MASTER THESIS

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

Title: Monitoring Tool for Distributed Java Applications

Author: 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 the is to create a monitoring platform and library which can be used to monitor distributed Java-based applications. This work is based on Google Dapper and shares a concept called "Span" with it. Spans 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 thesis consists of two parts: the native agent and instrumentation server. The users of this platform is 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. All classes marked for instrumentation are sent to the server which alters the byte code and caches the changed byte-code for the future instrumentation requests from other nodes.

Keywords: monitoring cluster instrumentation

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 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 or debugging such applications. Analyzing applications in distributed environment is inherently more complex task comparing to single-node applications where well-known debugging or profiling techniques may be used. Analysis of single-node applications usually focus on a single standalone application 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 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 are becoming affected. By collecting the relations between the nodes the analysis tool may be able to to use the information to 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 build 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 which is later used to build a so-called distributed stack-trace. Stack-trace in single node application represents the call hierarchy of 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 allows us to see the desired relations between the applications node. Google Dapper and Open-Zipkin are the most significant available monitoring tools and are discussed later in the thesis.

1.1 Project Goals

This thesis introduces monitoring tool for the similar purposes, sharing some of the concepts mentioned above, however the goals of this work are be different and should give the user a new generic and high-performance way how to monitor the 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 their instrumentation points.

Main requirements for the platforms 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.

- **Application-level Transparency and Universality**
  These two requirements are contradictory. The universal tool which could be used for monitoring majority of applications would either collect just basic information shared about all applications or the user would be required to manually specify the information specific to the application which leads to the loose of the application-level transparency. This tool tries to find compromise between these two goals and 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 for each JVM-based application to be able to be instrumented by minimizing the impact on the application itself. These

- **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 the 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 the users of this tool are required to use.

- **Modularity**
  The thesis 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 the user interface they prefer without significantly changed the code of the tool.

The discussion of different approaches for meeting the requirements above are discusses in the following section.

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 overview of several profiling tools, instrumentation and communication libraries. It also describes the relevant cluster monitoring tools like Google Dapper and OpenZipkin in more detail. This chapter ends with the Analysis section containing discussion of how specific requirements of the thesis are met. I also mentions the weaknesses of the other cluster monitoring tools and describes how the thesis tries to overcome them. The following Design section starts with an Overview section. This section depicts the architecture of the whole system. Further the Design chapter contains sections for each important parts of the application. It contains

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 consider for thesis or used in thesis are introduced. It covers libraries for byte code manipulation, communication, logging and also covers 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.

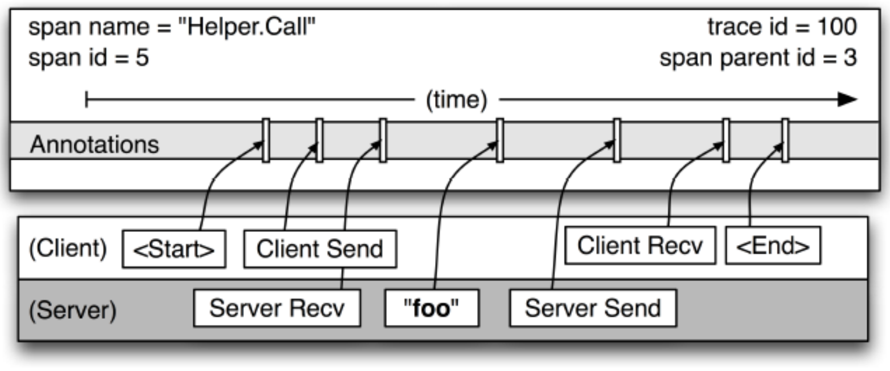


Figure 2.1: Example of Span in Google Dapper. Picture taken from the Google Dapper paper

