UNIVERSIDADE D
COIMBRA

Ricardo Ângelo Santos Filipe

Client-Side Monitoring of Distributed Systems

Tese no âmbito do Programa de Doutoramento em Ciências e Tecnologias da Informação orientada pelo Professor Doutor Filipe João Boavida de Mendonça Machado de Araújo e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Fevereiro de 2020

Client-Side Monitoring of Distributed Systems

Ricardo Ângelo Santos Filipe

Dissertation submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

February 2020

UNIVERSIDADE D
COIMBRA

Department of Informatics Engineering
Faculty of Science and Technology
University of Coimbra

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC). This work was partially carried out under the project PTDC/EEI-ESS/1189/2014 — Data Science for Non-Programmers, supported by COMPETE 2020, Portugal 2020-POCI, UE-FEDER and FCT and it was also supported by the advanced formation program of Altice Labs. I would also like to express my gratitude to the INCD - *Infraestrutura Nacional de Computação Distribuída*, for providing access to their computational resources.

This work has been supervised by Professor **Filipe João Boavida de Mendonça Machado de Araújo**, Assistant Professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

But why, some say, the moon? Why choose this as our goal? And they may well ask why climb the highest mountain? Why, 35 years ago, fly the Atlantic?

We choose to go to the moon. We choose to go to the moon and do the other things, not because they are easy, but because they are hard, because that goal will serve to organize and measure the best of our energies and skills, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win.

John F. Kennedy Moon Speech - Rice Stadium (adapted)

Abstract

From critical systems to entertainment, most computer systems have become distributed. Compared to standalone applications, distributed systems are more complex, difficult to operate and maintain, thus increasing the probability for outages or other malfunctions. Properly monitoring the system is therefore even more important. However, recovering a complete image of the system is a herculean task for administrators, who often need to resort to a large plethora of tools. Despite all these tools, the person that many times identifies the degradation or the system outage is the one that is somehow disregarded in the monitoring chain — the client. Almost daily, we have examples in the news from companies that had outages or system degradation perceived by the final client with a direct impact on the companies' revenues and image.

The lack of client-side monitoring and the opportunity to improve current monitoring mechanisms paved the way for the key research question in this thesis. We argue that the client has information on the distributed system that monitoring applications should use to improve performance and resilience.

In this work, we aim to evaluate the limits of black-box client-side monitoring and to extend white-box with client information. Additionally, we are very interested to understand what kind of information does the system leak to the client.

To evaluate this approach, we resorted to several experiments in distinct scenarios from three-tier web sites to microservice architectures, where we tried to identify performance issues from the client-side point-of-view. We used client profiling, machine learning techniques among other methods, to demonstrate that using client information may serve to improve the observability of a distributed system.

Properly including client-side information proved to be an interesting and challenging research effort. We believe that our work contributed to advance the current state-of-art in distributed system monitoring. The client has viable information that eludes administrators and provides important insights on the system.

Keywords:
Monitoring, Black-box monitoring, White-box monitoring, Client-side monitoring, Observability, Analytics, Microservices.

Resumo

Desde os sistemas críticos ao entretenimento, a maioria dos sistemas computacionais tornou-se distribuída. Quando comparados a aplicações monolíticas, os sistemas distribuídos são mais complexos, difíceis de operar e manter, aumentando assim a probabilidade de anomalias. A monitoria de um sistema distribuído é desta forma ainda mais importante. Todavia, obter uma imagem completa do sistema é uma tarefa árdua para os administradores, que frequentemente precisam de recorrer a uma grande variedade de ferramentas. Mesmo com a superabundância de ferramentas, a pessoa que muitas vezes identifica a degradação ou a interrupção do sistema é a mesma que de alguma forma é desconsiderada no fluxo de monitoria: o cliente. Quase diariamente, temos exemplos na comunicação social de empresas que tiveram interrupções ou degradação no serviço prestado percebido pelo cliente final, com impacto direto nas receitas e na imagem dessas empresas.

A falta de monitoria do ponto de vista do cliente e a oportunidade de melhorar a monitoria atual abriram o caminho para a questão chave de pesquisa nesta tese. Argumentamos que o cliente possui informação sobre o sistema distribuído que as ferramentas de monitora devem usar para melhorar o desempenho e resiliência.

Neste trabalho pretendemos avaliar os limites de uma monitoria do lado do cliente de uma forma "caixa-negra", e extender as soluções de "caixa-branca" com informação do cliente. Além disso, estamos também interessados em entender que tipo de informação é que o sistema escapa para o cliente.

