer has to have understanding of the monitored application and has to know the type of information, which are to be collected. The developer also takes care of defining where a new span starts and when existing span ends. The end-user of this final library is just responsible for starting the application with the monitoring tool attached and does not need to understand the application internals.

The advantage of this approach is that only core instrumentation library needs to be created that is universal and generic to all Java-based applications. On the other hand, the application's developer is required to extend the library in order to provide the desired monitoring functionality. However, for this price, the original application can remain unchanged and from the end-user point of view, the monitoring tool is completely transparent to the monitored application.

In more detail, the core monitoring library acts as the instrumentation server mentioned in the previous section. It can be seen in Figure 3.2 that by extending the core library, the developer creates an application-specific instrumentation server, which specifies the parts of the original application to be instrumented. The application-specific instrumentation server then communicates with the native agent, which is universal to all Java applications. It is important to mention that the instrumentation server is used only for instrumentation. The instrumented code is responsible for storing data to the user interface.

3.4 Easiness of Use

This requirement is highly connected to the previous one. The easiness of use is also important goal of the application. The usage of the core monitoring system is separated into two user groups, developers and end users. The goal is to not require from the end user to know the internal structure of the application and the monitoring platform. This is achieved by assuming that the developer, who is responsible for extending the library, can handle this technicalities. The user is supposed to work with the user interface and to read the results of the monitoring run.

However, it is also our goal to ensure that the developer responsible for extending the core monitoring tool needs to write a few lines of code as possible and does not need to have a deep knowledge about the JVM internals or application bytecode. This is also the reason why Java language and Byte Buddy library are used for the instrumentation on the server. The Byte Buddy library provides several very concise ways how to define the application-specific instrumentation. To shield the developer from the internals, all low-level core code is hidden from the developer in the native agent, which is universal to all Java applications and the developer is not supposed to change its implementation.

3.5 Easiness of Deployment

In order to ensure the easiness of deployment, the number of artifacts used by the tool should be as low as possible. Also the application should have understandable and relatively small configuration so the users can effectively set up the application for their desired needs.

Based on the discussion in the previous sections, the tool should have only two final artifacts - the universal native agent and the core monitoring server, which is supposed to be extended by the application developer for specific application needs.

The deployment of this tool should be also simplified at certain use-cases by starting the application-specific instrumentation server automatically from the native agent. In most cases, the user should only specify path to the instrumentation library and attach the native agent to the application. Several deployment strategies exist and are discussed later in the thesis.

3.6 Modularity

It is the also goal of the tool to allow the user to replace some of the default application modules of the whole platform by specific custom modules without the need of rewriting the complete application.

The extendable modules should be the interface presenting the observed data and the collectors bringing the data from the application nodes to the user interface. Whilst default implementation are available in the tool, the users have the possibility to plug-in custom user interface or more advanced data collectors. The modules have to meet some specific criteria in order to replace the default implementations, however the core implementation is left up to the user.

The main reason for this solution is that a lot of monitoring frameworks already exist. For example, an user interface provided by some different tool may be already deployed or some platform with already defined data collection may be used. The thesis tries to support these use-cases and tries to minimize the changes of the environment where this monitoring tool runs. This also leads to easier deployment of the platform.

3.6.1 Selection of the User Interface

Zipkin user interface was selected as the default user interface for the Distrace tool. The main reasons for its selection were the simplicity of the interface and ease of use. It also fulfills our visualization requirements as it allows the user to see dependencies between spans and also whole trace tree as well. However, as mentioned above, the monitoring platform is not tightly-coupled to this user interface. It is described later in the thesis how the user can create a plug-in allowing to use custom user interface.

3.7 Summary of Features

This section just summarize the list of desired and not-desired features based on the previous background and analysis.

Desired features and properties of the thesis based on the analysis are:

- Ability to collect distributed traces and spans via the instrumentation.
- Native agent implemented in C++ to be attached to a Java application. The native agent is universal to all Java applications and is not supposed to be changed.
- Instrumentation server implemented in Java used for the instrumentation. The instrumentation server acts as the core library, which can be extended by the application's developer for the specific application. Byte Buddy library is to be used within the server for the code transformations and definitions of the instrumentation points.
- Nanomsg library is to be used as the communication layer between the native agent and the instrumentation server.
- Support for the Zipkin user interface. The user interface is used to visualize the collected spans.
- Ability to replace the default user interface with the custom user interface.

Selected features and properties which are not desired by this thesis:

- Support for creating a custom user interface.
- Support for creating an universal instrumentation tool, which would not require the developer to specify the instrumentation points and specify start and end of spans.
- Implementing bytecode instrumentation in C++.
- Instrumenting the Java bytecode in the same machine as the running application.

4. Design

This chapter describes the design of the whole platform in details, however, implementation specifics of some parts of the Distrace tool are described in the following Chapter 5. The current chapter starts with the high-level overview of the complete platform and interactions between its parts. It is followed by a simple use-case to give the reader an idea how the Distrace tool is can be used.

Spans and their format are described next, followed by design of the native agent and instrumentation server. This chapter ends with description of the default Zipkin user interface and also JSON format, in which the user interface accepts the data from the instrumentation server.

4.1 Overview

The main purpose of the Distrace tool is to collect distributed traces. In order to achieve that, the Distrace tool is based on the concept of spans. Spans are used to denote some specific part of the communication between the communicating nodes and are important elements for building the whole trace trees. Trace trees consist of several spans and represent the complete task or communication, where a span inside the trace usually represents a few remote procedure calls between two neighboring nodes. The node initiating the trace creates so-called parent span and new calls started within the scope of this span create new nested spans. Created spans can be exported using different span exporters and can be sent to the user interface using various data collectors. Span exporters export spans in the desired format on disk or the network for further data collection. The collected data are used for spans visualization in the user interface. The user interface receives the spans from the span exporters or data collectors and presents them to the user in a form of trace trees.

Definition of when a new span is to be created and when an existing span needs to be closed is done by a developer by extending the core instrumentation server library. The created extended instrumentation server is then used for instrumenting the classes of the original application, however, spans are still located in the scope of the application itself. In order to obtain the class files for transformation, the native agent runs as part of the monitored application and sends the desired classes to the instrumentation server. The native agent is a core part of the whole platform. It is attached to the monitored application and additionally to providing the data to the instrumentation server, it is used to obtain various low-level information from the application.

The Distrace tool, therefore, consists of three main components:

- Native Agent
    - Is used to obtain bytecode for the instrumentation.
    - Is used to actually apply the instrumented bytecode.
- Instrumentation Server
    - Instruments the classes obtained from the native agent.

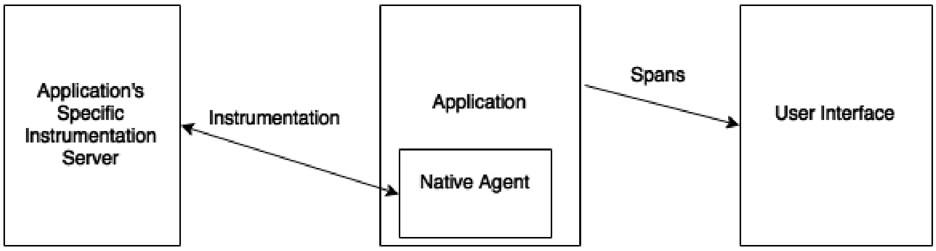

Figure 4.1: Basic relationship between the major components.

  - Is also the base library for custom application instrumentation server.
  - Can contain implementations of customized span exporters.

- User Interface

The Figure 4.1 denotes the basic relationships between the major parts. Instrumentation server communicates with the native again, mainly in order to instrument classes. The application communicates with the user interface by sending the spans to it. The spans can be sent either via data collection agent or via one of the default span exporters explained later in this chapter. Each part is described in more detail later in this chapter.

The Figure 4.2 shows how trace trees and spans are related on a simple two nodes example. Each trace is separated from each other and represents tracing of a single computation, which consists of several spans. Spans denote more local computation and can also contain additional application-specific information. In order to connect the information from multiple nodes, the trace information needs to be attached to the node communication. That is also a reason why in this case the methods `send()`, `receive()` and `process()` need to be instrumented. These methods also open and close spans at the correct places in the code.

4.2 Example Use Case

In order to demonstrate an use case for which this architecture may a good fit, a small example on a simple distributed application is shown in this section. The example consists of three modules. The client module used for submitting tasks, the execution module and the module used for exporting the data. These modules can be represented as separated threads in a single application or as different nodes of the distributed application. In this example, the user always passes a task to the client module. This module performs some pre-processing and sends the task to the execution module. This module performs the computation and once it's done, sends the data to the exporter module, which exports the data on disk and informs the client of the task completion. The architecture of the example can be seen in Figure 4.3.

The goal of this example is to record and visualize how long the transfers between different modules last and how long the processing on each module takes. It is also assumed that the platform does not collect this information already, otherwise the cluster monitoring tool would not be required. For simplicity, let's also assume that each module performs the functionality in a single method. The following code sections give the schematic code of each method.

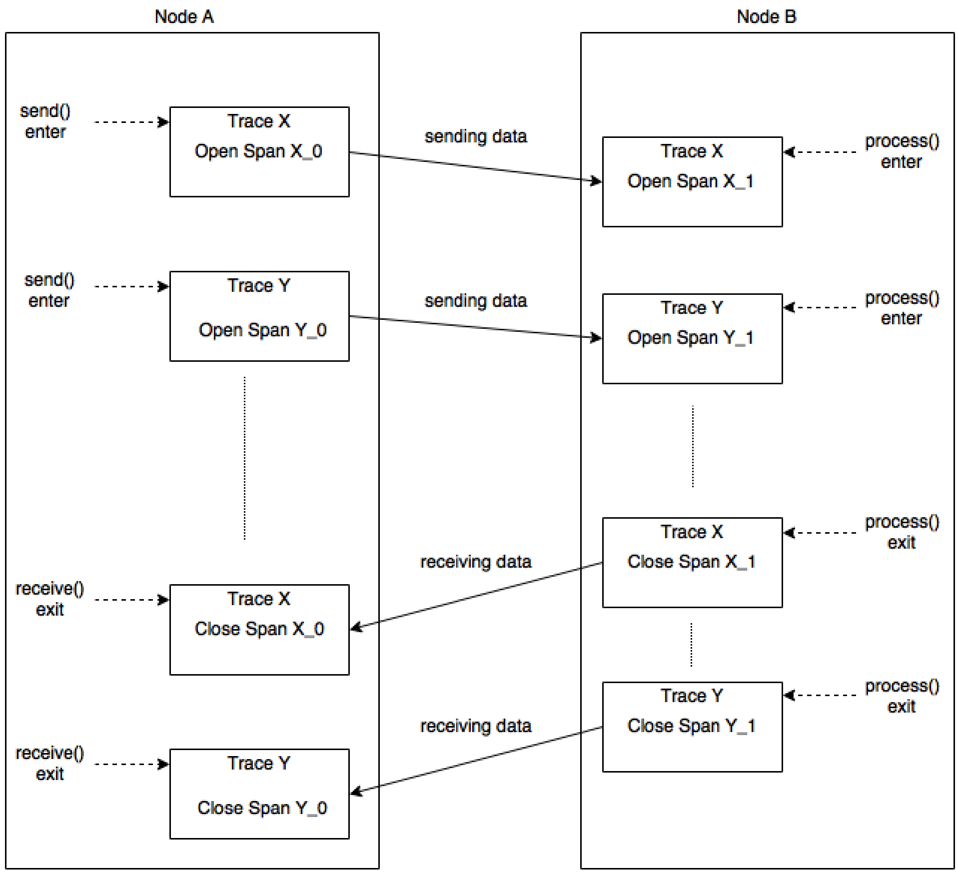

Figure 4.2: Trace and span demonstration.

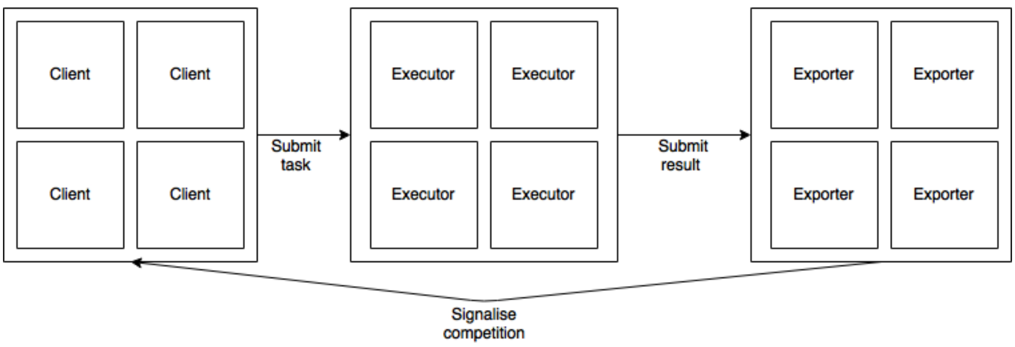

Figure 4.3: Example architecture.

- **The client:**

```
public acceptTask(Task task){
  preprocessTask(task)
  ...
  sendTaskForComputation(task)
  ...
  waitForExporterToFinish()
  ...
}
```

- **The executor:**

```
public execute(Task task){
  TaskResult result = executeTask(task)
  ...
  sendResultForVisuzalization(result)
}
```

- **The exporter:**

```
public export(TaskResult result){
  saveToDatabase(result)
  ...
  notifyClient ()
}
```

In order to collect this type of information and be able to reason about the relationship between the modules, these methods need to be instrumented. The instrumented code should look as in the schematic code below. Generally, the logic which keeps the track of the current trace and span needs to be injected into the code. To achieve this, developers are supposed to extend the core instrumentation server, which acts as the base library and provides them with several helper methods used to specify the instrumentation points for their applications.