- **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

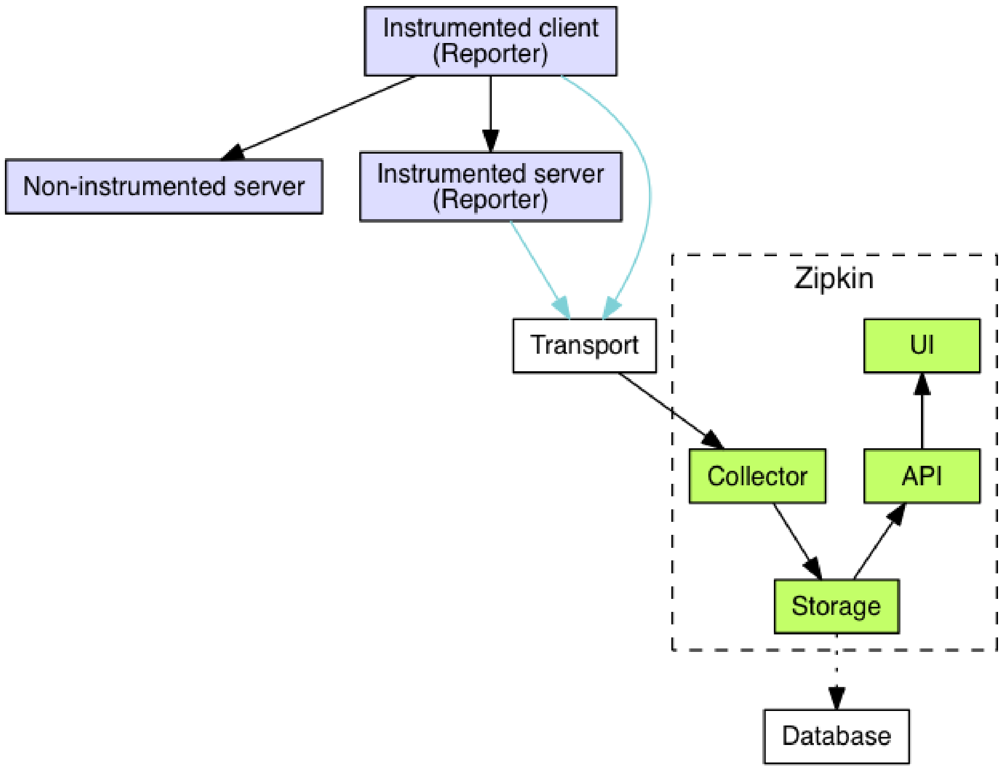


Figure 2.2: Zipkin architecture - http://zipkin.io/pages/architecture.html

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 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.

2.2.1 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.

2.2.2 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.

2.3 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:

- **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.

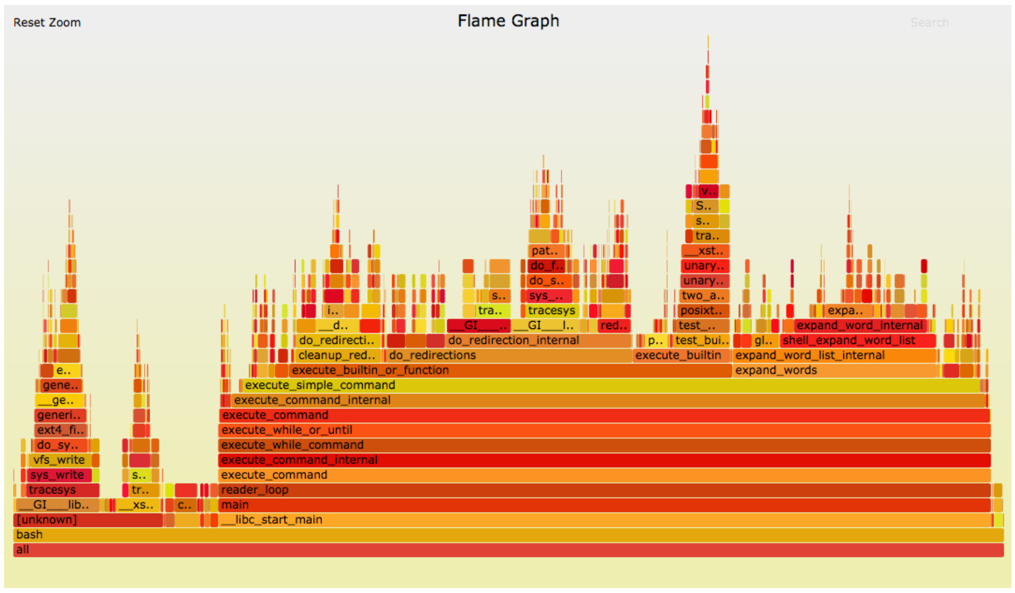


Figure 2.3: Flame Graph example

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.

2.4 Byte Code Manipulation Libraries

The thesis highly depends on the Java byte code instrumentation and this section gives overview of four byte code 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.

2.4.1 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.

2.4.2 Javassist

Javassist is well-known byte code 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 byte code. 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 byte code 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.

2.4.3 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.

2.4.4 Byte Buddy

Byte Buddy is fairly new, light-weight and high-level byte code 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 byte code but despite this, it gives the users full flexibility to redefine the byte code 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 byte code 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 byte code 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.

2.5 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.

2.5.1 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.

2.5.2 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.

2.5.3 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.

2.6 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.

2.6.1 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.

2.6.2 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.

2.6.3 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 byte code 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.

2.7 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.

2.8 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 environments so the user does not need to worry about configuration and downloading the correct dependencies for the application.

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

3. Analysis

This was achieved by limiting the number of artifacts to the bare minimum and therefore when it comes to using the tool, the user has only two files - native agent which needs to be attached to monitored application and the instrumentation server written in java which handles the instrumentation for the whole cluster application. Several deployment strategies exist and are discussed later in the 8 chapter.

The project should also be extensible in a way that information from additional low-level system monitoring tools can be attached to the monitored data - such as the memory usage or data allocation. We use native java agent written in C++ for the instrumentation purposes and the architecture is prepared to combine the monitored data from our tool with the other external tools in the future.

The project tries to make trade-offs between application level transparency and easiness of use. It is not desired from this tool to be an universal monitoring tool which could be used out of the box. but it can be thought of as an extendable library providing the developer with means how to instrument their specific application in high-level programming language such as Java. All instrumentation specific internals and low-level code is hidden from the user but low-level overhead is still achieved by multiple techniques discussed later. In order to use this platform on some particular application, the programmer has to extend the prepared library for the application by defining points where instrumentation should take place, but the original application's code remains unaffected and thus no recompilation of classes is required.

Using the native client we achieve low-overhead on the system and can query various interesting information such as number of loaded classes, garbage collection time and so on. To minimize performance and memory effects on the monitored applications, the instrumentation does not happen in the same JVM as the monitored applications runs. The native agents informs secondary helper JVM with our instrumentor running in it about the loaded classes and the instrumentor JVM decides whether the class should be instrumented or not and instruments the classes when required. This instrumentor JVM can also be shared by all the nodes in the cluster which has yet another advantage hat once any class has been instrumented by any node in the cluster, the other nodes just obtain the instrumented bytecode without the delays for instrumentation itself.