Para avaliar esta abordagem, recorremos a várias experiências em cenários distintos desde sites de três camadas até arquiteturas de micro serviços, onde tentamos identificar problemas do ponto de vista do cliente. Usámos técnicas de criação de *profiling* do ponto de vista do cliente, técnicas de *Machine Learning*, entre outros métodos, para demonstrar que o uso de informações do cliente pode servir para melhorar a observabilidade de um sistema distribuído.

A inclusão de informações do cliente provou ser um tópico de pesquisa interessante e desafiador. Acreditamos que o nosso trabalho contribuiu para avançar o atual estado da arte de monitoria em sistemas distribuídos. O cliente possui informações viáveis que escapam ao controlo dos administradores e fornece conhecimento importante sobre o sistema.

Keywords:
Monitoria, Monitoria Caixa-negra, Monitoria Caixa-branca, Monitoria via cliente, Observabilidade, Analítica, Microserviço

Acknowledgements

This work is dedicated to every people, from past to present, that in one way or another, has contributed to transform me into the person I am today.

Foremost, although in many ways a Ph.D. is a path of solitude and introspection, no man is an island, and without the support of some persons, it would be impossible to finish.

First, I would like to start by thanking Professor Filipe Araujo, for his guidance, knowledge, patience, and motivation. Without his belief and support it would be impossible to finish this work.

Secondly, to my colleagues with whom I worked with most closely. Serhiy Boychenko and Jaime Correia, I think I owe you some beers!

Third, I would like to show my gratitude to my employer Altice Labs and mostly to my colleagues in there. Thank you, for the support and patience, to achieve this objective.

Fourth, I also would like to thank all the anonymous reviewers that gave me insights to improve the quality of my work.

Last but not least, I would like to thanks my parents and brother. Thank you for believing in my strengths and by reinforcing that problems are "contingencies from life".

Thank you all by the support when I was rock bottom, and also by pull me down from the sky, when I was overjoyed with the achieved accomplishments.

List of Publications

This thesis relies on the published scientific research present in the following peer reviewed papers:

1. Ricardo Filipe, Serhiy Boychenko, Filipe Araújo, "Online Client-Side Bottleneck Identification on HTTP Server Infrastructures", The Tenth International Conference on Internet and Web Applications and Services (ICIW 2015), June 2015.

2. Ricardo Filipe, Serhiy Boychenko, Filipe Araújo, "On Client-Side Bottleneck Identification in HTTP Servers", 7º Simpósio de Informática, September 2015.

3. Ricardo Filipe, Filipe Araújo, "Evaluation of REST Interfaces for Reliable Invocation Semantics", The 15th International Conference on WWW/INTERNET (ICWI 2016), October 2016.

4. Ricardo Filipe, Serhiy Boychenko, Filipe Araújo, "An Experimental Evaluation of Performance Problems in HTTP Server Infrastructures Using Online Clients", International Journal On Advances in Internet Technology, Vol. 9, 2016.

5. Ricardo Filipe, Filipe Araújo, "Client-Side Monitoring Techniques for Web Sites", The 15th IEEE International Symposium on Network Computing and Applications (NCA 2016), November 2016.

6. Ricardo Filipe, Rui Pedro Paiva, Filipe Araújo, "Client-Side Black-Box Monitoring for Web Sites", The 16th IEEE International Symposium on Network Computing and Applications (NCA 2017), October 2017.

7. Fabio Pina, Jaime Correia, Ricardo Filipe, Filipe Araújo, Jorge Cardoso, "Monitoria de sistemas de micro-serviços", 10º Simpósio de Informática, September 2018.

8. Ricardo Filipe, Jaime Correia, Filipe Araújo, Jorge Cardoso, "On Black-Box Monitoring Techniques for Multi-Component Services", The 17th IEEE International Symposium on Network Computing and Applications (NCA 2018), November 2018.

9. Fábio Pina, Jaime Correia, Ricardo Filipe, Filipe Araújo, Jorge Cardoso, "Nonintrusive Monitoring of Microservice-based Systems", The 17th IEEE International

Symposium on Network Computing and Applications (NCA 2018), November 2018.

10. Ricardo Filipe, Jaime Correia, Filipe Araújo, Jorge Cardoso, "Towards Occupation Inference in Non-instrumented Services", The 18th IEEE International Symposium on Network Computing and Applications (NCA 2019), September 2019.

