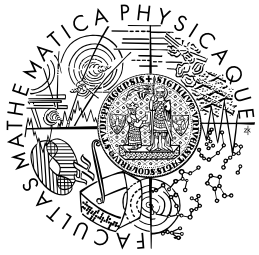

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 the 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 represents 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 RNDr. Pavel Parízek, Ph.D 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 RNDr. Michal Malohlava, Ph.D 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 growth of distributed applications there is also increasing demand for monitoring and debugging 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. Analysis of single-node applications usually focuses on a single standalone applications where most of the information required to reason about it is collected directly from the application. In case of distributed application it is desired to collect the same information as on single-node applications plus, and more importantly, the shared state between the communicating nodes. 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 information 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 for 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 send the required information. 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 already transferred between communication nodes. New analysis method may require new metrics which in this case would also mean recompiling and new deployment of the application.

For this reason several new monitoring and debugging tools have been developed. These tools are usually built on the code instrumentation technique. This method is used to alter the monitored application's code at run-time in order to collect all relevant information. The significant advantage of this method is that the original application does not have to be changed in order to add additional metrics. 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 give

the possibility to see the desired relations between the applications node. Google Dapper and OpenZipkin 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 and should give the user a new generic and high-performance way how to monitor distributed applications. Main goals of the thesis are to create an open-source generic monitoring library with small footprint on the monitored applications whilst giving the user the possibility to use high-level programming language to define instrumentation points.

Main requirements for the platform 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. The project 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 do collect the shared state between the application nodes. This information is later used for span collection and associating the relationships between spans.

- **Application-level Transparency and Universality**
  These two requirements are contradictory to each other. The universal tool which could be used for monitoring of majority of applications would either collect just basic information shared about all applications or the user would be required to manually specify the information to be collected specific to the application, which leads to loose of the application-level transparency. This tool tries to find a compromise between these two goals and attempts to support high-level of universality with minimizing the impact on the application itself. The project needs to do some trade-offs between the application-level transparency and the universality of the platform. The goal is to be able to instrument JVM-based applications to minimize the impact on the application's code itself.

- **Easiness of Use**
  The application should use high-level programming language for the instrumentation and specifying 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 complexity of deployment of this tool is also a significant aspect of the tool. In order for developers and testers to use this tool frequently, its deployment and usage has 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 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 significantly affecting the code of the tool.

The discussion of different approaches for meeting the requirements above are discusses in the Analysis chapter.

1.2 Thesis Outline

The thesis starts with the Background chapter. The purpose of this chapter is to give the reader overview of relevant tools to the thesis such as an list of several profiling tools, instrumentation and communication libraries. It also describes the relevant cluster monitoring tools like Google Dapper and OpenZipkin in more detail. The following Analysis chapter contains discussion of how specific requirements of the thesis are met and 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 Design chapter starts with the Overview section. This section depicts the architecture of the whole system and is followed by the Example Use Case section which describes how Distrace can be applied in this simple case to demonstrate it functions. Further, the Design chapter contains sections for each important part of the application such as explanation of Spans, Instrumentation server, the Native agent and the User interface. The next Implementation chapter describes several interesting implementation details in more depth and is followed by Big Example chapter. This chapter purpose is to show a more complex example of how Distrace can be used, in this case on the H2O machine learning platform. The example goal is be able to see the hierarchy of internal map reduce calls inside the H2O platform and also to see how long each map and reduce operation. The next Evaluation chapter mentiones the current limitations of the tool and also compares different deployment strategies of Distrace and give the comparisons between them.

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 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 in Distrace or used in thesis 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 based are Google Dapper and Zipkin. Both tools serve the same core purpose which is to monitor large-scale Java-based distributed applications. Zipkin is developed according to Google Dapper design which means that these two platforms share a few similar concepts. The basic concept shared between these two platforms is a concept called *Span* and it is explained in more details in the following section in the meaning of this thesis. For now, a span can be though of as time slots encapsulating several calls from one node to another with well-defined start and end of the communication.

The following two sections describe the basics of the both mentioned platform. Since both Zipkin and Google Dapper shares some basic concepts, only relevant and interesting parts to the thesis of each platform are described.

2.1.1 Google Dapper

Google Dapper is proprietary software used at 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 host at the same time. Parts of the monitored system does 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 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 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 at the Google use the same libraries and share similar control flow practices.

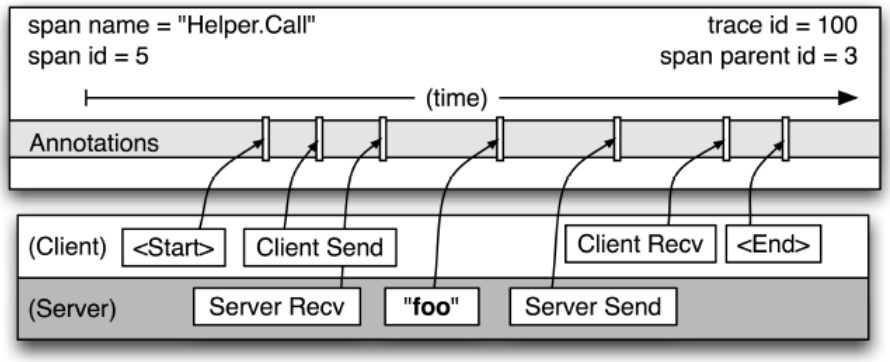

Figure 2.1: Example of Span in Google Dapper. Picture taken from the Google Dapper paper

- **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 which took part in the computation or communication.

There were two approaches proposed for obtaining the distributed traces when Google Dapper was developer: 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 schema 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 core system. This gives the developer of an application possibility to attach 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 represents relationship between parent span and children of this span. Usually the edges represents some kind of RPC calls or similar kind of communication. Each span has 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 serves exactly this purpose. Spans can also contain information from multiple hosts, usually from direct neighborhood of the span. Structure of a span in Google Dapper platform is described in the figure 2.1.

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 so the system has high-level of application transparency. The instrumentation points are:

- Dapper attaches so called trace-context as thread-local variable to the

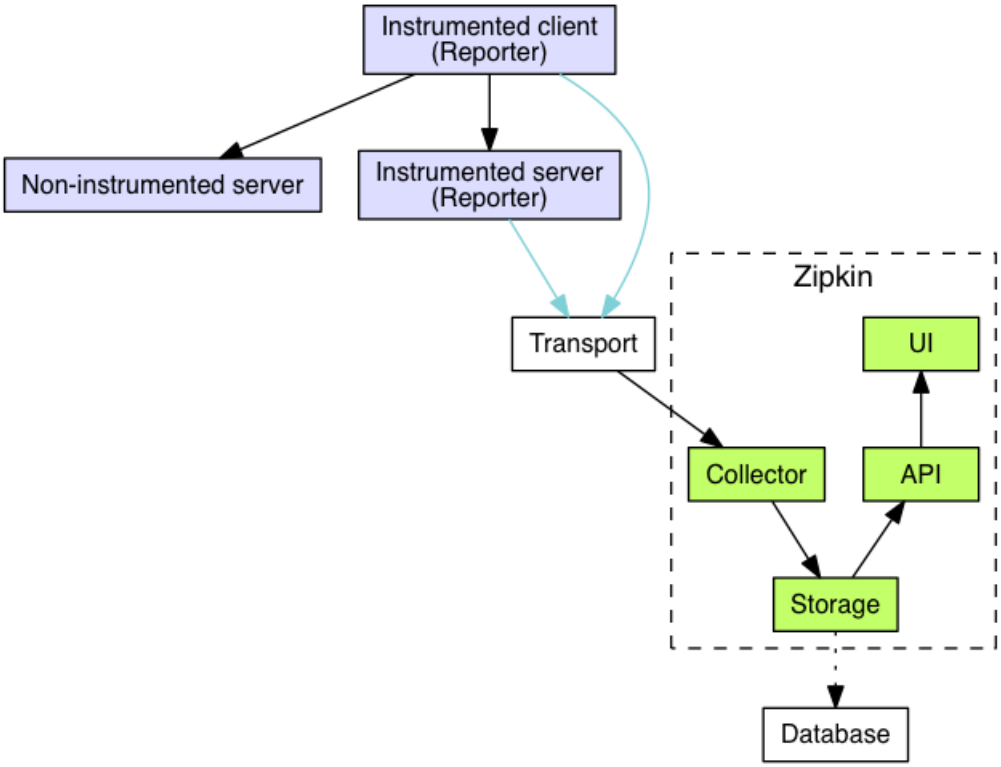

Figure 2.2: Zipkin architecture - http://zipkin.io/pages/architecture.html

thread when it handles any kind of control path. Trace context is small data structure containing mainly just reference to current and parent span via their ids.

- Dapper instruments the callback mechanism so when computation is deferred, the callbacks still carry around 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 to different languages. This library is instrumented as well to achieve the desired transparency.

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 at Google is significant so only samples are taken at a time.

2.1.2 Zipkin

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

Zipkin architecture can bee seen on 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, it is sent to Zipkin collector. In General, Zipkin consists of 4 components:

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

- **Storage**
  Data in Zipkin can be stored in a multiple ways, so this is a pluggable component. Data can be stored for example in Cassandra, MySQL or can be send to Zipkin UI right away without any intermediate storage at all. The last option is good for small amount of data since the user interface is not supposed to handle data storage.

- **Zipkin Query Service**
  This component acts as a query daemon allowing the user to query various information about span using simple JSON API.

- **Web UI**
  Basic, but very useful user interface. It allows the user to see whole trace trees and all spans with dependencies between them. The user interface accepts the spans in JSON format.

The Zipkin Web UI is used as front-end for the monitoring tool developed on this thesis. More information how it is used in the thesis is described in more detail in Zipkin UI section of Design chapter.