This chapter provides reasoning behind some architectural decisions of the monitoring platform. It describes different approaches for creating monitoring tool of this nature and provides arguments why the selected solution fits into the goals of this tool. The next section gives arguments why we decided to implement the application extensible instead of implementing all pieces from scratch. The last part of this chapter compares different approaches to the code instrumentation from the point of technical complexity and also the performance point of view

3.1 Platform Architecture

Several architectures for the whole platform was considered during the analysis stage of development. The main goals of this thesis is to achieve high level of application transparency, easy deployment and still affect the monitored application the least by the monitoring process itself.

One of the rejected approaches was to create a universal monitoring tool similar to Google Dapper, but support major portion of applications to be monitored. Whilst this would give the user great flexibility and universality and would also ensure that the uses have no need to extend the library, it's was decided as not feasible task. Every platform or application is different in it's architecture or in the way how it communicates. The communication can be implemented using various number of RPC frameworks and via HTTP using technologies like SOAP or REST. Such instrumentation tool could only instrument very basic information about Java-based programs, but since one of the main goals of this tools is the platform to be widely used and not tight to a specific platform, this approach was rejected.

The second and chosen approach for the monitoring tool was to design it as an general extendable instrumentation library. This library is supposed to hide all low-level details about core instrumentation and communication via JVMTI or JNI. The typical usage of this library is that developer can create monitoring tool for their particular applications based on this library just by specifying the points where the instrumentation should happens and what are the actions. The advantage of this approach is that only core instrumentation library needs to be created, generic to all platforms. However this library can't be used right away the programmer needs to build on top of it so the monitoring tool for their application can capture the application-specific control flow or data. The application transparency is therefore achieved in this approach by creating separation between the specific application and generic instrumentation and monitoring framework.

3.2 Application Modularity

During analysis of similar technologies and comparing different approaches for the monitoring it was decided that some modules of the application will be modular and the user could replace them with their own implementations. The extendable modules are the user interface presenting the observed data and the collectors bringing the data from application nodes to the user interface. Whilst default implementation exist and the user can use the application without changing these modules, we give the users the possibility tu plug-in custom user interface or more advanced data collectors. In order to be able to plug these modules in, they have to meet some criteria such as implement specific interfaces or extend specific classes, however the core implementation is let by the user.

The main reason for this solution is that a lot of monitoring solutions already exist and user are used to specific user interfaces or are using platform with already set-up data collecting. We wanted to support these use-cases so the environment where this monitoring tool runs does not have to be significantly changed. This leads to easier deployment of the platform.

3.2.1 Selection og User Interface

The reason why Zipkin UI was selected as the primary user interface for this work is mainly it's simplicity and ease of use. Also it fulfills the visualization requirements of the thesis as well, since we need to see dependencies between spans and also whole trace tree as well. However the monitoring platform is not tightly-coupled with this user interface. We will see later how to create custom span savers which can store data in any format suitable for different visualization tools.

3.3 Instrumentation Methods

This work is heavily based on byte-code instrumentation and the selection of instrumentation method heavily influences the rest of the platform. Usually the instrumentation can be done either via native or java agent. We briefly summarize the advantages and disadvantages of these 2 approaches and also provide arguments for the final solution which consists of combination of both techniques.

3.3.1 Java Agent

Java agent is 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 underlying bytecode. The distribution of Java Agents is also platform independent since they are packaged inside JAR files as the rest of Java classes. The disadvantages of this approach are usually the performance and the flexibility of the agent. Java agents are affected by garbage collection of the monitored application and can not be used to respond to internal JVM events. Also the additional objects created within the Java agent are put on the heap of monitored application. We can therefore influence the observations by the monitoring process itself. It is important to note that Java System classes can not be instrumented using Java Agent approach.

3.3.2 Native Agent

Native Agent is used for monitoring and instrumenting the applications on low-level programming language (C, C++) using JVMTI and JNI. These agents are written as native libraries on specific platforms so 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 give us great flexibility in the instrumentation and monitoring the JVM state. For example, all classes, 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 garbage collection, creation of new instance of specific class or switching threads. The big 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.

3.3.3 Special Instrumentation JVM

In the mentioned approaches, the instrumentation is done in the same process as the monitored application. Thus the application's application performance can be affected in both approaches. The goal of this thesis is to allow the user to write the instrumentation in the high level programming language. The assumption was that not many users would like to write the instrumentation on the level of byte arrays as in the case of the native agent. On the other hand, we wanted to give the user the possibility to instrument System classes as well and to be able to capture also information available only in native agent approach.

In order to take the best from the both approaches, we decided to create a so called Instrumentation Server. The instrumentation server is another JVM which receives the bytecode, performs the instrumentation and sends the bytecode back to the application. This approach takes the best from the 2 above since we are using the native agent on the application side and thus have access to all JVM internal information. When a class needs to be instrumented, the byte array containing the bytecode is sent to the instrumentation server for the instrumentation. On the instrumentation server, we load the bytecode using the ByteBuddy library and can perform instrumentation in high-level Java based language, even for System classes without affecting the performance of the monitored applications.

ByteBuddy was selected for the purposes of this instrumentation as it allows to write the instrumentation in Java but according the author, with considering the performance as well. Comparing 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 can be found easier. The API of the library is well-documented and the library is under active development. ByteBuddy is highly configurable which was also the main reason for choosing it as instrumenting classes for one JVM inside different JVM didn't work from scratch and turned out to be challenging part of the thesis.

