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 thesis 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 encapsulate set of calls among multiple communicating hosts and in order to be able to capture them without the need of changing the original application, instrumentation techniques are highly used in the thesis. The thesis consists of 2 parts: the native agent and instrumentation server. The users of this platform need 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 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 and theoretical advice.

Contents

1. Introduction

The volume of data processed is becoming larger every day. In order to process this data, the application are becoming distributed for reasons of scalability, stability and availability. However having these applications available is not sufficient. We need to be able to monitor and reason about distributed applications as well since only then we can efficiently find bugs and improve performance of such applications. However, reasoning about distributed applications is inherently more complex compared to single-node applications. In case of single node application we can use standard monitoring or profiling tools producing output we can use to reason about the application, but in the complex cluster solutions this can't be applied. Standard monitoring tools per each node in the cluster may be used however we still could reason only about one particular node and not the whole cluster.

Several monitoring tools for such a problem have been already developed. The most significant one is proprietorial software from Google called Google Dapper and open-sourced tool called Zipkin. They both build on a simple idea called distributed trace. In single node applications we can get current stracktrace which represents the call hierarchy at given moment. Distributed stack trace is the very same concept except that the dependencies between different nodes are preserved in that stack too. Therefore, using the distributed traces we can reason about more complex systems.

This thesis introduces another monitoring tool for the similar purpose, however it has some different concepts than the tools mentioned above. Google Dapper is build proprietorial and available only for Google applications. In case of Zipkin, the user has to instrument the application itself by introducing the annotation. This thesis tries to overcome these issues by creating open-source monitoring platform with small footprint on the monitored applications whilst giving the user the possibility to use high-level programming language to define their instrumentation points.

1.1 Project Goals

The project makes trade-offs between application level transparency and easiness of use. It is not an universal monitoring tool which could be used out of the bug 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.

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.

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.

The easiness of deployment is also the significant aspect of the introduced tool. In order for developers and testers to use this tool, it needs to be relatively easy to deploy it and use it. 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.

1.2 Thesis outline

The thesis starts

2. Background

This chapter covers technologies which are relevant to the thesis. It starts with the overview of similar monitoring tools for cluster based applications and follows by short overview of tools for debugging of large scale applications. Later different approaches to applications profiling are described. In the next several sections the technologies which have been consider or are used in the thesis in the current moment are introduced. It covers libraries for bytecode manipulation, communication, logging and Java relevant libraries such as JNI and JVMTI. Docker is briefly described at the end of this chapter as it is used as the main distributed package for the whole platform.

2.1 Cluster Monitoring Tools

The most significant platforms to this thesis are Google Dapper and Zipkin, where Zipkin is based on the previous. Both serves the same core purpose which is to monitor large-scale Java based distributed applications. This thesis is based mainly on Google Dapper but also uses helpful Zipkin modules such as the user interface. Since Zipkin is developed according to Google Dapper design, these two platforms shares very similar concepts. The most important concept is a Span and it is explained in more details in the following section. For now, we can think of a span as time slots encapsulating several calls from one node to another with well-defined start and end of the communication. The following two sections describes the basics the both mentioned platform. Both Zipkin and Dapper shares very similar concepts wo we just point out the most import parts relevant to the thesis.

2.1.1 Google Dapper

Google Dapper is proprietary software which was mainly developed as a tool for monitoring large distributed applications since debugging and reasoning about applications running on multiple host at the same time, sometimes written in different programming languages is inherently complex. Google Dapper has 3 main pillars on which is built:

- Low overhead It was assumed that such a tool should share the same life-cycle as the monitored application itself thus low overhead was on of the main design goals as well. Google dapper
- 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. 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 in the Google shares the same libraries and control flow.
- Scalability Such a system should perform well on large scale data.