- **The instrumented client:**

```
public acceptTask(Task task){
  TraceContext tc = TraceContext.create()
  tc.attachOnObject(task)
  Span s = tc.openSpan("Main Client Span")
  s.addAnnotation("tskReceived", timestamp)
  ...
  preprocessTask(task)
  s.addAnnotation("tskPreprocessed", timestamp)
  ...
  sendTaskForComputation(task)
  ...
  Result res = waitForExporterToFinish() // blocking method
  ...
  tc.closeCurrentSpan()
}
```

The `create` method creates a new trace and the `attachOnObject` method attaches the trace context on the `task` object, which is passed around the network. The method `openSpan` opens a new span encapsulating the client computation. The `addAnnotation` method is used to add application specific information to the current span. The `closeCurrentSpan` method is used to close the current span and export the content using the provided span exporter. In the default case, the data are sent to the user interface directly.

- **The instrumented executor:**

```
public execute(Task task){
  TraceContext tc = TraceContext.getFromObject(task)
  tc.openSpan("Executor_span")
  TaskResult result = executeTask(task)
  tc.attachOnObject(result)
  ...
  sendResultForExport(result) // non-blocking method
  tc.closeCurrentSpan()
}
```

  In this case, we don't create a new trace context but obtain the existing one from the `task` object on the input. A new nested span is opened within the scope of the span, created in the previous module. The current span marks the span from the previous module as its parent.

- **The instrumented exporter:**

```
public export(TaskResult result){
  TraceContext tc = TraceContext.getFromObject(result)
  tc.openSpan("Exporter_span")
  saveToDatabase(result)
  ...
  tc.closeCurrentSpan()
  notifyClient ()
}
```

  In this case, the meaning of the methods is the same as above.

The developer should extend the base instrumentation server to instrument the classes in order to have a similar format as above. The extended instrumentation server is described in Figure 4.4.

The extended instrumentation server is run on each node or on the network and is used to perform the instrumentation requested from the application. The native agent has to be attached to all nodes of the distributed application prior its start and the path to the extended instrumentation server JAR needs to be set as the mandatory argument. The default span exporter is used in this case and the collected spans are sent right to the Zipkin user interface. The default IP address and port of the user interface are used when they are not explicitly configured as the native agent argument.

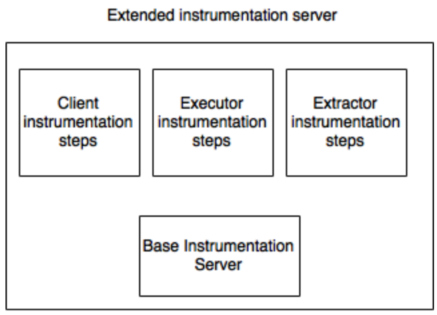

Figure 4.4: Structure of the extended instrumentation server JAR artifact.

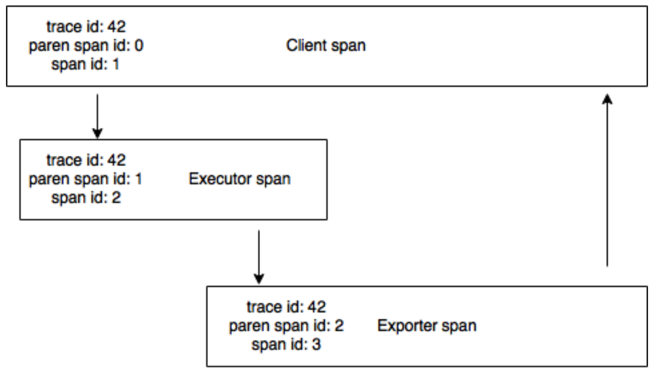

Figure 4.5: Example trace in case of the example application.

A single collected trace from this application should look as shown in Figure 4.5.

Therefore, it can be seen that the only part the developer needs to work on is the extension of the instrumentation server to specify the custom instrumentation points, otherwise the rest of technical work is done automatically. The end user is only responsible for starting the application with the agent attached.

4.3 Spans and Trace Trees

As mentioned briefly in the previous section, spans are used to gather the information about the distributed calls or so called, distributed stack traces. Points in the code where spans are created and closed are defined as part of the instrumentation server but since it's the most important concept in the thesis, we explain them in the separated section.

Spans are the main concept behind capturing the distributed traces. They are entities injected to the code of the instrumented application to keep track of the communication and state between the nodes of the distributed application. Usually, the initiator creates so called parent span and new calls started within the span create new nested spans. Collected spans can be processed using different span exporters and can be sent to the user interface using various data collectors.

Span has several mandatory and optional attributes. The mandatory attributes are trace id, span id and parent span id. Trace id identifies one complete distributed call among all interacting nodes of the cluster. This attribute is attached automatically when a new root span is created. A root span is the first span created inside the trace and does not have parent id attribute set up. Therefore, the user interface back-end can distinguish between regular spans and root spans and can identify the start of the whole trace. Parent id of a span is id of span in which scope the child span was created. The span and its parent span can be located on the same node or on different nodes as well. The first variant can be useful in cases where the developer requires to trace multiple threads as separated spans within a single application node.

Span has also several additional optional fields, which are later used in the user interface. The fields are:

- **Timestamp** - when the span started.
- **Duration** - how long the span lasted.
- **Annotations** - annotations used to carry additional timing information about spans. For example, time when the span has been received on the receiver side or the time the span has been processed at the receiver side can be set using these annotations.
- **Binary annotations** - annotations used to carry around application specific details. They can also be used to transfer information between communication nodes inside of a single span. For example, a sender can store number of bytes sent during the request and a receiver can use this information to calculate overall number of bytes received from this particular node.

Each span has also an internal field called **flags**. The developer may store tags important to the instrumentation and these flags are transferred as part of the spans. Flags are not sent to the user interface and are only used for instrumentation purposes. For example, they can be used in case of multiple spans exist at a single moment. In this case, these spans can be annotated with special flags and the decision can be made based on this to process these spans.

Each annotation, both binary and regular, has also an endpoint information attached. This element consist of:

- **IP** - IP of the node on which this event was recorded.
- **port** - port of the service which recorded the span.
- **service name** - a special name, which is used in the user interface to group and filter different traces by names.

4.3.1 Span IDs

It is also important to mention that span id is created randomly. This is done in order to allow parallel spans to coexist in the same control flow without overlapping as can be seen in Figure 4.6.

If the ids were not random at different nodes of the distributed system, the parallel spans would be creating child spans with ids in the same linear sequence and therefore these spans would be overlapping, as can be seen in Figure 4.7

The following sections contain information about how spans are exported for processing outside of the Distrace tool and also how spans are created using `TraceContext` and `TraceContextManager`.

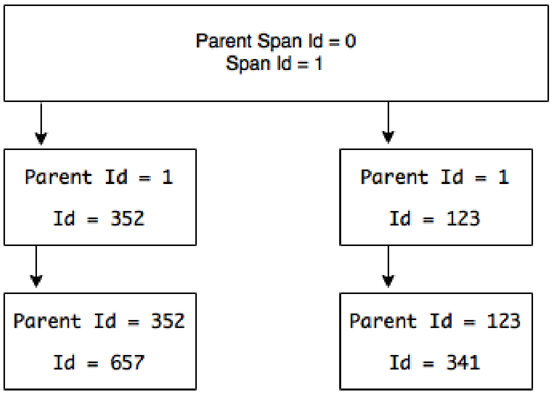

Figure 4.6: Generating span ids randomly ensures that they don't overlap when they are created in parallel.

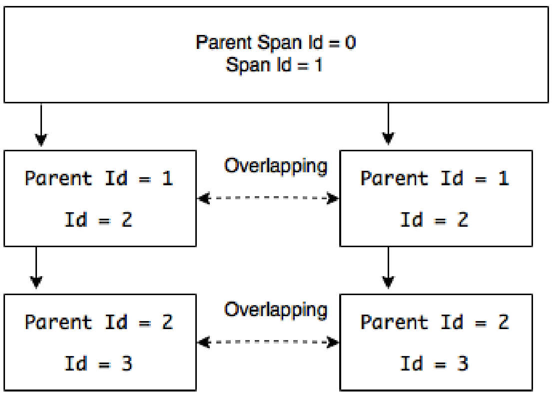

Figure 4.7: Generating span with the same linear sequence leads to span overlapping.

4.3.2 Span Exporters & Data Collectors

Spans can be exported from the application using so called span exporters. The type of the exporter can be configured via one of the native agent configuration properties. It is important to mention that the exporter is configured globally and each span in the application is using the same exporter[1]. The monitored application needs to have the chosen span exporter available at run-time, since the export is performed at the scope of the monitored application. Therefore, exporters are sent to the native agent during the initialization phase and the agent puts the exporters on the application classpath.

The Distrace tool includes two default implementation of span exporters, but also allows the developer to create new span exporters. Custom span exporters may be useful in cases when the developer wants to export the span data in a format used by different user interface or to use custom data collector. Data collector is a service, which collects data from the specified location and stores them in a central data storage available to all nodes in the distributed application. The Distrace tool does not implement data collection service as many services exist for this purpose.

Data collected within spans are internally represented in JSON format understandable to the Zipkin user interface. This is also the reason why Distrace contains support for working with JSON data and it is explained in more detail later in Section 5.3.4. This output format can be customized by the custom span exporter.

The implementation details are available at Section 5.1.1.

4.3.3 Trace Context

Trace context is used for storing the information about the current span and also for creating new spans and closing current spans. Trace context can be attached to a specific object and thread. This is done in order to allow multiple threads to have different computation state and therefore the platform is able to capture multiple distributed traces at the same time on the same node. Trace context manager is used for attaching trace context to threads and vice-versa. It provides several operations allowing to the developer to attach trace context to a specific thread and also to get trace context from a specific thread.

Each trace is represented by trace context and is uniquely identified by Universally unique identifier (UUID) of type one[2]. This UUID type combines 48-bit MAC address of the current device with the actual timestamp. This way it is ensured that two traces created at the same time on different nodes can not have the same identifier. The identifiers are generated in the native agent using C++ library called Sole[3] and are made available to the Java code via published native operation `getTypeOneUUID`.

The trace contact has `openNestedSpan` and `closeCurrentSpan` operations. The first operation is used to create a new nested span and set the newly created span as the current one. Nested span is opened within the scope of its parent

[1]Exporter is a static attributed of the `Span` entity.

[2]More UUID versions exist and are created based on different information

[3]The Sole library is available at `https://github.com/r-lyeh/sole` and can be used to create identifiers in C++ language.

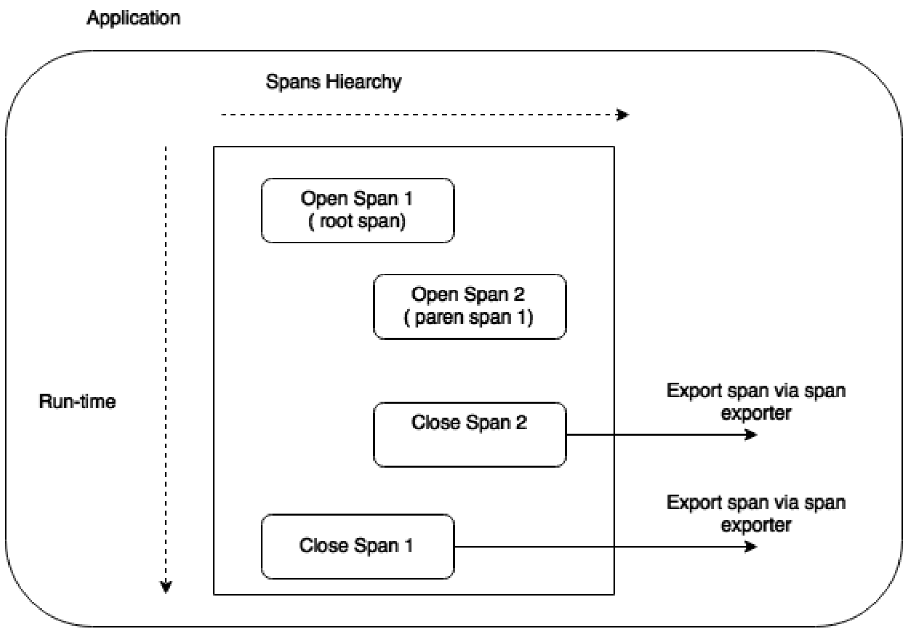

Figure 4.8: Creating and closing spans.

span and remembers the parent span. A root span is created in case no current span exists. The second operation is used to close the current span.

The second operation closes the the current span. This consist of two actions. The current span is exported using the configured span exporter and the parent span becomes the current span. The Figure 4.8 shows how spans are created, closed and exported.

4.3.4 Transferring Span Information

In order to capture the shared state between the nodes in the distributed application or between the threads on the same application node, the span details and trace context have to be transferred between the threads or the nodes. Distrace prepares several operations for attaching trace context to either a current thread or to an object acting as the carrier of the trace information.

Usually when transferring the trace context between the threads on the same node, the copies of the trace context should be used. This is not an issue when transferring the trace context between the different nodes of the distributed application.

The developer responsible for extending the instrumentation server can use operations like:

- `getTraceFromThread`
- `getTraceFromObject`
- `attachTraceToThread`
- `attachTraceToObject`

More information about the trace context API is available at 5.1.2.

There are also different variants of how spans can be closed. A span can be closed by the same node or thread who created it. In this case, attaching the trace context information to the current thread is usually sufficient. However, it is also possible that the span can be closed by different thread or node that originally created this span. In this case, the copy of the trace context should usually be created and attached to the object, which is used as the carrier of the trace context.

4.4 Native Agent

The native agent is used for accessing the internal state of the monitored application and also to instrument classes so they can carry the span and trace identifiers between the application nodes. The main task of the agent is to check whether a class is required to be instrumented and if yes, send the class for the instrumentation to the server and wait for the instrumented code.

The native agent consist of several modules. The most important ones are:

- **Bytecode parsing module**.
  The classes in this module are used to parse the JVM bytecode in order to discover the classes dependencies for further instrumentation. Bytecode parsing is a technical task described in Section 5.2.4.

- **InstrumentorAPI**.
  The `InstrumentorAPI` class provides several methods, which are used to communicate with the instrumentation server JVM. All the queries to the server go through the instance of this class.

- **AgentCallbacks**.
  All callbacks used in the native agent are defined in this namespace.