The disadvantage of the chosen approach is that the observed application have to send the the bytecode to the instrumentation server and wait for the instrumented bytecode. However several optimizations have been implemented to minimize this as much as possible. More information about these optimizations can be found in the Instrumentation section of Design chapter.

3.3.4 Two JVMs Inside a Single Process

The approach with the instrumentation server could be further optimized for some use-cases if more Java Virtual Machines could be running in a single native process. This would mean that we could communicate between the instrumentation JVM and the monitored application inside one process this avoiding inter-process or network communication delays whilst still being able to instrument classes in high-level language. However, as of JDK/JRE 1.2, creation of multiple VMs in a single process is not supported Mor.

3.4 NanoMsg as Communication Layer

NanoMsg has been chosen for communication layer. t hides the platform specific aspects of sockets comparing to raw sockets. It has also several performance benefits and general improvements over the well-known ZeroMq such as the better threading and better implemented zero-copy technique. The mappings mentioned in the Nanomsg section of Design chapter were tested and worked as expected.

4. Overview

This chapter gives an high-level overview of the whole platform, still omitting design and implementation specific details. Firstly, architecture in different deployment modes is described followed by explanation of distributed trace collection. Next, the overview of all pieces of whole platform - the native instrumentation agent and instrumentation server, the user interface used for presenting the observed results and the span savers and data collectors for transporting data from the application node to the user interface server. This chapter ends by a single use case which may help reader to image how this platform can be used in a real-world scenario.

4.1 Architecture Description

The whole platform consist of several pieces. The native agent which is attached to the Java application, the Instrumentation server used for performing the instrumentation in the separated JVM process and the user interface. Data are brought to user interface using various collectors. The platform was designed to be configurable and therefore we support several deploying methods. The instrumentation server can be either on the network available to all the application nodes and can be shared by all applications. This has an advantage of caching the instrumented classes. So when any class is instrumented for the first time, the instrumentation is not performed for other nodes but the class is immediately sent. The disadvantage of this solution is higher latence 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 4.1.

The other deployment method is that the instrumentation server runs on each application node. This has the advantage of faster communication since we can use inter-process communication methods to communicate between monitored JVM and the instrumentation server. The disadvantage of this solution is that we have to instrument all classes 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 4.2.

4.2 Spans

Spans are the main tool used for capturing distributed traces. They are special classes which instances are injected to instrumented 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 creates new nested spans. The only information required to be available in Span class is its id and parent span id and in order to be able to distinguish different spans, also unique trace id which is created during root span creation.

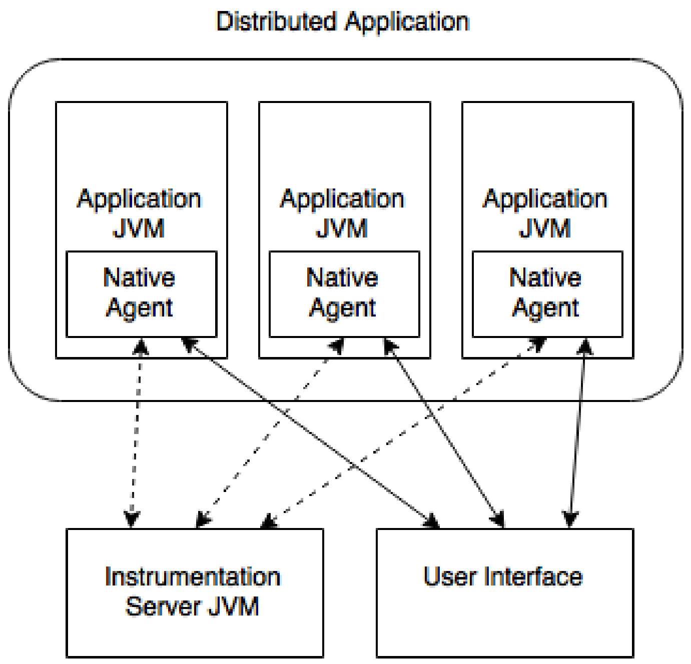


Figure 4.1: Architecture with shared instrumentation server. The dotted lines represents the communication between instrumentation server and the agent whilst the regular lines represents data collection from the agent to the UI