2.2 HTrace

HTrace https://github.com/cloudera/htrace 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 as Spans and Traces. In order to allow tracing of the application, the users need to manually attach Span identifiers to desired RPC's (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. Htrace has also support for custom Spans and 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 tool is to debug a 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 desired to know how the platform scales when it process significantly big data and what is the expected scalability trend. It can happen that the platform can run significantly slower on big data than expected when tested on smaller data. This issue is usually called a scaling bug. Tools which can be used to help discovering scaling bugs are for example Krishna and WuKong. 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, Wrishna, 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 Krishna, Wukong does not build one scaling trend of the whole application, but creates more 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 scalling bug, WuKong is able to locate us in the place in the code where the trend is probably violated.

2.3.2 Behavior-based Analysis

The second category of tools used for debugging large scale applications are 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 lowed down. These tools are especially helpful when discovering anomalies between different observed application runs. For example, STAT - Stack Trace Analysis Tool, 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 stacktraces 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. 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 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 in 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. Sampling profilers usually have less overhead compared to instrumentation profilers. These profiles are good to collect for example time how long a method run, caller of the method or for example the complete stacktrace. However they are not able to collect any application specific information.

- **Instrumentation Profilers**
  This can be solved by instrumentation profilers. These 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 ar recorded. Compared to sampling profilers, instrumentation profilers usually have slightly worse performance.

However, 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 done 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 which are able to collect traces from both system and JVM profilers are usually called mixed-mode profilers. JDK8u60 comes with the solution in a form of extra JVM argument *-XX:+PreserveFramePointer* Mix. Operating system is usually using this field to point to the most recent call on the stack frame and system profilers make uses 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 we are able to collect stack traced leading to:

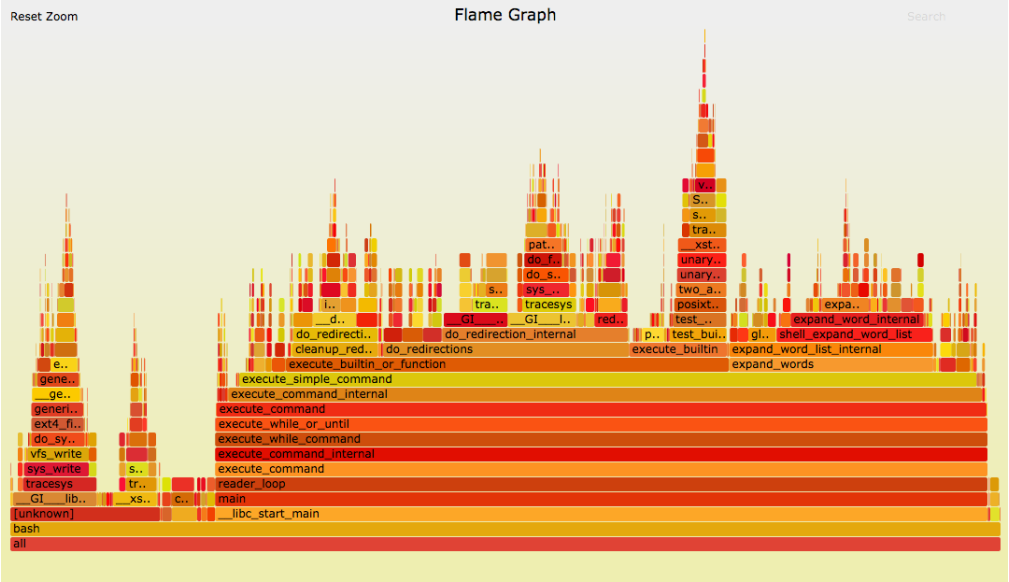

Figure 2.3: Flame Graph example

- **Page Faults** - these traces are useful to show what JVM code triggered main memory to grow.
- **Context Switches** - these traces are used to determine code paths which often lead to CPU switches.
- **Disk I/O Requests** - these traces show code paths leading to IO operations such as blocking disk seek operation.
- **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** - these traces show code paths leading to cache misses. This information can be used to optimize the Java code to make better use of the existing cache hierarchy.

All the information bellow can be described on a special chart called Flame charts.

Flame Graphs

Flame Graphs are special graphs introduced 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 performance in more comfortable way.

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 doesn't represent passage of time. The function calls are usually sorted alphabetically.
- The width of each box represents the time how long the function was 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.

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.
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 charts because it's one of the future plans to add support for flame charts into to monitored tools developed by this thesis. For more information about the flame charts please visit the Brendan Gregg's blog.

2.5 Byte Code Manipulation Libraries

The thesis highly depends on the Java bytecode instrumentation and this section gives overview of four bytecode manipulation libraries considered to be used at the thesis: Javassist, Byte Buddy, CGlib and. Since it's a core feature of the whole platform and affects both the performance and the usability of the whole platform, 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 Analysis section.

2.5.1 ASM

ASM is a low-level high-performance Java bytecode manipulation framework. It can be used to dynamically create new classes or redefined 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 operates on event-driven model as it makes use of Visitor design pattern 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 is 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 Java Strings. The disadvantage of this approach is that the 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. 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.

2.5.3 CGLib

CGLib as another byte-code manupulation library built on top of ASM. The main concepts are build 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 the result of previous methods or fields as we define. However CGLib lacks comprehensive documentation making harder to even understand the basics.

2.5.4 Byte Buddy

Byte Buddy 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. Also, classes created or instrumented by Byte Buddy does not depend on the Byte Buddy framework. Despite it's high-level approach, it still offers great performance and is used at frameworks such as Mockito or Hibernate. 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 it's super class. In order to intercept existing methods (change their behavior and return value), the method to be intercepted has to be identified using so-called `ElementMatchers`. 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 specifying the class name or using more complex structures. For example, the element marchers may be used to only consider all classes A extending class B whilst implementing interface C at the same type.

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 specified in the transformer using the element matchers. In more detail, `Transformer` interface has a method `transform` which has `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";
        }
}
```

Le'ts 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 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 needs to be initialized, which is in this case done in the static initializer. The initialization of interceptors is done using special helper class called `LoadedTypeInitializer`.

There are multiple ways how this behavior can be changed:

1. In Byte Buddy, the initialization strategy can be modified accordingly to the specific needs. No-op strategy can be used and `LoadedTypeInitializer` can be read right before the class is about to be instrumented. The initialization of the interceptor field can be handled manually later using observed initializer or we can even serialize the initializer together with the `Qux Interceptor` class, send them to different JVM where the instrumentation should take place and manually initialize the the interceptor field.

2. Instead of referring to `Qux` as a instance, it can delegated to as instance of `Qux` class. In this case the interception is performed via static methods and no intializers are required to be available, however the interceptor class still needs to be known at run-time.

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

Advice API

Advices are 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 original class's bytecode 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.

2.6 Communication Middleware

This thesis consist of several parts written in different languages which need be able to communicate. In order to achieve communication in such an 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 pros and cos. It give 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 platform.

2.6.2 ZeroMQ

ZeroMq 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 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. The library also allows the user to create typologies 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 stream of bytes.
- Automatic message queuing. The internals take 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.
- Language mappings to different languages.
- Ability to create different topologies. Example of a topology can be that one socket can be connected to multiple endpoints.
- Automatic TCP re-connection
- Zero-copy

Zero-copy in ZeroMQ

The library tries to apply 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.

Example of data copying can be transferring data from memory to network interface or from user application to 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 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 big performance benefit.

2.6.3 NanoMsg

NanoMsg [http://nanomsg.org/documentation-zeromq.html] is a socket library shadowing the differences between the underlying operation systems. It offers several communication patterns, is implemented on C and does not have any other dependencies. Generally, it offers very similar features to ZeroMQ since it's heavily based on it.

Unlike ZeroMQ, Nanomsg matches the full POSIX 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.

As in ZeroMQ, NanoMsq supports the following transport mechanisms:

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

- **IPC**
  Inter processes 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 just 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 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 reliable manner to a single recipient in a reachable network. The address in format `tcp://interface:port` needs to used when connecting to a node and when binding a node to specific address, the address in format `tcp://*:port` needs be used.

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

C++11 Mapping

Nanomsgxx [https://github.com/achille-roussel/nanomsgxx] is a C++11 mapping for Nanomsg library. It is a small layer build 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 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`

Java Mapping

Several Java bindings of Nanomsg library exists, but just jnanomsg library [http://niwinz.github
is described here. This language binding is build on top of JNA (Java Native Access) 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. Important Java concepts and classes relevant to the thesis are described in the following few sections.

2.7.1 JVMTI

The JVM Tool Interface [https://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html] is an interface used by development and monitoring tools for communication with JVM. It allows the user to monitor and control the 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 happening inside JVM and can react upon them. Agents run in the same process as the application itself which reduces the delay of 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 2 modes how an agent can can be started. It can be 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 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 library is specified and the library is loaded from there.

In the **Live** phase, the agent is dynamically attached to running application. This approach is more flexible since it is not required to to specify the agent library to 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 JVMTI please visit the official documentation. The following sections try to very briefly describe the important parts of JVMTI relevant to the thesis.

JVMTI Agent Initialization

When client is started, the method `Agent_OnLoad(JavaVM *jvm, char *options, void *reserved)` is called. This method should usually contain the agent custom initialization. Usually the agent initialization consist 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 depends 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, the events has 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` and `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. http://www.oracle.com/technetwork/articles/java/jvmpit 138768.html

JVMTI basic callbacks

As mentioned above, there are several events sent from the observed application. When instrumenting the applications code, the following are the most important events to record:

- `cbClassLoad` - triggered when class has been loaded by target JVM
- `cbClassPrepare` - triggered when 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 virtual machine obtains class file data but before the class is loaded. Usually, class instrumentation is based on this hook since the callback contains field for the to changed byte-code passed for further loading.
- `callbackVMInit` - triggered when virtual machine is initialized.
- `callbackVMDeath` - triggered when 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 which allows 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 instance of class `JNIEnv`. This environment keeps the connection to the 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 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 via `jobject` 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 is used exactly for this purpose. The following table is based on http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html. 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

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 elements in 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,
```

```
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 use later for the invocation itself. From performance reasons, it's good practice to cache the references to Java methods or objects which are accessed from JNI often, since getting 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 relevant to the thesis. It briefly describes the class loading process for dynamically loaded classes. This is followed by explanation of two important class loaders relevant to the thesis and lastly, `ServiceLoader` class is shortly described.

Class Loading Process

Java allows program to load classes dynamically at run-time. This is achieved by a 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 3 sub-phases:
   (a) **Verification** - Ensure that type in the binary format is correct and respects JVM restrictions.
   (b) **Preparation** - This phase consist of allocation the memory for fields inside the loaded type.
   (c) **Resolution** - This phase is optional (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 lazy way and delay resolution for the time when the type is accessed for the first time. Constant pool contains all references to variables and methods found during the compilation time at he class file.
3. **Linking Phase** - class variables are initialized to initial values

Relevant Class Loaders

There are several class loaders used natively in Java. This section however describes only two which are referenced later in the thesis.

- **Bootstrap class loader**
  This class loader is used to load system classes. When using 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 Sun JVM as the effect of a mechanism called *inflation*. Usually reflective access to method or fields is initially performed via JNI calls. When Sun JVM determines that there is a repetition in calling the same method or the same field via JNI (reflection), it creates synthetic class (classes created dynamically at run-time), 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. Service provider is an implementation of a service which is usually defined as 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 (interface or abstract class). The available service providers have to be defined in the META-INF folder of the application's jar distribution. For example, image there is a service A and two implementations, `Impl1` and `Impl2`. In that case META-INF folder should contain text file named A containing lines:

