Master's Degree in Informatics Engineering

Final Dissertation

Observing and Controlling Performance in Microservices

Author:

André Pascoal Bento

Supervisor:

Prof. Filipe João Boavida Mendonça Machado Araújo

Co-Supervisor:

Prof. António Jorge Cardoso

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

July 2019

This page is intentionally left blank.

Abstract

Microservice based software architecture are growing in usage and one type of data generated to keep history of the work performed by this kind of systems is called tracing data. Tracing can be used to help Development and Operations (DevOps) perceive problems such as latency and request work-flow in their systems. Diving into this data is difficult due to its complexity, plethora of information and lack of tools. Hence, it gets hard for DevOps to analyse the system behaviour in order to find faulty services using tracing data. The most common and general tools existing nowadays for this kind of data, are aiming only for a more human-readable data visualisation to relieve the effort of the DevOps when searching for issues in their systems, however these tools do not provide good ways to filter this kind of data neither perform any kind of tracing data analysis and therefore, they do not automate the task of searching for any issue presented in the system, which stands for a big problem because they rely in the system administrators to do it manually. In this thesis is present a possible solution for this problem, capable of use tracing data to extract metrics of the services dependency graph, namely the number of incoming and outgoing calls in each service and their corresponding average response time, with the purpose of detecting any faulty service presented in the system and identifying them in a specific time-frame. Also, a possible solution for quality tracing analysis is covered checking for quality of tracing structure against OpenTracing specification and checking time coverage of tracing for specific services. Regarding the approach to solve the presented problem, we have relied in the implementation of some prototype tools to process tracing data and performed experiments using the metrics extracted from tracing data provided by Huawei. With this proposed solution, we expect that solutions for tracing data analysis start to appear and be integrated in tools that exist nowadays for distributed tracing systems.

Keywords

Microservices, Cloud Computing, Observability, Monitoring, Tracing.

This page is intentionally left blank.

Resumo

A arquitetura de software baseada em micro-serviços está a crescer em uso e um dos tipos de dados gerados para manter o histórico do trabalho executado por este tipo de sistemas é denominado de tracing. Mergulhar nestes dados é díficil devido à sua complexidade, abundância e falta de ferramentas. Consequentemente, é díficil para os DevOps de analisarem o comportamento dos sistemas e encontrar serviços defeituosos usando tracing. Hoje em dia, as ferramentas mais gerais e comuns que existem para processar este tipo de dados, visam apenas apresentar a informação de uma forma mais clara, aliviando assim o esforço dos DevOps ao pesquisar por problemas existentes nos sistemas, no entanto estas ferramentas não fornecem bons filtros para este tipo de dados, nem formas de executar análises dos dados e, assim sendo, não automatizam o processo de procura por problemas presentes no sistema, o que gera um grande problema porque recaem nos utilizadores para o fazer manualmente. Nesta tese é apresentada uma possivel solução para este problema, capaz de utilizar dados de tracing para extrair metricas do grafo de dependências dos serviços, nomeadamente o número de chamadas de entrada e saída em cada serviço e os tempos de resposta coorepondentes, com o propósito de detectar qualquer serviço defeituoso presente no sistema e identificar as falhas em espaços temporais especificos. Além disto, é apresentada também uma possivel solução para uma análise da qualidade do tracing com foco em verificar a qualidade da estrutura do tracing face à especificação do OpenTracing e a cobertura do tracing a nível temporal para serviços especificos. A abordagem que seguimos para resolver o problema apresentado foi implementar ferramentas protótipo para processar dados de tracing de modo a executar experiências com as métricas extraidas do tracing fornecido pela Huawei. Com esta proposta de solução, esperamos que soluções para processar e analisar tracing comecem a surgir e a serem integradas em ferramentas de sistemas distribuidos.

Palavras-Chave

Micro-serviços, Computação na nuvem, Observabilidade, Monitorização, Tracing.

This page is intentionally left blank.

Acknowledgements

This work wouldn't be possible to be accomplished without the help and support of a lot of people. Thus, in this section I'd like to give my sincere thanks to all of them.

Starting by giving my thanks to my mother and to my whole family who have supported me through my entire academic course, and who always gave and always will give me some of the most important and beautiful things in life, love and friendship.

In second place, I would like to thank all the people that were involved directly to this project. To my supervisor, Professor Filipe Araújo, who contributed with his vast wisdom and experience, to my co-supervisor, Professor Jorge Cardoso, who contributed with is vision and guidance about the main road we should take and to Engineer Jaime Correia, who "breathes" these kind of topics through him and helped a lot with is enormous knowledge and enthusiasm.

In third place, I would like to thank the Department of Informatics Engineering and the Centre for Informatics and Systems, both from the University of Coimbra, for allowing and provide the resources and facilities for this project to be carried out.

In fourth place, to the Foundation for Science and Technology, for financing this project facilitating its accomplishment and to Huawei, for providing data, core for this whole research.

And finally, my sincere thanks to everyone that I have not mentioned and contributed to everything that I am today.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

This document presents the *Master Thesis* in *Informatics Engineering* of the student *André Pascoal Bento* during the school year of 2018/2019, taking place in the *Department of Informatics Engineering (DEI)* of the *University of Coimbra*.

1.1 Context

In today's world, software systems tend to become more distributed as time move on, resulting in new approaches that lead to new solutions and new patterns of developing software. One way to solve this is to develop systems that have their components decoupled, creating software with "small pieces" connected to each other that encapsulate and provide a specific function in the larger service. This way of developing software is called Microservices and has become mainstream in the enterprise software development industry [1]. However, with this kind of approach, the systems complexity is increased as a whole because with more "small pieces", more connections are needed and with this more problems related to latency and requests become harder to detect, analyse and correct [2].

To keep a history of the work performed by this kind of systems, multiple techniques like monitoring [3], logging [4] and tracing [5] are adopted. Monitoring consists on measuring some aspects like, e.g., Central Processing Unit (CPU) usage, hard drive usage and network latency of the entire system or of some specific node in a distributed system. Logging provides an overview to a discrete, event-triggered log. Finally, tracing is much similar to logging, however the focus is registering the flow of execution of the program through several system modules and boundaries. Lastly, distributed tracing, shares the focus on preserving causality relationships, however, is geared towards the modern distributed environments, where state is partitioned over multiple, threads, processes, machines and even geographical locations. This last one is better explained in Subsection 3.1.3 - Distributed Tracing. There are multiple approaches to gather information of this kind of systems, each with its benefits and disadvantages.

The main problem with this nowadays is that there are not many implemented tools for processing tracing data and none for performing analysis of this type of data. For monitoring it tend to be easier, because data is represented in charts and diagrams, however for logging and tracing it gets harder to manually analyse the data due to multiple factors like its complexity, plethora and increasing quantity of information. There are some visualisation tools for the Development and Operations (DevOps) to use, like the ones presented in Subsection 3.3.1 - Distributed Tracing Tools, however none of them gets

to the point of performing the analysis of the system using tracing, has they tend to be developed only for visualisation and display of tracing data in a more human readable way. Nevertheless, this is critical information about the system behaviour, and thus there is the need for performing automatic tracing analysis.

1.2 Motivation

The motivation behind this work resides in exploring and develop ways to perform tracing analysis in microservice based systems. The analysis of this kind of systems tend to be very complex and hard to perform due to their properties and characteristics, as it is explained in Subsection 3.1.1 - Microservices, and to the type of data to be analysed, presented in Subsections 3.1.3 - Distributed Tracing and 3.1.4 - Traces and Spans.

DevOps teams have lots of problems when they need to identify and understand problems with this systems. They usually detect the problems when the client complains about the quality of service, and after that DevOps dive in monitoring metrics like, e.g, CPU usage, usage, hard drive usage and network latency, and then in distributed tracing data visualisations and logs to find some explanation to what is causing the reported problem. This involves a very hard and tedious work of look-up through lots of data that represents the history of work performed by the system and, in most cases, this tedious work reveals like a big "find a needle in the haystack" problem. Some times, DevOps can only perceive problems in some services and end up "killing" and rebooting these services which is wrong, however, due to lack of time and difficulty in identifying anomalous services precisely this is the best known approach.

Problems regarding the system operation are more common in distributed systems and their identification must be simplified. This need of simplification comes from the exponential increase in the amount of data needed to retain information and the increasing difficulty in manually managing distributed infrastructures. The work presented in this thesis, aims to perform a research around these needs and focus on presenting some solutions and methods to perform tracing analysis.

1.3 Goals

The main goals for this thesis consists on the main points exposed bellow:

1. Search for existing technology and methodologies used to help DevOps teams in their current daily work, with the objective of gathering the best practices about handling tracing data. Also, we aim to understand how these systems are used, what are their advantages and disadvantages to better know how we can use them to design and produce a possible solution capable of performing tracing analysis. From this we expect to learn the state of the field for this research, covering the core concepts related work and technologies, presented in Chapter 3 - State of the Art.
2. Perform a research about the main needs of DevOps teams, to better understand what are their biggest concerns that lead to their approaches when performing pinpointing of microservices based systems problems. Relate these approaches with related work in the area, with the objective of understanding what other companies and groups have done in the field of automatic tracing analysis. The processes used to tackle this type of data, their main difficulties and conclusions provide a

better insight about the problem. From this we expected to have our research objectives clearly defined and a compilation of questions to be evaluated and answered, presented in Chapter 4 - Research Objectives and Approach.

3. Reason about all the gathered information, design and produce a possible solution that provides a different approach to perform tracing analysis. From this we expect first to propose a possible solution, presented in Chapter 5. The we implement it using state of the art technologies, feed it with tracing data provided by Huawei and collect results, presented in Chapter 6 - Implementation Process. Finally, we provide conclusions to this research work in the last Chapter 8 - Conclusions.

1.4 Research Contributions

From the work presented on this thesis, the following research contributions were made:

- Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo and Jorge Cardoso. On the Limits of Automated Analysis of OpenTracing. International Symposium on Network Computing and Applications (IEEE NCA 2019) (The paper is waiting review).

1.5 Document Structure

This section presents the document structure in this report, with a brief explanation of the contents in every section. The current document contains a total of seven chapters, including this one, Chapter 1 - Introduction. The remaining six of them are presented as follows:

- In Chapter 2 - Methodology are presented the elements involved in this work, with their contributions, has well as the work plan, with "foreseen" and "real" work plans comparison and analysis.
- In Chapter 3 - State of the Art the current state of the field for this kind of problem is presented. This chapter is divided in three sections. The first one, Section 3.1 - Concepts introduces the reader to the core concepts to know as a requirement for a full understanding of the topics discussed in this thesis. The second, Section 3.2 - Related Work presents the reader to related researches produced in the field of distributed tracing data handling. Finally, Section 3.3 - Technologies presents the result of a research for current technologies, that may be able to help solving this problem and produce a proposed solution to be implemented.
- In Chapter 4 - Research Objectives and Approach we present how we tackled this problem, the main difficulties that were found and the objectives involved to solve the issues that are presented. Also, in this chapter, a compilation of questions are presented and evaluated with some reasoning about possible ways to answer them.
- In Chapter 5 - Possible Solution a possible solution for the presented problem is exposed and explained in detail. This chapter is divided in four sections. The first one, Section 5.1 - Functional Requirements, expose the functional requirements with their corresponding priority levels and a brief explanation to every single one

of them. The second one, Section 5.2 - Quality Attributes, contains the gathered non-functional requirements that were used to build the solution architecture. The third one, Section 5.3 - Technical Restrictions, presents the defined technical restrictions for this project. The last one, Section 5.4 - Architecture, presents the possible solution architecture using some representational diagrams, and ends with an analysis and validation to check if the presented architecture meets up the restrictions involved in the architectural drivers.

- In Chapter 6 - Implementation Process, the implementation process of the possible solution is presented with detail. This chapter is divided in three main sections covering the whole implementation process, from the input data set through the pair of components presented in the previous chapter. The first one, Section 6.1 - Huawei Tracing Data Set, the tracing data set provided by Huawei to be used as the core data for research is exposed with some detail. Second, in Section 6.2 - Open Tracing Processor Component we present the possible solution for the first component, namely "Graphy OpenTracing processor (OTP)", that processes and extracts metrics from tracing data. The final Section 6.3 - Data Analysis Component presents the possible solution for the second component, namely "Data Analyser", that handles data produced by the first component and produces the analysis reports. Also, in the last two sections presented, the used algorithms and methods in the implementations are properly detailed and explained.

- In Chapter 7 - Research Objectives and Approach, the gathered results, corresponding analysis and limitations of tracing data are presented. This chapter is divided in three main sections. The first one, Section 7.1 - Anomaly Detection, the results regarding the gathered observations on the extracted metrics of anomalous service detection are presented and explained. Second, in Section 7.2 - Trace Quality Analysis the results obtained from the quality analysis methods applied to the tracing data set are presented and explained. The final Section 7.3 - Limitations of OpenTracing Data we present the limitations felted when designing a solution to process tracing data, more precisely OpenTracing data.

- Last, in Chapter 8 - Conclusions, the main conclusions for this research work are presented. The chapter is divided in three main sections. First, Section 8.1 - Brief Reflections, a reflection about the implemented tools, methods produced and the open paths from this research are exposed. Also a reflection of the main difficulties felted with this research regarding the handling of tracing data are presented. Second, Section 8.2 - Future Work, the future work that can be addressed considering this work is properly explained taking into consideration what is said in the previous section. Finally, Section 8.3 - Concluding Research Questions, the state of answers for the selected questions defined in this research are discussed.

Next Chapter 2 - Methodology, the elements involved in this work, their contributions and work plans for this research project are presented.

Chapter 2

Methodology

The methodology of work carried out in this research project is presented in this chapter. First, every member involved will be mentioned as well as their individual contribution for the project. Second, the adopted approach and process organisation of the collaborators involved will be explained. Finally, the work plan as well as the work performed, including the foreseen and real work plans for the whole year of work are presented.

