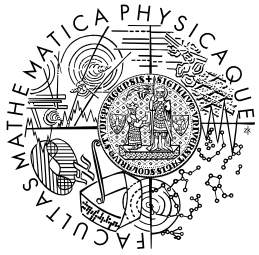

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Monitoring Tool for Distributed Java Applications

Author: 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 **4**
- 1.1 Project Goals . . . 4
- 1.2 Thesis outline . . . 5

2 Background **6**
- 2.1 Cluster Monitoring Tools . . . 6
  - 2.1.1 Google Dapper . . . 6
  - 2.1.2 Zipkin . . . 8
- 2.2 Tools for Large-Scale Debugging . . . 9
- 2.3 Profiling Tools . . . 10
- 2.4 Bytecode Manipulation Libraries . . . 12
  - 2.4.1 ASM . . . 13
  - 2.4.2 Javassist . . . 13
  - 2.4.3 CGlib . . . 13
  - 2.4.4 Byte Buddy . . . 13
  - 2.4.5 Important Annotations . . . 14
- 2.5 Communication Middleware . . . 14
  - 2.5.1 Raw Sockets . . . 14
  - 2.5.2 ZeroMQ . . . 14
  - 2.5.3 NanoMsg . . . 14
- 2.6 Java Libraries . . . 14
  - 2.6.1 Class Initialization Process . . . 14
  - 2.6.2 JVMTI . . . 14
  - 2.6.3 JNI . . . 15
  - 2.6.4 Relevant Class Loaders . . . 15
  - 2.6.5 Service Provider Interface . . . 15
- 2.7 Logging libraries . . . 15
- 2.8 Docker . . . 15
  - 2.8.1 Docker Compose . . . 15
  - 2.8.2 Example Docker Usage . . . 15

3 Analysis **16**
- 3.1 Instrumentation Library . . . 16
- 3.2 NanoMsg . . . 16
- 3.3 Instrumentation approaches . . . 16
  - 3.3.1 Java Agent Solution . . . 16
  - 3.3.2 Native Agent Solution . . . 16
  - 3.3.3 Instrumenting in secondary JVM . . . 16

4 Overview **17**
- 4.1 Architecture Description . . . 17
  - 4.1.1 Native Agent . . . 17
  - 4.1.2 Instrumentation Server . . . 17
- 4.2 Communication . . . 17

5 Design **18**
5.1 Basic Concepts . . . 18
5.1.1 Spans . . . 18
5.2 Native Agent . . . 18
5.2.1 Structure Overview . . . 18
5.2.2 Instrumentation . . . 18
5.2.3 Instrumentation API . . . 18
5.2.4 Native Agent Arguments . . . 18
5.2.5 Used JVMTI Callbacks . . . 18
5.3 Instrumentation Server . . . 18
5.3.1 Instrumentation Protocol . . . 18
5.3.2 Communication Modes . . . 18
5.3.3 Class Caching . . . 18
5.3.4 Custom Service Loader . . . 18
5.3.5 Public interfaces . . . 18
5.3.6 Extending the Server . . . 18
5.3.7 Class Loaders . . . 19
5.3.8 JSON Generation . . . 19
5.4 User Interface . . . 19
5.4.1 Zipkin UI Overview . . . 19
5.4.2 Zipkin Data Model . . . 19
5.4.3 Zipkin JSON Format . . . 19
5.5 Collectors . . . 19

6 Implementation Details **20**
6.1 Native Agent . . . 20
6.1.1 Byte Class Parsing . . . 20
6.1.2 Instrumentation . . . 20
6.2 Instrumentation Server . . . 20
6.2.1 Byte-Code Instrumentation . . . 20
6.3 Zipkin Integration . . . 20

7 Big Example **21**

8 Evaluation **22**
8.1 Known Limitations . . . 22
8.2 Platform demonstration . . . 22
8.2.1 Deployment Strategies . . . 22
8.2.2 Basic Building Blocks . . . 22
8.2.3 Basic Demonstration . . . 22
8.2.4 Optimizing the Solution . . . 22

9 Conclusion **23**
9.1 Comparison to Related Work . . . 23
9.2 Future plans . . . 23
9.2.1 Integration with well-known data collectors . . . 23
9.2.2 Add support for Flame charts . . . 23

Bibliography **25**

List of Figures **26**

List of Tables **27**

List of Abbreviations **28**

Attachments **29**

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.