```
Impl1
Impl1
```

Service loaders can therefore be used to extend the application capabilities without changing the source code. When a new implementation of the 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 defined service providers defined earlier.

2.8 Logging Libraries

One of the key aspects of the developed platform is low-overhead. Logging can have 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 wrong configuration. Since one of the thesis's requirements is low-overhead, the selection of logging library is important for the performance of the thesis as well.

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

2.9 Docker

Docker is an open source project used to pack, ship and run any application as a lightweight container [citate]. It is used to package applications in prepared envi-

ronments 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.

3. Analysis

This chapter contains discussion of different approaches for meeting the desired requirements. If also provides the arguments for the selection of some specific libraries later used in the thesis. It starts with a section summarizing the limitations of the similar monitoring solutions and is followed by several sections where each of these sections is dedicated for a single requirement. The requirement sections discuss in more detail what solution is the best for meeting the desired requirement. This chapter ends by a summary of the wanted and unwanted features in more detail compared to the few basic requirements.

3.1 Limitations of Similar Solutions

This thesis tries to overcome some of the limitations of the relevant monitoring tools and give the users the alternative solution. One goal of this tool is to be an open-source solution which is in contrast to the proprietorial Google Dapper. Google Dapper is also targeted to only specific type of applications sharing the same code structure and libraries used at Google and therefore there is a certain lack of the universality.

OpenZipkin 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 backed. These libraries try to minimize the amount of code changes in application sources, however the user is still required to change the application source code to add custom annotations or change the default tracing mechanism. The similar holds for very similar tool developed by Cloudera, HTrace. These tools also perform the instrumentation in the same JVM (Java Virtual Machine) where the application is running which can have performance impact on the application itself.

The tool proposed by this thesis tries 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 mentioned platforms.

3.2 Small Footprint and High Performance

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. One of the thesis requirements is to have a minimal footprint on the monitored application.

Since the instrumentation is a core functionality of the system it directly affects the performance of the application. Different instrumentation methods are described 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 - using the native or Java agent and 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 the 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. Usually, the programmer extends and creates custom class file transformers and the agent internals take 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 negative impact on the application itself. Also the objects created from the agent are put on the application's heap and therefore consume the applications' memory. 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 and it is important to note that Java system classes can not be instrumented when using this method.

3.2.2 Native Agent

Native Agent 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 non-Java language, 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 new instance of specific class or switching threads. The native agent is running as part of the Java process and therefore any resource-demanding computation can have negative performance impact on the application's performance as well as in the Java Agent method. However objects created in the native agent are not subject to garbage collection and are not created on the heap except when 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, the is a lack of stable instrumentation libraries written in C++ or C which could be used inside the agent. The developer of the native agent has therefore write all the required methods for extracting the relevant parts of the bytecode and the instrumentation itself.

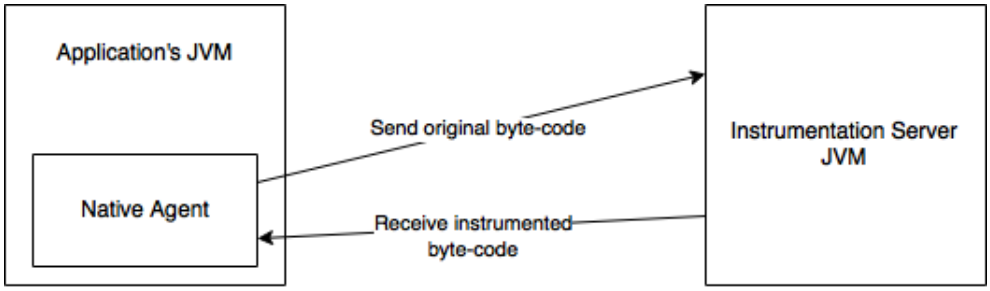

Figure 3.1: Sketch of the chosen approach.

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 high-level programming language to write the instrumentation code. For these reasons the compromise between the proposed solution has been chosen together with introduction of special process used specifically for the instrumentation. The solution can be seen on the figure 3.1.

In more detail, the native agent is used to to communicate with the application being monitored. When a class needs to be instrumented, the native agent sends the class's bytecode to 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 to collect important internal JVM information, send classes for instrumentation and receives the instrumented classes back. The instrumentation does not happen in the same JVM as the monitored application which allows the tool to have 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 operates on Java-language level and thus 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 really good performance results based on the official benchmarks available on the library's page. Compared to Javassist, the code is not written inside Java Strings, which means that the instrumentation code can be validated in today's IDE during compilation time and bugs in the instrumentation code can be found easier. The API of the library is well-documented, compared to CGLib, and the library is under active development. Byte Buddy is also highly configurable library which was also the significant reason for choosing it as the tool for instrumenting classes inside different JVM then 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 the Instrumentation section of Design chapter.

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 significant amount of time and also, it was not aimed of the thesis to create such a library. The performance impact on the application's was also 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, 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 Mor.

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 NanoMsg has been chosen. It hides the platform specific aspects of sockets comparing to Raw Sockets approach. It has also several performance benefits and general improvements over the well-known ZeroMq library such as the better threading and more sophisticated implementation of the zero-copy technique. The mappings of this library for Java and C++ languages mentioned in the Nanomsg section were also perfect fit into the tool.

3.3 Application-Level Transparency and Universality

On 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 the compromise between these two has been found.

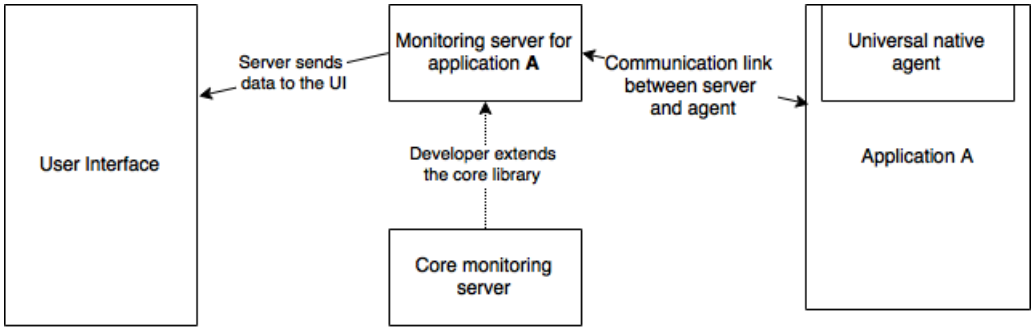

Figure 3.2: .

The rejected approach was to create an universal monitoring tool similar to Google Dapper, but be allowed to monitor mostly shared aspects of distributed applications. Whilst this would give the user great flexibility, universality and would also ensure that the users don't need to extend the library, it was decided as not a feasible task. Every platform or application is different in it's architecture or in the way how it communicates and therefore identifying the shared parts between all distributed applications is extremely challenging task. Also, such instrumentation tool could only instrument very basic information about Java-based programs.

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 have to have understanding of the application internals.

The advantage of this approach is that only core instrumentation library needs to be created which 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 as well. By extending the library, the developer creates an application-specific instrumentation server which knows the parts of the original application to be instrumented. The application-specific instrumentation server communicates with the native agent which is universal to all Java applications since it does not contain any state. The figure 3.2 represents the application's architecture.

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 goal of this thesis to ensure that the developer responsible for extending the core monitoring tool needs to write as less 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 were used for the instrumentation in the instrumentation server. 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 the code of it.

3.5 Easiness of Deployment

In order to ensure the easiness of deployment the number of artifacts used in the thesis 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 the server location 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 thesis to allow the user to replace some of the default application modules of the whole platform by specific custom modules without the need of rewrite 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 thesis, the users have the possibility tu plug-in custom user interface or more advanced data collectors. The modules have to met some specific criteria in order to replace the default implementations, however the core implementation is let up on the user.

The main reason for this solution is that a lot of monitoring solutions already exist and users may used to specific user interfaces or are using some platform with already set-up data collecting. The thesis tries to support these use-cases and tries to minimize the changes 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 UI was selected as the default user interface for the thesis. The main reasons for its selection were the simplicity of the interface and ease of use . It also fulfills the visualization requirements of the thesis 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 plug-in 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 instrumentation server for the instrumentation and specifying the instrumentation points.
- NanoMsg library to be used as the communication layer between the native agent and 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:

- Create a custom user interface.
- Create an universal instrumentation tool which would not require the developer to specify the instrumentation points and specify start and end of spans.
- Implement bytecode instrumentation in C++.

- Instrument the Java bytecode in the same machine as the running application.

4. Design

This chapter describes design of the whole platform in details, however implementation specifics of some parts of the tool are described in the following chapter. This chapter starts with an high-level overview of the complete platform and interactions between the parts and follows by a simple use-case to give the reader idea how this tool is mean to be used.

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

4.1 Overview

Main purpose of this tool is to collect distributed traces. In order to achieve that, the thesis is based on the concept of Spans. Spans are used to denote some specific part of the communication between the communicating nodes and are the important elements for building the whole trace trees. Trace trees consists of several spans and represent the complete task or communication where a span inside the trace represents usually a few remote procedure calls between two neighboring nodes. The node initiating the trace creates so called parent span and new calls started within this span create new nested spans. Created spans can be processed using different span savers and can be send to the user interface using various data collectors.

Spans are processed using Span Savers. Span savers are used to save spans in desired format on disk or on the network for further data collection and collected data are used for Spans visualizations in the user interface. The user interface receives the spans from the span savers or data collectors and present 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 instrumentation server is then used for instrumenting the classes of the original application at which information about spans needs to be preserved. In order to obtain the class files, the native agent runs as part of the monitored application and sends the instrumentation server the desired classes. The native agent is the core part of the whole platform. It is attached to the monitored application and additionally to the instrumentation, it is used to obtain various low-level information from the application.

The tool therefore consists of three main parts:

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

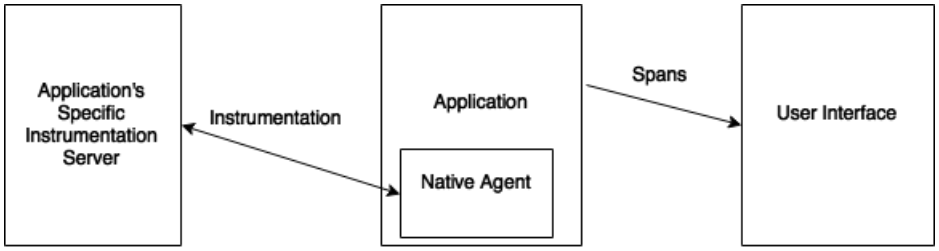

Figure 4.1: Basic relationship between the major components. Instrumentation server communicates with the native again mainly in order to instrument classes. The Application's communicates with the user interface by sending the spans. Spans can be send either via data collection agent or one of the default span savers explained later in this chapter.

  - Is also a base library for custom application instrumentation server
  - Can contain implementations of custom span savers

- User Interface

The graph 4.1 denotes the basic relationships between the major parts. Each part is explained in more detail later in this chapter.

4.2 Example Use Case

In order to demonstrate the use case for which this architecture is a good fit, a simple architecture is shown in this section. The example consists of three modules. The client module used for submitting a task, the execution module and the module used for exporting the data. These modules can be represented as a different threads in a single applications or as separated application nodes. In this example, the user always passes a task to the client module. The client module performs some pre-processing and sends the task to the computation module. This module performs the computation and sends the data to the exporter module which saves the data on disk and informs the client of the task completion. The architecture of the example can be seen on the figure 4.2.

The goal is to be able to discover how long the transfers between different modules last and how long the processing on each module last. 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.

- **The client:**

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

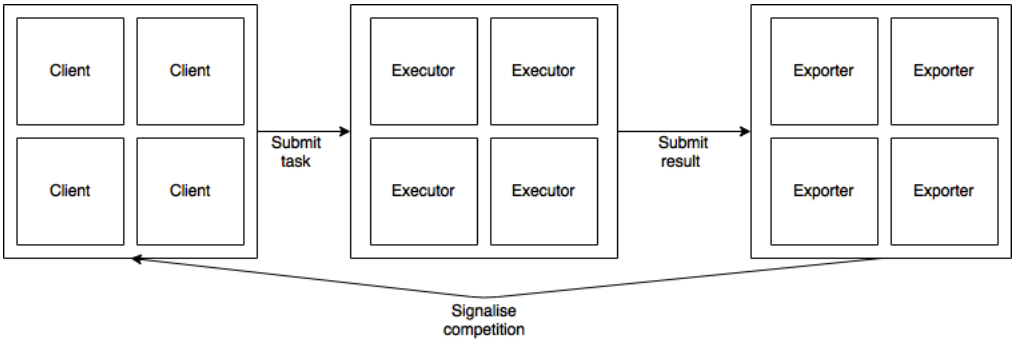

Figure 4.2: Example architecture.