The main people involved in this project were myself, André Pascoal Bento, student at the Master course of Informatics Engineering at Department of Informatics Engineering (DEI), who carried out the investigation and development of the project. In second, Prof. Filipe Araújo, assistant professor at the University of Coimbra, who contributed with his vast knowledge and guidance on topics about distributed systems and cloud computing. In third, Prof. Jorge Cardoso, Chief Architect for Intelligent CloudOps at Huawei Technologies, who contributed with his vision, great contact with the topics addressed in this work and with the tracing data set from Huawei Cloud Platform [6]. In fourth, Eng. Jaime Correia, doctoral student at DEI, who contributed with his vast technical knowledge regarding the topics of tracing and monitoring microservices.

CONTINUE FROM HERE!!!

As this work stands for an investigation, it was necessary to perform an exploratory work and there were no clear development methodology adopted. Instead of a development methodology, meetings were scheduled in the beginning to happen every two weeks. In this meetings, the people mentioned in the previous paragraph were gathered together to discuss the work carried out in the last two weeks and topics like the information gathered, the analysis about some existing tools, ideas and solutions were discussed between all. In the end, although there wasn't defined some development methodology, the meeting that were carried out were more than enough to keep the productivity and good work.

For the work plan and starting by some numbers, the time spent in each semester of the year by week are sixteen hours for the first semester, and forty hours for the second one. In the end, it was spent a total of 304 hours for the first semester, starting in 11.09.2018 and ending in 21.01.2019 (19 weeks times 16 hours per week), and it is expected to be spent a total of 840 hours for the second semester, starting in 04.02.2019 and ending in 30.06.2019 (21 weeks times 40 hours per week).

In the beginning, there was a work plan for two semesters given in the project proposition. For record, these plans are presented in Figure 2.1.

For effects of analysis, the real work plan carried out in the first semester is presented

in Figure 2.2.

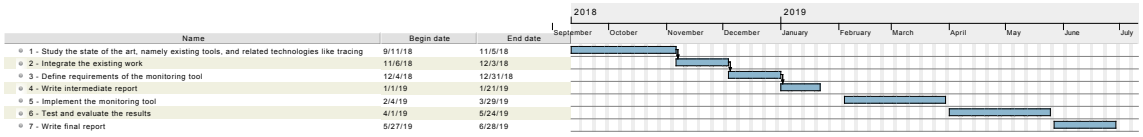

Figure 2.1: Proposed work plan for the first and second semesters.

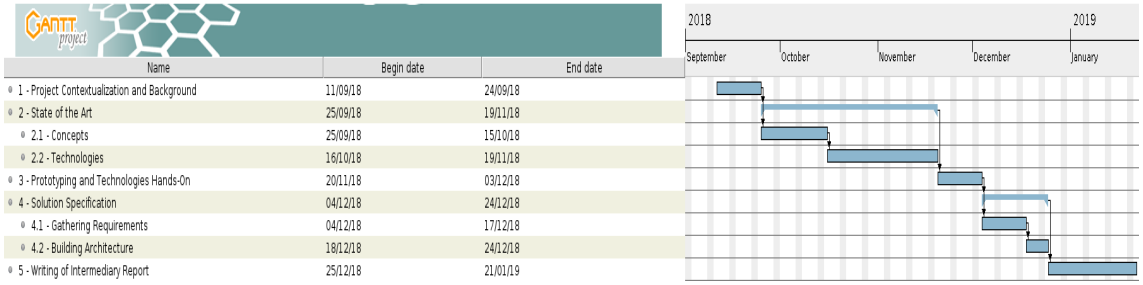

Figure 2.2: Real work plan for the first semester.

As we can see, the "foreseen" work plan for the first semester has suffered some changes, when comparing it to the real work plan. The predicted task 1 - Study the state of the art(...), was branched into two 1 - Project Contextualisation and Background and 2 - State of the Art, and took some more time to do because of the non-concrete and lack of documentation in the technologies related to the subject of this thesis. The predicted task 2 - Integrate the existing work, was replaced by 3 - Prototyping and Technologies Hands-On. This replacement was done because of the interest in test the technologies gathered in the state of the art and see some results with them, enhancing our investigation work and allowing us to get a better visualisation of the data that we had back then. The remaining tasks took almost the predicted time to do.

Finally, it was generated a foreseen work plan for the second semester even knowing that, with almost one hundred percent of certainty the work plan will change when we perform the real work, it is presented in the figure 2.3. This work plan was generated taken into account the proposed work presented in the figure **??** and the effort needed to implement the defined solution presented in the chapter 5 - Possible Solution. To estimate the effort for each task we decided to, first group tasks by four groups regarding their complexity and work load, second discuss if they were in the right group taking into consideration what is defined in the solution and the task background knowledge, and third assign the defined values to each group. This values represent the work days for each task, and in this case we defined the following four: 3(three), 5(five), 8(eight) and 12(twelve) working days as they are akin to the Fibonacci suites[7]. The only task that were not submitted to this, was the task 3 - Write the final report.

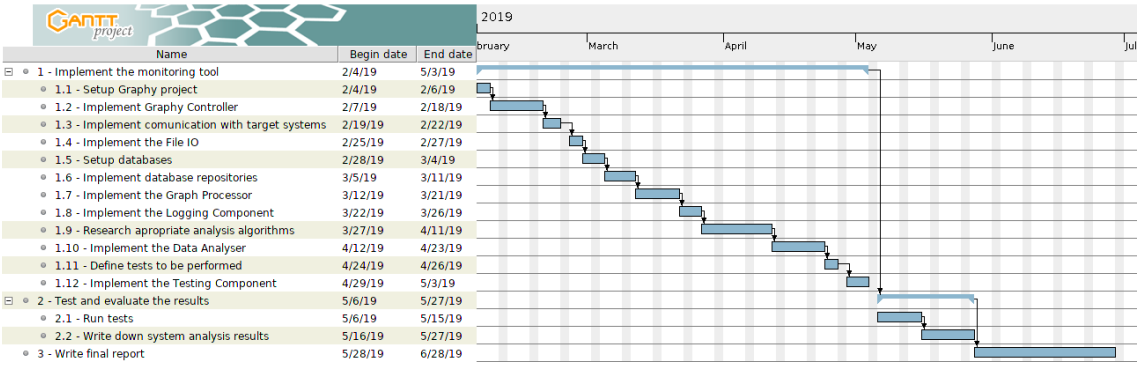

Figure 2.3: Foreseen work plan for the second semester.

Figure 2.4: Real work plan for the second semester.

All the figures to expose the work plans have been created by an open-source tool called GanttProject[8] that produce Gantt charts, a kind of diagram used to illustrate the progress of the different stages of a project.

This page is intentionally left blank.

Chapter 3

State of the Art

In this chapter, we discuss the core concepts regarding the project, related work and the most modern techniques and available technology for the purpose today. All the information that will be presented was the result of a work of investigation through published articles, knowledge exchange and web searching.

The main purpose of the section 3.1 - Concepts is to introduce and provide a brief explanation about the core concepts. In the second section 3.2 - Related Work some published articles and posts of related work are presented. In the final section 3.3 - Technologies, all the important technologies are analysed and discussed using tables, diagrams and plain text.

3.1 Concepts

The following concepts represents the baseline to understand the work related to this project.

3.1.1 Microservices

Microservices is "an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities"[9].

This kind of style has a very long history, and has being introduced and evolving since the first contact with topics like distributed computing, Application Programming Interface (API) and containers.

The core concept of microservices stands in isolation, or by other words, what everyone wants to achieve when building a software with microservices in mind, is to share less things between the services and deal with correlated failures. In this sense, a service is a small part of the entire system (e.g. Get messages microservice), and represents a tiny feature of the whole service (e.g. Chat Service). To do this, normally every microservice is encapsulated inside a container (e.g. Docker container[10]), and each runs in its own process and communicates with the other using lightweight mechanisms, often an Hypertext Transfer Protocol (HTTP) resource API. "A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another"[11].

On the other side, and for comparison purposes, we have another very well known architectural style, the monolithic. This style has a logically modular architecture, and the services are packaged and deployed in a single application using a single code base. To compare both architectural styles presented before we have the figure 3.1 that shows and provides a more clear insight about the differences between them.

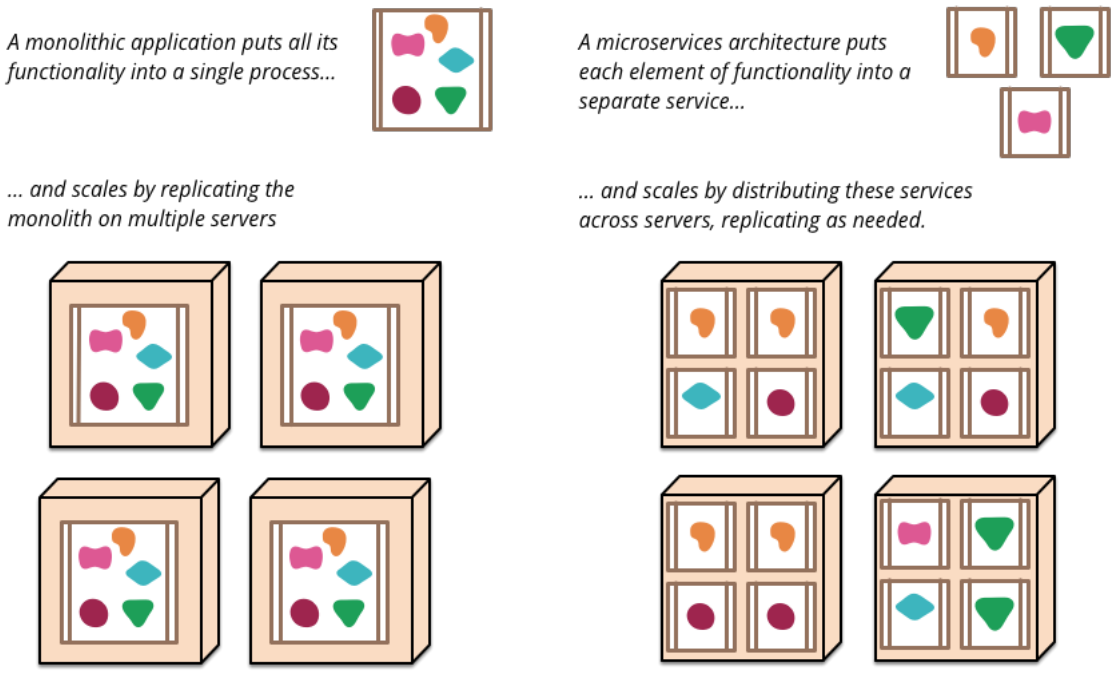

Figure 3.1: Monolithic and Microservices architectural styles[12].

Every style presented has its owns pros and throwbacks and its usages benefit from case to case. A brief example of pros and throwbacks of each one is: for very large teams developing a big and complex service that needs to scale, its good to use the microservices architectural style, because they can tackle the problem of complexity by decomposing application into a set of manageable services which are much faster to develop by individual members, and therefore its much easier to understand and maintain, however its harder to assemble and perform the deployment of the whole service composed by tinny granular parts. In the monolithic architecture, it is normally simpler to deploy, because we just have to copy a single packaged application to a server and run it, however when adding features to the application and it starts to grow in complexity, it gets harder to fully understand and made changes fast and correctly.

Microservices are simple enough to understand, but very hard to master in practice. Some people tend to think that microservices architecture reduce the complexity of the overall system over the monolithic architecture, however this isn't always like this. Communication is a big challenge in the realm of microservices and with a solution that integrates lots of interactions and connections throughout the whole system, the complexity starts to rise. Adding more microservices implies adding more complexity to the solution, and we have to be sure that new microservices can scale together with our existing ones.

"If you can not manage building a monolith inside a single process, what makes you think putting network in the middle is going to help?", Simon Brown[13].

Has been said before, the implementation of a microservices architecture raises the

complexity of the solution, and we need to be aware of that when we design and implement a solution based on this architectural stile. On the other side, after having the system running up, the debbug process is a lot more complex too because, instead of having a single unit to analyse, we may have hundreds or even thousands of "micro-units" to trace and analyse. This kind of problem takes us to an important topic, the observation and control of the system performance, taking into consideration that this system is a microservice based system.

3.1.2 Observability and Controlling Performance

Observing is "to be or become aware of, especially through careful and directed attention; to notice"[14].

Observability is an extension of observing and its definition is the following: "Observability is to measure of how well internal states of a system can be inferred from knowledge of its external outputs"[15].

The presented definitions represents the meaning of the words Observing and Observability are reflected exactly as it is in the project context. For example, observe the interaction between some microservices, regarding some data resulted from their interaction, to notice a certain fault in the whole/part of the system.

Controlling in control systems is "to manage the behaviour of a certain system"[16]. Controlling and Observability are dual aspects of the same problem[15].

When we want to understand the working and behaviour of a system, we need to watch it very closely and pay special attention to all details and data it provides. This kind of details and data may be in multiple structured text formats, and it can contain lots of information regarding the interaction between microservices and the corresponding access to them. The information generated by a microservices based system is normally represented as, what we call traces and spans, presented in the next subsection 3.1.4 - Traces and Spans. Therefore, we may work with this information as a starting point to perceive the characteristics of the system and build a tool that is able to be aware of it and that can perform an analysis of the data to detect some system failures.

3.1.3 Distributed Tracing

Google Dapper [17]

OpenTracing

// TODO: Write about Distributed Tracing: - Origin; - OpenTracing; - Specification; - Why; - How

3.1.4 Traces and Spans

First things first, we can think in a trace as a group of spans. A trace is a representation of a data/execution path through the system and a span represents the logical unit of work in the system. A trace can also be a span. The span has an operation name, the start time of the operation, its duration and some annotations regarding the operation itself. An example of a span can be an HTTP call or a Remote Procedure Call (RPC) call. For

a more clear insight of how spans are related with each other and with time, we have the figure 3.2.

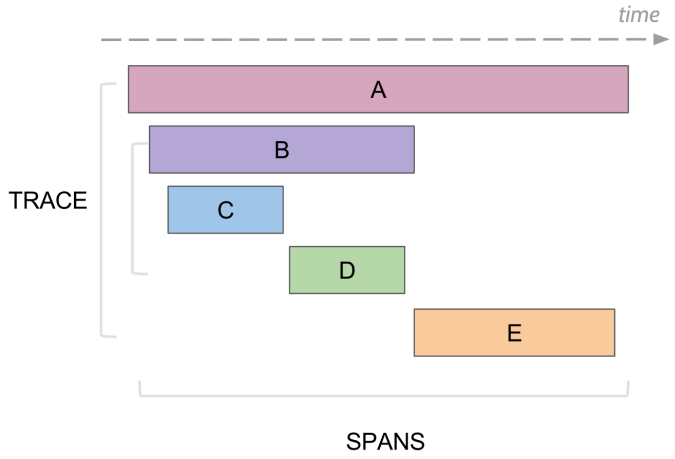

Figure 3.2: Traces and spans disposition over time.

