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

1.1 Project Goals

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 Spans 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 of both of the mentioned platform.

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.

Google Dapper collects so called distributed traces. The origin of the distributed trace is the communication/task initiator. This is demonstrated at the figure ...

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.

Trace trees and spans

2.1.2 Zipkin

2.2 Background and Tools

2.2.1 Tools for Large-Scale Debugging

2.2.2 Tools for Visualizations the Captured Data

Flame Charts

Graphite and Graphana

2.2.3 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 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