- **AgentArgs**.
  The `AgentArgs` class contains all the logic required for argument parsing.

- **NativeMethodsHelper**.
  The `NativeMethodsHelper` class is used for registering native methods defined in C++. These methods can be used from the Java code without worrying about the low-level implementation.

- **Utility module**.
  This module contains several utility namespaces. The most important are `AgentUtils` and `JavaUtils` namespaces. The first contains methods for managing the JVMTI connection and for registering the JVMTI callbacks and events. The second is used to simplify work with Java objects in the native code via JNI.

4.4.1 Agent Initialization

The agent is initialized through the same phases as described in Section 2.7.1. The following JVMTI events are especially important to the thesis: `VM Init`, `VM Start`, `VM Death`, `Class File load Hook`, `Class Prepare` and `Class Load`.

Callbacks are registered for all the mentioned events so the native agent can react to them accordingly in the code.

As part of the initialization process, the agent is responsible for either connecting to or starting a new instrumentation server. In case the native agent was started in the shared mode of the instrumentation server, the agent tries to connect to already existing server and the server is shared between all nodes of the distributed application. In the local instrumentation mode, the server is started as a separated process automatically and the connection is established with the server using the inter-process communication. In this case, each application node has dedicated instrumentation server.

The callback registered for the `VM Init` event is responsible for loading all additional classes from the instrumentation server as part of the initialization as well. The additional classes are for example `Span`, `TraceContext` or custom implementations of `SpanExporter` abstract class. These classes need to be available at run-time at the monitored application. They are required since the instrumented code is using trace context and spans for preserving the information about the current trace and the exporters for exporting the spans from the application. Therefore, these classes have to be available at the monitored application. The native agent is designed in a way that developers are not supposed to change the code of if. All the extension are supposed to be done as part of the extended instrumentation server. Therefore, the server is asked at the initialization phase for the list of all additional classes and they are sent to the native agent. The agent puts all the received classes on the application's classpath so they are available to the instrumented code.

4.4.2 Instrumentation

Code for handling the instrumentation is part of the callback for the `Class File load Hook` event. The callback has the bytecode for the class being loaded as its input parameter and allows the developer to pass a new instrumented bytecode as the output parameter. The process of instrumentation is described at this section, however some technical details are described in Chapter 5.

The process consists of several stages:

1. Enter `Class File load Hook` callback.

2. Enter the critical section. It can happen that the class file load hook is triggered multiple times and to prevent confusing the instrumentation server, the lock has to be acquired before the instrumentation of a class starts.

3. Firstly, we check whether the virtual machine with the application is started. If the JVM is initialized, the instrumentation continues, otherwise the instrumentation for currently a class currently being loaded is skipped. In this case, the instrumentation server is not contacted since the classes being loaded are at this point system classed and it is not desired to instrument Java system classes.

4. Attach JNI environment to the current thread. Since the JVMTI and JNI does not have automatic thread management, it is up to the developer to take care of correct threading management.

5. Discover the class loader which is loading the class.

6. Parse name of the class currently being loaded and the nape of the class loader loading the current class. Even though the callback provides input parameter, which should contain the name of the class, in some circumstances it can be set to `NULL` even though the class name is available in the bytecode. Instead of relying on this parameter, the bytecode is parsed and the class name is found manually.

7. Decide whether the instrumentation should continue. This check is based on the used class loader and name of the class being loaded. Classes loaded by the `Bootstrap` class loader and in the case of Oracle JVM, classes loaded by `sun.reflect.DelegatingClassloader` are not supposed to be instrumented. The `Bootstrap` class loader is used to load system classes and the second mentioned class loader is used to load synthetic classes. and in both cases, it's not desired to instrument classes loaded by these class loaders. There are also some ignored classes for which the instrumentation is not desired. For example, classes loaded during initialization phase from the instrumentation server and the auxiliary classes generated by the Byte Buddy framework should not be instrumented. Auxiliary classes are small helper classes Byte Buddy is using for example to access the super class of the currently being instrumented class. Therefore, the instrumentation continues only if the class is not ignored and not loaded by any of the ignored class loaders.

8. The instrumentation server is asked whether the class is marked for instrumentation. The agent does not know which classes are to be instrumented and therefore, it needs to query the server. The classes to be instrumented are marked by a developer when extending the instrumentation server library using simple Byte Buddy API. The process ends if the class should not be instrumented.

9. Check whether the instrumentation server already contains the class being loaded. If the server contains the class, we go to the next step and wait for the instrumented class. If the server does not contain the class, the native agent sends the class data to the instrumentation server, parse the class file for all the dependent classes and send all dependent classes for the instrumentation. This step is repeated through all dependencies recurrently until the loaded class does not have any other dependencies or until all dependencies are already available on the server. All dependencies for the currently instrumented class have to be available on the server in order to perform the instrumentation.

10. At this stage, the class is already on the instrumentation server and all dependencies for this class as well. The native agent waits for the instrumented bytecode to be sent from the server and sets the modified bytecode as the output argument of the `Class File load Hook` callback.

11. Exit the critical section.

12. Exit `Class File load Hook` callback.

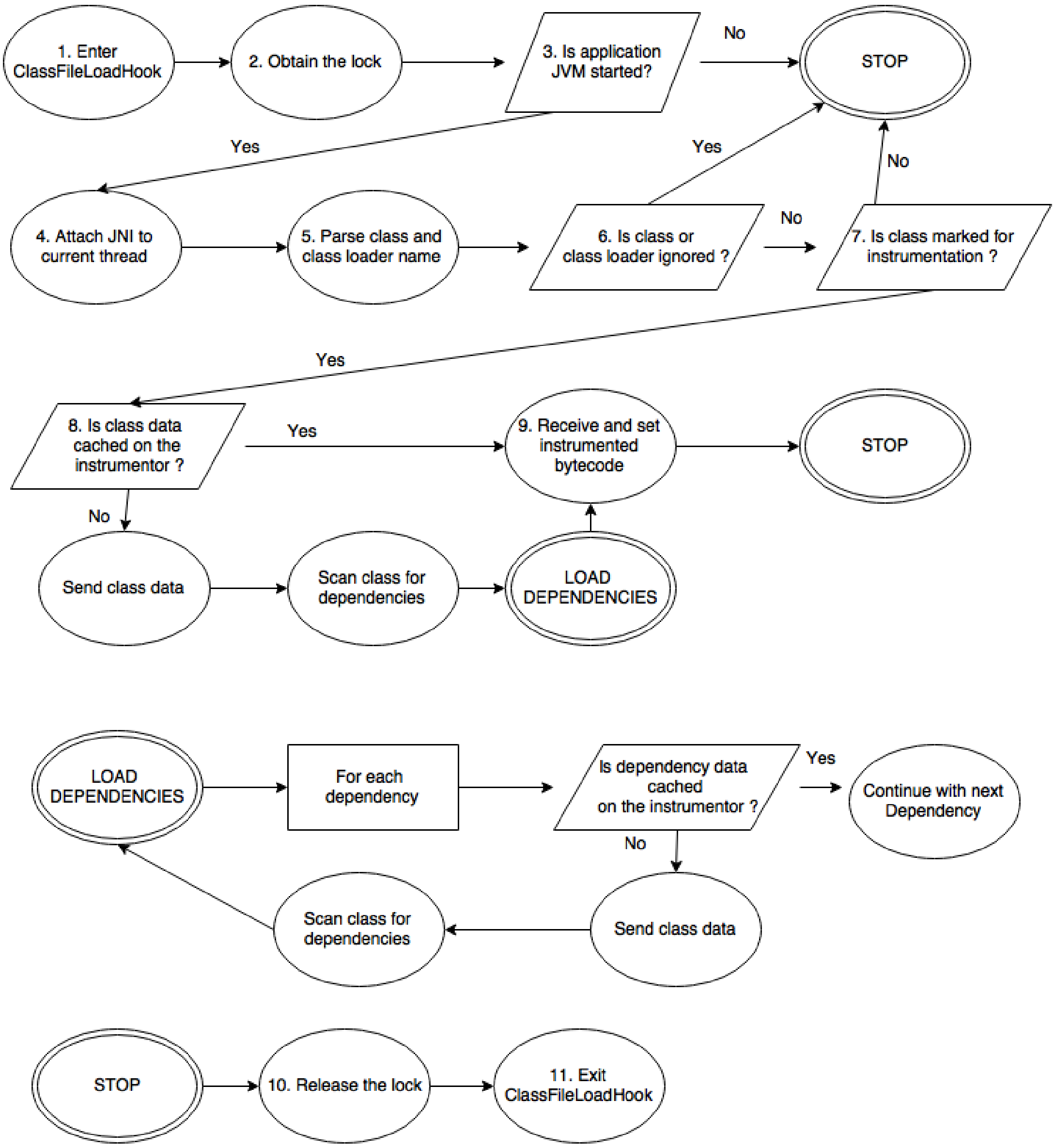

Figure 4.9: State machine diagram representing the instrumentation process.

The Figure 4.9 is the state machine diagram representing the instrumentation steps, where the numbers in selected steps correspond to the items in the list above.

Several technical difficulties had to be dealt with during the development. For example, cyclic dependencies between classes need to be properly handled during the instrumentation. We need to also ensure that the dependencies for instrumented classed are also instrumented in the correct order. The final solution to these problems is described in Section 5.2.1

The behavior of the agent may be configured using the arguments passed to the agent. Please see the Attachment 4 for the full list of native agent arguments.

4.5 Instrumentation Server

The instrumentation server is responsible for instrumenting the bytecode received from the native agent in the separate JVM and it also acts as the base library for instrumenting specific applications. The developer extending the instrumentation server can use prepared operations to define custom instrumentation points without touching the internals of the native agent.

This section covers several design aspects of the instrumentation server, leaving the implementation details on the following sections. The core instrumentation on the server is handled by the Byte Buddy code manipulation framework. The native agent asks the server if the class currently being loaded is required to be instrumented. If yes, the server receives the bytecode, performs the instrumentation and sends the data back to the agent. The server does not contain any application state, in particular, it does not take track about the distributed traces. The information about traces is contained in the instrumented classes within the monitored application.

The platform was designed to be configurable and deployment of the instrumentation server is supported via two approaches. The instrumentation server can be located either on the network to all nodes of the distributed application and can be shared by all nodes. This has the advantage of caching the instrumented classes. When any class is instrumented for the first time based on request from any node, it is saved and the instrumentation is not performed for other nodes. Instead, the class can be sent immediately. The disadvantage of this solution is higher latency between the agent and the instrumentation server since they are usually not on the same physical node. In this case, the instrumentation server has to be manually started in advance. Architecture of this scenario is depicted in Figure 4.10.

The other deployment method is that the instrumentation server runs on each node of the distributed application. This has the advantage of faster communication since in this case, inter-process communication is used to communicate between monitored JVM and the instrumentation server. The disadvantage of this solution is that all classes have to be instrumented on each node since there is no communication between the instrumentation servers. In this solution, the server is started automatically during the native agent initialization. Architecture of this scenario is depicted in Figure 4.11.

Except the cached classes, the server does not contain any application state and it just reacts to the agent requests. It can accept four type of requests:

- Request for code instrumentation.
- Request to store bytecode for a class on the server.
- Request to send all helper classes needed by the agent such as the `Span` class or `TraceContext` class.
- Request to check whether the server contains specific class or not.

The server interacts in more ways with the agent, however all communication is initiated by one of these four request types.

The instrumentation server needs to deal with several technical problems. The main issue is that the classes, which are about to be instrumented, require

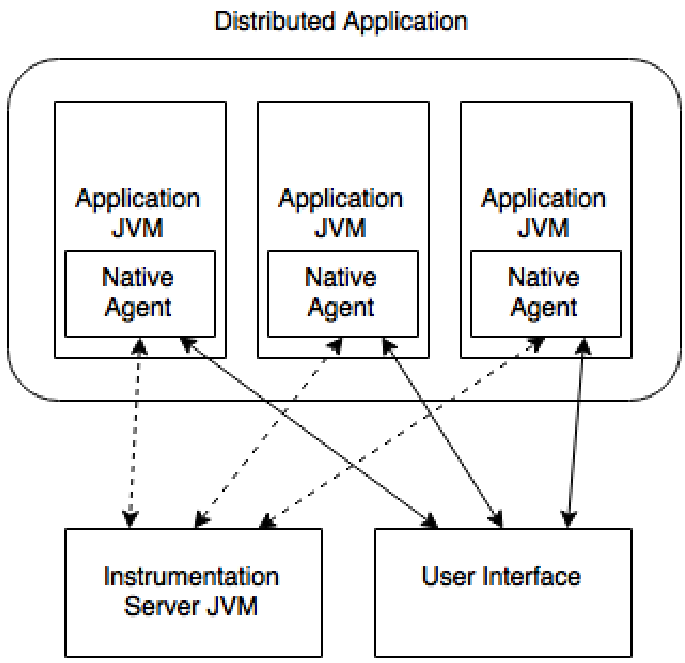

Figure 4.10: Architecture with the shared instrumentation server. The dotted lines represent the communication between the server and the agent, whilst the regular lines represent data transfer from the agent to the user interface.