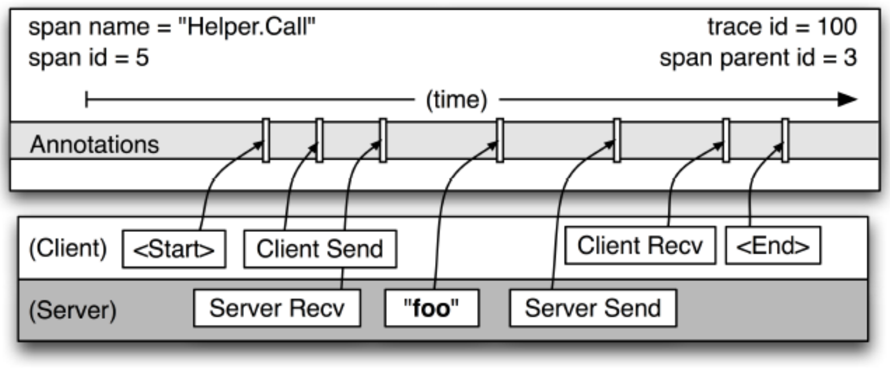

Figure 2.1: Example of Span in Google Dapper. Picture taken from the Google Dapper paper

Google Dapper collects so called distributed traces. The origin of the distributed trace is the communication/task initiator and the trace spans across the nodes in the cluster which took part as the computation/communication.

There were two proposals for obtaining this information - using the black-box and annotation-based monitoring approaches. The first one assumes no additional knowledge about the application whereas the second can use of additional information via annotations. Dapper is mainly using black-box monitoring schema since most of the control flow and RPC subsystems are shared among Google.

In Dapper, distributed traces are captured in so called trace trees, where tree nodes are basic units of work referred to as spans. Span is 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 its own id so it can be uniquely identified. In order to reconstruct the whole trace tree, we need to be able to identify the starting Span. Spans without parent id are called root spans serves exactly that purpose. Span can also contain information from multiple hosts, usually from spans from direct neighborhood. Spans structure in Dapper platform is described in the figure 2.1.

Dapper is able to achieve application-level transparency and follow distributed control paths thanks to instrumentation of a few common, mostly shared libraries among Google developers.

- Dapper attaches so called trace-context as thread-local variable to the thread when the thread 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 span ans current span id

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

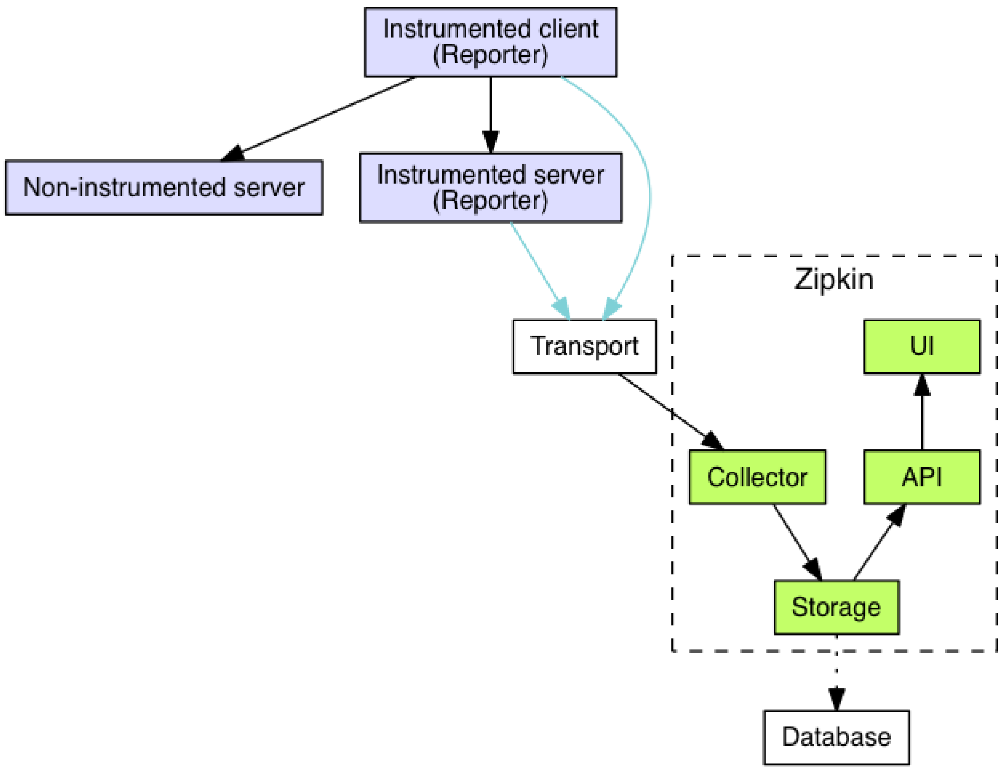

Figure 2.2: Zipkin architecture - http://zipkin.io/pages/architecture.html

Even though Dapper is mainly following black-box monitoring scheme mentioned bellow, it still have small support for adding custom annotation to the code. This gives the developer of an application possibility to attach additional information to spans which are very application-specific.