```
    ...
}
```

- **The executor:**

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

- **The exporter:**

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

In order to be able to collect this type of information and be able to reason about the relationship between the nodes, these methods needs to be instrumented. The instrumented code should look as in the schematic code bellow. For achieving this, the developer is supposed to extend the instrumentation server which acts as the base library and provides the developer with several helper methods used to specify the instrumentation points for their applications.

- **The instrumented client:**

```
public acceptTask(Task task){
    Inst.openSpan(task)
    Inst.currentSpan().addAnnotation("tskReceived", timestamp)
    ...
    preprocessTask(task)
    Inst.currentSpan().addAnnotation("tskPreprocessed", timestamp)
    ...
    sendTaskForComputation(task)
    ...
    waitForExporterToFinish()
```

```
  ...
  Inst.closeCurrentSpan()
}
```

The `Inst.openSpan` opens a new span, attaches it to the current thread and also attaches the span information to the task object which is passed around the network. In this case, the method actually creates a new trace tree since it's always start of the monitored communication. The `addAnnotation` is used to add application specific information to the monitored span. The `closeCurrentSpan` method is used to close the current span and export the content using the provided span saver. In the default case, the data are sent to the user interface directly.

- **The instrumented executor:**

```
public execute(Task task){
  Inst.openSpan(task)
  TaskResult result = executeTask(task)
  Inst.injectCurrentSpan(result)
  ...
  sendResultForExport(result)
  Inst.closeCurrentSpan()
}
```

In this case, the `openSpan` method gets the span information from the task object and attaches it to the current thread. The newly introduced method `injectCurrentSpan` is used to inject span information from the current thread to a new object.

- **The instrumented exporter:**

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

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

The developer should extend the base instrumentation server by steps how to instrument the classes in order to have a similar format as above. The extended instrumentation server is described on the figure 4.3.

The extended instrumentation server is then 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 distributed application prior it's start and the path to the extended instrumentation server JAR needs to be set as an mandatory argument. The default span sever is used in this case and the collected spans are send right to the Zipkin UI. The default endpoint for the user interface is used when not defined as an argument to the native agent.

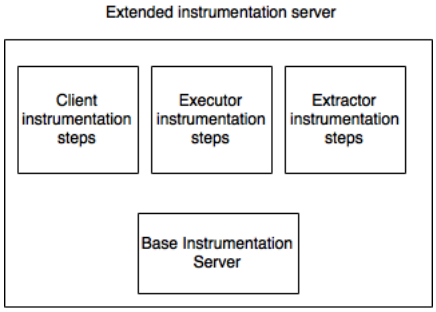

Figure 4.3: Structure of extended instrumentation server JAR.

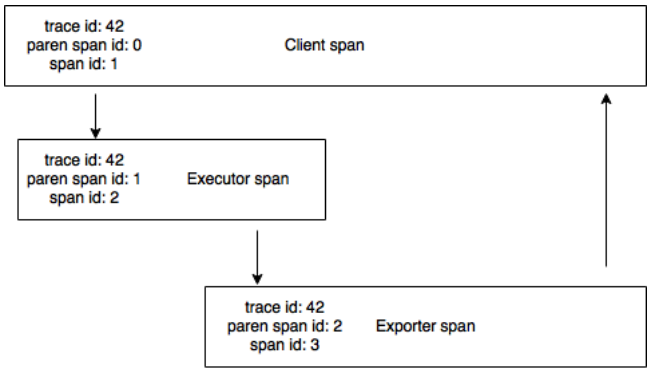

Figure 4.4: Example trace in case of the example application.

A single collected trace from this application should look as shown on the figure 4.4.

Therefore, it can be seen that the only part the developer needs to work with is the extension of the instrumentation server to specify the custom instrumentation points, otherwise the rest of the 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 special classes which instances are injected to instrumented application's classes to keep track of the communication and the state between the nodes in 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 savers and can be sent to the user interface using various data collectors.

Spans has several mandatory and optional fields. The mandatory fields are trace id, span id and parent span id. Trace id represents one complete distributed call among all interacting nodes on the cluster. This field is attached automat-

ically when a new root span is created. Root span is a first span created inside a trace. The root span does not have parent id field set up and therefore the user interface back-end can distinguish between regular spans and root spans and therefore can identify the start of the whole trace. Parent id of a span is always id of span from which the span received a request to perform some task. 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 several threads as separated spans within a single node.

Span have several additional fields which which are later used in the user interface. The fields are:

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

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

- **ip** - ip of the node on which this event was recorded
- **port** - port on which the service which recorded the span is running.
- **service name** - service name which is used to group different traces by names and can be later used to filter traces using this field in the user interface.

4.3.1 Span IDs

It is also important to mention that span id and parent span id are created randomly. This is done in order to allow parallel spans in the same control flow without overlapping as can be seen on the figure 4.5.

If the ids were not random, the parallel spans would be creating spans with ids in the same linear sequence and therefore these spans would be overlapping as can be seen on the figure 4.6

The following sections contain information about how spans are exported for external communication with the user interface and also how spans are created using `TraceContext` and `TraceContextManager` classes.

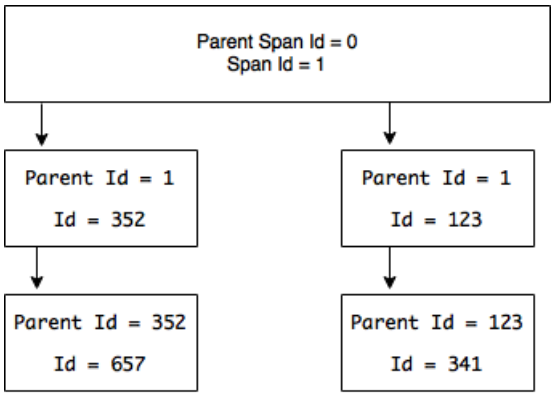

Figure 4.5: Generating span ids randomly ensures that they don't overlap when they are created in parallel.

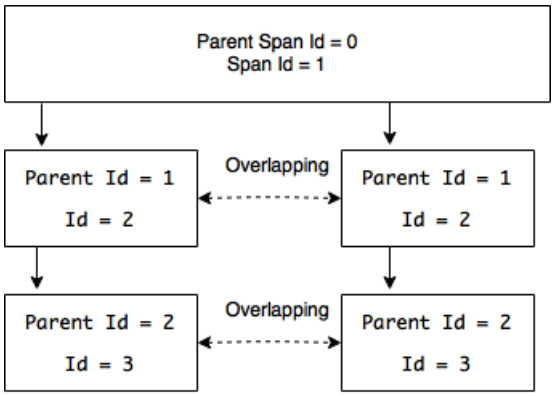

Figure 4.6: Generating span with the same linear sequence leads to span overlapping.

4.3.2 Span Savers & Data Collectors

Spans are exported from the application using the span savers. Span saver for the application may be selected via one of the native agent configuration properties. It is important to mention that all spans in the application are using the same, because span saver is static field of the `Span` class. Span savers need to be available to the monitored application and therfore are brought to the application from the instrumentation server via the native agent during the agent initialization phase.

The thesis provides two default implementation of span savers but also allows the developer to create new span savers. New span savers 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 the data from the specified place and store them in central data storage available by all application's node. This thesis does not implement data collection service as many services exists for this purpose.

Data collected within spans are internally represented in JSON format understandable by the Zipkin user interface. This is also the reason why the thesis contains support for working with JSON data and it is explained in more detail later in the Implementation chapter. This format can be again changed by the custom span saver.

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

- `saveSpan`. This method is used for saving the span. Custom span saver implementation may save the data on local disk or send over network. The destination is not limited by the code. Internally, this method is called asynchronously in a separated thread to allow asynchronous span processing which has a performance benefit.

- `parseAndSetArgs`. The instrumentation agent accepts also special argument which contains arguments for the defined span saver. Each span saver is responsible for parsing the span saver arguments.

As mentioned above, the tool provides two default simple span savers:

- `DirectZipkinSaver` - The default span saver sends the collected span asynchronously to the user interface right away without storing the data on disk to be collected by an external data collector. In this case the functionality of the span saver and the data collector are handled by this single saver. This span saver should be used for demonstration purposes only since it could overload the user interface or network when processing and sending high amount of collected spans in a short amount of time to the user interface as the UI is not prepared to store large amounts of data in the memory.

  This saver accepts a single argument which is ip and port of the Zipkin UI service. The diagram 4.7 shows how the Zipkin span saver is used.

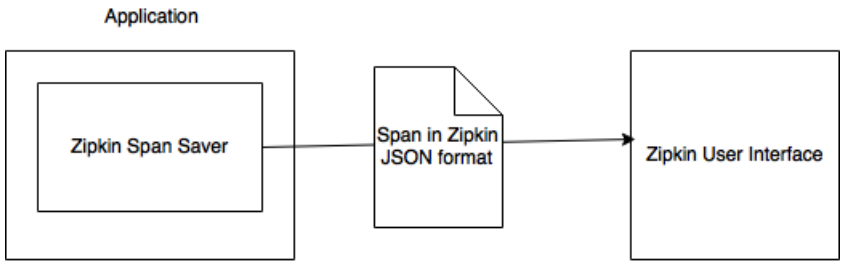

Figure 4.7: Using the Zipkin span saver to save spans directly to Zipkin user interface without the data collector agent.