As we can see in the figure 3.2, the spans spread over time, overlapping each other, since nothing prevents the occurrence of multiple calls in very close times. In the same figure we can see some boxes, two represent traces (box A and B) and four represent spans (boxes B, C, D and E). For a brief example, and for this case we may think of the following cases to define each operation inherent to each box: A - "Get user info", B - "Fetch user data from database", C - "Connect to MySQL server on 127.0.0.1'(10061)", D - "Can't connect to MySQL server on 127.0.0.1'(10061)" and E - "Send error result to client". The creators of OpenTracing have made a data model specification that says, "with a couple of spans, we might be able to generate a span tree and model a graph of a portion of the system"[18]. This is because they represents causal relationships in the system. As presented by the guys who defined this specification, and again for a more clear insight, the span tree can be like the one presented in the figure 3.3.

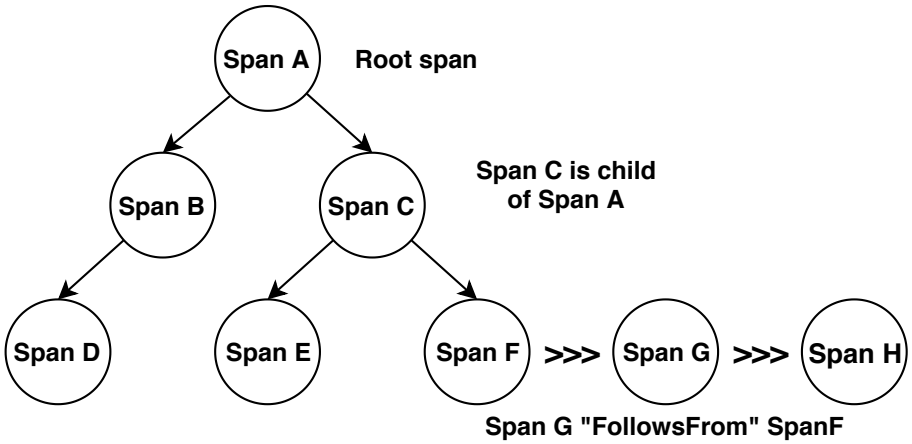

Figure 3.3: Span Tree example.

In the figure 3.3 it's represented a span tree with a trace made up of eight spans. Every span must be a child of some other span, unless it is the root span. With the information presented in the span tree, we can generate a multi-directed graph of the system (explained in the subsection 3.1.5 - Graphs).

This type of data is extracted and can be obtained, as trace files or by transfer protocols ex. HTTP, from technologies like Kubernetes[19], OpenStack[20], and other cloud or distributed management system technologies that implements some kind of system or code instrumentation using, for example, OpenTracing[21] or OpenCensus[22].

In the end and as explained before, traces and spans contains some vital system details as they are the result of instrumentation of a part or the whole system and therefore, this kind of data can be used as a starting point resource information to analyse the system.

3.1.5 Graphs

As it was briefly explained before, we might be able to model a graph of the system using a couple of spans. "A Graph is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices" [23].

Taking the very common sense of the term, a graph is an ordered pair G = (V, E), where G is the graph itself, V are the vertices/nodes and E are the edges. The figure 3.4 gives us, a simple visual representation, of what a graph really is for a more clear understanding. There we can see a graph composed by a total of five nodes that contains some labels in it and, in this case, five relationships between them.

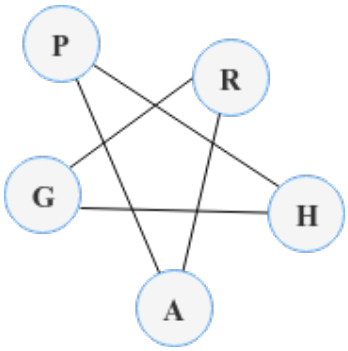

Figure 3.4: Graph visual representation.

This graph is the representation of the following information:

V = {'G', 'R', 'A', 'P', 'H'}

E = {{'G', 'R'}, {'R', 'A'}, {'A', 'P'}, {'P', 'H'}, {'H', 'G'}}

There are multiple types of graphs. In this term they can be undirected, where the set of edges don't have any orientation between a pair of nodes like in this example, or be directional, where the set of edges have one and only one orientation between a pair of nodes, or be a multigraph, where in multiple edges are more than one connection between a pair of node that represents the same relationship, and so forth.

Graphs can have many use cases, has they can model the representation of a lot of real life practical problems in the fields like physics, biology, social and information systems and for the purpose of this thesis, they are considered first class citizens.

A Graph Database (GDB) is "a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data"[24].

The composition of a GDB is based on the mathematics graph theory, and therefore this databases uses three main components called nodes, edges, and properties. This main

components are defined and explained in the following list:

- Node: Are the entities in the graph. They can hold any number of attributes (key-value pairs) called properties. Nodes can be tagged with labels, representing their different roles in your domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes.

- Edge (or Relationships): provide directed, named, semantically-relevant connections between two node entities (e.g. André STUDIES_IN Department of Informatics Engineering (DEI)). A relationship always has a direction, a type, a start node, and an end node. Like a Node it can attach metadata.

- Property: can be any kind of metadata attached to a certain Node or a certain Edge.

3.1.6 Time Series

Introduce Time-Series data

A Time Series Database (TSDB) is "is a database optimised for time-stamped or time series data like arrays of numbers indexed by time (a date time or a date time range)" [25].

This kind of databases are natively implemented using specialised database algorithms to enhance it's performance and efficiency due to the widely variance of access possible. The way this databases use to work on efficiency is to treat time as a discrete quantity rather than as a continuous mathematical dimension. Usually a TSDB allows operations like create, enumerate, update, organise and destroy various time series entries.

The TSDB and the GDB, presented in this subsection and in the subsection before respectively, are at the time, the most wanted and fastest growing kind of databases due to their use cases in the trending fields of Cloud and Distributed based Systems and in the *Internet of Things (IoT)*. The figure 3.5 presents the growing of this databases in the last two years. As we can see in the figure presented, the TSDB and GDB are distancing from the remaining databases in terms of popularity starting from the same spot in December of 2016. The predictions are that this databases will not stop increasing popularity, until this kind of systems described before start losing it too.

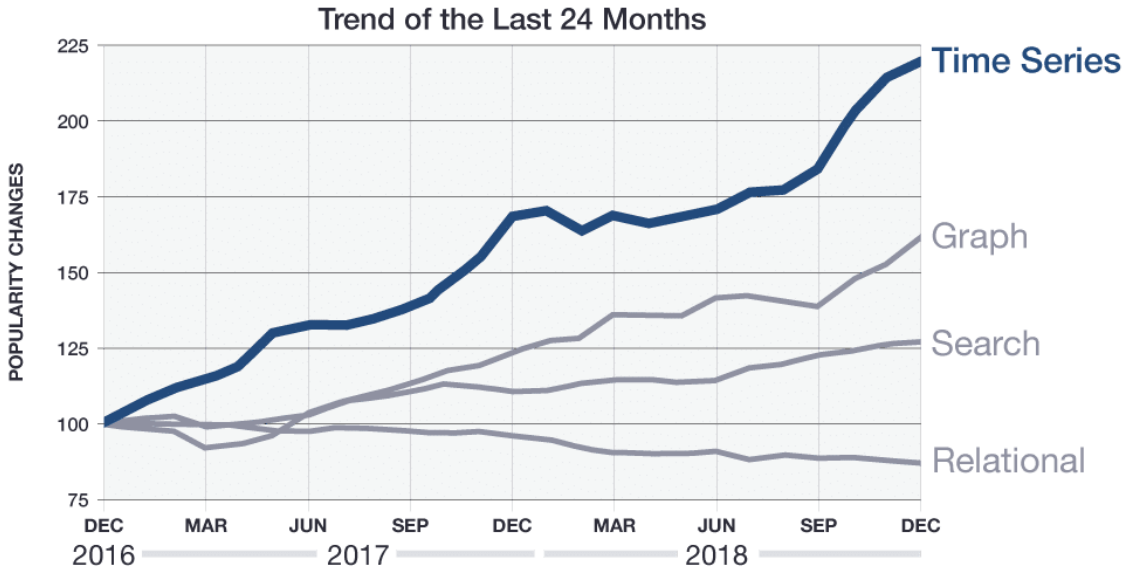

Figure 3.5: Fastest Growing Databases.[26]

After presenting the core concepts for this work, we follow to the next section 3.2 - Related Work where some related work with this thesis is presented.

3.2 Related Work

In this section are presented three section of the existing related work for tracing data analysis. After explaining each one, a brief reflection is made to note some directions of research for this project.

3.2.1 Mastering AIOps

// TODO: Explain what is done and how is done.

[27]

3.2.2 Anomaly Detection using Zipkin Tracing Data

// TODO: Explain what is done and how is done.

[28]

Wei Lee was contacted by email to understand better their research focus, however without success because no answer was given.

3.2.3 Analyzing distributed trace data

// TODO: Explain what is done and how is done.

[29]

3.2.4 Research possible directions

// TODO: Make a brief reflection about the related work.

3.3 Technologies

In this section are presented the technologies and tools that were researched, as well as the corresponding discussion considering the main objectives for this project. With the concepts presented in the section 3.1 in mind, the research were focused in a group of main topics regarding the problem we have in hands. The main topics were 3.3.1 - Distributed Tracing Tools, 3.3.2 - Graph Manipulation and Processing Tools, 3.3.3 - Graph Database Tools and 3.3.4 - Time-Series Database Tools.

3.3.1 Distributed Tracing Tools

This sub-section presents what are the most used and known distributed tracing tools. This tools are mainly oriented for tracing distributed systems like microservices-based distributed systems. What they do is to fetch or receive trace data from this kind of complex systems, treat the information, and then present it to the user using charts and diagrams in order to explore the data in a more human-readable way. One of the best features presented in this tools is the possibility to perform queries on the tracing (e.g. by trace id and by time-frame). The table 3.1 presents the most well-known OpenSource tracing tools.

Table 3.1: Tracing tools comparison.

Name	Jaeger	Zipkin
Repository	Jaeger GitHub [30]	Zipkin GitHub [31]
Brief description	It is a distributed tracing and monitoring system released as open source by Uber Technologies, that is used for monitoring and troubleshooting microservices-based distributed systems.	It is a distributed tracing and monitoring system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data. Zipkin's design is based on the Google Dapper paper.
Pros	OpenSource. Docker-ready. Can be used with some Zipkin functionalities, as it has a collector dedicated to it. Dynamic sampling rate. Browser UI.	OpenSource. Docker-ready. Allows lots of span transport ways (HTTP, Kafka, Scribe, AMQP). Browser UI.
Cons	Only supports two span transport ways (UDP and HTTP).	Fixed sampling rate.
Used mainly by	Uber	Lightstep

As we can see, this kind of tools are very similar and very good for monitoring and tracing a system as they provide a bunch of pros like being open source, containerized, support for some well known technologies for span transport and aggregate the spans in a good representational browser user-interface. However they are always focused on span and trace lookup and presentation, and do not provide a more interesting analysis of the system, for example to determine if there is any problem related to some microservice presented in the system. This kind of work falls into the user (Development and Operations (DevOps)) and he needs to perform the investigation and analyse the traces and spans with the objective of find anything wrong with them.

This kind of tools can be a good starting point for the problem that we face, because they already do some work for us like grouping the data generated by the system and provide some way to visualize it.

3.3.2 Graph Manipulation and Processing Tools

Considering that we have data to be processed and manipulated, we have to be sure that we can handle it for analysis. For this purpose, and knowing that the data is a representation and an abstraction of a graph, we needed to study the frameworks available this task. The table 3.2 presents the main technologies available at the time for graph manipulation and processing.

Table 3.2: Graph manipulation and processing tools comparison.

Name	Apache Giraph [32]	Ligra [33]	NetworkX [34]
Description	It's an iterative graph processing system built for high scalability. For example, it is currently used at Facebook to analyse the social graph formed by users and their connections.	A library collection for graph creation and manipulation, and for analysing networks.	A Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.
Licence[35]	Free Apache 2	MIT	BSD - New License
Supported languages	Java and Scala.	C and C++.	Python.
Pros	Distributed and very scalable. Excellent performance (Can process one trillion edges using 200 modest machines in 4 minutes).	Can handle very large graphs. Exploit large memory and multi-core CPU's (Vertically scalable).	Good support and very easy to install with Python. Lots of graph algorithms already implemented and tested. Mature project.
Cons	Uses the "Think-Like-a-Vertex" programming model that often forces into using sub-optimal algorithms and is quite limited and sacrifices performance for scaling out. Can't perform many complex graph analysis tasks because it primarily supports Bulk synchronous parallel.	Lack of documentation and therefore, very hard to use. Don't have many usage in the community.	Not scalable (single-machine). High learning curve due to the maturity of the project. Begins to slow down when there's a high amount of data (400.000 plus nodes).

With the information presented in the previous table, we can have a notion that this three frameworks don't work and perform in the same level in many ways.

One thing to consider when comparing them is the scalability and performance that each can provide, for instance, in this component the first one, Apache Giraph is the winner since it is implemented with the distributed systems paradigm in mind and can scale to multiple-machines to get the job done in no time, considering high amounts of data. In other way, we have the third framework, called NetworkX, that is different from the previous as it works in a single-machine and doesn't have the ability to scale to

multiple-machines. This can be a very problematic feature if we are dealing with very high amounts of data and we have to process it in short amounts of time. The last framework, called Ligra, works in a single-machine environment like the previous one, but it can scale vertically as it has the benefit of use and exploit multi-core CPU's.