The low-level overhead was also achieved by sampling the data. As is mentioned in the paper, the volume of data at Google is significant so only samples are taken at a time.

2.1.2 Zipkin

Zipkin is open-source distributed tracing system. It 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 captured automatically such as time when Span was created whereas some are optional and some even application-specific.

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 which works well with whole Zipkin infrastructure. Spans are stored asynchronously in Zipkin to ensure lower overhead.

Once the span is created, it is sent to Zipkin, in more details, to Zipkin collector. In General, Zipkin consists of 4 components:

- Zipkin Collector It 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 mulltiple ways, so this is a pluggable component. Data can be stored in for example in Cassandra, MySQL or can be send to Zipkin UI right away without storing it anywhere. The last option is good only for small amount of data.

- Zipkin Query Service This component act as a query daemon allowing us to query various informaion about span using simple JSON API.

- Web UI Basic, but very useful user interface. The user can see whole trace trees and all spans with dependencies between them.

In the thesis the Zipkin UI is used as front end for developed the monitoring tool and it's format is described in more detail in Zipkin UI section of Design chapter.

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.

2.2 Tools for Large-Scale Debugging

Standard techniques and tools can be used for debugging distributed applications, however when using these tools we lack the information about dependencies between different nodes in the cluster. There are many tools under the category of large-scale debugging but we just point out basic ideas behind two different approaches - discovering scalling bugs and behaviour based debugging.

2.2.1 Discovering Scaling Bugs

In distributed systems the scalability is very important. It is very important to know how our platform scales when it comes to significantly big data and what is the scalability trend we can expect. It can happen that on large data the platform can run significantly slower than expected when tested on smaller data. We call this issue as a scaling bug. Tools which can be used to help with these kind od 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. The observed scalling trend acts as a boundary. We observe the scalling bug when the scalling trend is violated. In the first tool, Wrishna, we can't tell which port of the program violated the scalling trend, however it is possible in the second tool, WuKong. In comparison to Krishna, Wukong doesn't build one scalling trend of the whole applications, but creates more smaller models, each per some control flow structure in desired programming language where the all these smaller models represent together the whole scalling trend. When we hit into scalling bug, WuKong can give us hints where the trend can be violated.

2.2.2 Behavior-based Analysis

The different category of tools used for debugging of large scale applications are based on behaviour analysis. The basic idea behind these tools is that the classes of equivalence are created from different program processes and different runs. Using this approach we lower down the number of data we need to inspect and the tools can help us to discover anomalies between different observed classes. 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. As the other example falling under the same category is AutomaDed. This tool creates several models from an execution and can compare them using clustering algorithm with (dis)-similarity metric to to discover anomalous behaviours. It also can point to specific code region which may be causing the anomaly.

2.3 Profiling Tools

Profiling is a form of dynamic code analysis used for analyzing for example how long each part of the system takes in the whole computation, where the computation spends the most time or the memory requirements of the whole program. Generally, we can group the profiling tools into two categories: sampling profilers and instrumentation profilers.

- sampling profilers

  Sampling profilers take statistical samples of an application at well-defined points such as method invocations. It usually have less overhead comparing to instrumentation profilers. The points were the application should take samples can be inserted at the compilation time by the compiler. Using these profilers we can collect how long the method run, who call it or for example the complete stacktrace. We however can't record any application specific information.

- instrumentation profilers This can be solved by instrumentation profilers. These profilers build on the instrumentation of the application's source code. They record the same kind of information as the sampling profilers but usually give us the ability to specify extra points in the code we are interested in and also to record application specific data.