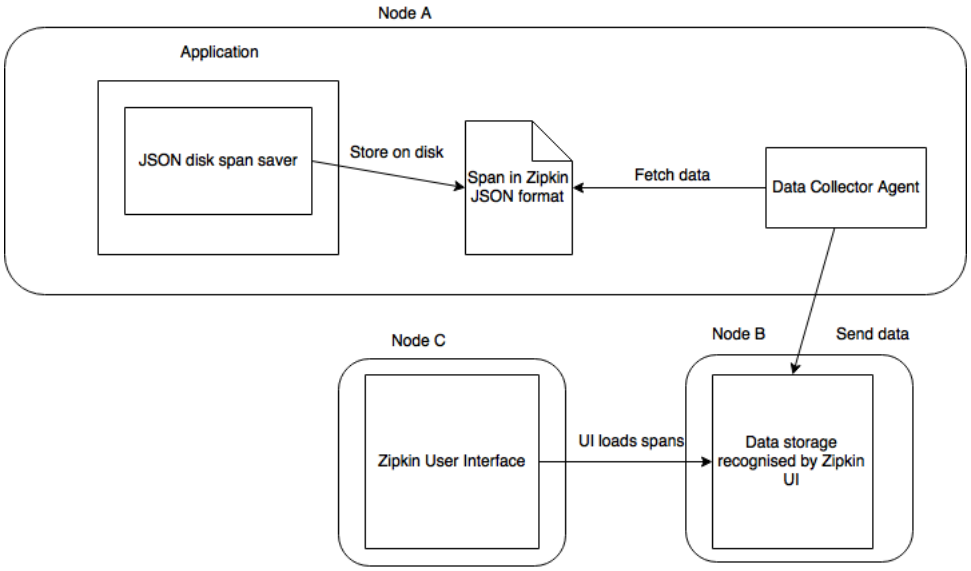

Figure 4.8: Using the JSON disk saver together with the data collection service but Zipkin user interface .

- `JSONDiskSaver` - The second available span saver saves the collected data asynchronously on disk in the format known to Zipkin UI for future collection to the Zipkin user interface by custom data collector. 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 saver accepts single argument which is a directory where collected spans are saved. The diagram 4.8 shows how JSON disk span saver is used.

Additionally, the diagram 4.9 shows how a custom span saver may be used.

In order to give the developer the flexibility to add new savers without changing the internals, the span savers 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 `SpanSaver` abstract class. The reason why the classes needs to be discovered is explained in the following Native Agent section.

To make the developer life easier, the **AutoService** library from *https://github.com/google/* is used. Instead of manually registering the implemented span savers into META-INF directory, they can be annotated in the code using the `AutoService` annotation with a single argument specifying the abstract parent, in this case `SpanSaver`.

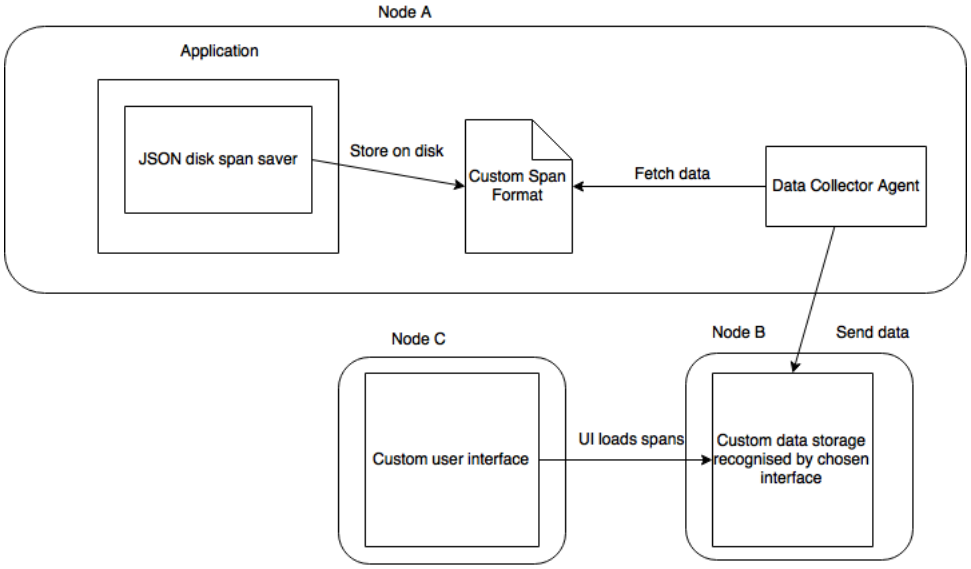

Figure 4.9: Using custom span saver together with the data collection service and custom user interface.

The library takes care of registering the classes automatically in the desired folder in correct format so the human error is minimized.

4.3.3 Trace Context

Trace context is a class used for storing the information about the current span and also for creating new spans and closing current spans. Trace context is always attached to a specific 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. Singleton instance of class `TraceContextManager` is used for attaching the threads to the trace contexts and vice-versa. It has a few methods allowing the developer to attach trace context to a specific thread and also to get trace context which is attached to a current thread.

Each trace represented by a trace context is uniquely identified by `Universally unique identifier (UUID)` of type one is created. This UUID version combines 48-bit MAC address of the current device with the current timestamp. This way it is ensured that two traces created at the same time on different nodes can't have the same identifier. The identifiers are created using a C++ library called sole (https://github.com/r-lyeh/sole) in the native agent and are made available to the Java code via a published native method.

The trace contact has method `sopenNestedSpan` and `closeCurrentSpan`. The first method is used to create a new nested span and set the newly created span as the current one. Nested span is a span which sets its parent id to the current span. A root span is created in case when no current span exists. The second method is used to close the current span. Closing the span triggers the span saving operation on the used span saver and the parent span becomes the current span.

The diagram 4.10 shows how spans are created, closed and saved. in this

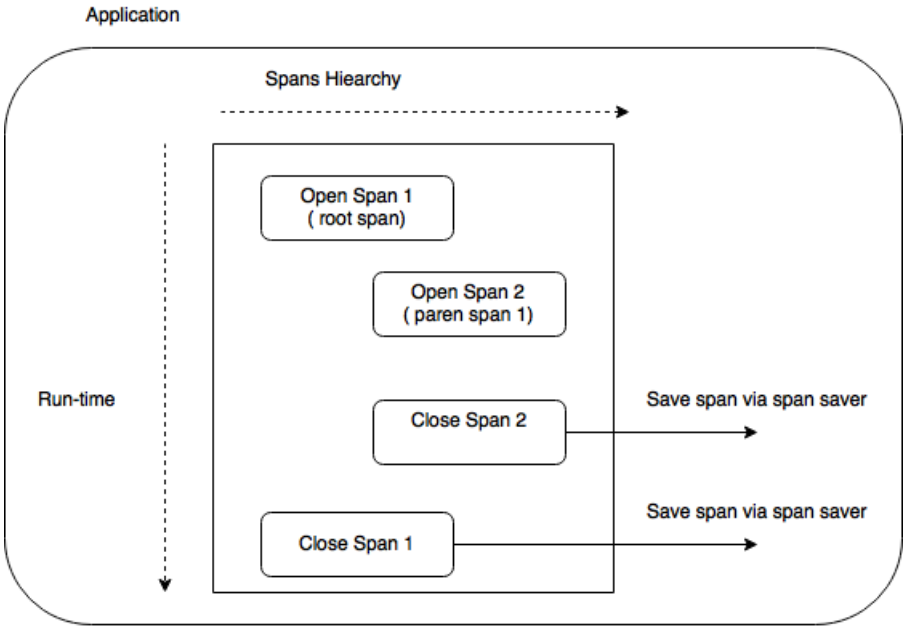

Figure 4.10: Creating and closing spans.

chapter.

4.3.4 Transferring Span Information

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

Usually when transferring the trace context between the threads on the same node the copies of the trace context have to be used. When transferring the trace context between the different nodes this is not an issue.

The method the developer can used for obtaining and attaching the trace context are:

- `createAndAttachTo(carrier)` - this method creates the trace context and attaches it to the carrier object and current thread. This method is usually used just once at at the time of trace creation.
- `getAndAttachFrom(carrier)` - this method obtains the trace context from the carrier object and attaches it to the current thread.
- `getWithoutAttachFrom` - this method obtains the trace context from the carrier object without attaching it to the current thread.
- `getCopyWithoutAttachFrom` - this method obtains a deep copy of trace context from the carrier object without attaching it to the current thread. It is used in cases where child spans are processes for example 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 object. This

is done in order to avoid different traces between the spans when processing by multiple threads.

- `attachOnObject(carrier)` - this method attaches the current trace context on the carrier object without creating it and attaching it to the current thread.
- `attachOnThread` - this method attaches the current trace context to the current thread.

There are also different variants how spans can be closed. Span can either closed by the same node or thread who created it. This is a simple situation and usually the methods attaching the trace context to the current thread are sufficient. It is also however possible that span can be closed by different threads or nodes then where they were initially created. In this case the copy of a trace context is usually created and attached to an object which acts as an carrier of the trace information.

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 agent task is to check whether a class is required to be instrumented and if yes, send the class for the instrumentation to the instrumentation server and wait for the instrumented code.

The native agent consist of several parts. The most important parts 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. Byte code parsing is a technical task described in the following fImplementation chapter.
- **InstrumentorAPI**.
  The `InstrumentorApi` class provides several methods which are used to communicate with the instrumentation server JVM. All the queries to the server are done via 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 later used from the Java code without worrying of the low-level implementation.

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

4.4.1 Agent Initialization

The agent is initialized via the same phases as described in the JVMTI Agent Initialization section of the Background chapter. 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 application's nodes. In the local instrumentation mode, the instrumentation 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 `SpanSaver` abstract class. These classes are used in the instrumented code and therefore have to be available to 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 within the instrumentation server. Therefore, the instrumentation 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 class-path 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 allow the developer to pass a new instrumented bytecode as the output parameter. The process of instrumentation is described here, however the technical details are described in the following chapter.

The process consist of several stages:

1. Enter the critical section. It can happen that the class file load hook is triggered multiple times and in order to not confuse the instrumentation server, the lock has to be acquired before the instrumentation of a class starts.

2. Firstly, the check whether the virtual machine is started is done. If the virtual machine is started and initialized, the instrumentation continues,

otherwise the instrumentation for currently loaded class is skipped without asking the instrumentation server since it the class is a system class and it is not desired to instrument Java system classes at this moment.

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

4. Discover the class-loader used for loading the class

5. Parse the name of the class being loaded. Even though the callback provides input parameter which should contain the name of loaded class, at 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.

6. 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 `Boostrap` class loader and in case of Sun JVM, from `sun.reflect.DelegatingClassl` are not supposed to be instrumented. The `Boostrap` class loader is used to load system class 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. Example of these classes are the classes loaded during initialization phase from the instrumentation server and the auxiliary classes generated by the Byte Buddy framework. Auxiliary classes are small helper classes Byte Buddy is using for instance for accessing the super class of the currently instrumented class. Therefore the instrumentation continues only If the class is not ignored and not loaded by ignored class loader.