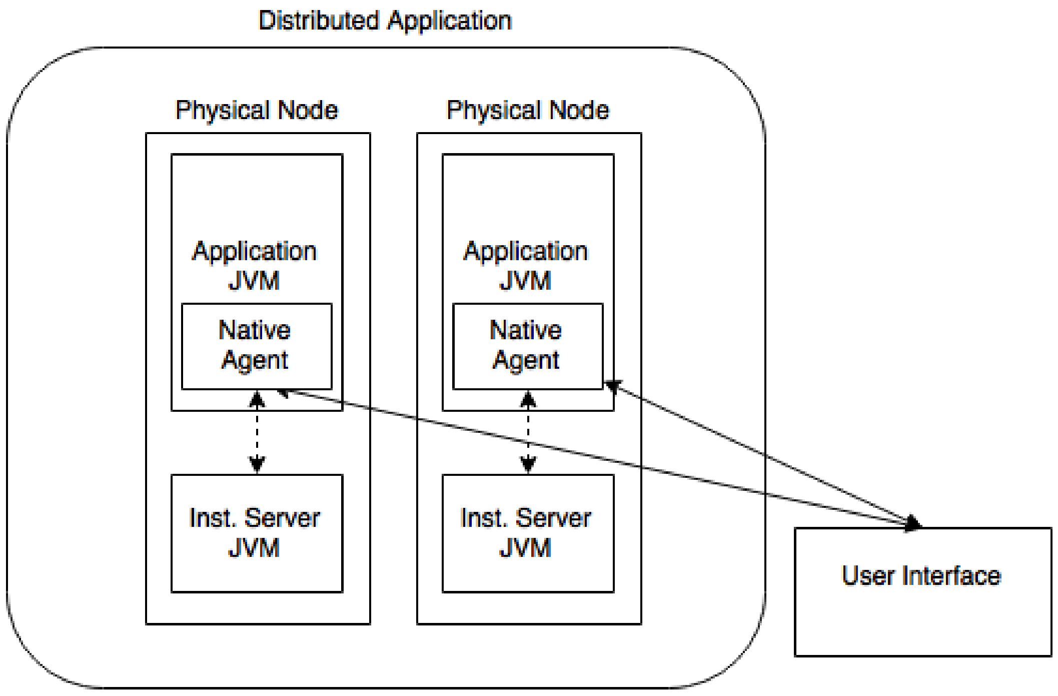


Figure 4.2: Architecture with separated instrumentation server. The meaning of the lines is the same as on the diagram above.

Created spans are processes using different span savers and can be send to the user interface using various data collectors.

4.3 Span Savers & Data Collectors

The platform allows the user to create custom span savers which can be used to plug-in custom data collectors. By default, the platform supports 2 simple span savers:

- The default span saver sends asynchronously the collected span to the user interface right away. In this case the functionality of span saver and data collector is covered by this single saver. This span saver should be used for demonstration purposes only since it could overload the user interface or network when there is a high amount of collected spans in a short amount of time.
- The second available span saver saves asynchronously the collected data on the disk in the format known to Zipkin UI. This data can be collected using various data collectors. This is a preferred way when putting this tool to production with combination in some well-known data collection agent.

4.4 User Interface

The user interface receives the data from the span savers or data collectors and present them to the user. It connects the spans collected from different nodes using the span and parent span ids which means we can see the distributed traces graphically and see how the computation went node by node. For the purpose of this tool, Zipkin User Interface is used for this purpose since it's open source and fulfills exactly the required service. However as mentioned above, the users can write custom span savers to save the data in a format known to different user interface in order to not depend on Zipkin UI.

4.5 Native Agent

The native agent is the core part of the whole platform. It is attached to the monitored application and allows us to obtain various low-level information from the application run. The native agent is responsible for starting the instrumentation server in case of local instrumentation mode or connecting to the instrumentation server when sharing the server between all application nodes.

The main task for the agent is to check weather a class is required to be instrumented and if yes, send the class for the instrumentation to the server and wait for the instrumented code. The agent does not know which classes are to be instrumented and it therefore needs to query the server. The other approach could be to specify which classes should be instrumented as arguments to the client but that would be hard to manage since it would be static list of classes. In this approach the developer can specify which classes can be instrumented on the instrumentation server using simple Byte Buddy API.

The communication protocol between the agent and instrumentation server as well as the technical aspects of the instrumentation are described in the following chapters.

4.6 Instrumentation Server

The instrumentation server is responsible as the name suggest for instrumenting the byte code. The native agent asks the server weather the class is required to be instrumented, the server then received the byte code, performs the instrumentation and sends the data back to the agent. It does not contain any application state and does not know about the distributed traces.

The instrumentation server also acts as the base library for the instrumentation for specific applications. Usually the user needs to have the code for instrumentation server available as a dependency and can use prepared method to define custom instrumentation points without touching the internals of the native agent.

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 as well. The other issue is for example 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 the issues and the optimizations mentioned above are described in the following sections.

4.7 Example Use Case

In order to have a better understanding how this platform can be used we provide simple example of client-server application as an use-case, so far without the code. Clients query the server and server provides them with the response. We would like to instrument the client and the server and monitor the communication between the client and the server.

Because client and the server are different applications, we need to create 2 different instrumentation servers and specify what is the method and a class which is required to be instrumented and what should happen in the instrumented code. In this case we just want to add custom annotations to recorded spans which are specific to the client-server communications.

The native agent has to be attached to both the client and the server prior it's start and the path to the instrumentation server jar needs to be set on both client and server to corresponding server. 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.

We can see that the only part the user needs to worry about is the extension of the instrumentation server to specify the custom instrumentation points, otherwise the rest of the work is done automatically.

5. Design

This chapter describes design of the whole platform in details, however implementation details of some specific parts is described in the following chapter. Spans and their format is described first followed by design of native agent and instrumentation server. This chapter ends by describing the selected Zipkin user interface and also JSON format in which the UI accepts the data from the instrumentation server.

5.1 Spans

As mentioned briefly in the previous section, spans are used to gather the information about the distributed calls or so called, distributed stacktraces. Spans 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 has several mandatory and optional fields. The mandatory fields are trace id, id and parent id. Trace id represents one complete distributed call among all interacting nodes on the cluster. The field is attached automatically when a new root span is created. Root span is a first one which is created inside a trace. The root span does not have parent id field set up so the user interface backend can distinguish between regular spans and root spans. Parent if of a span is always id of span from which we received a request to perform some task. The span and its parent span can be located on the same node and on different nodes as well. The first can be useful in cases we would like to capture the communication between different threads on the same node in the same manner as the rest of cluster communication

Span have several additional fields which are set and are used in the Zipkin UI. Each span has also

- **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 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 has also endpoint information attached. This consist of:

- **ip** - ip of node on which this event is recorded
- **port** - port on which the service which recorded the span is running

- **service name** - service name can be used to group different traces by names and can be later used to filter such traces in the user interface

The following sections contains information about how span are exported for external communication and also how spans are created using `TraceContext` and `TraceContextManager` classes.