However, we can look on profilers from different point of view and categorize them based on the level on which they operate and are able to record the information - system profilers and application specific profilers. At application specific profilers, we are the most interested in profilers targeted for JVM platform.

- System profilers System profilers operate on OS-level. They are great at showing system code paths, but are not able to capture method calls done for example in Java application.

- JVM profilers These profilers show Java methods, but usually not system code paths.

The ideal solution for monitoring purposes 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 the profiler types are usually called mixed-mode profilers. JDK8u60 comes with the solution in a way of extra JVM argument *-XX:+PreserveFramePointer* Mix. Operating system is usually using this field to point to most recent call of 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 optimalizations.

This option ensures that JVM abides the frame pointer register and will not use it as general purpose register and therefore we can get both system and JVM stack frames in a one call hierarchy. Using the JVM mixed-mode profilers we are able to collect information about:

- page faults They allow us to see what leads to from of JVM resident memory.
- context switches Context switches are interesting to see code path leads to leaving the CPU.
- disk i/o requests Show code paths leading to IO operations such as blocking disk seek operation.
- TCP events Show code paths leading from high-level Java code to low-level system methods such as connect or accept, so we can reason about performance and good design of network communication in much more better detail.
- CPU cache misses Show code paths leading to cache misses. Using this information we can optimize the Java code to make better use of the existing cache hierarchy.

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

Flame Charts

Flame Chart is a concept by a developer Brendan Gregg. Flame graphs are aa visualization for samples stack traces, which allows the hot paths in the code to be identified quickly. The output of sampling of instrumentation profiler can be significantly big and therefore visualizing can help to reason about performance in more comfortable way.

The description:

- 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.

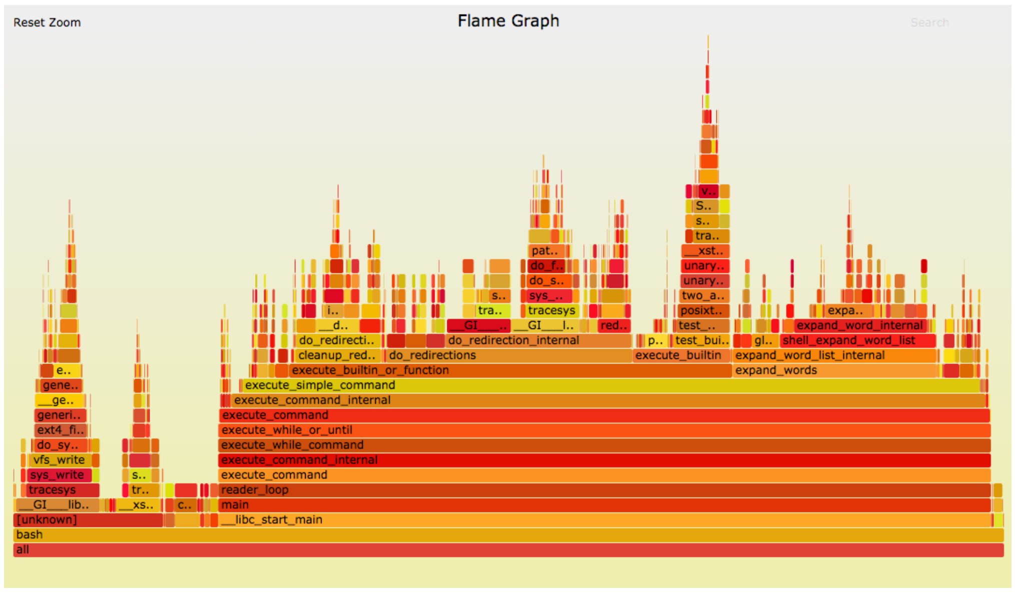

Figure 2.3: Flame Graph example

- The **x-axis** shows the population of traces. It doesn't represent passing of time. The function calls are usually sorted alphabetically.
- The width of each box represents the time 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 we can use profiler of our choice.
2. Fold stacks We need to prepare the stacks so Flame graphs can be created out of them. For this, there are several scripts prepared for different profiler types.
3. Generate the flame graph itself, again using the prepared script provided on the link above.