7. The instrumentation server is asked whether it already contains the loaded class or not and also if the class should be instrumented. The agent does not know which classes are to be instrumented and it therefore needs to query the server. The classes for instrumentation are marked by developer when extending the instrumentation server library using simple Byte Buddy API.

   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 to the instrumentation. This step is repeated throughout the dependency scan recurrently until the loaded class does not have any other dependencies or until all dependencies is 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.

8. 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 send from the server.

9. Exit the critical section.

Even though the class is fully instrumented and the instrumented bytecode is available to the agent, the process is not completely done. The instrumentation library used at the instrumentation sever (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 - class defining the instrumentation code. This field is automatically set by Byte Buddy framework in most of the cases, but since in case of this thesis the instrumentation is done on different JVM then where the code is actually running, it needs to be handled explicitly. In order to set this field by corresponding `Initializer` class, both the initializer class and interceptor class need to be available on the agent. The instrumentation server sends the initializer class together with the instance of interceptor during the instrumentation of the class and the agent registers the interceptor and initializers with the instrumented class for later use since the static interceptor field can be set up when the class is used for the first time. The initializers are loaded during `Class Prepare` event. This event is triggered when the class is prepared but no code has been execute so for.

The callback for `Prepare` event is also used to register the native methods for the class being loaded. Registering the native method to the class makes it available from Java programming language.

Several technical difficulties had to be dealt with during the development. For example, cyclic dependencies when instrumenting the class had to be properly handled. Also ensuring that the dependencies for the instrumented class are also instrumented in the correct order has been a significant challenge. The different attempts for the solution and the final solution is described in the following chapter since it's highly implementation specific.

4.4.3 Instrumentation API

The Instrumentation API is used to communicate with the instrumentation server. It provides low-level methods for sending data in form of byte arrays or strings and the corresponding methods for receiving the data. On top of these method several methods are built to make the communication easier. The most important methods are:

- `sendClassData` method sends bytecode to the instrumentation server.
- `isClassOnInstrumentor` method checks whether the bytecode for the given class is already on the instrumentation server or not.
- `instrument` method triggers the instrumentation and returns the instrumented bytecode.
- `loadInitializersFor` method is for loading the initializers for specific class.
- `loadDependencies` method is used to load all dependent classes and upload them on the instrumentation server. The dependency is uploaded only in case it's not already available on the instrumentation server.
- `shouldContinue` method checks if the class on its input is allowed to be instrumented.

- `loadPrepClasses` method loads all dependent classes in the agent initialization phase.

4.4.4 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 sever 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 case of local instrumentation server mode.
- **connection_str** - specifies the type of connection between 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 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 case of Unix. However this value is set automatically at run-time if not explicitly specified as an 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.
- **saver** - specifies the span saver type. The value can be either `directZipkin(ip:port)` where ip:port is address of the Zipkin UI interface or `disk(destination)` where destination sets the output directory for the captured spans. Custom span savers are supported as well. In that case the format of the value is a fully qualified name of the span saver with arguments in parenthesis, for example as `com.span.saver(arguments)`
- **config_file** - specifies path to a configuration file containing the agent configuration. It can contain all arguments mentioned above, each argument per one line of the configuration file.

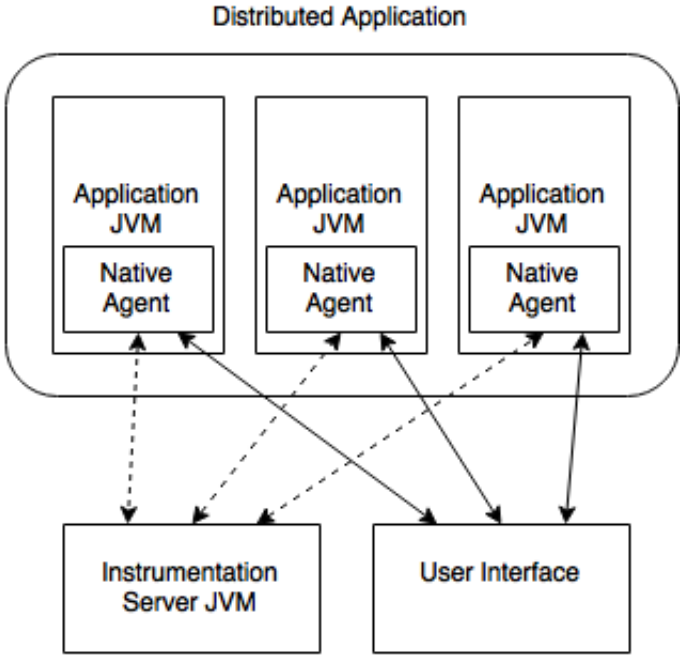

Figure 4.11: Architecture with shared instrumentation server. The dotted lines represents the communication between instrumentation server and the agent whilst the regular lines represents data collection from the agent to the UI

4.5 Instrumentation Server

The instrumentation server is responsible for instrumenting the bytecode received from the native agent in separated JVM and also acts as the base library for the instrumentation for specific applications. The developer extending the instrumentation server can use prepared method 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 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 application's instrumented classes.

The platform was designed to be configurable and deployment of instrumentation server is supported via two approaches. The instrumentation server can be either on the network available to all the application nodes and can be shared by all applications. This has the advantage of caching the instrumented classes. So when any class is instrumented for the first time, it is saved and the instrumentation is not performed for other nodes but the class is immediately sent. The disadvantage of this solution is higher latency between the agent and the instrumentation server since they are usually not on the same node. In this case the instrumentation server has to be manually started in advance. Architecture of this scenario is depicted on the diagram 4.11.

The other deployment method is that the instrumentation server runs on each

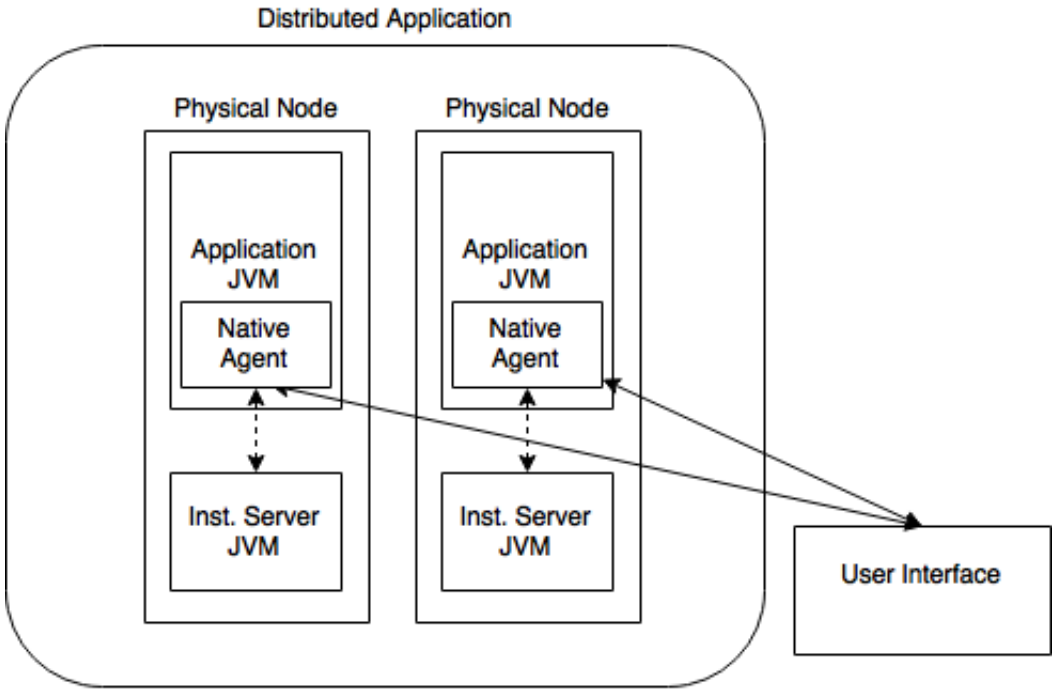

Figure 4.12: Architecture with separated instrumentation server. The meaning of the lines is the same as on the diagram above.

application node. This has the advantage of faster communication since 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 on the diagram 4.12.

Except from 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 for storing bytecode for a class on the server.
- Request for sending 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 they are just sub-parts of the communication initiated by one of these 4 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 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 or minimizing the communication when possible. The technical aspects of these issues and the optimizations mentioned above are described in the following sections.

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 client whether the class is already on the instrumentation server or not. The second stage is the instrumentation itself. The first stage is initiated by the agent. The server performs the check for class availability in 3 phases:

1. Check whether the instrumented bytecode for this class is available.
2. If not, check whether 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 application classes on the instrumentation server classpath for optimization reasons.

The server informs the agent if it does not have the bytecode for the class available and in that case the agent sends the class to the server. The server registers the received bytecode under the class name. The agent therefore 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 instrumented class is sent right away without instrumenting the class again. If the cache is empty, the class is instrumented and put into the cache.

The code instrumentation is handled by `CustomAgentBuilder` and `BaseAgentBuilder` classes. The instrumentation server expects instance of `CustomAgentbuilder` on the input of its `start` method. This is an abstract class containing single abstract method `createAgent(BaseAgentBuilder builder, String pathToGeneratedClasses)` where the builder is a wrapper around the Byte Buddy `AgentBuilder` class which is used to define the class transformers.