11. Ricardo Filipe and Filipe Araújo, "Client-Side Monitoring of HTTP Clusters Using Machine Learning Techniques", The 18th IEEE International Conference on Machine Learning and Applications (ICMLA 2019), December 2019.

Other published works with authorship or contributions from the author during the period:

12. André Pinho, Pedro Furtado, Helena Margarida, Ricardo Filipe, "Experimental Comparison and Tuning of Time Series Prediction for Telecom Analysis", The 4th International work-conference on Time Series (ITISE 2018), September 2018

13. Jaime Correia, Fábio Ribeiro, Ricardo Filipe, Filipe Araújo, Jorge Cardoso, "Response Time Characterization of Microservice-Based Systems", The 17th IEEE International Symposium on Network Computing and Applications (NCA 2018), November 2018.

14. Diogo Alves, Bruno Valente, Ricardo Filipe, Maria Castro, Luís Macedo, "A Recommender System for Telecommunication Operators' Campaigns", The 19th EPIA Conference on Artificial Intelligence (EPIA 2019), September 2019.

Table of Contents

List of Figures

List of Tables

Abbreviations

AJAX	**A**synchronous **J**avascript **a**nd **X**ML
API	**A**pplication **P**rogramming **I**nterface
APM	**A**pplication **P**erformance **M**anagement
AWS	**A**mazon **W**eb **S**ervices
CC	Pearson **C**orrelation **C**oefficient
CDN	**C**ontent **D**elivery **N**etwork
CPU	**C**entral **P**processing **U**nit
CSV	**C**omma—**S**eparated **V**alues
DAG	**D**irected **A**cyclic **G**raph
DB	**D**ata**B**ase
DevOps	**D**evelopment (and) **O**perations
DNS	**D**omain **N**ame **S**ystem
DOM	**D**ocument **O**bject **M**odel
HDFS	**H**adoop **D**istributed **F**ile **S**ystem
HTML	**H**ypertext **M**arkup **L**anguage
HTTP	**H**ypertext **T**ransfer **P**rotocol
IaaS	**I**nfrastructure **a**s **a** **S**ervice
I/O	**I**nput / **O**utput
IP	**I**nternet **P**rotocol
IPM	**I**nfrastructure **P**erformance **M**anagement
JSON	**J**ava**S**cript **O**bject **N**otation

KPI	**K**ey**P**erformance **I**indicator
MAE	**M**ean **A**bsolut **E**rror
MSE	**M**ean **S**quare **E**rror
NN	**N**eural **N**etwork
PaaS	**P**latform **a**s **a** **S**ervice
RAM	**R**andom **A**ccess **M**emory
RBF	**R**adial **B**asis **F**unction
REST	**RE**presentational **S**tate **T**ransfer
RMI	**R**emote **M**ethod **I**nvocation
RUM	**R**eal **U**ser **M**onitoring
SEO	**S**earch **E**ngine **O**ptimization
SLA	**S**ervice **L**evel **A**greement
SLR	**S**imple **L**inear **R**egression
SQL	**S**tructured **Q**uery **L**anguage
SVM	**S**upport **V**ector **M**achine
SVR	**S**upport **V**ector **R**egression
TC	**T**raffic **C**ontrol
TCP	**T**ransmission **C**ontrol **P**rotocol
TSDB	**T**ime-**S**eries **D**atabase
URL	**U**niform **R**esource **L**ocator
VM	**V**irtual **M**achine
W3	**W**orld **W**ide **W**eb
WEKA	**W**aikato **E**nvironment **K**nowledge (for) **A**nalysis
XHR	**X**ml **H**ttp **R**equest
XVFB	**X** **V**irtual **F**rame **B**uffer

Chapter 1

Introduction

From video streaming to entertainment, from machine-to-machine frameworks to health applications, distributed systems lie at the heart of modern businesses. Users demand responsive systems, which in turn depend on resilience (Laprie, 2008) and elasticity (Herbst et al., 2013; Manifesto, 2019). A complete component monitoring, including relationships between services, is key to ensure system resiliency. However, efficient monitoring has proven to be a difficult task, especially in highly dynamic systems (Halpern, 1987).