Purpose of this really short section was just to introduce the idea of Flame charts since it's one of the future plans the thesis could be extended to support. For more information about the flame charts please visit the Brendan Gregg's blog.

2.4 Bytecode Manipulation Libraries

This thesis highly depends on the for which the byte-code manipulation is a core feature. Since the work is written in Java, we are mainly interested in instrumentation and byte-code manipulation libraries based on Java. This section covers . The purpose of this section is to introduce 4 standard bytecode manipulation libraries - Javassist, ByteBuddy, CGlib and ASM - and give their comparison.

Since it's a core feature of the whole platform and affect the performance and the usability of the whole platform, the library was thoroughly reviewed before selected. ByteBuddy was selected to be used in the thesis and the reasons why are mentioned bellow as well.

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 bytecode manipulation library built on top of ASM. It allows to Java programs to define new classes at runtime and also to modify a class files prior the JVM loads them. It works on higher level abstraction so the user of this library is not required to work with the low-level bytecode. The code to be injected to the existing bytecode is expressed as Java Strings which has the disadvantage that the code to be injected is not subject to code inspection in most of the current IDEs. The advantage of Javassist is that the injected code does not depend on the Javassist library at all. The strings representing the code are compiled on runtime by special javassist compiler which works well for most of the common programming structures but just to point out auto-boxing and generics are not supported by the compiler. 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.

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 runtime. The proxified class is then used to 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 from the users.

2.4.4 Byte Buddy

ByteBuddy is fairly new, light-weight and high-level bytecode manupulation library. The library depends only on visitor API of the ASM library which does not further have any other dependencies. It does not require from the user to understand format of java bytecode but despite this, it gives the users full flexibility to redefine the byte code according 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 we want to create a subclass, in the most generic way we can create subclass from Object class. The newly created class can introduce new methods or intercept methods from it's super class. In order to intercept existing methods and change their behavior and return value, the method to be intercepted has to be identified using so-called ElementMatchers. These matchers allow us to identify methods using for example their names, number of arguments, return methods or associated annotations. The whole list of matchers and also examples how code can be generated is greatly described on the project Github page.

As mentioned earlier, the power behind Byte Buddy is also that it can be used to redefine classes at runtime. This is achieved several concepts, mainly Transformers, Interceptors and Advice API.

Code Transformation

In order to tell byte buddy when it needs to intercept a method or field, we need to identify the place in the code which triggers the interception. First, the class containing method to be instrumented need to be located. It can be done by simply specifying the class name or using more complex structures. For example we can 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 what interceptor or advice should handle the instrumentation for the particular matched method. Such methods can be again identified using the ElementMatchers mentioned in the section above. In more detail, transformer interface has a method transfrom which has DynamicType.Builder as it's argument. This builder is used to create a single transformer wrapping all the previous ones for all classes in the code so the result of this builder can be thought of as a dispatcher of the instrumentation. Methods to be instrumented are specified on the builder instance using the ElementMatchers as well as what interceptor or advice API will be used to handle the transformation.

As mentioned above there are 2 ways how to instrument a class in ByteBuddy.

Interceptors

Interceptor is a class defining the new or changed desired behavior for the method to be instrumented. We demonstrate how Byte Buddy uses interceptors on a small example. Let's assume we have original class Foo:

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

Le'ts also assume that our Interceptor is of type Qux. The interception of the class Foo using our 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);
    }
}
```

We can therefore see 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 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, initialization strategy can be modified according to the specific needs. We can define no-op strategy and read `LoadedTypeInitializer` right before the class is about to be instrumented. Later we can perform the initialization our self using observed initializer or we can even serialize this 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, we can delegate to `Qux` as a class and call the interception logic via static methods. In this case the interceptor class still needs to ne known at runtime, but there is no need to perform the interceptor initialization.

3. Instead of using interceptors, advice api which inlines the code to the class itself may be used.

Advice API

Advices are another approach how code can be instrumented in Byte Buddy. Compared to `Interceptor`s it is more limited, but on the other hand, in cases where it's possible to use it, the code is in-lined into the class's bytecode and therefore no other dependencies are required. It is also stated in Byte Buddy documentation that performance of Advice API is better compared to using interceptors.

However, instrumentation using Advice API is only allowed before or after the matched method which is achieved using the `Advice.onMethodEnter` and `Advice.onMethodExit` annotations.

2.5 Communication Middleware

This thesis consist of several parts which need be able to communicate. The communication is also complicated by the fact that the parts are written in different programming languages - Java and C++. In order to achieve communication in such an environment, following libraries has been inspected.

2.5.1 Raw Sockets

We are not referring to a library but to using raw sockets on their low-level API. Using raw sockets has several pros and cos. It give us a full flexibility and the highest possible performance since there isn't any additional layer between our application data and the socket itself. However, integrating different platforms and different languages can be time-consuming. Several frameworks have already been created to achieve this so the user does not need to know about the language or underlying platform.

2.5.2 ZeroMQ

ZeroMq is a communication library build 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, using TCP or also using multicast.

The library also supports to create typologies using one of many supported communication patterns like publisher-subscriber or request-reply.

- Hiding the differences of underlying operating systems.
- Message framing - delivering whole messages instead of stream of bytes
- No need to worry about queuing messages. 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 a topologyy. For example, one socket can be connected to multiple endpoints.
- Automatic TCP re-connection
- Zero-copy

Zero-copy in ZeroMQ

The library also tries to apply concept called zero-copy if 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 is transferring data from memory to network interface or from user application to underlying kernel. We can see that zero-copy can't be implemented at all layers because for example without copying the data from the kernel to network interface, we could not actually exchange any data. 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 plus.

2.5.3 NanoMsg

NanoMsg [http://nanomsg.org/documentation-zeromq.html] is a socket library shadowing the differences in 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++ when using C++ STL containers for example. Also compared to ZeroMQ, objects are not tightly bound to particular threads this it gives the user flexibility to create their custom threading models without big limitations. NanoMsg should also implement zero-copy technique at additional layers which again leads to performance benefits.

As in ZeroMQ, NanoMsq supports the following transport mechanisms:

- **INPROC** 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 It enables 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 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 but backslash. On both mentioned platforms, the address has to start with `ipc://` prefix.
- **TCP** TCP is used to transport messages in a reliable manner to a single recipient to in a reachable network. When connecting to a node, the address in format `tcp://interface:port` needs to used and when binding a node, address in format texttttcp://*:port should be used.

NanoMsg can be used in via it's core C library, but also several language mappings for different languages exist.

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 core library making the

API more C++11 like friendly. Especially, there is no need to tell when to release resources, since it's handled automatically in desctuctors. 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 on technologies on which this thesis heavily depends. Firstly, Java Virtual Machine Tool Interface (JVMTI) is described followed by basic introduction to 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 a interface used by development and monitoring tools for communication with JVM. It allows its user to monitor and control the execution running in Java virtual machine. An application communicating with the JVM using JVMTI is usually called agent. Agents are notified of the events happening inside JVM and can react upon them.

Agents run in the same process as the application itself this reducing the communication. Since JVMTI as interface written in C, agents can be written in C or C++. The agent has to be attached to the application via

JVMTI supports 2 modes how agent can can be started, either in OnLoad phase or in Live phase. In OnLoad phase, 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 can be though as more flexible since we don't have to specify agent library to monitored application in advance, but it brings several limitation as well.

We don't aim to describe full JVMTI functionality here, please consider this just as brief introduction to the interface and inspect the documentation for more information. In the following sections we aim 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. In this method we can do custom initialization for our agent.

Usually the initialization consist of several phases:

1. Optionally, parse arguments passed to 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 we want the JVMTI to support. We can specify what are the operations our JVMTI agent can perform. The agent can be for example allowed to re-transform classes, signal threads or generate different class hook events.
4. Register events we are interested in observing. JVMTI does not inform the agent about all events by default, the events has to be manually defined.
5. Register callbacks for the events we are interested in. In case of agent used for instrumentation we are mostly interested in events `cbClassLoad`, `cbClassPrepare`, `cbClassFileLoadHook`, `callbackVMInit` and `callbackVMDeath`.
6. Optionally, initializing phase is also good for creating various locks required for synchronization between different JVMTI threads.

The user of JVMTI is also required to manually implement queening and locking when processing multiple JVMTI events at the same time since the framework does not handle this. http://www.oracle.com/technetwork/articles/java/jvmpitransition-138768.html

JVMTI basic callbacks

As mentioned above, there are several events send from the observed application. When instrumenting, we are mostly interested in the following events:

- `cbClassLoad` - triggered when class has been loaded by target JVM
- `cbClassPrepare` - triggered when class has been prepared by target JVM. At the point the class is prepared all static fields, methods and implemented interfaces are available 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 happens based on this hook since the callback allows as to change the bytecode passed for further loading.
- `callbackVMInit` - triggered when virtual machine is initialized
- `callbackVMDeath` - triggered when virtual machine has been closed either using standard way or forcibly.

2.6.2 JNI

Java Native Interface is a framework which allows Java code running in a 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 operations require instance of class `JNIEnv`. This environment keeps the connection to the virtual machine. When calling the methods from native application, the method has to be first found. This is achieved by specifying the types and method signature.

Java Types Mapping

For each Java primitive type there is corresponding native type in JNI. Native types always start with the `j` as the prefix, for example `boolean` is Java type whereas `jboolean` as native type. All other JNI reference types are referred to via `jobject` class. This means that java arrays are accessed via `jobject` as well.

The most important is however how we can specify the types in method signatures. There is a mapping giving each type a signature which can be used exactly for this purpose. This table is base on http://docs.oracle.com/javase/7/docs/technotes/guides/j

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

So for example the method:
`xx.yy.Person foo(int n; boolean[] arr, String s);` would have 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

On the method bellow we can see how JNI can be used to call a Java method `getClassLoader`.