The developer 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, please read more about Byte Buddy in the Byte Buddy section. The server provides several helper methods for creating the transformers and interceptors which are less verbose then the standard Byte Buddy approaches.

Each created transformer has to have associated 'n interceptor which defines the code to be injected. Each interceptor implementation has to implement `Interceptor` interface. This is required for the server to be able to discover all interceptors at run-time without the need for changing the internals of the server. Each implementation of the interceptor needs to register itself in the META-INF directory of the generated JAR in the same way as the span savers mentioned in the previous section. Custom service loader is then used to locate all classes implementing the `Interceptor` interface.

Even though Byte Buddy takes care about the instrumentation, the `BaseAgentBuilder` class is internally properly configured so the instrumentation happens exactly as desired. The class implements four Byte Buddy listeners used for informing us about the instrumentation progress and allow us to react on the process of the instrumentation. The listeners are:

- `onTransformation` listener is called immediately before the class is instrumented. Implementation of the listener in the thesis also sends 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 instrumented. The class is not instrumented when the user 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 process completed. It is called after both of `onTransformation` and `onIgnored` listeners.

Byte buddy requires dependencies for the instrumented class to be available. They are needed because the instrumentation framework needs to know signature of all methods in several cases, for example when the method is overridden in the child class. The dependencies are all the classes specified in the class file such as type of the methods return value or arguments, super class or implemented interfaces. By default, Byte Buddy tries to find these dependencies using two classes - `LocationStrategy` and `PoolStrategy`. The first class is used to tell Byte Buddy where to look for the raw bytecode of dependent classes. The classes are loaded by context class loader by default, but since the classes are received over the network, custom `InstrumentorClassloader` class loader is used to handle the class loading. It is a simple class loader which keeps the cache of the classes received from the agent and when a request for instrumentation comes, instead of looking into the class files, it loads the data from the cache in the memory.

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

So in overall, 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.

4.5.2 Custom Service Loader

In order to allow the developer to extend the base instrumentation library service loaders for loading the extensions are used. The service loader is used used for two types:

- Custom span savers. Each span saver inherits from the abstract class `SpanSaver`.
- Custom Interceptors. Each interceptor implements the interface `Interceptor`.

The user can create custom span savers and interceptors by either inheriting the desired class or implementing the required interface and put the name of the class inside the text file in the META-INF directory in the JAR file. The text file has to have tha same name as the abstract class or the interface the implementation is for. For example, when 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's only required to obrain `Class` object representing the available implementation. Therefore a custom service loader was created for this purpose. This loader works in very similar way as the standard Java one, but instead of returning the instances of loaded services it just returns classes of available services.

4.5.3 JSON Generation

The data inside spans are internally stored as instances of `JSONValue` class since in order to support the communication with the default Zipkin UI they need to be exported as JSON. JSON is a lightweight format for exchanging data where the syntax is based on Javascript object notation.

The JSON handling is based on the https://github.com/ralfstx/minimal-json library, however custom simplified implementation was created which fits the theses requirements. Also the number of dependencies is lowered by this decision.

This JSON support is designed via several classes:

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

Each **JSONValue** can be printed as valid JSON string where the printing is driven by`JSONStringBuilder` class. This class is also responsible for escaping the characters according to JSON standards. The default printer prints the data without any formatting as one line, however `JSONPrettyStringBuilder` prints

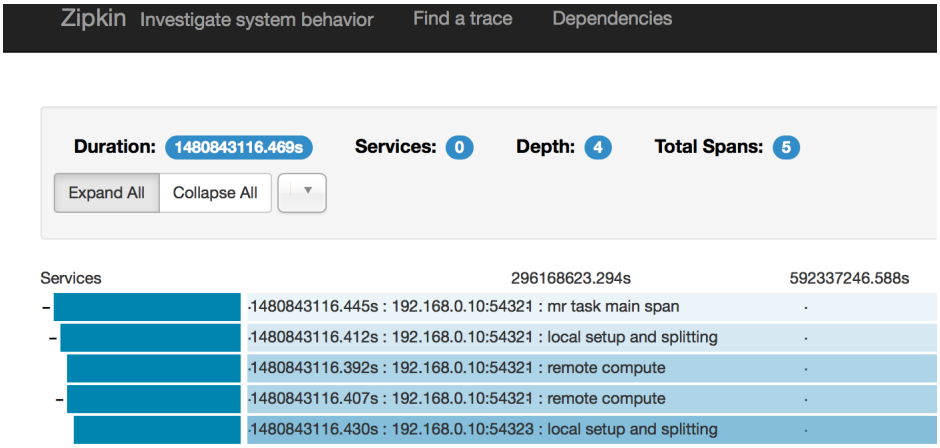

Figure 4.13: Example of Zipkin UI

the data in more human-readable format. The second printer is usually used for debugging purposes and the first one for real usage as the size of the data is smaller in this case.

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 single one and presents all annotations under the given span. This allow the 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 sending the data back on forth to send them as one single complete span.

The thesis is using Zipkin as default user interface. The default data 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 saver class. This section gives an overview of Zipkin user interface and describes the Zipkin data model

Each span in the UI is clickable and all the additional information can bee seen at that level. In this thesis the stack trace are also collected at each span for monitoring purposes. Example of such information screen can be seen on the figure 4.14.

4.6.1 Zipkin Data Model

Zipkin requires data to be sent in JSON format. Requests to UI are sent as JSON arrays where the array elements are the spans. Zipkin understands the following

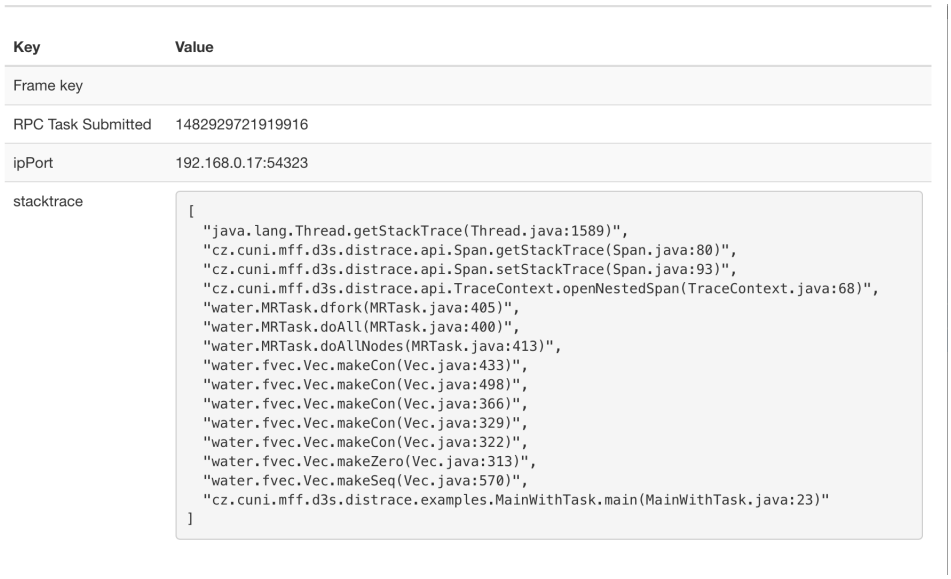

Figure 4.14: Example of the detail span information.

fields 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 UI supports span ids only to be 64-bit long.
- **parentId** - parent id of the current span.
- **timestamp** - the time when the span was created.
- **duration** - the duration of the span. It is the duration between the span creation and span closing.
- **annotations** - array containing standard Zipkin annotations. These annotations can be handled by user interface in specific way since the user interface understands the meaning of the content. The documentation specifies the following annotations:
  - **cr** : timestamp of client receiving the span
  - **cs** : timestamp of client sending the span
  - **sr** : timestamp of server receiving the span
  - **ss** : timestamp of server receiving 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* fields, the fields are of simple string or number type. Annotations are objects with the three fields - annotation value, annotation name and the endpoint. Endpoint is another object specifying the address and port at the code 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 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. It starts with explanation of the bytecode parsing and instrumentation at the native agent part of the complete tool. The following section covers relevant parts of the instrumentation server which are the instrumentation together with creation of transformers and estimators. This chapter ends with a brief explanation of how the spans are exported to the Zipkin user interface and also, how the spans can be exported to a custom data format.

5.1 Native Agent

The native agent consists of several interesting technical parts. This section covers the instrumentation itself and also explains considered approaches 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 explain as well.

In the following part, the internals of how JVM bytecode is parsed is explained.

5.1.1 Instrumentation

In general, the native agent does not perform the instrumentation but gets bytecode for the the required class, sends the bytecode to the instrumentation server and applies the instrumented bytecode after receiving it from the client.

The instrumentation server required all dependencies to be available for the currently instrumented class. This means that all other classes mentioned as part of method and field signatures, super classes or interfaces has to be available on the instrumentation server. To achieve this, two solutions have been tried but only the second solution shown to be feasible.