Distributed systems may fail due to component problems or due to successive accumulated failures causing a waterfall effect (Gray; Oppenheimer et al., 2003). To be useful, a monitoring technique should have the following properties: high accuracy and coverage, few false-alarms, deployability and maintainability. Monitoring systems may operate in three distinct layers (MonitoringTools, 2015; SemaText, 2019): low-level (e.g. machine and protocol tests, such as heartbeats, pings, and HTTP error code monitors), application performance monitoring (APM) and real-user monitoring (RUM). Normally, a monitoring solution is the combination of several low-level monitors (to check the overall status of the machine), some application monitor (that may be out-of-the-box or customized), and even some client tracking.

This is the main subject of this thesis — monitoring —, a key aspect to a proper administration and operation of distributed systems. In a highly dynamic distributed system, we are forced to instrument and monitor applications to have the overall status of the infrastructure. In current solutions, there are two main unwanted problems: first, the

overall monitoring application is reactive, meaning that operators must act to already occurring incidents; secondly observing and monitoring the system is only possible if developers and administrators work together to instrument every single point-of-failure. Hence, it is interesting to understand the feasibility of timely inferring system status with as few data as possible.

1.1 Problem Statement and Motivation

Monitoring distributed systems is not easy. Leslie Lamport mentioned that "A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable." (Lamport, 1987). This sentence has some important aspect to our work: distributed systems have components that will fail and cause an overall system degradation or outage. Additionally, creating a larger ecosystem of components may increase the difficulty to monitor and prevent failures. This may lead to failure events with enormous impact on companies' reputation, often with consequences for business.

For example, Target Corporation suffered a system outage that prevented customers from making purchases in their stores, due to a technical issue (CNBC, 2019). U.K. mobile telecommunication operator O_2 removed all Ericsson's technology from their network after two outages that left more than 2 million persons without communications for an entire day (Whittaker, 2012). Many other cases exist, such as Facebook and WhatsApp outages due to misconfiguration and technical problems (DailyMail, 2015; Engadget, 2019; Verge, 2019).

What is the common aspect in the previous examples? All of them had a direct impact in the client, and some of them were firstly detected by clients, before being acknowledged by companies.

Monitoring thus aims to help systems and networks to achieve high resilience standards. Additionally, several studies have confirmed that the lack of investment in monitoring may cause economic and image damage to a company (Gartner, 2013; Owezarski et al., 2013; Rimal et al., 2009; Verizon, 2013).

In Oppenheimer et al. (2003), three of the largest-scale Internet services, "operate their own 24x7 System Operations Centers staffed by operators who monitor the service and

respond to problems". For Joyce, "monitoring of distributed systems involves the collection, interpretation, and display of information concerning the interactions among concurrently executing processes." (Joyce et al., 1987). Almost two decades since Oppenheimer' statement and three decades after Joyce's, very little has been made to change this approach to system administration (Ewaschuk and Beyer, 2016; Spotify, 2018; Vector, 2018). We still resort to alarms, dashboards, agents, and a plethora of other tools to assist administrators in the task of finding patterns and failures in distributed systems. Interestingly, with all these tools, the client continues to be somehow disregarded in the monitoring flow. The client or machine outside of the monitored system may contribute with information that eludes system operators, but improves system monitoring.

A significant part of recovery time (and therefore availability) is the time required to detect and locate service failures. Quickly detecting the root cause of a problem can be the the most important hurdle to improve availability of the system. Classic techniques suffer from false positive alarms, thus reducing the operator confidence in the tool. Dashboards, for example, put the responsibility to find any kind of problem on the operator (Bodic et al., 2005; Kiciman and Fox, 2005; Padmanabhan et al., 2006).

In spite of the numerous observation and monitoring tools available, building responsive distributed systems remains a difficult challenge. Part of the problem lies on the lack of client-side information that could either complement or altogether replace existing monitoring systems.

1.2 Main Objectives and Approach

There are a plethora of software solutions to monitor and alert system administrators for problems occurring in the application. These tools are normally classified as low-level, application and user-level solutions. Internal monitoring solutions running at the system or middleware layer, provide feedback for the overall status of the machine. In this category, we have programs like agents, probes, and system tools such as `Nagios` or `Zabbix`. Although very simple, these tools lack the application view that provides other important insights to administrators.

At the application level, there is a large diversity of solutions. Some companies and even academic research aim to create customized tools that fill some monitoring gaps.

Other implement more intrusive frameworks, such as tracing, to inspect correlation among services, thus being able to create system-wide graphs or other complex system visualizations. Other tools, mostly Application Performance Management tools (APM) aim to create notifications or dashboards for the system (AppDynamics, 2017; DynaTrace, 2017; NewRelic, 2017). Each APM competitor brings its own features to the table, to enable the creation of a small (but precise) picture of the system. However, they lack the most important part of the system: the final customer.