```
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);
}
```

First we need to get a reference to a method, which we use later for the invocation itself. From performance reasons, it's good practice to cache the references to methods or objects in Java which we access 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 when for dynamically loaded classes. This is followed by explanation of 2 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 runtime. This is achieved by a following process:

1. **Loading** - Load the bytecode from class file
2. **Linking** - Linking is the process of incorporating a new class to the runtime state of the JVM. It phase consists of 3 sub-phases:
   (a) **Verification** - Ensure that type in the binary former is correct and respects JVM restrictions.
   (b) **Preparation** - This phase consist of allocation 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 being actually used. Constant pool contains all references to variables and methods found during compilation time.
3. **Linking Phase** - class variables are initialized to initial values

Loading step loads data from class files in a binary from known as the bytecode.

Relevant Class Loaders

There are several class loaders used natively in Java. However we describe only 2 which are references in the thesis later.

- **Bootstrap class loader**
  This classloader is used to load system classes. When using native agent, even classes loaded by bootstrap classloader 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 there is a repetition in calling the same method or using the same field via JNI (reflection), it creates synthetic class (class created dynamically at runtime), which is used to perform this call without using 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 some service which is usually defined as set of methods. The service is often defined as abstract class or interface.

ServiceLoader allows us to specify the service type for which we want to load all service providers and then load the desired service providers. The available services have to be defined in the META-INF folder of the application jar distribution. Let's say we have a service A and 2 implementations, Impl1 and Impl2. In that case META-INF folder with contain text file with name A containing lines