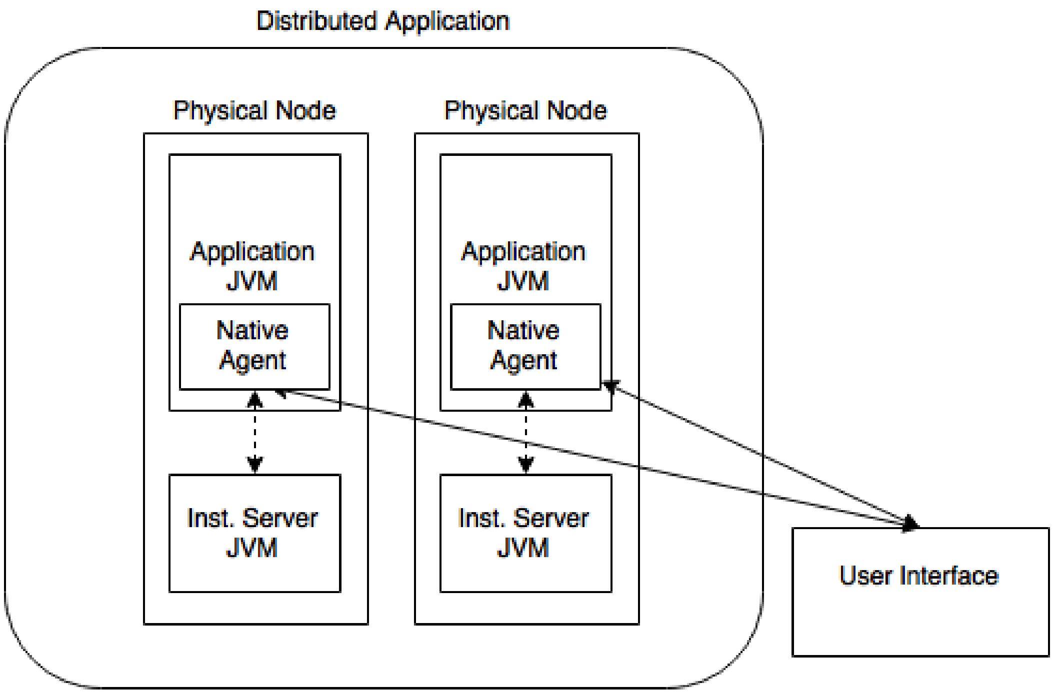

Figure 4.11: Architecture with separated instrumentation server. The dotted lines represent the communication between the server and the agent, whilst the regular lines represent data transfer from the agent to the user interface.

all other dependent classes to be available. The other issue is instrumenting the classes with circular dependencies. The server also performs several optimizations to provide faster response to the agent such as caching the instrumented classes and minimizing the communication when possible. The technical aspects of these issues and the optimizations mentioned above are described in Chapter 5.

4.5.1 Instrumentation

The instrumentation of the class is triggered by the agent and it's done in two stages. The first stage informs the agent whether the class is already available on the instrumentation server or not. The second stage is the instrumentation step itself. The first stage is initiated by the agent and the server performs the check for class availability in three phases:

1. Check if the instrumented bytecode for this class is available.
2. If not, check if the original bytecode for this class is available.
3. If not, check if the class can be loaded using the server's context class loader. This handles the cases where the user builds the instrumentation server together with the application classes or adds the classes on the server classpath for optimization reasons.

The server informs the agent if it does not have the bytecode for the class available. In that case, the agent sends the class to the server and the server registers the received bytecode under the class name. Therefore, the agent does not have to send the class next time since it's already cached on the instrumentation server. The second stage follows the first stage immediately. If the server already contains the instrumented class in the cache, the transformed class is sent right away without instrumenting the class again. If the cache is empty, the class is instrumented and put into the cache.

More information about instrumentation process on the server is described in more details in Section 5.3.1.

4.5.2 Custom Service Loader

Service loaders are used for loading the extensions to the Distrace tool and creating the extended instrumentation server. The service loader is used for two object types:

- **Custom span exporters** - Each span exporter inherits from the abstract `SpanExporter` class.
- **Custom Interceptors** - Each interceptor has to implement the interface `Interceptor`.

The user can create custom span exporters and interceptors by either inheriting the desired class or implementing the required interface. In order to allow the platform to locate the custom implementations, the name of the class has to be written inside the text file in the META-INF directory in the extended instrumentation server JAR file. The text file has to have the same name as the abstract

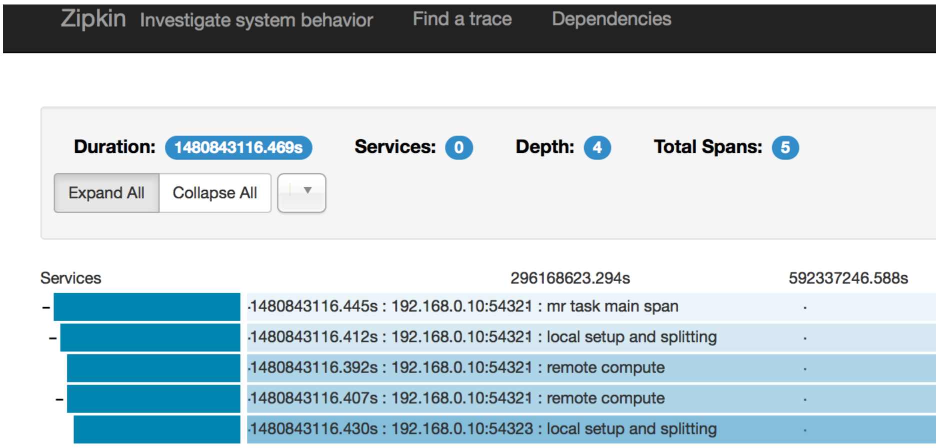

Figure 4.12: The example of trace output in the Zipkin user interface.

class or the interface the particular implementation is for. For example, when the user creates a new interceptor called `x.y.InterceptorA`, the file `Interceptor` in the META-INF folder has to contain line `x.y.InterceptorA`.

Java provides service loader for this purpose. However the standard Java implementation looks up the classes defined as above and automatically creates new instances using the well-known constructors. For the thesis purposes this was unwanted as it is only required to obtain the `Class` object representing the available implementation. Therefore a custom service loader was created. This service loader works in a very similar way as the standard Java implementation, but instead of returning the instances of loaded services, it just returns classes representing the available services.

4.6 User Interface

The user interface receives spans and presents them in a hierarchical way so the relationships between different nodes can be seen easily. The important feature of the user interface is that the data for a single span can be sent incrementally. This means that several JSONs representing the same span can be sent with different annotations and the user interface merges these spans into the single one and presents all annotations under the given span. This allows the Distrace tool to send part of data from the sender side and part of data from the receiver side directly to the user interface, instead of exchanging the data between the communicating nodes in order to send them at the end as one single complete span.

The Distrace tool is using Zipkin as default user interface. The default format for exporting spans is designed in order to be understandable by this user interface. The user is however still able to change the data format to support custom user interface via custom span exporters. This section gives an overview of Zipkin user interface and describes the Zipkin data model. The example output of a single trace in the Zipkin user interface can be seen in Figure 4.12.

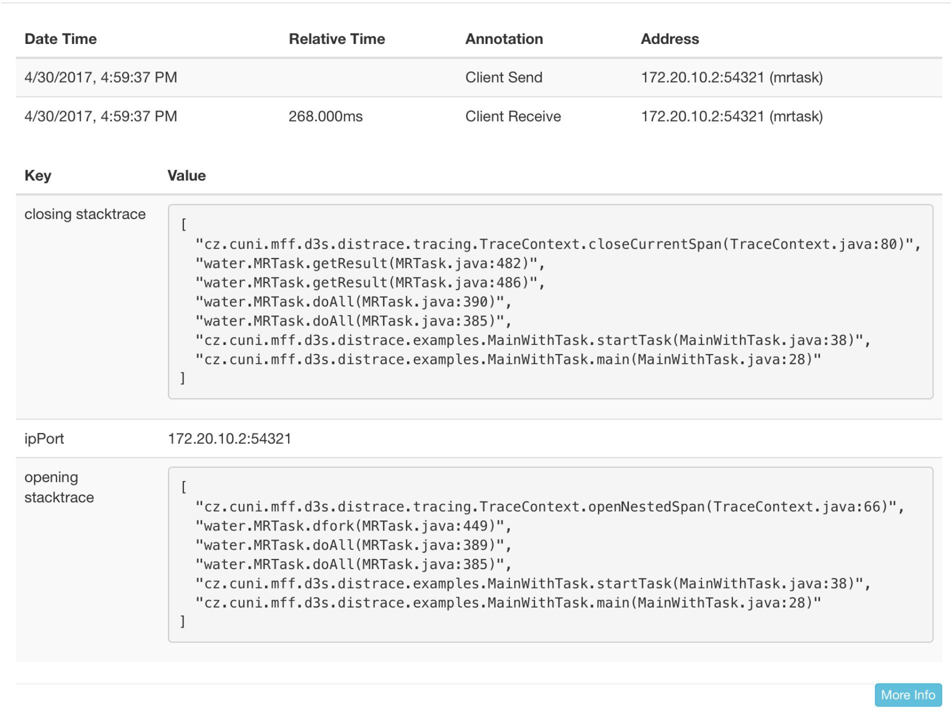

Figure 4.13: The example of the detail span information.

Each span in the user interface is clickable and all the additional information can be seen at that level. The Distrace tool also automatically collects the stack trace at the time of span creation and closing. The example of such information screen can be seen in Figure 4.13.

4.6.1 Zipkin Data Model

Zipkin requires data to be sent in JSON format. Requests to the user interface are sent as JSON arrays, where the array elements are the spans. Zipkin understands the following attributes of span object:

- **traceId** - unique id representing the complete trace. It can be either 128 or 64 bit long.
- **name** - human readable span name.
- **id** - id of this span. At the current implementation, Zipkin user interface supports span ids only to be 64-bit long.
- **parentId** - parent id of the current span.
- **timestamp** - the time of the span creation.
- **duration** - the duration of the span. It is the duration between the span creation and closing.
- **annotations** - array containing standard Zipkin annotations. These annotations can be handled by the user interface in a specific way, since the user interface understands the meaning of content of these annotations. Following annotations are available:

- **cr** : timestamp of the client receiving the span.
- **cs** : timestamp of the client sending the span.
- **sr** : timestamp of the server receiving the span.
- **ss** : timestamp of the server sending the span.
- **ca** : client address.
- **sa** : server address.

- **binaryAnnotations** - array of custom annotations. For example, collected stack traces are sent as a binary annotation.

Except the **annotations** and **binaryAnnotations** attributes, the attributes are of simple string or number type. Annotations are objects consisting of three additional attributes - annotation value, annotation name and the endpoint. Endpoint is another object specifying the address and port of the node in the distributed application, where the span or particular annotation was recorded. Endpoints can also specify service name, which may be used to search for particular spans. Full example of data sent to Zipkin user interface can be:


```
[
 {
   ”traceId”: ”123456789abcdef”,
   ”name”: ”query”,
   ”id”: ”abcd1”,
   ”timestamp”: 1458702548467000,
   ”duration”: 100743,
   ”annotations”: [
     {
       ”timestamp”: 1458702548467000,
       ”value”: ”sr”
       ”endpoint”: {
         ”serviceName”: ”example”,
         ”ipv4”: ”192.168.1.2”,
         ”port”: 9411
       },
     }
   ],
   ”binaryAnnotations”: [
     {
       ”key”: ”bytes_sent”,
       ”value”: ”1783”
       ”endpoint”: {
         ”serviceName”: ”example”,
         ”ipv4”: ”192.168.1.2”,
         ”port”: 9411
       },
     }
   ]
 }
]
```

5. Implementation Details

This chapter explains several technical implementation details. The first section describes span exporters and API of Trace Context. The following section focuses on implementation details of the native agent and the last section focuses on the implementation details of the instrumentation server.

5.1 Span and Trace Trees

This section explains two implementation specific areas related to spans and trace trees. It describes span exporters in more detail and gives an overview of the Trace Context API.

5.1.1 Span Exporters

An implementation of a span exporter has to extend from the abstract ancestor defining common methods for each span exporter. Also, in order to be able to use the exporter automatically in the code, it has to have a constructor with single `String` argument accepting exporter arguments. The arguments format is defined in the case of default span exporters, however, the developer may use any format in case of custom span exporters. `SpanExporter` abstract class is the common ancestor for each exporter and has two abstract methods:

- `export`. This method is used for exporting the span. Custom span exporters implementation may save the data on local disk or send over the network. The destination is not limited by the code. Internally, the `export` method is called asynchronously in separated threads to allow asynchronous span exporting, which can lead to a performance benefit.

- `parseAndSetArgs`. The native agent has configuration property, which the user can use to configure arguments for the span exporter. Each span exporter is responsible for parsing its own arguments.

As mentioned in Section 4.3.2, the Distrace tool provides two default implementations of span exporters:

- `DirectZipkinExporter` - This span exporter sends the collected span asynchronously to the user interface right away without storing the data on disk to be collected by any data collection agent. In this case, the functionality of the span exporter and the data collector are handled by this single exporter. This span exporter should be used only for demonstration purposes since it could overload the user interface or network when processing a high number of spans, because the Zipkin user interface is not prepared to handle and store a large amount of data in the memory. However, this is a default span exporter at this moment.

  This exporter accepts a single argument, which is the IP address and port of the Zipkin user interface. Figure 5.1 shows, how the Zipkin span exporter is used.

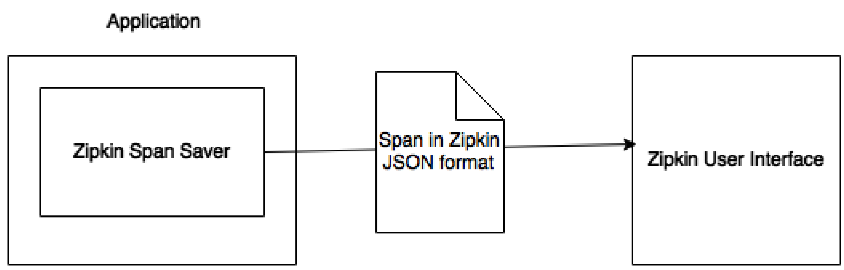

Figure 5.1: Using the Zipkin span exporter to export spans directly to Zipkin user interface without the data collection agent.