5.1.1 Span Savers

Spans are internally represented in a JSON format. The thesis contains support for working with JSON data and it is explained in more detail later in the Implementation chapter. In order to be able to send span data to corresponding user interface or just to save them on disk the spans can be processes using various span savers.

Each saver has to extend from 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.

SpanSaver abstract class has 2 abstract methods:

- `saveSpan`. This method is used for saving span. Custom span saver implementaion may save the data on local disk or send over network. The destination is not limited.

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

Spans are saved asynchronously using executor service. The thesis offers 2 default span saver implementation - `DirectZipkinSaver` and `JSONDiskSaver`. The first one sends spans to Zipkin user interface without storing it on disk. It accepts a single argument which is ip and port of the Zipkin UI service. The second saver is used to save data on disk which can be further collected by some data collector. It accepts single argument which is a directory where spans are saved.

In order to be able to allow the flexibility to add new savers, we have register them in the META-INF directory of the generated 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 we use the **AutoService** library from *https://github.com/google/auto/tree/master/service*. Instead of manually registering the implemented span savers into META-INF directory, we can annotate them with `AutoService` annotation with a single argument specifying the abstract parent. The library then takes care of registering the classes automatically so the human error is minimized.

5.1.2 Trace Context

Trace context is class used for storing the information about the current span and also for creating new 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 us 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 1 is created. The version 1 of UUID combines 48-bit MAC address of the current device with the current timestamp. This way it is ensured that 2 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 `openNestedSpan` and `closeCurrentSpan`. The first 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 saver attached to the span and moves one level higher in the span hierarchy to the previous current span.

5.2 Native Agent

The native agent is used for accessing the internal state of the monitored application and to instrument classes to allow us to attach the span and trace identifiers to classes transferred between the application nodes.

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. Bytecode parsing is a technical task described in the following Implementation chapter.
- **InstrumentorAPI**. This class is provides several methods which are used to communicate with the instrumentor JVM. All the queries to the instrumentor are done via instance of this class.
- **AgentCallbacks**. All callbacks used in the native agent are defined in this namespace.
- **AgentArgs**. This class contains all the logic required for argument parsing.
- **NativeMethodsHelper**. This 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 namespances are **AgentUtils** and **JavaUtils**. The first one is contains method for managing the JVMTI connection together with method for registering the JVMTI callbacks and events. The second one is used for easier work with Java objects in the native code via JNI.

5.2.1 Agent Initialization

The agent is initialized via the same phases as described in the JVMTI Agent Initialization section of the Background chapter. For the thesis purposes, we are interested in the following events: `VM Init`, `VM Start`, `VM Death`, `Class File load Hook`, `Class Prepare` and `Class Load` event. Callbacks are registered for all the mentioned events so we can react to them accordingly in the code.

As part of the initialization process we need to either connect to or start a new instrumentation server. In case the native agent was started in the shared mode of the instrumentation server, the code tries to connect to already existing server. In the local instrumentation mode, the instrumentation server is started as a separated process and the connection is established with the server using the inter process communication.

The callback registered for the `VM Init` event is 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 and have to be available to the monitored application since the instrumented code require them as dependencies. The native agent is designed as a code which users should not need to touch and define all application specific code in the instrumentation server extension. 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.

5.2.2 Instrumentation

Code handling the instrumentation is part of the callback for the `Class File load Hook` event. The callback has the bytecode for the class which is being loaded on its input and allow as 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, we have to lock before we start instrumentation of a class.

2. Firstly, we check if the virtual machine is started since we don't need to instrument 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 us to take care of correct threading management.

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

5. Parse the name of 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 is correct. Instead

of relying on this parameter, we parse the class name from the bytecode ourselves.

6. Decide weather we should continue with the instrumentation. This check is based on the used class loader and name of the class being loaded. Classes loaded by the `Boostrap` classloader and in case of Sun JVM, from `sun.reflect.DelegatingClassloader` are not supposed to be instrumented. The `Boostrap` is used to load system class and the second mentioned classloader is used to load synthetic classes and in both cases, it's not desired to instrument classes loaded by these classloaders. 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 example for accessing the super class of the currently instrumented class. Therefore we proceed to the instrumentation only If the class is not ignored and not loaded by ignored class loader.

7. We ask the instrumentation server whether it already contains the loading class or not. If the server does not contain the class, we send the class data to the instrumentation server, parse the class file for all the dependent classes and send all dependent classes to the instrumentatiot. We repeat the dependency scan recurrently until the class does not have any other dependencies or until we know that the dependency 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. Therefore, at this step we can proceed with the instrumentation itself and ask the server for the instrumented class.