The second, and also most important thing to consider, is the support and quantity of implemented graph algorithms in the framework, and in this field the tables turned, and the NetworkX has a lot of advantage as it have lots of implemented graph algorithms defined and studied in graph and networking theory. The remaining frameworks don't have very support either because they don't have it documented or because the implementation and architecture that where considered don't allow to implement it.

For a more clear insight of the position of the presented technologies in the previous table, we have the figure 3.6 [36].

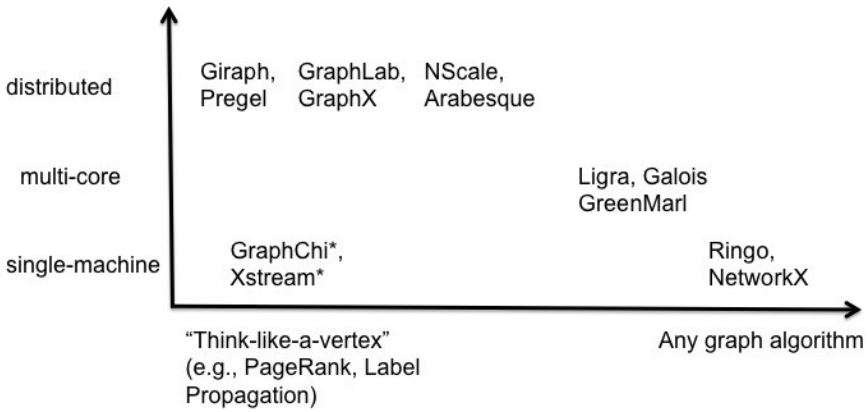

Figure 3.6: Graph manipulation tools comparison, regarding scalability and graph algorithms.

With the presented figure, our perception of what we might choose when considering this tools is more clear, but we have always some trade offs we cannot avoid. The best approach, and if it's possible, is to consider the usage of an hybrid environment where Giraph and NetworkX coexist one with another, as one fills the gaps of the other, but always taking into consideration that a bottleneck will occur between them **graph˙frameworks˙performance˙evaluation** and that there are almost none implementation where they coexist **graph˙frameworks˙performance˙evaluation** because of their disparity.

3.3.3 Graph Database Tools

Manipulating and process graph data is not enough, we need to store this data somewhere, and to do this we need a GDB. The results of the research for the best graph databases tools available are presented in the table 3.3.

As we can notice by the data provided by the presented table, the state of the art about graph databases is not very good. The offers are very limited and all of them lack something when we start to see them in detail. The interest in this databases is increasing as graph technology tend to have many use cases and solve lots of problems nowadays.

Facebook detains the most powerful and robust system for this purpose, but as it is

the base of their business because they need to perform large operations in their huge social graph in reduced times, it is a proprietary technology and is only referenced in some articles [38].

The remaining two tools, are very supported by the community because of their license and demand, however based on the stars and forks of their repositories, Neo4J is more well received by the community and tends to become more popular. It doesn't implement horizontal scalability by design and this can be a risk when using it in systems with scalabilty in mind, but there are some authors that report they were able to perform implementations and surpass the scalability issue, however with many snags[41]. ArangoDB supports scalability by default as we can see in the figure 3.7[42], but it has a very hard query

Table 3.3: Graph databases comparison.

Name	ArangoDB [37]	Facebook TAO [38]	Neo4J [39]
Description	It's a NoSQL database developed by ArangoDB Inc. that uses a proper query language to access the database.	TAO, "The Associations and Objects", is a proprietary database, developed by Facebook, that stores all the data related to the users in the social network.	It's the most popular open source graph database. Has been developed by Neo4J Inc. and is completely open to the community.
Licence	Free Apache 2	Proprietary	GPLv3 CE
Supported languages	C++ Go Java JavaScript Python Scala	—	Java JavaScript Python Scala
Pros	Multi data-type support (key/value, documents, graphs). Allows the combination of different data access patterns in a single query. Supports cluster deployment.	Very fast(=100ms latency). Accepts millions of calls per second. Distributed.	Supports ACID(Atomicity, Consistency, Isolation, Durability)[40]. High-availability. Has a visual node-link graph explorer. REST API interface. Most popular open source graph database.
Cons	Needed to learn a new query language called AQL(Arango Query Language). High learning curve. Has paid version with high price tag.	Not accessible to use.	Can't be distributed (It needs to be vertically scaled).

language with a high learning curve inherent to it, and it is payed to use some special features like SmartGraphs storage[43] that improves the writing of graph in distributed databases.

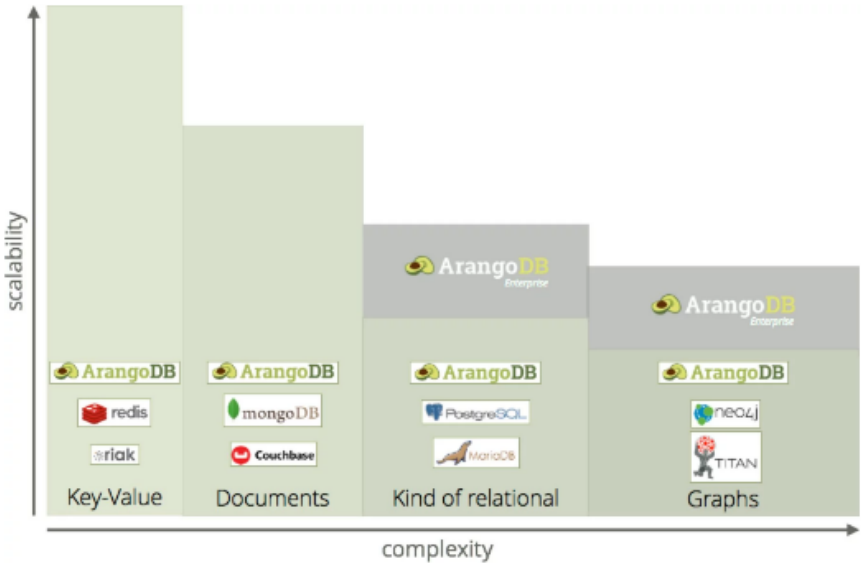

Figure 3.7: ArangoDB vs. Neo4J scalability over complexity.

3.3.4 Time-Series Database Tools

As we intend to extract useful data from span trees and graphs, we need to store it somewhere. We already know that the spans and trace data are directly related with time based information, explained in the subsection 3.1.4 - Traces and Spans, so the best way to store the gathered or calculated information from them is in a TSDB.

The figure 3.8 present the ranking of the TSDB at the current time[44].

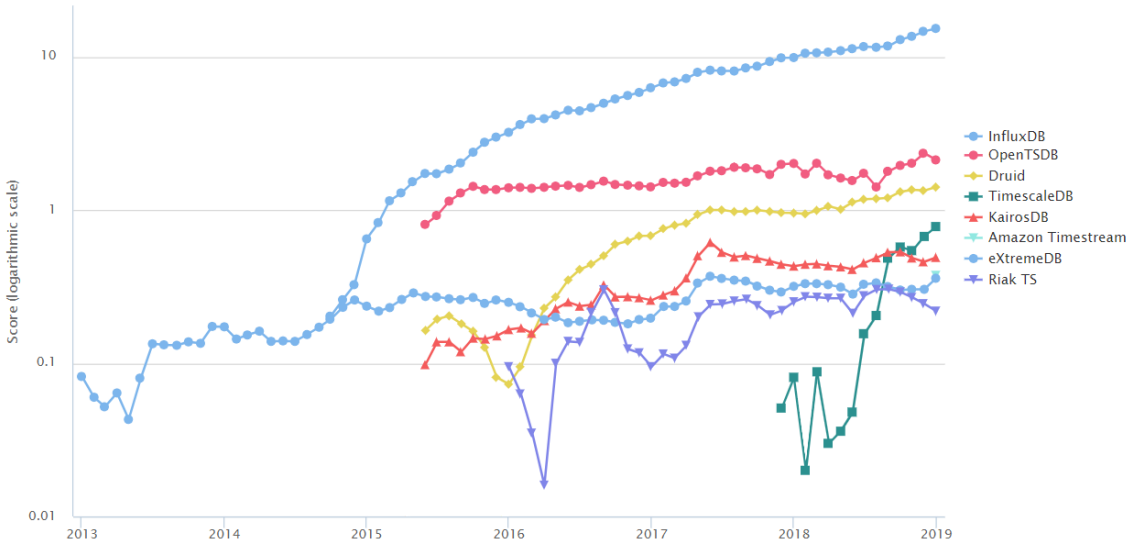

Figure 3.8: TSDBs ranking from 2013 to 2019.

The table 3.4 exposes a comparison between the two top databases presented in the ranking, the *InfluxDb* and *OpenTSDB*, two very well known databases in the world of TSDB, in order to understand the advantages and disadvantages of each one.

Table 3.4: Time-series databases comparison.

Name	InfluxDB [45]	OpenTSDB [46]
Description	It is an open-source time series database developed by InfluxData written in Go and optimised for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics.	It is a distributed, scalable Time Series Database (TSDB) written on top of HBase. OpenTSDB was written to address a common need: store, index and serve metrics collected from computer systems (network gear, operating systems, applications) at a large scale, and make this data easily accessible and graphable.
Licence	MIT	GPL
Supported languages	Erlang Go Java JavaScript Lisp Python R Scala	Erlang Go Java Python R Ruby
Pros	Scalable in the enterprise version. Outstanding high performance. Accepts data via HTTP, TCP, and UDP protocols. SQL like query language. Allows real-time analytics.	It's massively scalable. Great for large amounts of time-based events or logs. Accepst data via HTTP and TCP access protocols. Good platform for future analytical research into particular aggregations on event/log data. Doesn't have paid version.
Cons	Enterprise high price tag. Clustering support only available in the enterprise version.	Expensive to try. Not a good choice for general-purpose application data.

Based on the information presented in the referenced table, we can notice that this two databases are very similar on what they offer like the access protocols and scalability capabilities. In the point of licence, both are open source, however the first one, InfluxDB, has an enterprise paid version that is not very well exposed in its documentations and much people don't even notice it, contrarily to OpenTSDB which is completely free. The enterprise version of InfluxDB provides clustering support, high availability and scalability[47], features that OpenTSDB offer for free, however in terms of performance, InfluxDB outperforms OpenTSDB in almost every benchmark by a far distance as we can see in the figures 3.9 and 3.10.

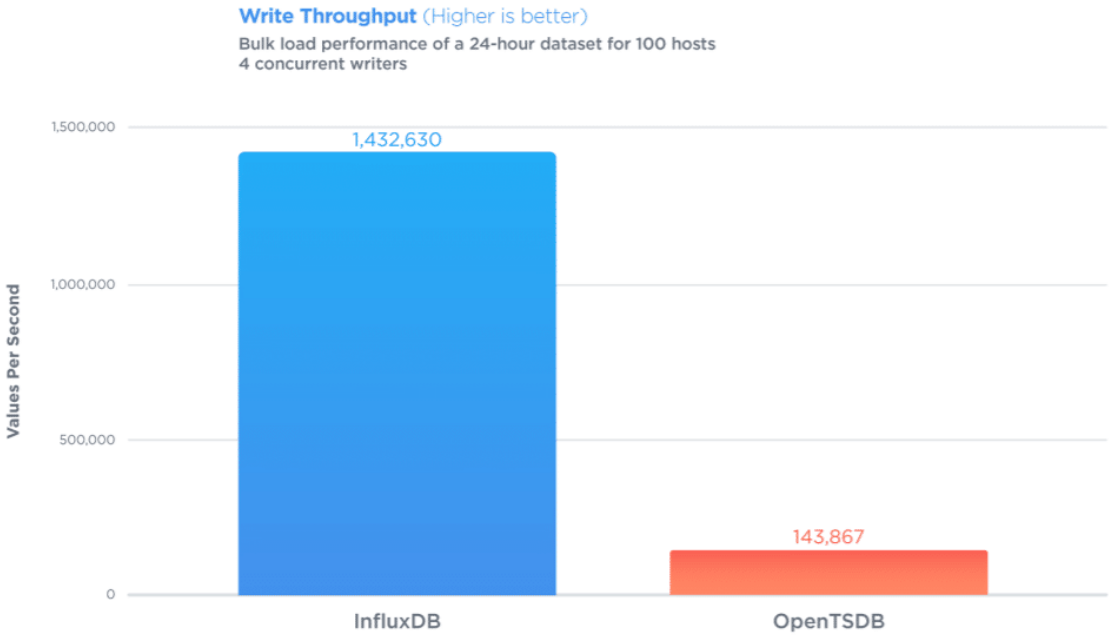

Figure 3.9: InfluxDB vs OpenTSDB write throughput performance[47].

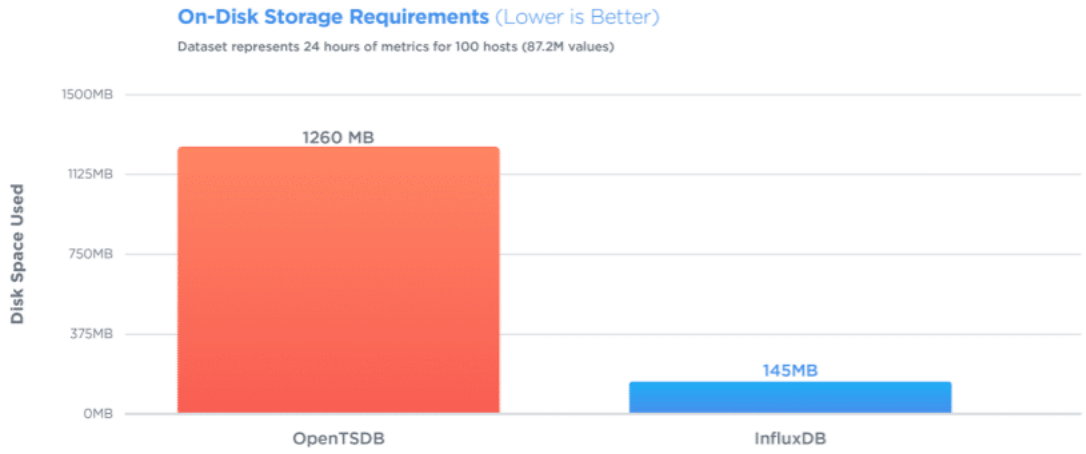

Figure 3.10: InfluxDB vs OpenTSDB storage requirements[47].