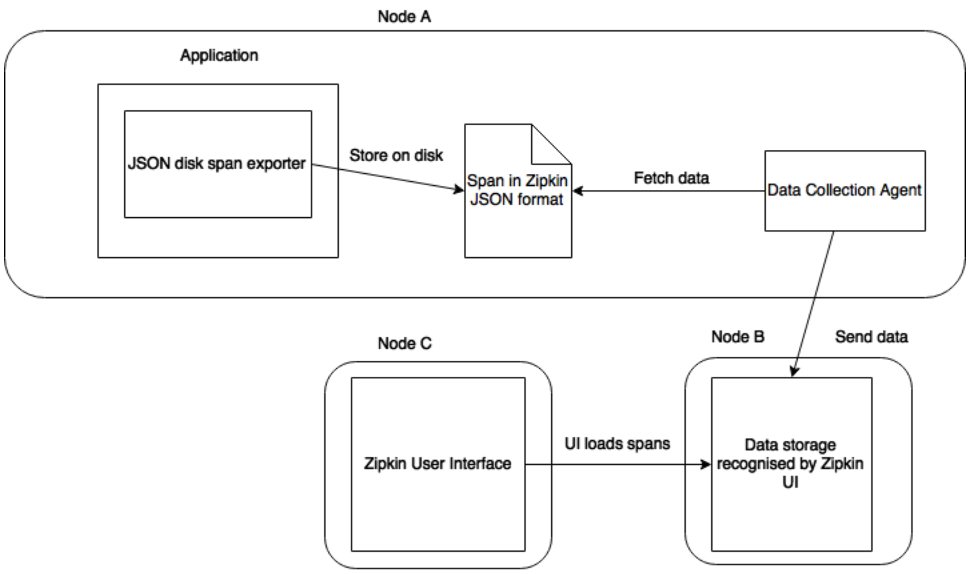

Figure 5.2: Using the JSON disk exporter together with the data collection agent together with the Zipkin user interface.

- `JSONDiskExporter` - The second available span exporter saves the collected spans asynchronously on disk in the format known to the Zipkin user interface. The exported spans may be collected in the future by a custom data collection agent and for example, sent to the user interface or database. Together with some well-known data collection agent, this is a preferred way of transferring spans from the application to the Zipkin user interface in the production. This exporter accepts a single argument, which is a destination directory for exported spans. Figure 5.2 shows how JSON disk span exporter is used.

Additionally, Figure 5.3 shows how a custom span exporter may be used.

In order to give the developer the flexibility to add new exporters without changing the internals, the custom service loader is used and the span exporters have to be registered in the META-INF directory of the extended instrumentation server JAR file. This ensures that the service loader can find all implementations of the `SpanExporter` abstract class. The reason why the classes need to be discoverable by the service loader is explained in Section 4.4.1.

To make the developer life easier, the **AutoService** library[1] may be used when

[1] The AutoService library is available at `https://github.com/google`.

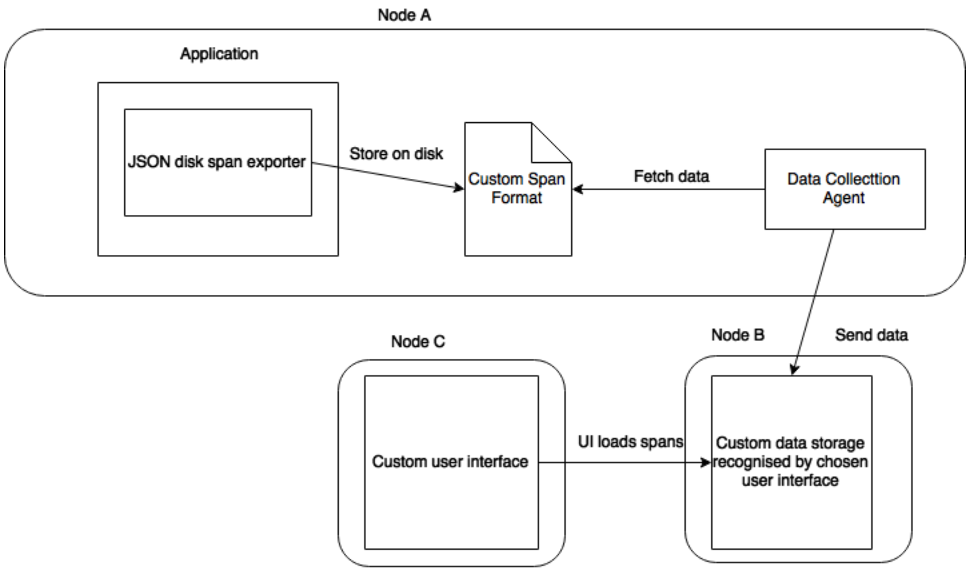

Figure 5.3: Using the custom span exporter together with the data collection agent and custom user interface.

extending the core server library. Instead of manually registering implementations of custom span exporters into META-INF directory, they can be annotated in the code using the `AutoService` annotation. This annotation takes a single argument specifying the abstract parent, in this case, `SpanExporter`. The library takes care of registering the classes automatically in the desired folder in the correct format so the human error is minimized.

5.1.2 Trace Context API

The following methods can be used for obtaining and attaching the trace context:

- `static create()` - creates a new trace context.
- `static getFromObject(holder)` - gets the existing trace context from the holder object.
- `static getFromThread(thread)` - gets the existing trace context from the specified thread.
- `static getFromCurrentThread()` - gets the existing trace context from the current thread.
- `attachOnObject(holder)` - attaches the trace context to the holder object.
- `attachOnThread(thread)` - attaches the trace context to the specified thread.
- `attachOnCurrentThread()` - attaches the trace context to the current thread.
- `deepCopy` - creates a deep copy of the trace context. It is usually used in cases where child spans are processed in parallel by multiple threads. In

this case, the copy of trace context with the same id is shared among all these threads, but they operate on very own objects. This is done in order to allow monitoring of parallel spans within a single trace without having to face race conditions on a single trace context object.

The methods above can also be chained and, for example, a trace context can be obtained from the holder object, deep copy created and the newly created copy attached to a new holder object.

5.2 Native Agent

This section covers specific parts of the native agent in more detail. It starts with the explanation of considered approaches for instrumentation during the development. The problem of instrumentation server requiring the dependencies for each instrumented class is explained together with the problem of instrumenting the classes with cyclic dependencies. The final solution is described as well. Further, the instrumentation API, which is used for communication with the server, is explained. The last section describes the bytecode parsing.

5.2.1 Instrumentation Details

The native agent does not perform the instrumentation but asks the server to carry out the transformation. The agent obtains the original bytecode for the class, sends the bytecode to the instrumentation server, waits for the transformed bytecode and lastly, applies the instrumented bytecode.

The instrumentation server requires all dependencies to be available for the class currently being instrumented. This means that all other classes referenced inside the class file need to be available on the instrumentation server. This includes:

- The argument types of all methods.
- The return type of all methods.
- The type of all fields.
- The type of a super class.
- The type of implemented interfaces.

The dependencies have to be loaded also for the referenced types. To achieve this, we tried two solutions, but only the second solution shown to be feasible.

Unsuccessful Solution

The first and unsuccessful solution was based on the fact that several `Class File load Hook` callbacks may be executed multiple times in different threads. When the application loads a class, the `Class File load Hook` event is triggered and bytecode of this class is made available. In this method, the new `Class File load Hook` event was artificially enforced via the `RetransformClasses` JVMTI method. This method accepts an array of classes for which the hook should be re-thrown. In order to continue with the instrumentation of the original class, all dependent classes have to be instrumented first. However, classes with cyclic dependencies are not supported in this approach. In order to instrument a class with some cyclic references, all dependencies have to be instrumented first, which is also the class itself.

This solution faced also a different problem. Since the number of dependencies can be significant, the problem of too many threads being opened at a single time has also appeared.

Chosen Solution

The second and currently used solution is based on the fact that Java class files may be accessed as a resource using the class loader, which is loading the class. The class file can be accessed using the `getResourceAsStream` method. In this solution, the instrumentation server is first asked whether the class currently being loaded should be instrumented. If the class is marked for instrumentation, its bytecode is sent to the instrumentation server[2]. Then, all references are scanned in the class file. For this, we need to parse the raw JVM bytecode. More details about this process are explained in Section 5.2.4. Loading of dependent classes is recursively called for each references class until the class does not have any other dependencies, or if all the dependencies are already uploaded on the instrumentation server. Once all dependencies for the class have been sent to the server, the instrumentation process is started and the agent waits for the transformed bytecode.

The disadvantage of this solution is that developers may override this method in their applications and not provide access to the class files. This is a limitation of the thesis. However, when such event happens, the instrumentation does not end with the exception, but the attempt to load the class using a different class loader created artificially is done.

Initializers and Interceptors

The instrumentation library used at the server (Byte Buddy) is using so-called `Initializer` class to set up special interceptor field in the instrumented classes. It is a static field, which references the instance of the class interceptor. An interceptor is a class, which defines the instrumentation code. This interceptor field is automatically set by Byte Buddy framework using the corresponding initializers in most use-case of this library. However, in the case of the Distrace tool, the instrumentation is performed in different JVM than where the code is actually running and Byte Buddy can't handle this case automatically. Therefore, initialization of the interceptor field has to be handled explicitly.

In order to set this field by corresponding initializer, both the initializer class and interceptor class need to be available to the agent. The instrumentation server sends the initializer class together with the instance of interceptor during the instrumentation of the class. Upon receiving, the agent registers the interceptor and initializers with the instrumented class for later to be applied. The interceptor

[2]This step is done only in case if the bytecode for the class is not already available

field is static and can be set up only when the class is used for the first time. Therefore, the initializers are loaded during `Class Prepare` event triggered by the JVM with the application and set up the interceptor field of the class. This event is triggered when the class is prepared, but no code has been executed so far.

This is not required in case the Advice API is used for the instrumentation. More information about the Advice API is in Section 2.5.4.

5.2.2 Auxiliary Classes

Auxiliary classes are created at run-time during the instrumentation of a class by Byte Buddy framework. The developer can annotate the class to be instrumented with Byte Buddy annotations. These annotations tell the framework to create for example a proxy to a super class or proxy classes to fields of the instrumented class. These proxies then can be used inside the instrumented code to access the objects outside the scope of the instrumented method. Any instrumented class, which is using these auxiliary classes, requires them to be available at run-time on the JVM with the application. Therefore, the native agent asks the server during the instrumentation for bytecode of the auxiliary class associated with the currently instrumented class. After receiving, the native agent saves the bytecode as a Java class file on disk and makes the class available to the application by adding the class to the application's classpath.

5.2.3 Instrumentation API

The `Instrumentation API` provides several methods used internally to communicate with the instrumentation server. It defines low-level methods for sending data in form of byte arrays or strings and the corresponding methods for receiving the data. Several more complex methods are built on top of these basic ones to make the communication easier. The most important methods are in the API are:

- `sendClassData` - sends bytecode to the instrumentation server.
- `isClassOnInstrumentor` - checks, whether the bytecode for a given class is already available on the instrumentation server.
- `instrument` - triggers the instrumentation and returns the instrumented bytecode.
- `loadInitializersFor` - loads initializers for a specific class.
- `loadDependencies` - loads all dependent classes and sends them to the instrumentation server. A dependent class is uploaded only in case it's not already available on the instrumentation server.
- `shouldContinue` - checks if the class on its input is marked for the instrumentation.
- `loadPrepClasses` - loads all classes, which are required by the monitoring tool to be available at run-time on the JVM with the application. These are for example `TraceContext` and `Span` classes.

5.2.4 Bytecode Parsing

Bytecode parsing is a necessary feature of the Distrace tool and is required for discovering the list of all dependent classes of a class currently being loaded. No sufficient C++ implementation has been found and therefore, a custom parsing module has been implemented. Bytecode parsing module in the Distrace tool is inspired by the Apache Commons BCEL library[3] written in Java. We created a very simplified C++ equivalent of this library with features required for our needs.

The main entry point for parsing is the `ClassParser` class, which contains `parse` method accepting the bytecode of a class to be parsed. The `ClassParser` class also defines several accessors for the parsed information. For example, we can get the name of the super class, the list of all implemented interfaces, the list of all methods or the list of all defined fields and their types.

The bytecode structure consists of several parts:

- **Magic id** - Magic id is the first integer stored in each bytecode and is always set to 0xCAFEBABE hexadecimal number.
- **Version** - Version consists of two numbers of `short` type. The first short represents minor Java version and the second major Java version.
- **Constant pool** - Constant pool is a table, which contains a mapping from id, representing a Java type, to the fully qualified type name. The id can represent interface names, field names and also other important constants.
- **Class Info** - Class Info contains the information whether the currently parsed bytecode represents a class or an interface. It also contains the name of this class and name of the parent class.
- **Interfaces** - This part of bytecode contains the number of interfaces this class implements. This number is followed by id of type `short` for each interface. The fully qualified name of an interface can be looked up using the class pool.
- **Fields**. This part of the bytecode contains the number of fields this class defines together with some additional information for each defined field. The fully qualified type of a field can be looked up using the class pool.
- **Methods**. This part contains the number of defined methods in the bytecode together with some additional information for each method such as the number of arguments. The fully qualified types of return value and arguments of the method can be looked up using the class pool.

More information about the class file structure can be found in the official Java documentation[4]. Each section of a class file is parsed separately. `ByteReader` class is used as a reader of the raw bytecode and contains several methods for reading different types of data from the bytecode array.

[3]More information about this library can be found at `https://commons.apache.org/proper/commons-bcel/`.

[4]The documentation of the class loading process is available at `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html`.

Parsing the magic id and both, minor and major versions, is straightforward as they are just numbers and can be read using the `ByteReader` class directly. Parsing of the constant pool is more complex. For each entry in the constant pool, a constant representing the entry is read. Each constant represent a specific object. For example, the constant can represent a class, a string constant, a type of a field or a return value and arguments of a method. Once the constant pool is parsed, it can be queried for the specific symbols using their id. Class name, super class name, interfaces, fields, and methods are read from the constant pool using their ids.

5.3 Instrumentation Server

This section describes several technical parts of the instrumentation server. The details of the instrumentation itself are provided first, followed by an overview of the optimizations the server does to speed up the communication with the native agent. The next section describes how native methods defined in Java on the instrumentation server are bound to their implementations on the native agent side. The last section describes how JSON objects, which represent spans, are generated.

5.3.1 Instrumentation Details

The classes to be instrumented are marked using the `MainAgentBuilder` and `BaseAgentBuilder` classes. The instrumentation server expects the instance of `MainAgentBuilder` on the input of its `start` method. This builder is the abstract class containing single abstract method `createAgent(BaseAgentBuilder builder, String pathToHelperClasses)`, where the builder is a wrapper around the Byte Buddy `AgentBuilder` class, which is used to define the class transformers.