9. Exit the critical section.

Even though the class is not fully instrumented and the instrumented byte-code 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 the interceptor field in the instrumented classes. It is a static field which references the instance of the class interceptor. This field is automatically set by Byte Buddy framework in most of the cases but since we are instrumenting on different JVM then where the code is actually running, we need to handle this case as well. In order to set this field by corresponding `Initializer` class we need to have both the initializer class and interceptor 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 register the interceptor and initializers with the instrumented class for later since we need to wait for the class to be used for the first time to set the static field to a corresponding interceptor. The initializers are loaded during `Class Prepare` event handling. This event is triggered when the class is prepared but no code has been execute so for.

This callback for this event is also used to register the native methods for the class being loaded. By registering the native method to the class we make it available from Java programming language.

Several technical difficulties had to be dealt with during the development. For example, we need to properly handle cyclic dependencies when instrumenting the class. 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.

5.2.3 Instrumentation API

The Instrumentation API is used to communicate with the instrumentation server. It provides low-level method 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 is build to make the communication easier. The most important methods are:

- `sendClassData` method sends byte code to the instrumentation server.
- `isClassOnInstrumentor` method checks weather 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 weather the class on its input is allowed to be instrumented.
- `loadPrepClasses` method loads all dependent classes in the agent initialization phase.

5.2.4 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 the server these classes if they need to be instrumented or if some class to be instrumented depend 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 agent configuration. It can contain all arguments mentioned above, each argument per 1 line of the configuration file.

5.3 Instrumentation Server

Instrumentation server is responsible the the code instrumentation in separated virtual machine then the application is running. In this section we cover several design aspects of the instrumentation server, leaving the implementation details on the following sections. As already mentioned, the server can run locally on each application node or it can be shared among all the application nodes. The core instrumentation on the server is handled by the Byte Buddy code manipulation framework.

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 byte code for a class on the server.

- Request for sending dependency classes needed by the agent
- 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.

5.3.1 Instrumentation

The instrumentation of the class is triggered by the agent and it's done in 2 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 who asks the server whether the class is available on the server or not. The server performs this check in 3 phases:

1. Check whether the instrumented bytecode for this class is available
2. If not, check wether the original bytecode for this class is available
3. If not, check if we can load the class using our 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.

The server informs the client if it does not have the class, the agent sends the class and the server registers the received byte-code 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 the first one. If the server already contains the instrumented class in the cache, it sends it 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 abstract class containing single abstract method `createAgent(BaseAgentBuilder builder, String pathToGeneratedClasses)` where the builder is wrapper around the Byte Buddy `AgentBuilder` class which is used to define the class transformers.

The user 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 some interceptor which defines the instrumented code. Each interceptor 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 sever. Each implementation of the interceptor needs to register itself in the META-INF directory of the generated jar in a same way as it 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 for 4 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. Our implementation of the listener 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 override in the child class. The dependencies are all the classes specified in the class file such as type of return value, arguments, super class or 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 byte code of dependent classes. The classes are loaded by context class loader by default, but since the classes are received over the network we use our `InstrumentorClassloader` 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 byte code 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 contains the 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 one 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 2 steps:

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

5.3.2 Custom Service Loader

In order to keep the instrumentation extendable we use concept of service loaders for loading the extensions. This is done for 2 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 the name of 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` has to **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 we need to have the `Class` object representing the class available. 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 loaded service classes.

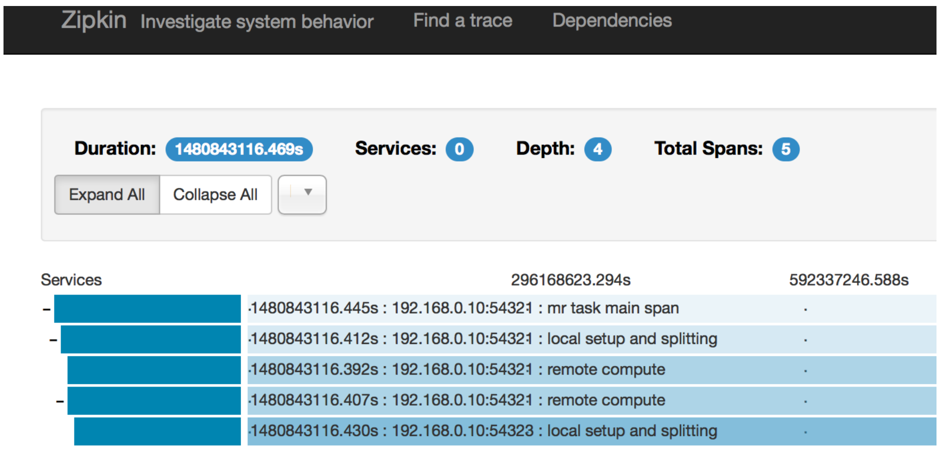
5.3.3 JSON Generation

The 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 we created custom simplified implementation which fits the theses requirements. Also the number of dependencies is lowered by this decision.

