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:	Mgr. 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: Mgr. 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 Mgr. Pavel Parízek, Phd. 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 Mgr. Michal Malohlava, Phd. 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 reasoning about distributed applications is inherently more complex. In case of single node application we can use standard monitoring or profiling tools which give us data to reason about the application, but in the complex cluster solutions this can't be applied. We could use standard monitoring tools per each node in the cluster 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.

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.

1.1 Project Goals

The project should make some trade-offs between application level transparency and easiness of use. Therefore, we can think of this solution more as a 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. So the user-programmer has to extend the prepared library for their specific application by defining points where instrumentation should take place, but the original application can still remain unaffected.

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. Also, 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.

One of the most important aspects of the introduced tool is the deployment. In order for developers and testers to use this tool, it needs to be really easy to deploy 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 2 files - native agent which needs to be attached to monitored application and Instrumentation server written in java which handles the instrumentation for the whole cluster application. Several deployment strategies exist and are discussed later in the 7 chapter.

2. Analysis

This chapter gives overview of two significant related platforms on which this work depends - Google Dapper and Zipkin. It continues with description of several background concepts and tools which are to some level relevant to the thesis, such as tools for large scale debugging, tools for visualizing the monitored data and also several profiling tools and their comparison. The libraries for bytecode instrumentation and different communication middle-wares are described in detail as the selected libraries affect the platform at very low level. This chapter ends with a comparison of different approaches to instrumenting Java applications.

2.1 Related Work

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.

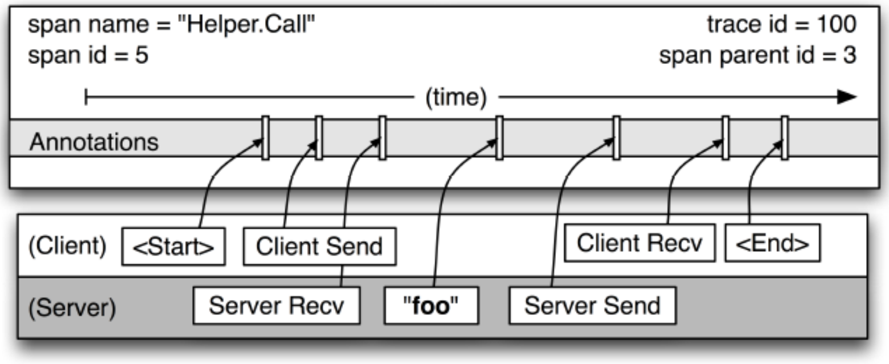

Figure 2.1: Example of Span in Google Dapper. Picture taken from the Google Dapper paper

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.