The developer extending the core instrumentation server needs to implement this method and specify on which classes and on which methods the instrumentation should happen. Since Byte Buddy is used for writing transformers and interceptors, more information about Byte Buddy library is located in Section 2.5.4. In short, transformers are used to identify the class to be instrumented. They consist of advice or interceptors identifying the particular methods to be instrumented on the given class. The advice and interceptors also contain the code to be injected to the instrumented methods. The server provides several helper methods for creating the transformers and interceptors, which are less verbose than the standard Byte Buddy approaches.

Each interceptor has to implement the `Interceptor` interface. This is required so the server can discover all interceptor implementations at run-time without the need of changing the internals of the server. Each implementation of the interceptor needs to register itself in the META-INF directory of the generated JAR file in the same way as the span exporters mentioned in Section 5.1.1. Custom service loader is then used to locate all classes implementing the `Interceptor` interface. The interceptors need to be discovered since the instrumented classes depend on the interceptors and require them at run-time. Therefore, the instrumentation server has to send them to the native agent to make them available to the monitored application.

The advice may be used without any special annotations since Byte Buddy in-lines the code defined by the advice into the original code. Therefore, there is no need to transfer the advice implementations to the monitored application.

Even though Byte Buddy takes care of the internals of the instrumentation, the `BaseAgentBuilder` class is internally properly configured so the instrumentation is defined exactly as desired. This class implements four Byte Buddy listeners used reporting about the instrumentation progress. These listeners allow us to react to the process of the instrumentation. The listeners are:

- `onTransformation` listener is called immediately before the class is instrumented. Implementation of the listener in the Distrace tool also sends to the agent all auxiliary classes required by the instrumented class and the initializers used for setting the static interceptor field on the instrumented class.
- `onIgnored` listener is called when the class is not marked for instrumentation. The class is not instrumented if the developer does not define any transformer for the specified class.
- `onError` listener is called when some exception occurred during the instrumentation.
- `onComplete` listener is called when instrumentation sucessu completed. It is called after both of `onTransformation` and `onIgnored` listeners.

Byte buddy requires all dependent classes for the instrumented class to be available. They are needed because the instrumentation framework needs to know the signature of all methods so it can correctly identify the methods to be instrumented. The dependencies are all classes referenced in the class file such as the type of the method return value and arguments, the super class and the implemented interfaces.

By default, Byte Buddy library attempts to find these dependencies using `LocationStrategy` and `PoolStrategy` classes. The first class is used to tell Byte Buddy where to look for the raw bytecode of dependent classes. By default, the classes are loaded by the context class loader, but since the classes to be instrumented are received over the network, custom `InstrumentorClassloader` class loader is used to handle the class loading. It is a simple class loader which loads the class data from the agent and caches them. When there is a request for instrumentation, instead of looking into the class files, this class loader loads the bytecode from the cache and passes it to the Byte Buddy.

However, Byte Buddy internal API does not work directly with raw bytecode. It uses classes `TypeDescription` and `PoolStrategy`. The first class has a constructor accepting the `Class` class. The instance of this class contains metadata for the class passed to the constructor, such as the signature of all methods and fields, the list of all interfaces or, for example, the list of constructors. The second class is used for caching the type descriptions so they are not created every time the class is accessed.

In overall, the class lookup is done in the following two steps:

1. Check whether type description for the class is available. If yes, load the type description from the cache.
2. If the type description is not available, load the class using the `InstrumentorClassloader`, create type description for the class and put it in the cache.

Trace Context Field Injection

We also need to inject trace context details to the instrumented classes. The trace information is attached to the class by adding a new synthetic field with name `____traceContext`. This field can contain trace context representing the current trace and is used in the code to obtain a reference to the current trace context and also the current span. This new field is created using the Byte Buddy instrumentation builder with the `defineField` method.

5.3.2 Optimizations

The instrumentation server performs several optimizations to speed the communication with the native agent. The first optimization is caching of the classes sent to the instrumentation server from the native agent and also caching of already instrumented classes. This behavior is useful in cases where the native agents are sharing the instrumentation server. When a class is received from any agent, it is cached and the rest of the agents don't need to send the original class again when they request the instrumentation from the server. The server also performs the instrumentation only once and caches the instrumented classes. When any agent queries the server to instrument already instrumented class, the server can send the class immediately from the cache.

The second way how the communication can be optimized is influenced by the user. The user may compile the extended instrumentation server with the application classes or add these classes to the classpath of the server. When a native agent asks the server for instrumentation of a class, the server first checks if it can load the class locally and avoid transferring the bytecode from the native agent.

5.3.3 Binding the Native Methods

This section explains how methods implemented on the native agent can be used in the classes defined on the instrumentation server. Some classes created at the instrumentation server, such as **SpanExporter** class, have to use data from the native agent, but the native agent runs in different JVM. This is achieved by creating a helper method at the agent side, which returns the required data, and by declaring the corresponding native method on the Java side. When a class, which defines these native methods, is sent to the native agent and used for the first time, the native method in Java is bound to the corresponding implementation in C++. The methods are bound together inside the callback for the `Prepare` event. This ensures that we can define native methods in Java and bind them with their implementations on the separated machine, in this case, the machine with the native agent. Also, this can have performance benefits, since these methods are written as native methods.

For example, this technique is used for accessing the span exporter type inside the `SpanExporter` abstract class. This class is defined at the instrumentation server, however, the exporter type is passed as an argument to the native agent. This class contains the native method named `getSpanExporterType`, which returns the span exporter type. The `SpanExporter` class is sent to the native agent during the agent initialization and when it's used for the first time, the `getSpanExporterType` method is bound to the corresponding C++ implementation, which provides the value of this argument.

5.3.4 JSON Generation

The collected data inside spans are internally stored as instances of `JSONValue` class representing any JSON value. JSON format is chosen since default Zipkin user interface expects spans in this format. JSON is a lightweight format for exchanging data with the syntax based on Javascript object notation.

The JSON handling is inspired by the minimal-json library[5]. The simplified custom implementation was created which provides features required by the Distrace tool. Also, the number of dependencies required to build the Distrace tool is lowered since this code is part of the Distrace sources.

This JSON support is designed via several classes:

1. **JSONValue** - The abstract ancestor for all JSON types. This type defines common methods to all implementation.
2. **JSONString** - A class representing string types.
3. **JSONNumber** - A class representing numeric types.
4. **JSONLiteral** - A class representing the literals **null**, **true** and **false**.
5. **JSONArray** - A class representing the JSON arrays. It has support for adding new elements into the array.
6. **JSONObject** - A class representing the JSON objects. It has support for adding new items into the object.

Each **JSONValue** can be exported as a string where the export is driven by the `JSONStringBuilder` class. This class is also responsible for escaping the characters according to JSON standards. The default printer exports the data without any formatting into a single text line, however, `JSONPrettyStringBuilder` exports the data in more human-readable format. The second printer is usually used for the debugging purposes and the first one is used for exporting spans in real scenarios as the size of the data is smaller in this case.

[5]The library is available at `https://github.com/ralfstx/minimal-json`.

6. Realistic Case Study

This chapter demonstrates how the Distrace tool can be used on a bigger example and also serves as the user manual for creating custom monitoring applications. We will go through all the steps from creating the application up to running the application and accessing the observed results.

This example is based on the H_2O[1] open-source fast scalable machine learning platform. This platform supports various methods for building machine learning models, methods such as deep learning, gradient boosting or random forests. The core of the H_2O platform is written in Java and clients for different languages exist as well. Internally, H_2O is using map-reduce computation paradigm[2] to perform various tasks across the cluster.

The goal of this example is to monitor a subset of map-reduce tasks and see visualizations of the computation process. This can help reasoning about the performance of the platform and can discover unwanted delays in the computations. This chapter first describes relevant parts of the H_2O platform in more details. In the following sections, we describe in steps how to extend the core instrumentation library for H_2O purposes. Lastly, we show how this example can be started and how visual output can be interpreted. This full example is also available in the attached source code of the thesis in the Attachment 1. More examples are available and their complete list together with the instructions on how to run them are in the Attachment 2.

6.1 H_2O In More Details

H_2O is in-memory machine learning platform. The computations are performed in the cluster, where the cluster consists of several H_2O nodes. All nodes in the cluster are equal and each of them can initialize the computation. Each computation is performed as a map-reduce task. This section describes the internal format of data used in the H_2O platform, how data are stored on the nodes and lastly, how the computations are performed in the cluster.

6.1.1 Data in H_2O

Data are stored in H_2O in so-called `Frames`. A `Frame` is an in-memory distributed data table with columns and rows. The `Frame` is designed in a way, that it can handle data too big to fit into a memory of a single machine. Each column is represented by the `Vec` class. This class represents a vector of data that is again distributed across nodes in the cluster. Further, each vector is split into multiple `Chunks` and each chunk represents the part of the vector physically available on a single node.

[1]More information about H_2O can be found on `https://www.h2o.ai` and `https://github.com/h2oai`

[2]Map-reduce is a programming model in distributed systems. The basic idea is to split tasks into smaller parts and perform the map operations. The intermediate results are then combined together using the reduce calls until the complete result has been assembled from all the sub-tasks.

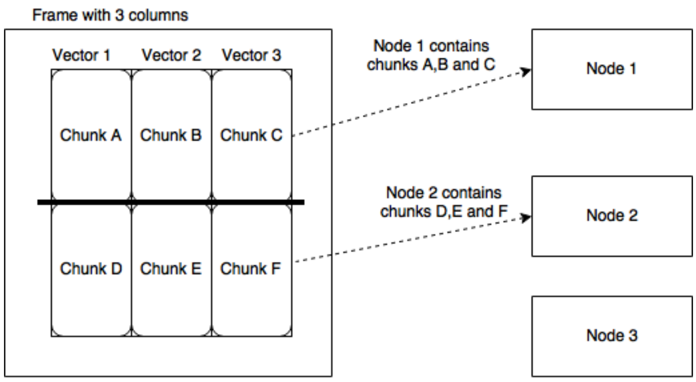

Figure 6.1: Structure of the H_2O frame and its distribution in the cluster.

It is possible for one node to contain multiple chunks from a single vector and therefore, the number of chunks in the vector does not represent the number of nodes on which the data are distributed. Typically, data imported to H_2O are distributed via chunks equally among all the nodes in the cluster, but algorithm may also decide to distribute the data on just several nodes in the cluster. It is also possible to create the frame manually and specify on which nodes the chunks should be stored. Therefore, the frame may be distributed only on a portion of the cluster. Usually, when chunks are being created on some specific node, chunks of the same size for each column are created on that node. This means that each node storing the data usually have corresponding (neighbor) chunks for all the columns. This can be thought of as that each node storing some data has a subset of rows from the full table with all columns.

The Figure 6.1 shows the structure of the frame with three columns, where each column is split into two chunks. It also shows how chunks may be distributed in the cluster of size three. We can also see in the figure that corresponding chunks for each vector have the same size and are stored on the same nodes.

6.1.2 Computation in H_2O

When the user tells H_2O to create a deep learning model based on the data on the input, H2O sees this as a **Job**. Jobs are used for tracking long-lifetime user interactions and encapsulate the whole computation from the user point of view. The job can consist of several map-reduce tasks. In H2O, the class `MRTask` is used as the core implementation of a map-reduce task. The map-reduce task is always bound to some H_2O frame on which the computation needs to be performed. This class is used to encapsulate the task, partition it to smaller tasks and run remote computations among the whole cluster. The `map` operations are called on each node, which contains locally available chunks for the frame this task is based on. The `reduce` calls are used to reduce the result from two sub-tasks into a new task with combined result from the children.

In more detail, the class `MRTask` extends from `DTask`. This class is a general class used in H_2O to represent task remotely executed. Further, `DTask` class extends from the `H2OCountedCompleter` class. The last class is a simple wrapper around the Fork/Join (F/J) execution framework[3] allowing the plat-

[3]For more information, please read Java documentation for Fork/Join framework available

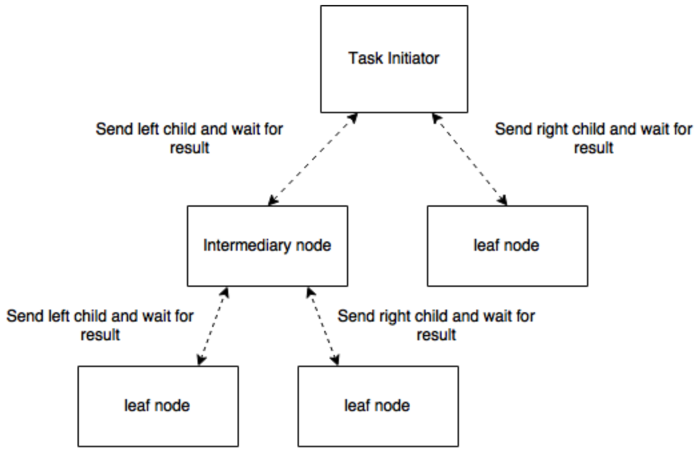

Figure 6.2: High-level overview of execution hierarchy.

form to prioritize tasks. The Fork/Join framework is an implementation of Java `ExecutorService`, which helps with job parallelization on multiple processors. The Fork/Join framework executes tasks in separated threads and can move tasks between threads to ensure the highest possible performance. Each H_2O `MRTask` is executed as `ForkJoinTask` inside this execution framework. A `ForkJoinTask` is a task wrapper, which can run inside a single thread. It is a light-weight wrapper and a large number of tasks may be served by a smaller number of actual threads.

The way how H_2O perform computation from the high-level point of view can be seen in Figure 6.2. The task initiator receives the `MRTask` from the parent `Job` or from the user. It splits the task into two new sub-tasks and sends these tasks to up to two different nodes in the cluster. These nodes again split the task from the input and send the task to another selected nodes in the cluster. Each splitting decreases the number of nodes on which the task has to be sent. Once this number os one, the node is marked as a leaf node and does not perform any other splitting and re-sending.