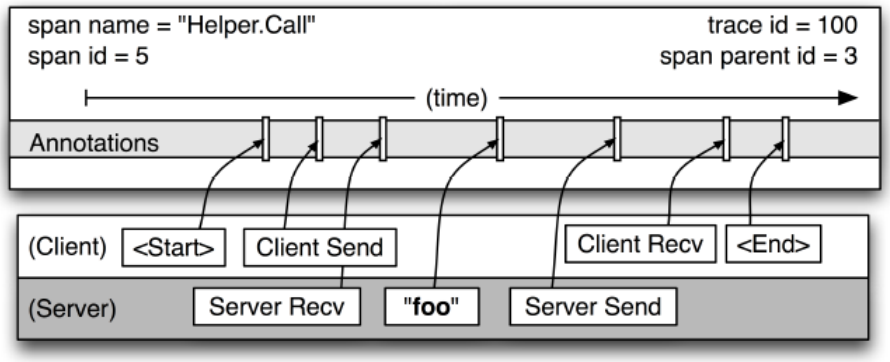

Figure 2.1: Example of Span in Google Dapper. Picture taken from the Google Dapper paper

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.

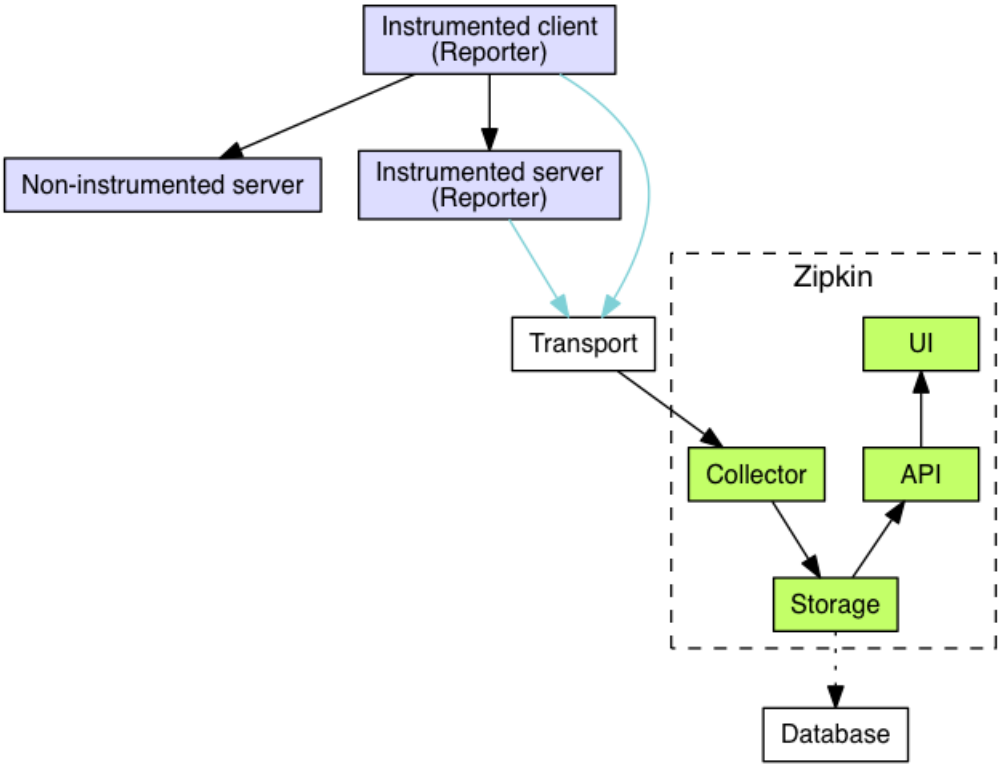

Figure 2.2: Zipkin architecture - http://zipkin.io/pages/architecture.html

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.

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.

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.
- 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