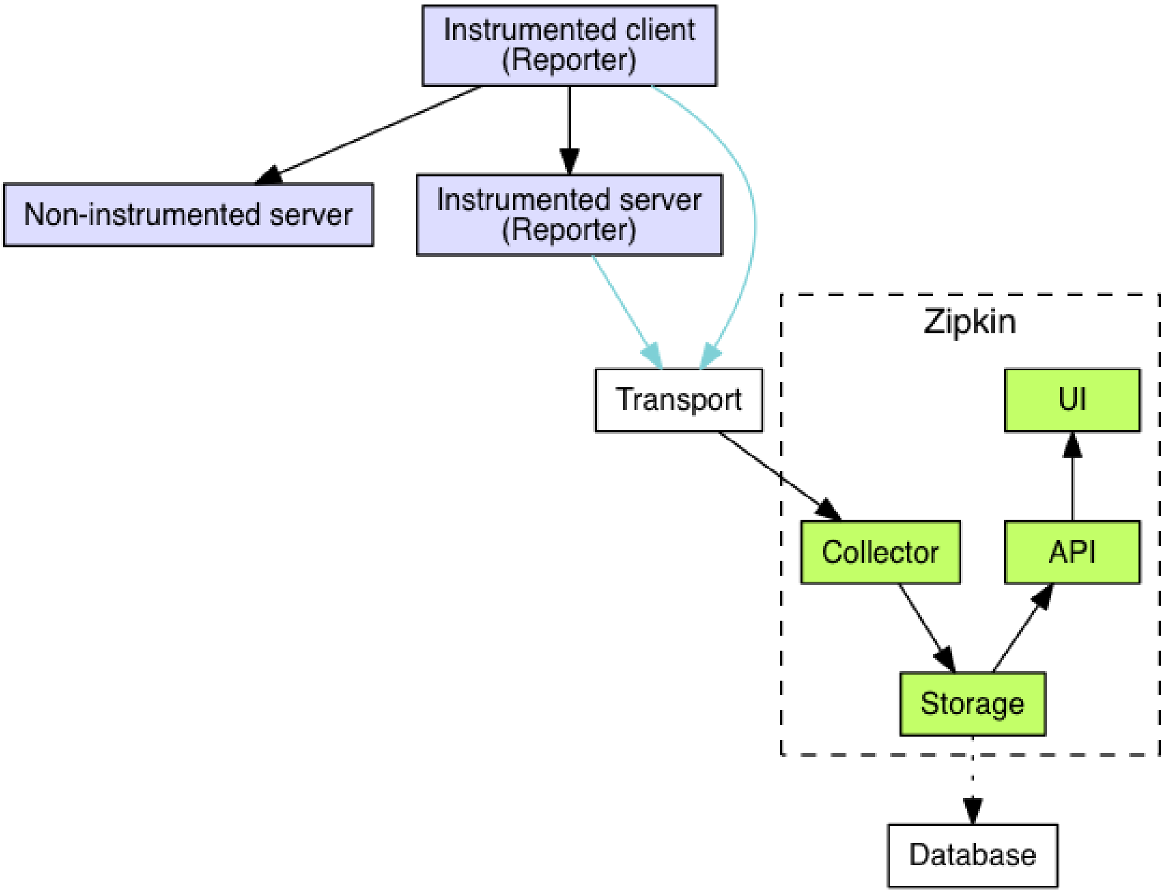

Figure 2.2: Zipkin architecture - http://zipkin.io/pages/architecture.html

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 Background and Tools

This section covers related tools to the thesis. They are not necessarily used in the thesis, but are important tools used for the monitoring and debugging various kind of applications. The most important tools mentioned in this section are the ones for debugging large scale applications, tools for visualizing the captured data and the most importantly, two kinds of profiling tools.

2.2.1 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 causing the anomaly.

Graphite and Graphana

Flame Charts

2.2.2 Profiling Tools

System Profilers

JVM Profilers

Write about AsyncGetCallTrace

2.3 Instrumentation Libraries

2.3.1 Javassist

2.3.2 ByteBuddy

2.3.3 CGlib

2.3.4 ASM

.. just give brief overview what were the instrumentation libreries choices. The selected one will be describied in the next section

2.4 Communication Middleware

give comparison between the possible communication middle-wares

2.4.1 Raw Sockets

2.4.2 ZeroMQ

2.4.3 NanoMSG

2.5 Comparison of Agent Approaches

give introduction to various instrumentation techniques and compare the 2 approaches

2.5.1 Java Agent Solution

2.5.2 Native Agent Solution

3. Related Technologies

This chapter will talk about selected technologies. It will not say why we chose this particular technology since it's done in the previous section, but will talk about particular technology aspects in more details

3.1 Java

3.1.1 Class Initialization Process

3.1.2 JVMTI

JVMTI Overview

Basic Hooks

..maybe more subsubsections later

3.1.3 JNI

JNI Overview

Java Types Mapping

Example Java Calls C++

..maybe more subsubsections later

3.1.4 Relevant Class Loaders

3.1.5 Service Provider Interface

3.2 ByteBuddy

3.2.1 Main Concept

3.2.2 Transformers

3.2.3 Interceptors

3.2.4 Class File Locator

3.2.5 Advice API

3.2.6 Selected ByteBuddy Internals

Auxiliary Classes

Initializer Classes

3.2.7 Important Annotations

3.3 NanoMsg

3.3.1 API Overview

3.3.2 Available Communication Modes

3.3.3 Language Mappings

C++11 Mapping

Java Mapping

3.4 spdlog

logging library used

3.5 Docker

Docker Compose

Example Docker Usage

used for easy of use

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. Evaluation

7.1 Known Limitations

here mention limitations with the instrumentation

7.2 Platform demonstration

7.2.1 Deployment Strategies

Instrumentor per Application Node

Instrumentor per Whole Cluster

Optimizing the Deployment

7.2.2 Basic Building Blocks

7.2.3 Basic Demonstration

7.2.4 Optimizing the Solution

8. Conclusion

8.1 Comparison to Related Work

8.2 Future plans

8.2.1 Integration with well-known data collectors

8.2.2 Add support for Flame charts

An example citation: **?**

List of Figures

List of Tables

List of Abbreviations

Attachments