It is important to say that the `MRTask` is always distributed to all nodes in the cluster. This figure shows how it is ensured that each node participates in the computation, however, the computing step itself is still missing. Each node of the cluster, who receives a task, also submit the task for computation into the Fork/Join execution framework. This computation performs the mapping operation on all the chunks, which are available locally on the node executing this task, for the frame associated to the task. The `reduce` operation follows the `map` operation, however, we need to first ensure that the child tasks, from which we want to combine results together, are already finished. This is ensured by child tasks signaling to the parent tasks when the work has been finished. Therefore, parent tasks are informed when they can start reducing the results.

The computation on a single node is shown in the following pseudo-code:

```
MRTask task = ... // task received from the parent or from the user
MRTask left = split(task, start1, end1)
MRTask right = split(task, start2, end2)
remoteCompute(left)
remoteCompute(right)
```

at `https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`.

```
H2O.submitTask(task) // submit this task into local F/J
..
// task is taken from F/J for execution
task.map(..)
task.waitForComplete() // wait for the child tasks to finish
task.reduce(left, right)
notifyComplete(task) // notify parent of completion
```

The split method accept indices representing nodes in the cluster on which this sub-task and further sub-sub-tasks may be executed.

The last missing piece of information is a description of what is done when the node has more chunks available for the frame associated with the task since each `map` operation is executed only per single chunk. In case the node has multiple chunks available for the frame, the node always locally submits two new tasks into the F/J framework, each having half of the chunks. This is recursively repeated until we have single chunk tasks, which are processed normally. These locally-split tasks inform their parents that they are done. This signalization goes up the tree until the original task is marked finished.

6.1.3 Methods for Instrumentation

For purpose of this example, a special `MRTask` called `SumMRTask` has been created. This task just performs distributed sum of a range of numbers. In order to be able to visualize the computation process of this task, the following `MRtask` methods are important for the instrumentation

- `dfork` - this method is called at the initiator node and starts the computation of the whole task.
- `getResult` - this method is called at the initiator and blocks until the distributed computation finishes.
- `setupLocal0` - this method splits the task, creates sub-tasks for child nodes and finally, submits the sub-tasks to the target child nodes.
- `map` - the map operation.
- `reduce2` - the reduce operation.
- `compute2` - this method handles the computation itself by calling the `map` implementation.
- `onComplete` - this method waits for the sub-tasks to finish and then calls `reduce2` on them.

We are also interested in how long the remote computation lasts on child nodes. For this reason, we need to instrument the following two methods on the `RPC` class:

- `call` - this method is called when the remote computation has been submitted.
- `response` - this method is called when the remote computation has been finished and the child node is signaling that its work is done.

6.2 Building the Core Server and Native Agent

In order to be able to extend the core instrumentation library, we need to build it first. This won't be necessary once the core instrumentation server is published to some online repository of JAR packages[4] The native agent also needs to be built on the platform where H_2O will be running.

Please see the Attachment 3 for the information on how to build the project from sources or how to run this example from the prepared Docker machine.

6.3 Extending the Core Instrumentation Server

In order to capture the relationships between tasks and their computations, we need to instrument the methods mentioned in the previous section. It is important to always find a good pair of methods, which open and close a single span. In the case of H_2O, the following calls have been identified to form good spans. The pairs are ordered from the top level spans encapsulating the whole computation to the local spans encapsulating single mapping or reduce operations.

1. `dfork` - `getResult`. This pair is used to open the main span, which encapsulates the complete computation of a single `MRTask`. This is possible since the `dfork` method is only called on the node where the computation starts and the same holds for the `getResult` method after computation finished. The span is opened when the first method is entered and closed when the latter method is left.

2. `setupLocal0` - `onCompletion`. This pair forms the span representing the complete communication on a single node. It encapsulates the local work as well as the waiting for the remote work to complete. This span is opened when the first method is entered and closed when the latter method is left. This span is well-defined since the `onCompletion` method is called for each task and represents that the work has been finished in all children tasks.

3. `call` - `response`. This pair forms the span used to track only remote computation. It encapsulates the remote computation and all following sub-tasks created by the remote task. This span is opened when the first method is entered and closed when the second method is left.

4. `compute2` - `onCompletion`. This pair forms the span used to encapsulate the computation on a local node. It encapsulates all the cases of local work. In case multiple chunks exist for the given local task on the same node, this method is recursively called on split tasks until we have tasks representing a single chunk. The calls are also recursively confirmed by the `onCompletion` call.

   In multi-chunk case, only one child task is submitted for execution into F/J thread. The second task is executed immediately in the same thread. This way it is ensured that we can reuse the existing thread as much as possible. This span is opened when the first method is entered and closed when the later method is left.

[4]For example, Maven Central Repository.

5. `map` - `map`. This pair represents the single mapping operation. This span is entered when the `map` method is entered and closed when the `map` method is left. Therefore, this span lasts only for the duration of the `map` method call.

6. `reduce2` - `reduce2`. This pair represents the single reduce operation. This span is entered when the `reduce2` method is entered and closed when the `reduce2` method is left. Therefore, this span lasts only for the duration of the `reduce 2` method call.

For all of the pairs above, the Advice API is used for the instrumentation because it's sufficient in our case to be able to capture just method enter and exit events. In case we would like to perform more complex instrumentation, we could use Interceptors API, which is described in the Byte Buddy documentation and briefly also in Section 2.5.4.

Instrumenting these methods is a technical task and the code can be seen in the corresponding example available in the Attachment 1. The trace context information is always attached to the task transferred between the nodes. The trace context is initially created during the `dfork` method call since it's the main entry point of the computation. When instrumenting `compute2` and `setupLocal0` methods, the deep copy of the trace context is attached to the transferred task because the `onComplete` method is usually called from a different thread and this ensures there are no collisions caused by accessing the same trace context from multiple threads.

Also, when instrumenting a few of the mentioned methods, we need to ensure that correct pairs of spans are created and closed. For this purpose, we use flags available on the `Span` class. Flags allow us to attach additional information to the trace context, which may be used when closing the span. This is, for example, useful in cases when a method is used to close several spans at the same time and we need to properly distinguish between the spans to close the correct one.

Once all the advice or interceptors are created for each of the mentioned methods above, we need to create the transformers. The transformer defines on which method from the code of the application which advice or interceptor is applied. For this purpose, Distrace provides several helper methods allowing us to create transformers in a very concise way. For example, the transformer for `RPC.class` defining instrumentation of the `call` and the `response` method looks like:

```
new BaseTransformer() {
@Override
public DynamicType.Builder<?>  defineTransformation(
        DynamicType.Builder<?> builder) {

        Method call = ReflectionUtils.getMethod(RPC.class, "call");

        return builder. visit (Advice.to(RPCAdvices.call.class).on(is( call ))).
         visit (Advice.to(RPCAdvices.response.class).on(named("response")));
}
});
```

The `RPCAdvices.call` static class and `RPCAdvices.response` class are the advice, which defines the instrumentation.

Once the transformers for each instrumented class are created, we need to associate the transformers with the classes from the application on which they should operate. We also need to create a main entry point of the extended instrumentation server. This is demonstrated on the following code snippet:

```
public static void main(String args[]) {
        new Instrumentor().start(args, new MainAgentBuilder() {
                @Override
                public AgentBuilder createAgent(
                        BaseAgentBuilder builder,
                        String pathToHelperClasses) {

                        return builder.type(isSubTypeOf(MRTask.class))
                        .transform(mrTaskTransformer)
                        .type(is(RPC.class))
                        .transform(rpcTransformer)
                }
        });
}
```

In this code, we associate a class to each transformer and also, we are starting the instrumentation server using the API provided by the core instrumentation server with the `MainAgentBuilder` instance. This instance is later used as a dispatcher of the instrumentation of the whole application.

Later, when starting the application with the native agent attached and configuring the agent, we need to explicitly specify the class, which is used as the main entry point. This is exactly the class containing the `main` method with the content above.

Now we need to build the extended instrumentation server. This server has two dependencies: the core instrumentation server and also the H_2O application sources. The first dependency is obvious as we are using the API defined in the library. The second library is required since we are using several application classes when defining the instrumentation points. This may not be necessary when instrumenting different applications since we can identify the class to be instrumented for example by its name as `named("fully.qualified.class.name)` instead of `is(Example.class)`. The second option, however, gives us more freedom when defining the instrumentation points and has also a performance benefit. In this case, the application classes are already located on the server and when they are requested to be instrumented, their original bytecode doesn't have to be sent from the native agent.

After this step, we should have two artifacts - the native agent library built for our platform from the previous steps and also the extended instrumentation server JAR file from this step.

6.4 Configuring and Running the Application

For the purposes of this example, we start the cluster of three H_2O nodes. Two nodes are the regular nodes and the last node contains the main method in which the **SumMRTask** is executed. This task is used to sum up a range of numbers in distributed manner.

An arbitrary H_2O node can be started as:
`java -jar h2o.jar -name cluster_name`.
When operating on the network with multi-cast communication enabled, multiple nodes started with the same cluster name automatically forms a single cluster.

If we want to start the application with the monitoring agent attached, we can use the `-agentlib` java option. Any H_2O node can be started with the monitoring feature enabled by calling the following command:

```
java -agentpath:"$NATIVE_AGENT_LIB_PATH=$AGENT_ARGS" -jar h2o.jar
-name cluster_name
```

The `$NATIVE_AGENT_LIB_PATH` variable needs to point to the location of the native agent library and `$AGENT_ARGS` shell variable may contain any arguments passed to the native agent. The arguments are in the format `key=value` and are separated by the semicolon. The complete list of available arguments is in the Attachment 4.

In the case of this example, we let the native agent start the instrumentation server locally for each node automatically. Therefore, the inter-process communication is used and we don't need to configure it explicitly. Only two arguments need to be specified - the path to the instrumentation server and the fully qualified name of the main entry class of the instrumentation server.

Therefore, the full command starting H_2O with the monitoring agent enabled is:

```
java -agentpath:"/home/agent.so=instrumentor_server_jar=
/home/instrumentor.jar;instrumentor_main_class=main.entry.pint"
-jar h2o.jar -name cluster_name
```

In order to start the cluster of size three, we need to call this command three times. Twice with the regular H_2O node and once with the H_2O node containing the execution of the `SumMRTask`. It is also important to start the Zipkin user interface to which the results are published. The user interface server may be started as: `java -jar zipkin.jar`[5].

6.5 The Results

Once all three nodes have been started, the computation starts and the results based on our instrumentation are shown directly in the Zipkin user interface. By default, the user interface is available at port 9411.

We need to click on the *Find Traces* button to show all traces, which satisfy our search conditions. We should see in the output a single trace and once we click on it, we should see a similar result to the one in Figure 6.3. This figure shows just a portion of the whole trace but contains the important observed information.

[5]Zipkin Jar file may be downloaded at `https://github.com/openzipkin/zipkin` or is available at the attached CD.

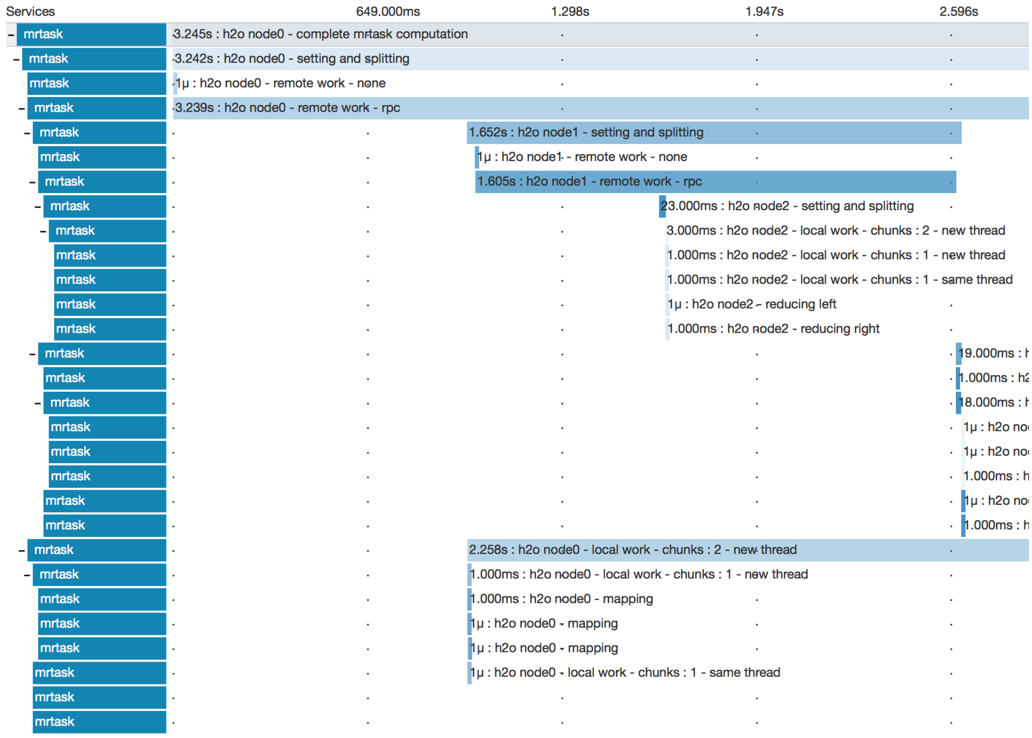

Figure 6.3: Example trace from the distributed computation on H_2O platform.

All the operations and their timing can be seen on the output and we can see how long each operation lasted and when it started. The spans are also organized hierarchically as they were created. We can see the main span encapsulating the whole computation, the spans for computation on each node and also spans encapsulation the remote computations. The local `map` and `reduce` calls are displayed as well. It is, for example, interesting to see how long it took the platform to perform the `reduce` operation after the `map` operation has been called.

7. Evaluation

This chapter firstly shows the measurements of how long the monitored application starts with and without the tracing enabled. These measurements are based on the case study available in Chapter 6. This section is followed by a description of known limitations of the Distrace tool.