The previous systems do not take into consideration if the client is having some technical difficulty, even when the system appears to be fully functional. Real user monitoring suites (RUM), like Pingdom (Pingdom, 2017), Monitis (Monitis, 2017) or Bucky (Bucky, 2017) do this to some extent, by relying on clients' data, but they mostly serve to create dashboards and trigger notifications according to a set of rules, or are more oriented to search engine optimization (SEO) tasks.

Even with all these monitoring solutions, problems regarding the status of the system still persist. First, the majority of the monitoring solutions are intrusive, being difficult to operate and maintain in legacy systems. Secondly, they are closely coupled to the system in terms of technology and infrastructure. Normally, we have white-box solutions, intrusive and with internal access to the system. Unlike this, black-box solutions have no access to the internal state of the distributed system. They must use metrics observable from the exterior, e.g., from the client. A client-side black-box solution could help decoupling the distributed system from monitoring, thus providing independence between monitoring and infrastructure (e.g. language and technology). Client-side information can also enrich current white-box solutions. Combining white and black-box can improve monitoring solutions and the overall health of the system. The main goal of this thesis is, therefore, to develop monitoring solutions that involve the use of data collected by the client.

In the pursuit of understanding how would client-side monitoring work, we need to address the following sub-objectives:

1. Automatically infer the internal state of the server, using only client reads (black-box). For this, we aim to understand the effects of distinct bottlenecks on the clients and how to improve the overall monitoring in more complex and real scenarios, with the purpose of having autonomous and online prediction mechanisms.

2. Determine how to incorporate client reads in state-of-the-art monitoring tools (white-box). With the insights and information collected from the client, we aim to understand the viability of adding client data to monitoring tools. For this reason, we explored the limits of this kind of solution and propose a hybrid solution, where we can merge information collected and observed by the final clients with system internal information.

3. Understand the limits of client-side monitoring. A full black-box monitoring approach is difficult, since we depend on information that the system may leak to the client. We need to understand what kind of information is leaked by the system in different operational scenarios (e.g. system affected by some sort of bottleneck) and identify patterns in observable metrics foreseen by clients such as end-to-end request and response times.

1.3 Results and Contributions

This thesis aims at improving the current state of the art of distributed systems monitoring. The next chapters are the result of our endeavours.

Chapter 2 presents the current state of the art. This state of the art allowed us to understand limitations and challenges of current tools that need to be addressed. Additionally, this chapter allowed us to understand what goals and results we might achieve. In this chapter we summarize knowledge that is transversally used in the rest of the thesis.

In our survey of Chapter 3, we analyze the most used web sites worldwide, based on the majority of errors that a client can perceive, characterizing them by connection errors, logical errors and JavaScript errors. The classification is done by resorting to standard tools and frameworks using only client-side data. In our survey we observe that even the most used websites are bound to failures, therefore providing us with an extra motivation to improve monitoring solutions.

Our efforts to understand how performance issues and bottlenecks are identified outside of the system take us to Chapter 4. The work of this chapter allowed us to understand patterns for distinct synthesized bottlenecks in a laboratory experiment. Using client-side data makes this solution independent from the system. Additionally, we analyzed

several web sites to understand if we could identify the same patterns in an uncontrolled environment.

We then tried to extract useful data combining information from several clients. Since we wanted to extract patterns and resort to autonomous bottleneck detection in Chapter 5, we used machine learning techniques. We provided data from several clients and used supervised techniques to understand if they could identify patterns associated with bottlenecks.

Chapter 6 gives insights on how to achieve monitoring in very dynamic microservice architectures. This naturally extends our previous approach, because each microservice behaves like a client for the next microservice in the chain.

In detail, this thesis main research contributions are:

1. A survey on the main issues involved in current monitoring solutions regardless of their level of intrusiveness (i.e. white-box or black-box solutions). This allowed us to identify gaps in the current state of the art.
2. An analysis of the major HTTP solutions, such as web sites or REST interfaces. This analysis can help us to understand which kind of problems are more prone to appear to the client.
3. A laboratory experiment where we synthesize several bottlenecks and evaluate how do they show up on the client. This is very useful to understand the limits of a full black-box solutions.
4. An autonomous bottleneck identification technique, using machine learning.
5. Extending the previous methods to the increasingly popular microservices architecture using nothing more than the information collected from the invoking side.