This JSON support is designed using 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 number type.
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.



```
-1480843116.445s : 192.168.0.10:54321 : mr task main span
-1480843116.412s : 192.168.0.10:54321 : local setup and splitting
-1480843116.392s : 192.168.0.10:54321 : remote compute
-1480843116.407s : 192.168.0.10:54321 : remote compute
-1480843116.430s : 192.168.0.10:54323 : local setup and splitting
```

Figure 5.1: Example of Zipkin UI

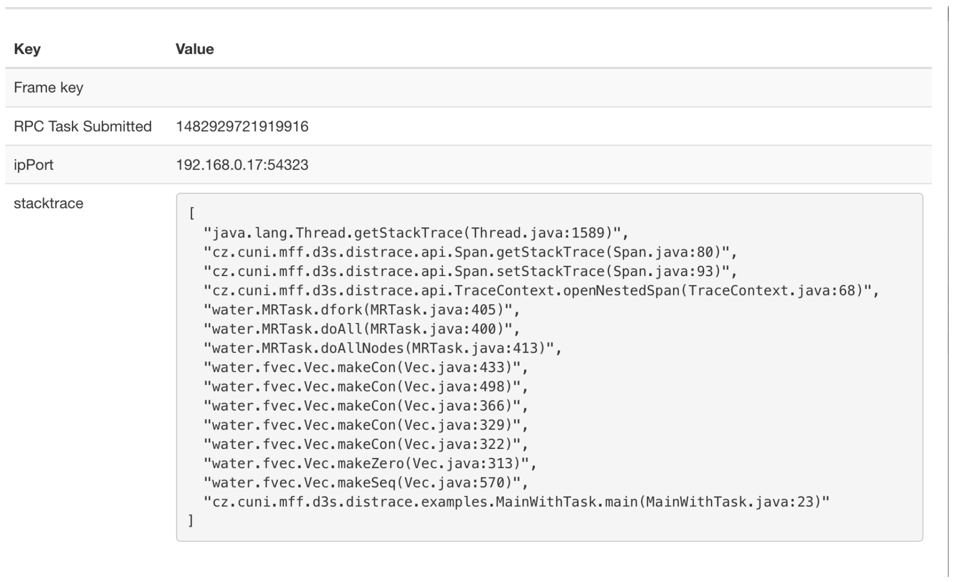
Each **JSONValue** can be printed as valid JSON string and the printing is driven by a class `JSONStringBuilder`. This class is also responsible for escaping the characters according to JSON standards. The default printer prints the data without any formatting as one line, however `JSONPrettyStringBuilder`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

5.4 User Interface

The thesis is using Zipkin as default user interface. The default data format for exporting spans is build 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 .

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 that the data for a single span can be sent incrementally. This means that we can send several JSONs representing the same span with different annotations and the user interface merges these spans into single one and presents all annotations under the given span. This allow us to to send part of data from the sender side and part of data from the receiver site right to the user interface instead of sending the data back on forth to send them as one single complete span.

Each span in the UI is clickable and all the additional information can bee seen at that level. In this thesis we collect stack trace at each span for monitoring purposes. Example of such information screen can be seen on the figure 5.2.



Key	Value
Frame key	
RPC Task Submitted	1482929721919916
ipPort	192.168.0.17:54323
stacktrace	see below

```
[
  "java.lang.Thread.getStackTrace(Thread.java:1589)",
  "cz.cuni.mff.d3s.distrace.api.Span.getStackTrace(Span.java:80)",
  "cz.cuni.mff.d3s.distrace.api.Span.setStackTrace(Span.java:93)",
  "cz.cuni.mff.d3s.distrace.api.TraceContext.openNestedSpan(TraceContext.java:68)",
  "water.MRTask.dfork(MRTask.java:405)",
  "water.MRTask.doAll(MRTask.java:400)",
  "water.MRTask.doAllNodes(MRTask.java:413)",
  "water.fvec.Vec.makeCon(Vec.java:433)",
  "water.fvec.Vec.makeCon(Vec.java:498)",
  "water.fvec.Vec.makeCon(Vec.java:366)",
  "water.fvec.Vec.makeCon(Vec.java:329)",
  "water.fvec.Vec.makeCon(Vec.java:322)",
  "water.fvec.Vec.makeZero(Vec.java:313)",
  "water.fvec.Vec.makeSeq(Vec.java:570)",
  "cz.cuni.mff.d3s.distrace.examples.MainWithTask.main(MainWithTask.java:23)"
]
```

Figure 5.2: Example of the detail span information.

5.4.1 Zipkin Data Model

Zipkin requires the data to be sent in JSON format. Requests to UI are sent as JSON arrays where the array elements are the spans them selfs. Zipkin understands the following 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 time from the creating the span until closing the span.
- **annotations** - array containing standard Zipkin annotations. These annotations can be handled by user interface in specific way since the user interface is 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 reciving the span
  - **ss** : timestamp of server receiving the span
  - **ca** : client address
  - **sa** : server address

- **binaryAnnotations** - array of custom annotations. For example collected stack trace is send as a binary annotation.

Except the *annotations* and *binaryAnnotations* fields, the fields are of simple string or number type. Annotations are objects with the 3 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
        },
      }
    ]
 }
]
```

6. Implementation Details

Mention interesting parts of the implementation

6.1 Native Agent

6.1.1 Byte Class Parsing

explain byte code structure

6.1.2 Instrumentation

Handling auxiliary classes and loaded type initializers

6.2 Instrumentation Server

6.2.1 Byte-Code Instrumentation

6.3 Zipkin Integration

Sending Data to Zipkin

7. Big Example

8. Evaluation

8.1 Known Limitations

here mention limitations with the instrumentation

Appendix- will contain arguments for the native agent.

8.2 Platform demonstration

8.2.1 Deployment Strategies

Instrumentor per Application Node

Instrumentor per Whole Cluster

Optimizing the Deployment

8.2.2 Basic Building Blocks

8.2.3 Basic Demonstration

8.2.4 Optimizing the Solution

grafy

9. Conclusion

9.1 Comparison to Related Work

9.2 Future plans

9.2.1 Integration with well-known data collectors

9.2.2 Add support for Flame charts

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