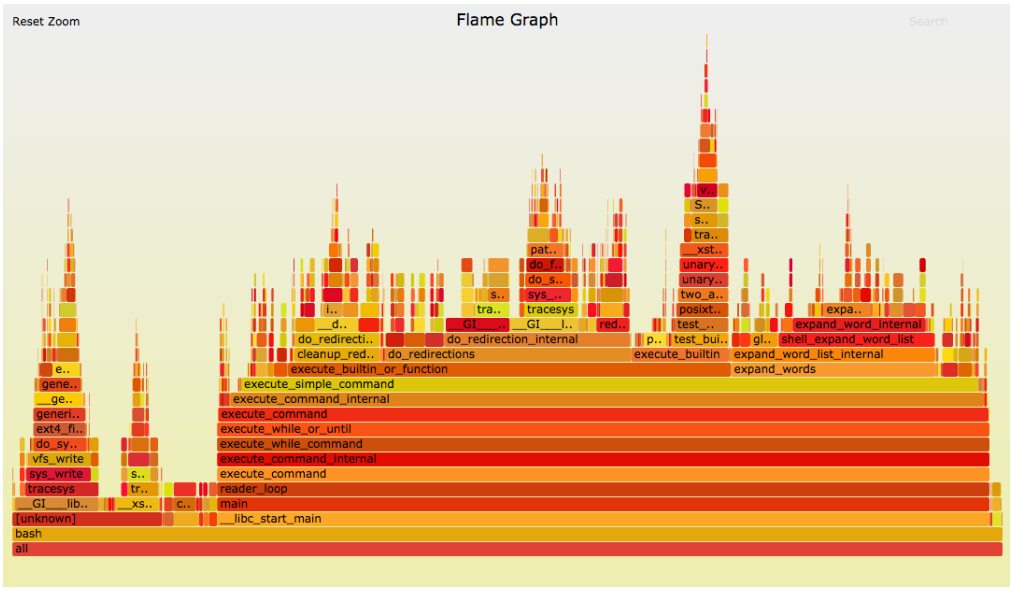

Figure 2.3: Flame Graph example

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 instrumentation 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 byte-code 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.

Main Concept

Transformers

Interceptors

Class File Locator

Advice API

Selected ByteBuddy Internals

Auxiliary Classes

Initializer Classes

2.4.5 Important Annotations

2.5 Communication Middleware

give comparison between the possible communication middle-wares

2.5.1 Raw Sockets

2.5.2 ZeroMQ

2.5.3 NanoMsg

API Overview

Available Communication Modes

Language Mappings

C++11 Mapping

Java Mapping

2.6 Java Libraries

2.6.1 Class Initialization Process

2.6.2 JVMTI

JVMTI Overview

Basic Hooks

..maybe more subsubsections later

2.6.3 JNI

JNI Overview

Java Types Mapping

Example Java Calls C++

..maybe more subsubsections later

2.6.4 Relevant Class Loaders

2.6.5 Service Provider Interface

2.7 Logging libraries

spd log on c++ side and logging library used

2.8 Docker

2.8.1 Docker Compose

2.8.2 Example Docker Usage

used for easy of use

3. Analysis

mention comparison of different approaches to instrumenting Java applications.

3.1 Instrumentation Library

Describe why bytebuddy has been chosen

3.2 NanoMsg

Descrive why NanoMsg has been chosen

3.3 Instrumentation approaches

3.3.1 Java Agent Solution

3.3.2 Native Agent Solution

3.3.3 Instrumenting in secondary JVM

give introduction to various instrumentation techniques and compare the 2 approaches

4. Overview

4.1 Architecture Description

4.1.1 Native Agent

4.1.2 Instrumentation Server

4.2 Communication

5. Design

5.1 Basic Concepts

5.1.1 Spans

5.2 Native Agent

mention here the issue with running more JVMs inside one process

5.2.1 Structure Overview

5.2.2 Instrumentation

mention issues with circular dependencies but leave how it is implemented into the next chapter

5.2.3 Instrumentation API

5.2.4 Native Agent Arguments

5.2.5 Used JVMTI Callbacks

5.3 Instrumentation Server

5.3.1 Instrumentation Protocol

5.3.2 Communication Modes

5.3.3 Class Caching

5.3.4 Custom Service Loader

5.3.5 Public interfaces

5.3.6 Extending the Server

.. instrumentation server can run on the same node or over the network. Instrumentation server can have client code attached or not.

5.3.7 Class Loaders

5.3.8 JSON Generation

5.4 User Interface

5.4.1 Zipkin UI Overview

5.4.2 Zipkin Data Model

5.4.3 Zipkin JSON Format

5.5 Collectors

Should I mention the collectors ? It may be sufficient to have send data right to zipkin for demonstration purposes

6. Implementation Details

Mention interesting parts of the implementation

6.1 Native Agent

6.1.1 Byte Class Parsing

6.1.2 Instrumentation

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!

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.

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

List of Figures

2.1 Example of Span in Google Dapper. Picture taken from the Google Dapper paper 7
2.2 Zipkin architecture - http://zipkin.io/pages/architecture.html . . 8
2.3 Flame Graph example 12

List of Tables

List of Abbreviations

Attachments