The first and unsuccessful solution was based on the fact that several on class file load hooks may be executed at several time in different threads. When the application loads a class, the class load hook event is triggered for it and its bytecode is made available. In this method, the new class file load hook event was artificially enforced via the `RetransformClasses` method. This method accepts 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 in this solution first. This also means that in this approach, the classes with cyclic dependencies are not supported. In order to instrumented such a class, all dependencies have to be instrumented first which is also the class itself.

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

The second and currently used solution is based on the fact that the Java class files may be accessed as a resource using the class loader of the class currently being load. Disadvantage of this solution that the developer may override the `getResourceAsStream` method on the custom class loader and not provide

access to he class files. This is a limitation of the thesis. However, when a such event happens, the instrumentation does not end with the exception but first, the attempt to load the class using a different class loader is done.

In this solution, the instrumentation server is first asked whether the current class should be instrumented based on the server's extensions. If the class is designed to be instrumented, its bytecode is send to the instrumentation server (only in case if the bytecode for the class is not already available). Then, dependencies are scanned via parsing the raw JVM bytecode which is explained in detail in the following section. Dependency loading is recursively called for each new dependent class until the class does not have any other dependencies or if all the dependencies are already uploaded to the instrumentation server. Once all dependencies for a class have been sent to the server, the instrumentation is invoked and the agent waits for the new bytecode.

Also several helper classes are sent back to the native agent at the stage where the class is checked whether it should be instrumented or not. The classes are auxiliary Byte Buddy classes and also instances of `LoadedTypeInitializer` class. The initializers are sent as serialized instances and therefore their defining class has to be available in the application. This is achieved in the agent initialization phase where several required classes are sent to the application from the server. The instances are saved to a map which maps the initializer name to its serialized representation. This initializer is later using during the class preparation phase to set the static interceptor field of the instrumented class as mentioned in the previous sections.

The auxiliary classes are classes created at run-time during the instrumentation on the server and have to be available on the applications machine as well. This is achieved by loading the bytecode for the auxiliary class, saving the the class as a java class file on the disk and making it available by adding the class on the application's class path.

5.1.2 Byte Code Parsing

Byte code parsing is necessary feature of the thesis and is required in order to be able to get the list of dependent classes on a class currently being loaded. No sufficient C++ implementation has not been found and therefore a custom parsing module has been implemented. The parsing module is based on the Apache Commons BCEL Java package and the simplified version but still with the same logic has been rewritten to C++.

The main class used for parsing is `ClassParser` which contains `parse` method accepting the bytecode of a class and defines also several accessors for the parsed data such as the super class name and complete reference, list of all implemented interfaces, list of all methods or list of all defined fields and their types.

The bytecode starts with the several important parts:

- **Magic id**. Magic id is a first integer stored in each bytecode and contains always 0xCAFEBABE number.
- **Version**. This part contains actually two shorts, where the first short represents minor Java version and the later one represents major Java version.

- **Constant pool**. Constant pool is a table representing class and interface names, field names and also other important constants. It contains mapping from id representing the type to the fully qualified type name.
- **Class Info**. Class information contains the information whether the bytecode represents a class or an interface, denotes class name and also super class name.
- **Interfaces**. This part contains the number of interfaces this class implements followed by id of type short of each interface. The 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.
- **Methods**. This part contains the number of defined methods in the bytecode together wit additional information per each method such as the number of arguments.

More information about class file structure can be found at https://docs.oracle.com/javase/specs 4.html. Each part of the class file mentioned above is parsed separately. For accessing the raw bytecode a `ByteReader` class is used. It contains methods fro reading different types of data from the bytecode array.

Parsing the magic id and both minor and major versions is straightforward as their are just numbers and can be read using the byte reader class directly. Parsing of the constant pool is more complex. For each entry in the constant pool a constant representing the entry is read. The constant can be of several type such as constant representing the Class symbol, String symbol, Method types of Field types for example. Once the Constant pool is parsed, it can be queried for the specific symbol by its id. Class name and super class name, interfaces, fields and methods are read from the constant pool by using their ids.

5.2 Span Injection on Instrumentation Server

This short section explains how additional fields such as the span information are internally attached to the instrumented classes. The trace information is attached to the instrumented class by adding a new synthetic field with name `____traceId`. This trace id represents the current trace and is used in the code to obtain reference to a current trace context and also current span. A new field is created using the Byte Buddy instrumentation builder using the `defineField` method.

5.3 Determine the Current Span Saver

This section explains how span saver type and also other arguments passed to the native agent are made accessible to the instrumentation saver. The span saver type is defined as part of the native agent and needs to be used at the instrumentation server as well. This is achieved by creating and registering the native method to a class which is defined as part of the instrumentation server. In span

saver case, the abstract class `cz.cuni.mff.d3s.distrace.storage.SpanSaver` is defined at the instrumentation server and contains also native agent for getting the span saver type. The abstract `SpanSaver` class is sent to the native agent during the initialization phase. When this class is used for the first time, the native method implementation is bound to the method `getSpanSaverType` defined in the `SpanSaver` class. Therefore, even the the classes defined as part of the instrumentation server can use method defined on the native agent with a really small overhead. Actually there may be a performance gain since these methods are written as native methods.

6. Big Example

7. Evaluation

7.1 Known Limitations

here mention limitations with the instrumentation

7.2 Platform demonstration

7.2.1 Deployment Strategies

Instrumentor per Application Node

Instrumentor per Whole Cluster

Optimizing the Deployment

7.2.2 Basic Building Blocks

7.2.3 Basic Demonstration

7.2.4 Optimizing the Solution

7.3 Comparison to Related Work

7.4 Future plans

7.4.1 Integration with well-known data collectors

7.4.2 Add support for Flame charts

charts of performance when having shared server and local instrumentation servers

8. Conclusion

An example citation: Anděl [2007]

Bibliography

Java Mixed-Mode Flame Graphs at netflix, javaone 2015. http://www.brendangregg.com/blog/2015-11-06/java-mixed-mode-flame-graphs.html. Accessed: 2017-03-13.

The Invocation API. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/invocation.html. Accessed: 2017-03-21.

J. Anděl. *Základy matematické statistiky*. Druhé opravené vydání. Matfyzpress, Praha, 2007. ISBN 80-7378-001-1.

List of Figures

List of Tables

List of Abbreviations

Attachments