After providing the state of the art involved with this work to the reader in this chapter, the document flows to the next chapter 4 - Research Objectives and Approach, where the objectives of this research and the approach taken to each component of work are presented and discussed.

This page is intentionally left blank.

Chapter 4

Research Objectives and Approach

This chapter presents the research objectives and approach used in this thesis. We will start to discuss how we faced the problem, and what were the main difficulties that we found when handling this kind of problem as well as the ways we have taken to deal with it.

The debugging process in distributed systems and microservice based systems is not an easy task to perform, because of the way the system is designed using this kind of architecture style, as explained in the subsection 3.1.1 - Microservices. Reasoning about concurrent activities of system nodes and even understanding the system's communication topology can be very difficult. A standard approach to gaining insight into system activity is to analyze system logs, but this task can be very tedious and complex process. The main existing tools are the ones presented in **??**, but they only do the job of gather and present the information to the user in a more gracefully way, however they rely on the user perception to do the search to find issues that exist in their platform by performing queries to the spans and trace data. So, with this problem ahead, we started looking for the needs of Sysadmins and/or Development and Operations (DevOps) when they wanted to scan and analyse their system searching for issues.

To do this and narrow the problem we were facing, we decided to talk with some DevOps personal and expose them the situation, with the objective to gather their main needs and ideas in mind, we putted ourselves in their perspective when talking to them to try and find what are the main difficulties when they perform their search for issues in the system in a *"As a DevOps i want to..."* situation. The kind of questions that were placed were like: *"What are the most common issues?"*, *"What are the variables involved in this kind of issues?"* and *"What are the correlations between this variables and the most common issues?"*. From this discussions and conversations emerged the following eight core questions:

1. What is the neighbourhood of one service?
2. Is there any problem (Which are the associated heuristics)?
3. Is there any faults related to the system design/architecture?
4. What is the root problem, when A, B, C services are slow?
5. How are the requests coming from the client?
6. How endpoints orders distributions are done?

7. What is the behaviour of the instances?
8. What is the length of each queue in a service?

The next step was to work on the questions presented above. We decided to split them in more concise questions, refine and filter the most relevant to define our objective, and after that, check with someone involved in the DevOps field if the final questions represent some of their needs. First, to handle the information presented in this eight initial questions, we decided to create what we called a "Project Questions Board". This board consists on a Kanban [48] style board present in the project git repository, were everyone involved in the project could access and modify it. The board was defined with four lanes: "To refine", "Interesting", "Refined" and "Final Questions", and the process were to cycle the questions through every lane, generating new ones and filtering others. After this, and to check if the final questions were really some that represented the needs of a DevOps, some colleagues that work directly in the field were contacted and the questions were exposed to them. In the end, the ten questions that were produced in the final lane represented right what are some of their needs. The final questions, their corresponding description**(D)** and explanation of the expected work**(W)** that must be performed to each one are exposed bellow:

1. What is the neighbourhood of one service, based on incoming requests?
   - D. The neighbourhood of one service is a very important information to know due to the simple fact that it represents the interaction between the microservices. The incoming requests can map the interactions between the microservices and with this kind of information, we can check and analyse the service dependencies.
   - W. Implies generate a graph, based on the spans and traces, using the outgoing connections, from a certain node, that are correlated with the incoming connection(s).
2. What is the neighbourhood of one service, based on outgoing requests?
   - D. Similar to the previous question, but this time, instead of incoming requests we focus on the outgoing requests of one service.
   - W. Implies generate a graph, based on the spans and traces, using the incoming connections, from a certain node, that are correlated with the outgoing connection(s).
3. How endpoints orders distributions are done, when using a specific endpoint?
   - D. The distributions of microservices in a system allows us to understand if a certain endpoint groups with others or if it is an isolated service, which stands for its relevance to the whole system.
   - W. Implies generate a graph, based on the spans and traces, then calculate the degree of a certain node that represents the endpoint, to finally check if it is an isolated, a leaf or a dominating (high or low depending on the degree of the other degrees) endpoint.
4. How requests are being handled by a specific endpoint?
   - D. This question has the objective of analyse the status of the requests that arrive or depart from a specific endpoint. This status represents if the requests was well succeed or not.

W. Implies to analyse the data from the requests that pass through a specific endpoint. Based on the annotations presented in the spans and traces, we are able to check if the requests are resulting in success or in error.

5. Which endpoints are the most popular?

D. The popularity of a certain endpoint is very important because it represents the importance of this endpoint to the system.

W. Implies to retrieve the most popular service, based on the spans and traces, and get the services with more incoming connections sorted by the number of incoming connections.

6. Is there any problem related to the response time?

D. Response time is must watch variable because, for example, strange high values may represent a problem in the system performance.

W. Implies to get the response time of every trace (difference between end and start time of every span in the trace) and then calculate and store some measurements like the average time, the maximum time, the minimum time and variance. After having some stored values, the system must perform calculations and check if there is too much disparity between them to determine if there is a problem in the response time.

7. Is there any problem related to the morphology?

D. The morphology of the system allows us to understand if some endpoints are common in the system (they usually exist), or if they only are instantiated in specific situations.

W. Implies generate multiple graphs, based on a certain group of spans and traces that are contained in a certain time interval. Then we need to store the graphs gradually using some graph storing mechanism to perform the difference of subsequent stored graphs. This result of the difference between graphs must be stored in a time-series storing mechanism, to be accessed later and determine if there were hard changes that could lead to morphology problems in the system thought time.

8. Is there any problem related to the entire workflow of (one or more) requests?

D. Analyse the request workflow through the system is a good practice, as it represents the interaction triggered by the request in the system and its resulting behaviour. This can lead to find out if the system has cycles and if they are normal or represent a problem to solve.

W. Implies to generate the graph of the system, identify the path of some request(s) in the system and then perform the calculation to verify and identify if there were cycles presented in the graph involved in the path of the request(s). The results of this calculations must be stored in a time-series storing mechanism, to be accessed later and determine if this cycles are normal, or if they represent a problem related to the request(s) work-flow, based on the kind of request.

9. Is there any problem related to the occupation/load of a specific endpoint?

D. The occupation/load of a specific endpoint in this case is represented by the number of requests in queue of a specific endpoint. This value is very important because it represents if the endpoint is in overflow or not.

W. Implies to get the number of requests in queue of a specific endpoint and then calculate and store some measurements like the average, maximum, minimum and variance of the number of requests in queue. After having some stored values, the system must perform calculations and check if there is too much disparity between them to determine if there is a problem in the occupation/load.

10. Is there any problem, related to the number/profile of the client requests?

D. The number of client requests and their corresponding profile represents the behaviour of clients when using the system. This can be used to identify problems of bad system usage from the clients, like for example a DDoS (Denial Of Service).

W. Implies calculate the number of accesses to the system, based on the spans and traces annotated with client requests of a certain time interval, and store the calculations for every node. After having some stored values, the system must determine the level regions of access based in the available data (profile of requests, ex.: high, moderate and low), and check if there were to much requests outside of the defined level regions.

These final questions, with a slight reformulation, could be exported to high level of abstraction functional requirements of the monitoring tool that we want to develop. After having this questions, we decided to study the current state of art to check how things are done nowadays regarding this subject and we found that some tools perform the process of convert spans and traces to a graph, that represents the system at that current time interval, however they do not perform any kind of analysis and study over the span tree and the graph after that[49].

Considering this, what we decided to do was to develop a simple prototype tool to test some state of the art tools. What we were able to achieve was to do the reconstruction of the graph, using our own data (this data was provided by Prof. Jorge Cardoso, representing an approximate two hour collection of spans and traces, about 400.000 spans, generated by one of their clusters). At this point, and since we have already held the hands-on of some tools at the moment, we were ready to start and think about the solution we need to build. Therefore, we decided to specify the solution, and considered to build a monitoring tool named by ourselves, *Graphy*.

In a very briefly explanation, we want that *Graphy* be able to calculate relevant metrics from the span trees and the generated graphs, and to work with this kind of metrics to perform the system analysis and answer the questions exposed above. To perform some of this work, it will be resourcing to machine learning algorithms that we will need to study in parallel with the implementation, as we cannot predict what we might encounter when retrieving the metrics at the time. The machine learning algorithms are to process the metrics and perform some deductions regarding the system behaviour over the time.

Chapter 5

Possible Solution

In this chapter we present and discuss the possible solution to be implemented regarding the research objectives of this work. To present the solution and explain it, we will cover some aspects and topics that are considered when defining a software based solution. This topics are the well known functional requirements 5.1, the quality attributes or non-functional requirements 5.2 and finally, the architecture 5.4 that is produced based on all the previous topics.

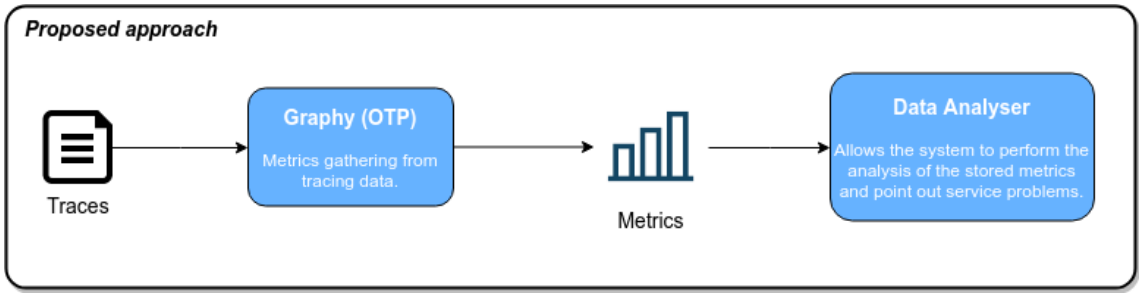

Figure 5.1: Proposed approach for tracing analysis.

5.1 Functional Requirements

The functional requirements are in software and systems engineering, by definition, a declaration of the intended function of a system and its components. For the purpose of this specification, we decided to present a brief description of each functional requirement, composed by an id, the corresponding name and its priority. The notation used in the priority was based on the urgency that we have to implement a certain functionality, and we decided to use three levels: High, Medium and Low priorities.

Therefore, the functional requirements for the proposed solution, obtained from the questions presented in the chapter 4, are briefly specified in the table 5.1 sorted by priority levels.

As the table 5.1 shows us, the functional requirements can be grouped in three main groups due to their priority levels. The first four (FR-1 to FR-4) are presented with high level of priority, because they do not represent a very high level of difficulty to implement, however they provide some base to the implementation of the following four. This four functional requirements (FR-5 to FR-8) represent a much higher degree of difficulty

Table 5.1: Functional requirements specification.

ID	Name	Priority
FR-1	The system must know what are the neighbours of a certain service, based on the service incoming requests.	High
FR-2	The system must know what are the neighbours of a certain service, based on the service outgoing requests.	High
FR-3	The system must know which endpoints are the most popular.	High
FR-4	The system must know if requests to a certain endpoint results in success or error.	High
FR-5	The system must be able to identify if there is any problem related to the response time.	Medium
FR-6	The system must be able to identify if there is any problem related to its morphology.	Medium
FR-7	The system must be able to identify if there is any problem related to the entire work-flow of one or more requests.	Medium
FR-8	The system must know how endpoints orders distributions are done when using a specific endpoint.	Medium
FR-9	The system must be able to identify if there is any problem related to the occupation/load.	Low
FR-10	The system must be able to identify if there is any problem related to the number/profile of the client requests.	Low

has there are much more questions to solve involving research and the implementation (explained in the chapter 4 - Research Objectives and Approach). The final two (FR-9 and FR-10) have a low priority level, because they do not represent to much relevance of functionality to our core issues.

5.2 Quality Attributes

When designing a system, one of the most important things to consider is to specify and describe well all the quality attributes or non-functional requirements. These kind of requirements are usually Architecturally Significant Requirements and they are the ones that require more of the architect's attention, as they reflect directly all the architecture decisions and aspects. Sometimes, they are also named the "itilities" because most of them share this suffix in the word.

To specify them, we usually use a representation called a utility tree, in which we insert the Quality Attribute (QA) by a certain order of priority, for the architecture and for the business inherent to each one, and in order to consider the trade-offs and decide the weight of each in the produced architecture. The codification of the order of priority is the following:

- H. High
- M. Medium
- L. Low

To describe them, we must pay attention and try to include six important things in the definition: the **stimulus source**, the **stimulus**, the **environment**, the **artifact**, the **response** and the **measure of the response**.

The figure 5.2 contains all the raised QA for this project exposed in a utility tree format, sorted alphabetically by their general QA, and after by the architectural impact.