7.1 Measuring the Tool Overhead

This section measures the overhead of the Distrace tool on the H_2O example. We measure how long the example run from the start to end when the agent is attached and monitoring enabled and also in the opposite case. This measurement is highly specific on the type of the target program since it depends on several factors, such as the number of classes being instrumented or whether we optimised the performance of the server by adding the application classes on the classpath of the instrumentation server.

In this measurement, we use instrumentation server which already has application classes on its classpath since it's the advice and most generic use-case of the Distrace tool. We also run the example in the local instrumentation server mode, which means that each native agent starts the server for the local application automatically. The measurements have been performed on H_2O cluster of size three with 16 GB memory available and Intel Core I7 quad-core CPU.

The following Table 7.1 contains numbers in seconds how long it took to start the whole cluster of size three, with and without the instrumentation enabled.

Run Number	Monitoring off	Monitoring on
1	37,378s	40,912s
2	30,699s	50,104s
3	36,902s	45,844s
4	36,709s	46,502s
5	31,063s	47,503s
6	30,799s	50,440s
7	36,799s	44,695s
8	37,358s	50,504s
9	37,844s	47,444s
10	36,969s	44,909s
average	**35,252s**	**46,886s**
standard deviation	**3,054s**	**3,026s**

Table 7.1: The start-up time of the whole cluster.

We can see that in this case, the average run time with monitoring enabled is on the average 10 seconds slower. This can be explained as the overhead of starting the instrumentation server from the native agent on a single machine for all three H_2O nodes in the cluster.

The following Table 7.2 shows how long only the computation of the first map-reduce task lasted with monitoring enabled and disabled.

Run Number	Monitoring off	Monitoring on
1	12,232s	16,810s
2	12,359s	14,467s
3	12,293s	15,681s
4	12,331s	13,196s
5	12,229s	11,055s
6	12,360s	15,037s
7	11,867s	11,839s
8	12,399s	17,246s
9	12,256s	11,088s
10	12,323s	15,500s
average	**12,265s**	**14,192s**
standard deviation	**0.151s**	**2.283s**

Table 7.2: Run time of the first map-reduce task.

The overhead in the case when the monitoring is enabled is caused by the instrumentation of the classes when they are first needed. We can also see that the deviation is higher in the case when instrumentation is enabled. This may be explained by the variety in transfer times of the classes between the instrumentation server and the agent.

Lastly, the final Table 7.3 shows how long each subsequent computation run, also in the case when monitoring is enabled and disabled. This means that we omit the first map-reduce task computation and measure only following calls when all instrumentation has already finished for the required classes.

Run Number	Monitoring off	Monitoring on
1	2,025s	3,279s
2	2,025s	4,143s
3	2,030s	3,153s
4	1,079s	4,225s
5	1,025s	3,653s
6	1,071s	3,367s
7	0,990s	3,055s
8	1,064s	2,659s
9	0,264s	3,254s
10	0,999s	2,721s
average	**1,257s**	**3,351s**
standard deviation	**0.583s**	**0.527s**

Table 7.3: Run time of the following map-reduce tasks.

We can see that there is still overhead by the introduced monitoring, but not a significant one. This overhead is caused by the extra work introduced by the monitoring tool. This can be, for example, checking whether we are closing the correct span or exporting the span.

7.2 Known Limitations

The Distrace tool has a few limitations, which we would like to address in the future.

- **Required Java version**
  Distrace requires Java 8 to be available. The platform has been tested on several Java 7 implementations and several internal Java bugs occurred. These problems are already fixed in the Java 8. Even though Java 7 is being replaced by Java 8 nowadays, it still can be seen as a limitation of this tool.

- **Overriden `getResourceAsStream` method**
  The instrumentation process requires that the `getResourceAsStream` is able to return a class file for a class. However, developers may override this method and not provide the class files when we ask for them. When this happens and we are not able to load the class file using this method, another attempt to load the class file with a different approach is done, however, this is still a limitation of the Distrace tool.

- **Attaching Agent at Run-Time**
  Currently, the native agent has to be attached prior the application start using the `-agentlib` or `-agentpath` options. However, Java provides the attachment API allowing the agents to join at run-time of the application. This has the benefit that the application can be started without any additional arguments. The thesis contains the sub-project called **agent-attacher**, which is using the attachment API and attaches the agent to the running application, but currently, the agent does not perform any tasks when it's attached at run-time.

  The agent is disabled for this use-case since we would need to properly handle and separate the instances of instrumented classes before and after the instrumentation. It is possible that an instance of some class has been already created, the class has been instrumented, and new instances of this class have been created. Therefore we would have instances of the same class, first instrumented and the second not. This could be a problem in some applications and therefore, this feature still needs a further investigation before it's allowed.

8. Conclusion

The main goals of this thesis were to create a monitoring tool for distributed Java applications with a small footprint on the monitored application and high-level application transparency and tool universality. It was also desired to ensure the usage and deployment of the final tool are easy.

The instrumentation overhead can be still observed even though the classes are instrumented in a separated instrumentation machine, however, it is just a constant overhead based on the nature of the injected code to the original classes. The Distrace tool universality was achieved by implementing the native agent universal to all Java applications and by creating a core instrumentation server. This core server can be extended by developers and they can create application specific instrumentation tools. This also ensures that the final users of the monitored application does not need to know about the monitoring and can work with the application as usual. The developer has also the possibility to extend the monitoring platform by a custom user interface and can also specify a custom format for spans being exported from the application. This ensures that the Distrace tool may be integrated easily into already existing environments. The interface at the core instrumentation server is kept simple in order to make the usage straightforward also for developers.

Comparing to the related project Google Dapper, the Distrace tool introduced by this thesis is released as open-source and allows higher application transparency since the purpose of Google Dapper is to monitor only Google applications. Comparing to Zipkin, the user does not need to change the application sources in order to attach span and trace information. This ensures that the source code of the original application remains unchanged. The tool provided by this thesis does not aim to replace any of the mentioned tools, however, it tries to create a universal tool with keeping performance in mind and ensuring that the usage is a simple as possible for the end-user.

This Distrace tool is also planned to be extended in the future in the following areas:

- **More Additional Span Exporters**
  Currently, the Distrace tool provides two default span exporters and allows the user to extend the `SpanExporter` abstract class and implement custom ones. However, we would like to create more exporters in the future, which would be able to store spans into different storage types and also in different formats. At this moment, the output is in the JSON format understandable to the Zipkin user interface and the data are exported either to a disk or are send to the user interface right away. We could, for example, create a span exporter, which could export spans into a database, from which the arbitrary user interface could fetch the data.

- **Support for Flame Graphs**
  The second future plan is to add support for flame graphs. The native agent could be used to capture the stack-traces of the running application and later, a flame graph representing the distributed computation could be created. For example, this integration would give us the ability to in-

spect the memory usage or performance cluster-vise using the flame graphs visualisations.

Bibliography

[1] Zipkin Architecture, 2017. URL http://zipkin.io/pages/architecture.html.

[2] David Buck. Inflation System Properties, feb 2014. URL https://blogs.oracle.com/buck/inflation-system-properties.

[3] Shigeru Chiba. Getting Started with Javassist, 2015. URL https://jboss-javassist.github.io/javassist/tutorial/tutorial3.html#boxing.

[4] Brendan Gregg. Java Mixed-Mode Flame Graphs at Netflix, JavaOne 2015, nov 2015. URL http://www.brendangregg.com/blog/2015-11-06/java-mixed-mode-flame-graphs.html.

[5] Andrew Josey. POSIX® 1003.1 Frequently Asked Questions (FAQ Version 1.15), jun 2015. URL http://www.opengroup.org/austin/papers/posix_faq.html.

[6] Ignacio Laguna, Zhezhe Chen, Feng Qin, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, and Bowen Zhou. Debugging High-Performance Computing Applications at Massive Scales. *Communications of the ACM*, 58(9):72–81, aug 2015. doi: 10.1145/2667219. URL https://doi.org/10.1145%2F2667219.

[7] Kelly O'Hair. The JVMPI Transition to JVMTI, jul 2004. URL http://www.oracle.com/technetwork/articles/java/jvmpitransition-138768.html.

[8] Oracle. The Invocation API, 2014. URL http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/invocation.html.

[9] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010. URL https://research.google.com/archive/papers/dapper-2010-1.pdf.

[10] Martin Sustrik. Differences between nanomsg and ZeroMQ. URL http://nanomsg.org/documentation-zeromq.html.

[11] Rafael Winterhalter. Testing the performance of 4 Java runtime code generators: CGLib, Javassist, JDK proxy & Byte Buddy, jul 2014. URL https://zeroturnaround.com/rebellabs/testing-the-performance-of-4-java-runtime-code-generators-cglib-javassist-jdk-proxy-byte-buddy/.

Attachments

1. Attachment 1: CD with the source codes of the Distrace tool.
2. Attachment 2: List of examples available in Docker and instructions on how to run them.
3. Attachment 3: Instructions on how to build and run.
4. Attachment 4: List of native agent arguments.

Attachment 2: List of Examples

The source code of the thesis contains several examples. They can be run manually by first compiling the sources and then starting using the provided scripts. The examples are also available in the provided Docker machine. This docker machine contains all compile and run-time dependencies for the tool to be able to run and examples can be run on this machine as well.

The list of available examples:

1. **DependencyInstrumentation**
   This example just demonstrates the basic instrumentation functionality on dependent classes. It does not have any output do the user interface.

2. **H2OSumMRTask**
   This larger example demonstrates the tool on the H_2O machine learning platform. The cluster of three H_2O nodes is started and a simple map-reduce task is executed within this cluster. The internals of map-reduce task are monitored and the associated spans are shown in the Zipkin user interface.

3. **SingleJVMCallback**
   This example demonstrates how the Distrace tool can be used to instrument and monitor the callbacks. This example is using a single JVM with multiple threads.

4. **SingleJVMThread**
   This example shows how threads can be instrumented and monitored. This example is running on a single JVM with multiple threads.

Attachment 3: Building And Running

To build both, the native agent and the core instrumentation server, the developer needs to obtain the Distrace tool sources[1] and call `./gradlew build`[2] command, which builds the tool and produces artifacts for the core instrumentation server and the native agent as well. The build process requires several dependencies to be available.

The instrumentation server requires Java 8 to be available. It has dependencies on several libraries, but these are downloaded automatically by the build process.

The native agent depends on several libraries, which need to be available on the system:

- Boost-filesystem
- Boost-system
- Nanomsg
- Nanomsgxx
- Cmake

The last dependency is required to be able to build the native agent project. Any C++11 compliant compiler needs to be available as well.

Once the artifacts are built the user may extend the core instrumentation server with the instrumentation definition. The application can be further run using the following incantation:

`java -agentpath:"$NATIVE_AGENT_LIB_PATH=$AGENT_ARGS" -jar app.jar`

The `$NATIVE_AGENT_LIB_PATH` shell variable should point to the location of the native agent library and `$AGENT_ARGS` shell variable may contain any arguments passed to the native agent. The arguments are in the format `key=value` and are separated by the semicolon. Please see the Attachment 4 for the full list of available native agent arguments.

In order to be able to see the spans in the user interface, we need to start the Zipkin user interface first. The user interface server may be started as: `java -jar zipkin.jar`[3].

8.1 Running Docker Examples

In order to run this example without the need to set up all dependencies, the Docker container with this example is prepared. This container contains all Distrace dependencies and when started, is also automatically starts the Zipkin user

[1] Distrace is available at `https://github.com/jakubhava/Distrace`.

[2] On Windows, `./gradlew.bat build`

[3] Zipkin Jar file may be downloaded at `https://github.com/openzipkin/zipkin` or is available at the attached CD.

interface on the default port. To run any example in Docker, Docker needs to be available, the Distrace source directory needs to be available as well[4] and the following script should be called: `./docker/run-test.sh`[5].

Both these scripts expect a single argument representing the example name. When this argument is missing, the list of available examples is printed to the console.

[4]The sources are not actually needed for running the examples in docker, but the script which is used to start the docker machines is available there as well.

[5]On Windows, `./docker/run-test.cmd`

Attachment 4 : Native Agent Arguments

The native agent accepts several arguments which can be used to affect the agent behavior. In local instrumentation server mode, several arguments affect also the server started from the agent. Available arguments are:

- **instrumentor_server_jar** - specifies the path to the instrumentation server JAR. It is a mandatory argument in case the instrumentation server is supposed to run per each node of monitored application.
- **instrumentor_server_cp** - specifies the classpath for the instrumentation server. It can be used to add application specific classes on the server classpath which has the effect that the monitored application does not have to send to the server these classes if they need to be instrumented or if some class to be instrumented depends on them.
- **instrumentor_main_class** - specifies the main entry point for the instrumentation server. It is a required argument in the case of local instrumentation server mode.
- **connection_str** - specifies the type of the connection between the native agent and the instrumentation server. It is a mandatory argument in shared instrumentation server mode in which case the value is in format `tcp://ip:port` where ip:port is the address of the instrumentation server. Otherwise, the agent and server communicates via inter-process communication and the argument can be set in format `ipc://identifier` where identifier specifies the name of pipe in case of Windows and name of the file used for IPC in the case of Unix. However, this value is set automatically at run-time if not explicitly specified as the argument.
- **log_dir** - specifies the log directory for the agent and when running in local server mode, specifies the log directory for the server as well.
- **log_level** - specifies the log level for the agent and when running in local server mode, specifies the log level for the server as well.
- **span_exporter** - specifies the span exporter type. The value can be either `directZipkin(ip:port)`, where `ip:port` is the address of the Zipkin user interface or `disk(destination)`, where destination represents the output directory for the captured spans.

  Custom span exporters are supported as well. In that case, the format of the value is fully qualified name of the span exporter with arguments in parenthesis, for example as `com.span.exporter(arguments)`
- **class_output_dir** - specifies the output directory for several helper classes received from the instrumentation server. This value is automatically set if not configured explicitly.

- **config_file** - specifies the path to a configuration file containing the agent configuration. It can contain all arguments mentioned above, except the configuration file argument. Argument entries in the configuration file are in the format `arg=value` and each entry is on a new line of the configuration file.