```
Impl1
Impl1
```

Service loaders can therefore be used to extend the application without changing the source code. When the user of the application needs to provide another implementation of the service, it can create service provider, register it inside the META-INF folder and the application will use the new service provider as well the rest of the 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 but sometimes it's necessary to have information from various application runs. That is why the selection of logging library is important for the performance of the thesis as well.

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

2.8 Docker

Docker is an open source project to pack, ship and run any application as a lightweight container [citate]. It is used to package the applications in a 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 build on top of docker allowing us to specify multi-container startup-script. This script can define dependencies between different containers which leads to a simple and automated way on how to start whole bunch of related applications in separated environments using one single call.

3. Analysis

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 extendable 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.

3.3 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 2 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.

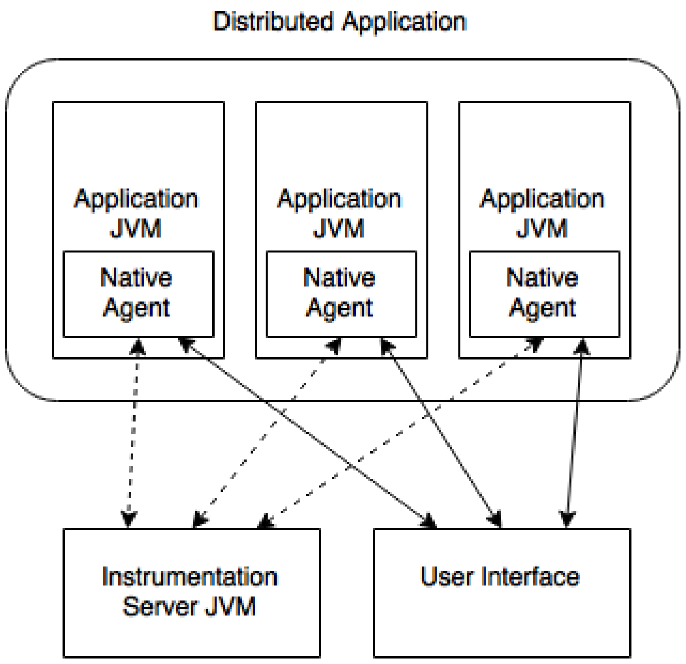

Figure 4.1: Architecture with shared instrumentation server. The dotted lines represents the communication between instrumentation server and the agent whilst the regular lines represents data collection from the agent to the UI

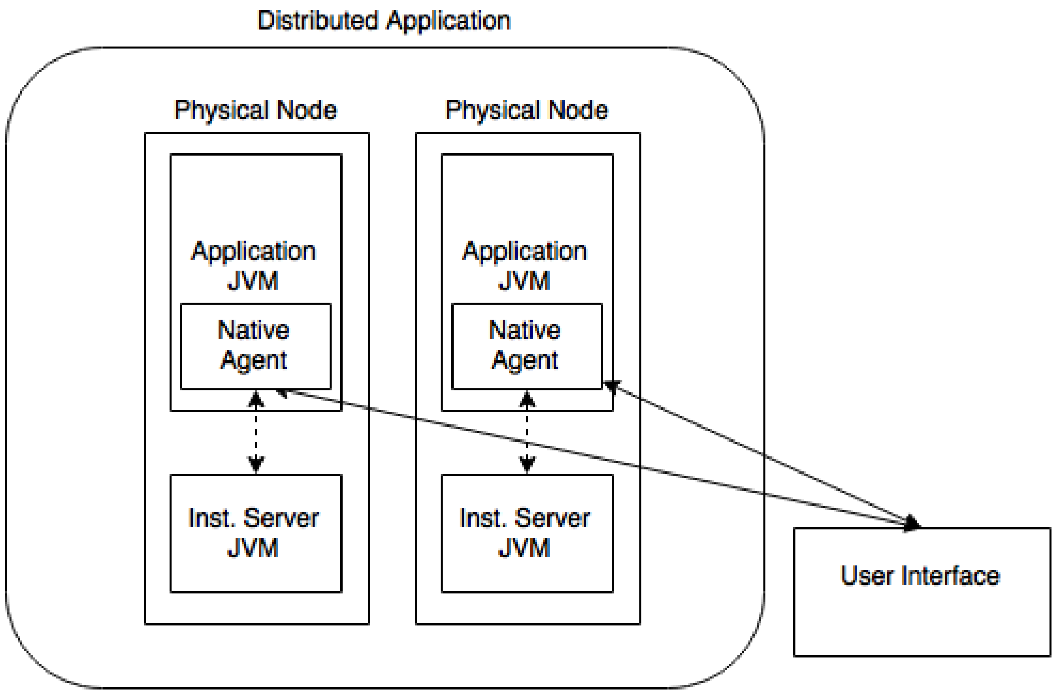

Figure 4.2: Architecture with separated instrumentation server. The meaning of the lines is the same as on the diagram above.

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.

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

5.4.1 Zipkin UI Overview

5.4.2 Zipkin Data Model

5.4.3 Zipkin JSON Format

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