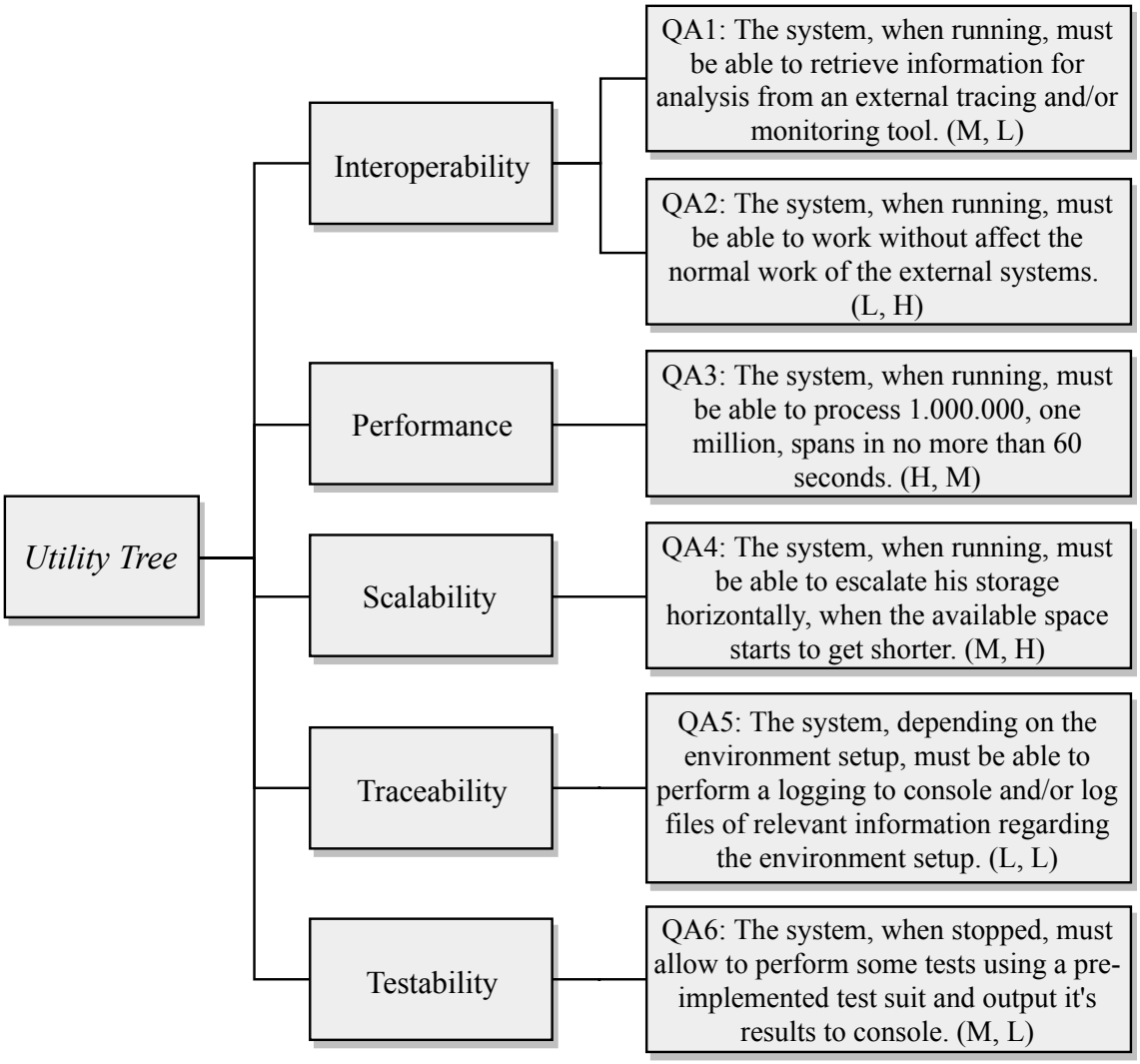

Figure 5.2: Utility tree.

As we can see by the information presented in the utility tree, there are the following QA's: Interoperability, Performance, Scalability, Traceability and Testability. A briefly explanation for every QA is exposed bellow:

QA1 (Interoperability): Since the wanted solution is a monitoring tool, it has to access an external system in order to obtain the necessary data to analyse and therefore, access to an internal monitoring tool presented in the external system. As this is considered the starting point to obtain our data, we considered a Medium level for the architecture and a Low for the business.

QA2 (Interoperability): Since the solution will be accessing an external system or outputs generated by it, all interactions with it must not cause conflicts. This is very important in the business perspective, because if our solution is not co habitable

with other systems it may be completely rejected. For the architectural perspective it doesn't represent a big impact.

QA3 (Performance): We defined this QA taking into account the number of spans produced in an hour, by the system were we gathered our data. As it produces approximately 200.000 spans in an hour, and to ease our research work when using the tool, we decided to set the target for our solution as 1.000.000 spans in about a minute. This QA will have an high architectural impact, as it can define a certain technology for graph processing. For the business perspective, we considered a Medium level, as it presents some interest.

QA4 (Scalability): Due to the amount of data needed to process and store over time, our system has to be able to store the data into multiple machines because it may start running out of space. We decided to give a medium level of architectural impact as it can change the solution in terms of storage components. For the business this has an high impact, as it need more machines to run the solution if this QA is fully considered against the remaining.

QA5 (Traceability): This QA was considered due to the simple fact that we need to bee able to see what's the system is doing, when it's processing the data. For this QA we decided to give low levels for both architecture and business, as it does not represent relevance to any of them.

QA6 (Testability): As the system will be analysing external systems, we need to be sure that it analyses it in a correct way, and to be sure of this we need to test our solution. We considered a medium level for the architectural impact for this QA, because its implementation needs to analyse some components from within the system. For the business we considered a low level, because the main interest to test the system is ours in order check if it is working correctly.

5.3 Technical Restrictions

In this section we present the technical restrictions involved in this solution.

Normally, when specifying a solution in software engineering, after presenting the functional requirements and non-functional requirements, comes the specification of business restrictions. However, in this project none were raised due to the simple fact that this work is focused on the exploration and research and that there isn't any formal client defined.

To define the technical restrictions, we used an id and its corresponding description. The raised technical restrictions are presented in the table 5.2 followed by an explanation.

Table 5.2: Technical restrictions specification.

ID	Description
TR-1	Use OpenTSDB as a Time-Series database.

We raise only one technical restriction, as we can see in the table 5.2. This technical restriction was considered because prof. Jorge Cardoso suggested it, due to the simple fact that OpenTSDB is the database that they are currently using in their projects where a time-series database is needed. However, this project does not have a concrete and

formally defined client, it is good to use a technology used by the people that will use the tool to ease their work and possibly introduce changes.

5.4 Architecture

In this section, the architecture will be presented based on all the previous topics with resource to the defined Simon Brown's C4 Model[50]. This approach of defining an architecture uses the following four diagrams to do it: 1 - Context Diagram, 2 - Container Diagram, 3 - Component Diagram and 4 - Code Diagram. For the definition of this architecture, only the first three representations will be considered. Every representation will be exposed with a briefly explanation of the decisions taken to draw that specific diagram. After presenting the representations and the corresponding explanations, we will cycle thought all the QA, in order to explain where it is reflected and the considerations taken to produce the current architecture.

5.4.1 Context Diagram

In this subsection the context diagram is presented. This kind of diagram allows us to see "the big picture" of the system as it represents the system as a "big box" and its interactions with the users and other systems. The figure 5.3 presents the context diagram for this solution.

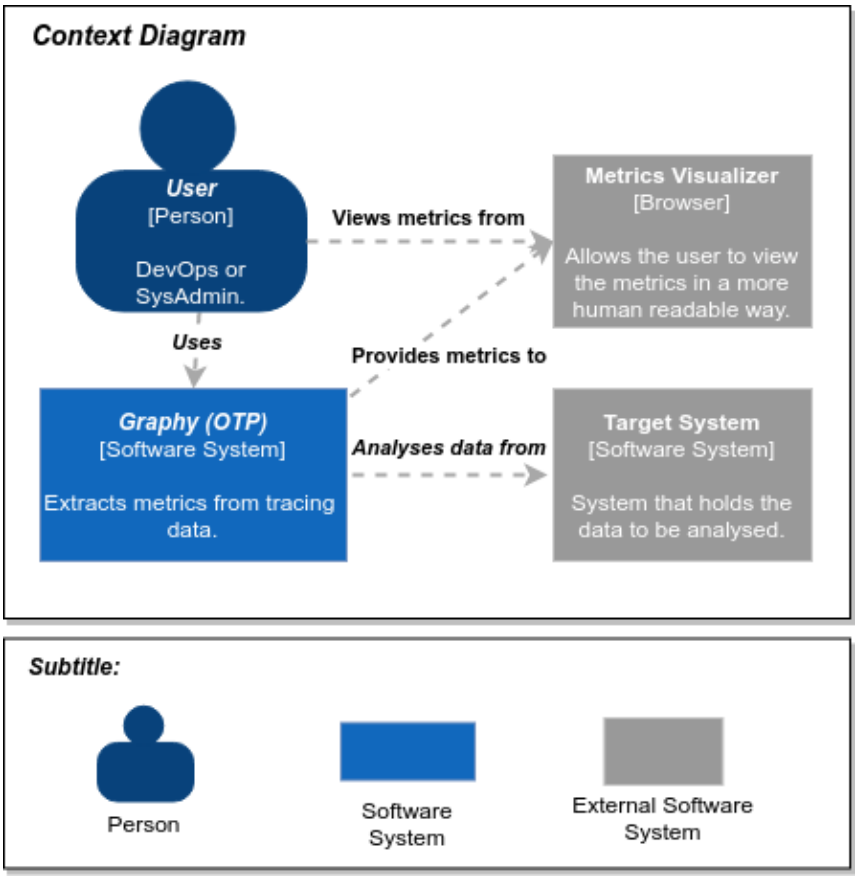

Figure 5.3: Context diagram.

In the presented diagram we can see that the solution, *Graphy*, will receive interactions

from the user, as it need someone to control its operations and will analyse data from an external system infrastructure, defined as the target system.

5.4.2 Container Diagram

In this subsection the container diagram is presented. This kind of diagram allows us to perform a "zoom-in" in the context diagram, and get a new overview of this system, therefore in this diagram we can see the high-level shape of the software architecture and how responsibilities are distributed across it. The figure 5.4 presents the container diagram for this solution.

In this figure we can see the main containers involved in the Graphy system. The first one, from top to bottom, is the *Access Console* and this container was considered as it is needed for the user to be able interact with the *Graphy Application API*. The *Graphy Application API* controls the entire Graphy system, that uses a communication protocol to retrieve information from the system it must analyse, and two databases to store the information resulted from the processing, a Graph Database (GDB) and a Time Series Database (TSDB).

5.4.3 Component Diagram

In this subsection, the last diagram is presented, the component diagram. This kind of diagram gives us a more deeper vision about the system, and therefore, it presents us with its main components. The figure 5.5 presents the component diagram for this solution.

In this last diagram, we have a lower level of abstraction sight of the *Graphy Application API* container, composed by a total of eight components. At its core we have the *Graphy Controller*, that has the responsibility of controlling the remaining components involved in the system, orchestrating the actions requested by the user thought the *Access Console* and performed by the whole system. In the remaining components, we have two that have greater relevance, the *Graph Processor* and the *Data Analyzer*. These components are responsible of processing the data converted from the Spans and Traces data, use the corresponding repository component to store or retrieve information from its database, and perform the analysis of the data. Finally we have the *Testing Component*, which is responsible of test the systems algorithms, the *Logging Component* which is responsible of log the relevant information and the *File IO* which is responsible of handle the files involved in the system processes.

To check the architecture produced, and as said before, we will now cycle thought all the QA and check were they are reflected in the architecture presented for this solution, and explain the trade-off involved and what were our considerations about each one.

QA1 and QA2 are satisfied by the simple fact that the system is able to analyse data from an external system, using a transmission protocol where data is exchanged thought HTTPS in JSONL format. As the data is only read from an exposed endpoint presented in the target system, this will not affect the normal work of the external system as this preserves its independence.

QA3 is satisfied when we decide to use NetworkX as the technology to process our graphs. This technology does not scale horizontally, however it has a very decent performance and it's able to retrieve a certain measure of a graph with about 100.000 nodes, in near 15 seconds[51]. From 1.000.000 spans, in normal conditions, we will never be able to

get a graph of this kind of size, as with our experiments, with 100.000 spans we were able to get a graph of almost 20 nodes, and with 200.000 spans we were able to get a graph of almost 30 nodes. In the end, we are considering this time and span quantity to be sure that our tool will give us good times and ease our work of performing the research and implementation of this kind of tool.

QA4 is satisfied because we decided to use two databases that are scalable horizontally

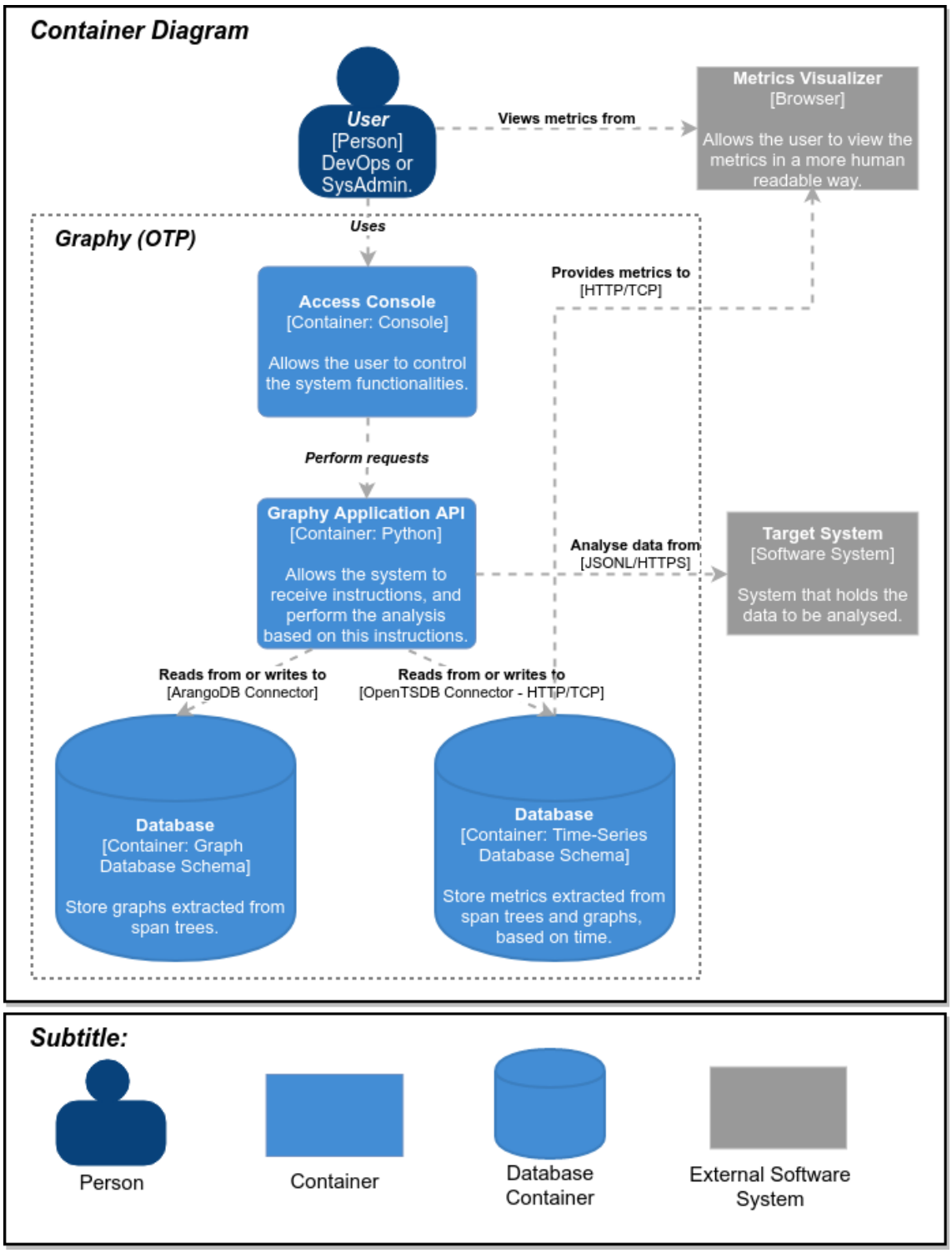

Figure 5.4: Container diagram.

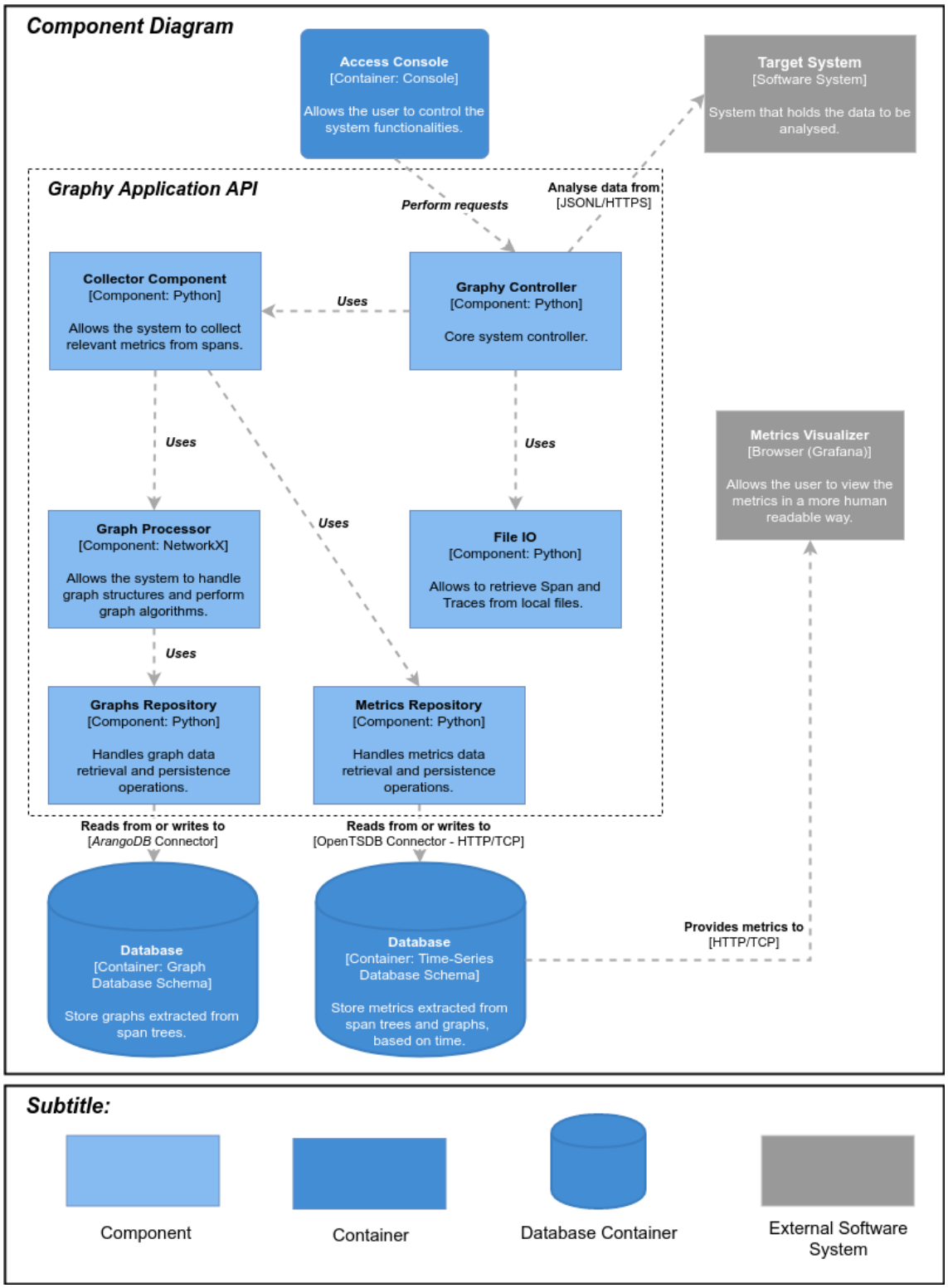

Figure 5.5: Component diagram.

by design, the ArangoDb for a GDB and the OpenTSDB for a TSDB, both presented in the 3.3.3 and 3.3.4 subsections respectively. In the end, this QA can not be fully satisfied because we can not scale our system entirely, due to the fact of the technology that we chose to perform the graph processing. However, we have chosen this technology because it can perform much more graph algorithms, as we can see in the figure 3.6, and this is

much more relevant for our main purpose.

QA5 is satisfied by the existence of the component *Logging Component*, that allows the system to perform logging of relevant information. For the technology here, we decided to use click-log[52], a python library used for logging purposes as it has all the main capabilities needed here to perform the logging.

QA6, like the previous one is satisfied by the existence of a certain component, the *Testing Component*, which implements all the capabilities and functionalities to perform tests and check if the systems is working correctly.

Finally, for the only technical restriction raised, we can see that it is satisfied by the usage of the OpenTSDB as our TSDB.

With all that was presented before, we conclude that our solution satisfies all the quality attributes, business constraints and technical restrictions, and therefore, we may claim that this architecture fits our needs as a solution, taking into consideration the trade-offs involved and explained before.

This page is intentionally left blank.

Chapter 6

Implementation Process

This chapter presents the implementation process of the proposed solution explained in the previous chapter. Every step is explained taking into consideration the most important points to implement the methods. This chapter covers three main sections: First, **??** - **??**, the data provided to perform this research is briefly exposed and analysed. Second, **??** - **??**, the implementation of Graphy (OpenTracing processor (OTP)), our proposed solution to collect metrics from tracing data presented in the chapter 5, is explained in detail. Finally, **??** - **??**, the methods for analysis of the stored observations are presented with the respective results and answers to the research questions.

6.1 Huawei Tracing Data Set

Nothing comes from nothing, and in this project the starting point for every method developed was a data set provided by Huawei, represented by professor Jorge Cardoso. To be able to get access to this information, a NDA: Non-disclosure agreement were signed by both parts. This data set contains the results of tracing data gathered from an experimental cluster used by the company for testing purposes. In terms of time this data is a representation of two days of the systems workload, therefore two files were provided, one for each day. These files were generated in 2018-07-10 and for protection, some fields of the data set were obfuscated during the generation process.

Table 6.1: Data set provided for this research.

	2018-06-28	2018-06-29
File	traces-2018-06-28.jsonl	traces-2018-06-29.jsonl
Spans count	190 202	239 693
Traces count	64 394	74 331

As we can see by the information presented in the table 6.1, each file represents a day of traces and both were written in JSONL format [53]. This type of file format is an extension to the normal lightweight data-interchange standard JSON: JavaScript Object Notation file type, however, in JSONL multiple JSONs are separated by a new line character. Each Span is presented by a single JSON, so for each line we have a Span encoded in JSON format. Further in this sections is a representations of the amount of traces by hour, to do this and to perform the traces count we have to map the span trees, the explanation and algorithm are presented in the next section **??** - **??**.

Span data format is defined in a specification, but companies and software developers do not need to strictly follow it, they can produce their own span data format and this means that each software company can have their own span specification. To ensure the understanding and follow up with the span data format, there was a file with instructions about the spans. In this file, the fields were explained, this consisted in their specific names and the supposed data type. A brief sample of the explanation provided in the file is exposed in the points bellow.

1. traceId - Unique id of a trace (128-bit string).
2. name - Human-readable title of the instrumented function.
3. timestamp - UNIX epoch in milliseconds.
4. id - Unique id of the span (64-bit string).
5. parentId - Reference to id of parent span.
6. duration - Span duration in microseconds.
7. binaryAnnotations:
   (a) protocol - 'HTTP' or 'function' for RPC calls.
   (b) http.url - HTTP endpoint.
   (c) http.status_code - Result of the HTTP operation.
8. annotations:
   (a) value - Describes the position in trace (based on Zipkin format). Could be one of the following values or other: 'cs' (client send), 'cr' (client receive), 'ss' (server send) or 'sr' (server receive)
   (b) timestamp - UNIX epoch in microseconds.
   (c) endpoint - Wich endpoint generated a trace event.

Two notes were also provided within this file. To point each one, has they are very important, we present them bellow.

1. Time units are not consistent, some fields are in milliseconds and some are in microseconds.
2. Trace spans may contain more fields, except those mentioned here.

From the point presented above, we get a notion that we need to consider some things about the values that we might encounter in the spans fields. Some fields are required by the OpenTracing specification as presented in 3.1.3, and to sanitise this whole topic, it is very important that a company, when producing this kind of information, stick with the specification, even if it is an open-source specification. So "traceId", "name", "timestamp", "id" and "parentId" are required fields, this is because they are the foundations to identify a span, produce relation between them and to place them in time. Other topic that is very important to understand is that the specification is not precise, we do not know clearly what is available when working with a span, this means that there is no specific data structure for fields like "binaryAnnotations" and "annotations", and this is a big problem as we might have spans that are completed or incomplete and we can not

know it as there is nothing to tell us about it. For example, in the "binaryAnnotations" the information is disposed in key value objects, and as said in the second point of the notes presented above: "Trace spans may contain more fields, except those mentioned here", this causes a tremendous explosion in possibilities, because there might be keys with their corresponding values for some particular spans and it gets hard to generalise this whole span structure.

At the current time of writing, the clarity of span data only depends if the company that produced the software or the software owner care about what the system will produce in terms of tracing data. To be sure that the teams developing distributed software produce good tracing data, the company must implement a standard for every team to follow. The normalisation and unification of the span format should be one thing to consider in the near future, this is because it is very hard to perceive and distinguish multiple span types. For example, in the data provided spans can be of two types: Hypertext Transfer Protocol (HTTP) Span or Remote Procedure Call (RPC) Span, and the only field that distinguishes them is that the second one has an "exec" field, which stands for the execution process id, and that is not presented in the HTTP span type. Another example, for the measure units of some fields, if the field have the same key, they all should be in one single measure unit, this is because some tools are expecting this, and they assume wrong values when spans have time-stamps in milliseconds and others in microseconds like in this case.

To give a brief notion of how the traces and spans are spread throughout time, we have used our tool, Graphy OTP, to generate two charts that represents the counting of traces and spans for each hour in each day. The decision to generate two spitted charts comes from the simple fact that we have one file for each day. To be able to count the spans in time, in this case by hour, the tool only needs to group every span by hour and count them, however for traces, the tool has a lot more work has it have to merge all spans in their corresponding trace (explained in **??**. After having all traces, we can count them like we count spans. Note that if a span or trace starts at a given time (t1) contained in a time-frame (tf1), and with its duration (d1) surpasses the next time-frame (tf2), (t1 + d1 >tf2), it is considered to be in the tf1, or by other words, only the starting time of the trace or span is considered for the counting. The figures 6.1 and 6.2 presents the data set traces and spans counting throughout time.

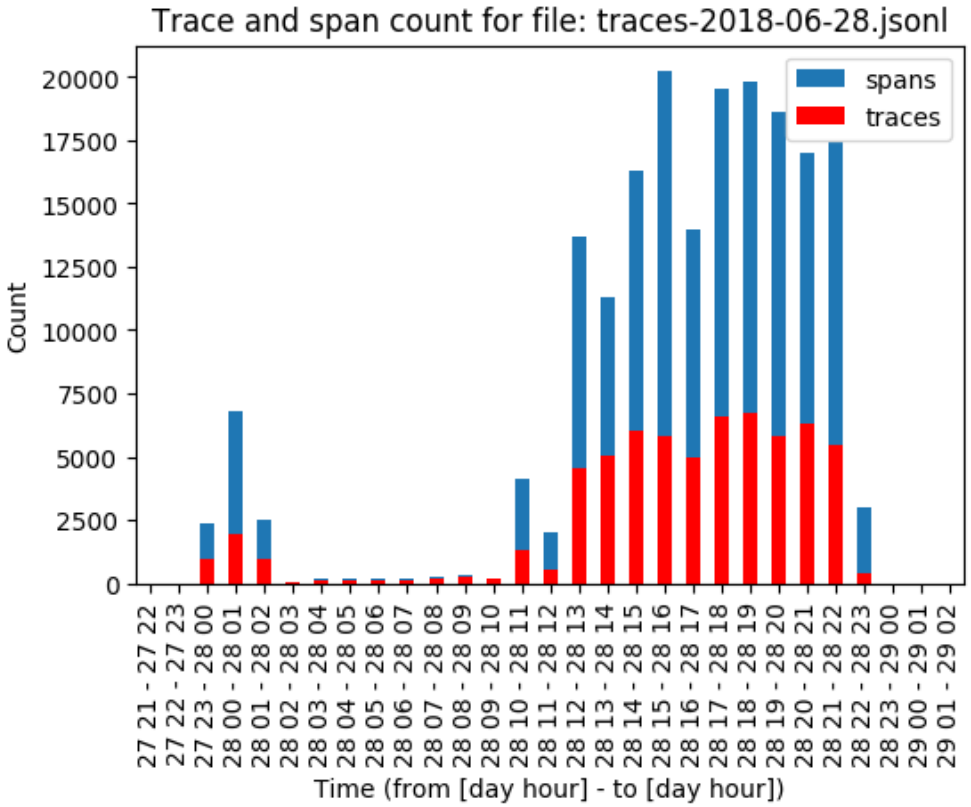

Figure 6.1: Trace file count for 2018-06-28.

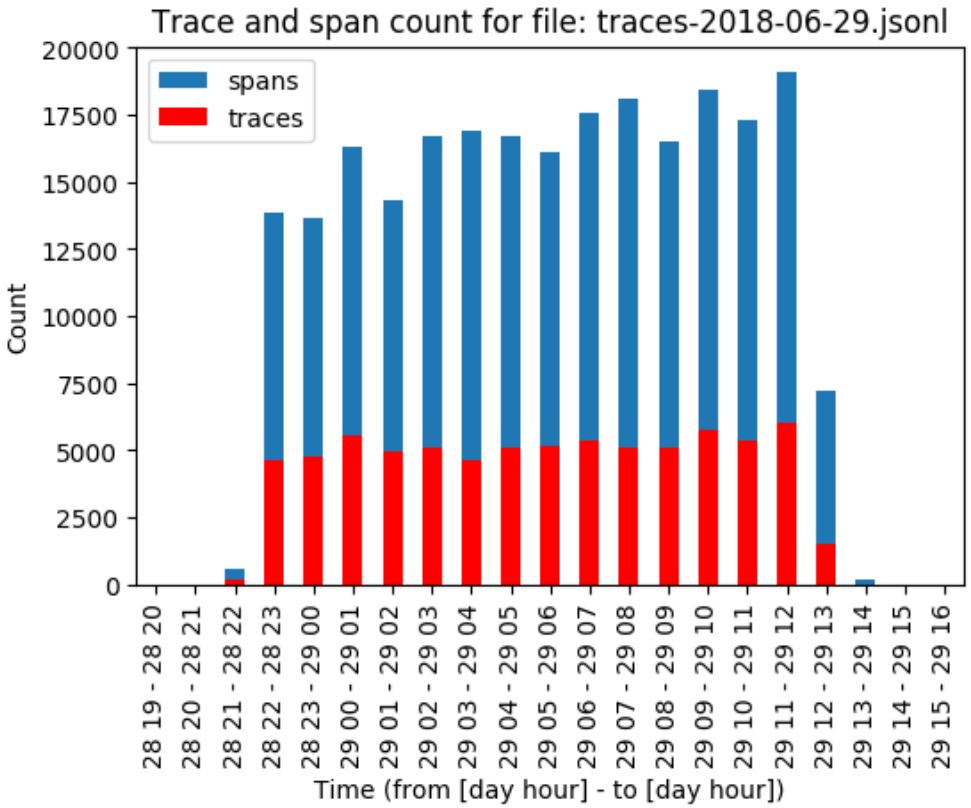

Figure 6.2: Trace file count for 2018-06-29.

// A big region without traces, why? Can't answer that!

// Less data spreaded through time than the previous one.

// Maybe include here the Quality of tracing analysis and possibilities in their own analysis.

6.2 Open Tracing Processor Component

// TODO: Start by explaining how the metrics relate with the questions, and what metrics we decided to extract and were, how do we put it there.

Explain how to map spans into span trees. to process the information presented in the data set and store it for our own analysis purposes

... as explained in 3.1.4, each spans relates with another, in this case, by a field called parent id. After mapping the spans into trees, we end up with a list of tress, each for a specific trace, and we just have to count them. To map the spans into trees, we index it by the

// TODO: Regarding database setup and implementations, the choices were to use OpenTSDB as a Time-Series database and ArangoDB as a graph database. However, the second choice wasn't the better one, due to some lack of support and terrible Application Programming Interface (API) documentation. Some issues were opened in the GitHub client API for the programming language Python, but the answer was always that it have the expected behaviour [54]. This have lead to some difficulties when implementing the component Graphs Repository, presented in Graphy API. Some difficulties were felted when trying to store graphs with custom names and setting up different type values to some data structures presented in the pyArango client API. The solution was to fetch all the API, perform some changes and use our custom pyArango client. This changes were committed for review to the original project stored on GitHub. Mitigating this kind of problems was something that was impossible to predict when choosing the Graph storing technology.

Algorithm 1: How to write algorithms

```
Result: Write here the result
initialization;
while While condition do
    instructions;
    if condition then
        instructions1;
        instructions2;
    else
        instructions3;
    end
end
```

aaa

// Finally present the display of information in the Grafana with some prints. Refer that the lack of data in day 28. And do some observations.

6.3 Data Analysis Component

// TODO: Talk about Unlabeled data, the best algorithms to handle this kind of situation and how we did it.

Chapter 7

Results, Analysis and Limitations

Section intro...

7.1 Anomaly Detection

...

7.2 Trace Quality Analysis

...

7.3 Limitations of OpenTracing Data

...

This page is intentionally left blank.

Chapter 8

Conclusions

This chapter presents the conclusions to this whole work. Therefore in this chapter we decided to split it into three sections, first there is a brief reflection about this whole work in **??** - **??**, in second a section about the future work that can follow the whole research carried out by this investigation.

8.1 Brief Reflections

This thesis presents an attempt to further the field of distributed system based tracing analysis.

// TODO: Talk a bit about everything

A reflection about the tools and methods produced and the open paths from this whole research are exposed. Also a reflection of the main difficulties felted with this research are presented

8.2 Future Work

// TODO: Talk a bit about the possible work in this field and what can be achieved using this work.

the future work that can be addressed considering this work is properly explained taking into consideration what is said in the previous section

8.3 Concluding Research Questions

// TODO: Talk a bit about everything

This page is intentionally left blank.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow", in *Present and Ulterior Software Engineering*, Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 9783319674254. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://hal.inria.fr/hal-01631455.

[2] P. Di Francesco, I. Malavolta, and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption", in *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, 2017, pp. 21–30, ISBN: 9781509057290. DOI: 10.1109/ICSA.2017.24. [Online]. Available: http://cs.gssi.infn.it/ICSA2017ReplicationPackage.

[3] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems", *ACM Transactions on Computer Systems*, vol. 5, no. 2, pp. 121–150, Mar. 1987, ISSN: 07342071. DOI: 10.1145/13677.22723. [Online]. Available: http://portal.acm.org/citation.cfm?doid=13677.22723.

[4] S. P. R. Janapati, *Distributed Logging Architecture for Microservices*, 2017. [Online]. Available: https://dzone.com/articles/distributed-logging-architecture-for-microservices.

[5] OpenTracing.io, *What is Distributed Tracing?* [Online]. Available: https://opentracing.io/docs/overview/what-is-tracing/.

[6] *Huawei Cloud Platform*. [Online]. Available: https://www.huaweicloud.com/.

[7] V. Osetskyi, *How to Calculate Man-Hours for The Software Project: Explanation with an Example*. [Online]. Available: https://medium.com/existek/how-to-calculate-man-hours-for-the-software-project-explanation-with-an-example-50f2fbe111d2 (visited on 01/03/2019).

[8] The GanttProject Team, *GanttProject*. [Online]. Available: https://www.ganttproject.biz/ (visited on 11/29/2018).

[9] C. Richardson, *Microservices Definition*. [Online]. Available: https://microservices.io/ (visited on 10/17/2018).

[10] Docker Inc., *Why Docker?* [Online]. Available: https://www.docker.com/why-docker (visited on 12/24/2018).

[11] ——, *What is a Container*. [Online]. Available: https://www.docker.com/resources/what-container (visited on 12/23/2018).

[12] M. Fowler and J. Lewis, *Microservices, a definition of this architectural term*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 01/07/2018).

[13] S. Brown, *Distributed big balls of mud*. [Online]. Available: http://www.codingthearchitecture.com/2014/07/06/distributed%7B%5C_%7Dbig%7B%5C_%7Dballs%7B%5C_%7Dof%7B%5C_%7Dmud.html (visited on 12/23/2018).

[14] Thefreedictionary.com, *Observing definition*. [Online]. Available: https://www.thefreedictionary.com/observing (visited on 10/13/2018).

[15] Wikipedia, *Observability*. [Online]. Available: https://en.wikipedia.org/wiki/Observability (visited on 01/02/2019).

[16] ——, *Control system*. [Online]. Available: https://en.wikipedia.org/wiki/Control%7B%5C_%7Dsystem (visited on 01/02/2018).

[17] B. H. Sigelman, L. Andr, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure", Tech. Rep. April, 2010, p. 14. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/pt-PT//archive/papers/dapper-2010-1.pdf%20https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf.

[18] OpenTracing, *OpenTracing Data Model Specification*. [Online]. Available: https://github.com/opentracing/specification/blob/master/specification.md (visited on 12/10/2018).

[19] Cloud Native Computing Foundation, *What is Kubernetes?* [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 11/29/2018).

[20] OpenStack, *What is OpenStack?* [Online]. Available: https://www.openstack.org/software/ (visited on 11/29/2018).

[21] OpenTracing.io, *What is OpenTracing?* [Online]. Available: https://opentracing.io/docs/overview/what-is-tracing/ (visited on 11/29/2018).

[22] Google LLC, *What is OpenCensus?* [Online]. Available: https://opencensus.io/ (visited on 11/29/2018).

[23] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition - Graphs*. 2019. [Online]. Available: https://algs4.cs.princeton.edu/42digraph/.

[24] Wikipedia, *Graph database*. [Online]. Available: https://en.wikipedia.org/wiki/Graph%7B%5C_%7Ddatabase (visited on 10/15/2018).

[25] ——, *Time series database*. [Online]. Available: https://en.wikipedia.org/wiki/Time%7B%5C_%7Dseries%7B%5C_%7Ddatabase (visited on 12/11/2018).

[26] InfluxData, *Time Series Database (TSDB) Explained*. [Online]. Available: https://www.influxdata.com/time-series-database/ (visited on 12/11/2018).

[27] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly Detection and Classification using Distributed Tracing and Deep Learning", 2018, [Online]. Available: https://pt.slideshare.net/JorgeCardoso4/mastering-aiops-with-deep-learning.

[28] W. Li, *Anomaly Detection in Zipkin Trace Data*, 2018. [Online]. Available: https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1.

[29] B. Herr and N. Abbas, *Analyzing distributed trace data*, 2017. [Online]. Available: https://medium.com/@Pinterest%7B%5C_%7DEngineering/analyzing-distributed-trace-data-6aae58919949.

[30] JaegerTracing, *Jaeger GitHub*. [Online]. Available: https://github.com/jaegertracing/jaeger (visited on 12/10/2018).

[31] OpenZipkin, *Zipkin Repository*. [Online]. Available: https://github.com/openzipkin/zipkin (visited on 12/10/2018).

[32] Apache Software Foundation, *Apache Giraph*. [Online]. Available: http://giraph.apache.org/ (visited on 12/03/2018).

[33] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory", Pittsburgh, [Online]. Available: https://www.cs.cmu.edu/%7B~%7Djshun/ligra.pdf.

[34] N. developers, *NetworkX*. [Online]. Available: https://networkx.github.io/ (visited on 12/03/2018).

[35] Wikipedia, *Software License*. [Online]. Available: https://en.wikipedia.org/wiki/Software%7B%5C_%7Dlicense (visited on 10/16/2018).

[36] A. Deshpande, *Surveying the Landscape of Graph Data Management Systems*. [Online]. Available: https://medium.com/@amolumd/graph-data-management-systems-f679b60dd9e0 (visited on 11/24/2018).

[37] ArangoDB Inc., *ArangoDB Documentation*. [Online]. Available: https://www.arangodb.com/documentation/ (visited on 10/16/2018).

[38] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph", [Online]. Available: https://cs.uwaterloo.ca/%7B~%7Dbrecht/courses/854-Emerging-2014/readings/data-store/tao-facebook-distributed-datastore-atc-2013.pdf.

[39] Neo4J Inc., *No Title*. [Online]. Available: https://neo4j.com/docs/ (visited on 10/16/2018).

[40] Wikipedia, *ACID(Computer Science)*. [Online]. Available: https://en.wikipedia.org/wiki/ACID%7B%5C_%7D(computer%7B%5C_%7Dscience) (visited on 10/16/2018).

[41] K. V. Gundy, *Infographic: Understanding Scalability with Neo4j*. [Online]. Available: https://neo4j.com/blog/neo4j-scalability-infographic/ (visited on 12/15/2018).

[42] ArangoDB Inc., *What you can't do with Neo4j*. [Online]. Available: https://www.arangodb.com/why-arangodb/arangodb-vs-neo4j/ (visited on 12/15/2018).

[43] ——, *ArangoDB Enterprise: SmartGraphs*. [Online]. Available: https://www.arangodb.com/why-arangodb/arangodb-enterprise/arangodb-enterprise-smart-graphs/ (visited on 12/15/2018).

[44] DB-Engines.com, *Time-Series DBMS Ranking*. [Online]. Available: https://db-engines.com/en/ranking%7B%5C_%7Dtrend/time+series+dbms (visited on 01/09/2019).

[45] InfluxData, *InfluxDB GitHub*. [Online]. Available: https://github.com/influxdata/influxdb (visited on 12/12/2018).

[46] OpenTSDB, *OpenTSDB*. [Online]. Available: https://github.com/OpenTSDB/opentsdb (visited on 12/12/2018).

[47] C. Churilo, *InfluxDB Markedly Outperforms OpenTSDB in Time Series Data & Metrics Benchmark*. [Online]. Available: https://www.influxdata.com/blog/influxdb-markedly-outperforms-opentsdb-in-time-series-data-metrics-benchmark/ (visited on 12/12/2018).

[48] Wikipedia, *Kanban Board*. [Online]. Available: https://en.wikipedia.org/wiki/Kanban%7B%5C_%7Dboard (visited on 12/09/2018).

[49] W. Li, *Anomaly Detection in Zipkin Trace Data*, 2018. [Online]. Available: https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1 (visited on 12/11/2018).

[50] S. Brown, *The C4 model for software architecture*. [Online]. Available: https://c4model.com/ (visited on 12/12/2018).

[51] T. Hladish, E. Melamud, L. A. Barrera, A. Galvani, and L. A. Meyers, "EpiFire: An open source C++ library and application for contact network epidemiology".

[52] M. Unterwaditzer, *Click-log: Simple and beautiful logging*. [Online]. Available: https://click-log.readthedocs.io/en/stable/ (visited on 12/22/2018).

[53] I. Ward, *JSON Lines*. [Online]. Available: http://jsonlines.org/ (visited on 04/18/2018).

[54] A. Bento and T. Daouda, *pyArango Issues*, 2019. [Online]. Available: https://github.com/ArangoDB-Community/pyArango/issues/137.