It is important to mention that Chapter 6 was a team effort. The "exponential variable sum algorithm", in Section 6.2.1, and the "tandem queue model fitting", in Section 6.3.1, models are not from my authorship. I include them here for self-containment, as I need to use them as a baseline for comparison to the machine learning approaches. The main author of these algorithms was another PhD student called Jaime Correia.

In addition to the aforementioned direct contributions, we believe that this thesis can pave the way to better monitoring solutions. Through more observable distributed components, we aim at improving what is most important: the client's quality-of-experience.

1.4 Thesis Structure

Chapter 1 introduces the research topic, the motivation, problem characterization, and main contributions.

Chapter 2 presents the background and state of the art in monitoring distributed systems.

Chapter 3 presents a study on a large list of online HTTP application, such as web sites or REST interfaces. We aimed to understand if server-side errors are common and whether clients can detect them.

Chapter 4 presents an experiment on how to detect bottlenecks in a distributed system from the client-side point-of-view. We used some classic statistical analysis to create a client profile concerning the web page time retrieval.

Chapter 5 extends the previous scenario, by having multiple clients invoking the same resource. We correlated all this information to identify bottleneck patterns. For this, we used machine learning techniques to pinpoint distinct performance issues.

In Chapter 6 we analyze a microservice architecture. The monitoring solution is built resorting to Netflix components and to simulation. In this chapter we try to understand if the same methods that we use in the previous chapters can be adapted to this kind of applications.

Finally, Chapter 7 concludes the thesis and proposes future directions for this research.

Chapter 2

State of the Art on Distributed Systems Monitoring

Monitoring takes an important role in distributed systems, to ensure quality-of-service to the client. The importance of a proper monitoring is a thematic that has been evolving from monolithic to distributed solutions. Distributed systems — an evolution of monolithic architectures —, have created even more difficult, complex and unpredictable situations to system administrators.

As stated in Section 1.1, distributed systems are highly dynamic, may have distinct technologies and may be part of a large ecosystem. The plethora of problems and monitoring solutions for such systems is consequently intimidating.

In this chapter we present some of the most advanced monitoring solutions for distributed systems, as well as some fundamental concepts. We divide the topics of this chapter into several Sections. First, in Section 2.1, we present the main concepts relevant to our work such as microservice, observability, tracing, logging or debugging. In Section 2.2, we present the common use cases associated with monitoring distributed systems, from anomaly detection, system modelling and occupation prediction to improving system efficiency. Afterwards, in Section 2.3, we present techniques and studies addressing these use cases. In Section 2.4 we present concrete tools and frameworks used for monitoring. Section 2.5 is focused on studies associated with web site reliability, as some of our work is related to this topic.

Finally, in Section 2.6 we present our conclusions and how we can contribute in this demanding field of research.

2.1 Concepts

2.1.1 Monitoring

Monitoring can be defined as the act to observe and retrieve relevant data and tracking the events occurring in a system. With this information, monitoring aims to supervise and report the overall health of the system to administrators (Techopedia, 2019b).

For Google, monitoring is "collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes." (Ewaschuk, 2019; Ewaschuk and Beyer, 2016). These authors describe some of the Google's challenges in terms of monitoring. From the definition and taxonomy of some monitoring concepts to the description of some of the monitoring solutions implemented, this study gives a general overview of the difficulties to maintain a simple (but effective) tool in a production environment.

We can decompose monitoring into two main branches: white-box and black-box.

White-box monitoring

White-box monitoring is "based on metrics exposed by the internals of the system, including logs, interfaces like the Java Virtual Machine Profiling Interface, or an HTTP handler that emits internal statistics." (Ewaschuk, 2019). This kind of monitoring although very invasive, forcing the instrumentation of the code or the need of agents, provides a complete view of the system.

To collect data, probes and agents gather, collect and transmit information about the machine to a central point. This information can then be monitored to understand if a threshold is exceeded, or a specific pattern occurs (e.g. by using anomaly detection techniques).

In the literature, the aggregation of agents, probes and information collected about the infrastructure can be named "Infrastructure Performance Management" (IPM). IPM

provides a complete overview about the system with an emphasis on CPU, network or memory used on each machine. Some examples of these components are `Zabbix` or `Nagios` (Nagios, 2019; Zabbix, 2019).

Application Performance Monitoring (APM) solutions can also be considered white-box. APMs are intrusive, but can generate a clear image of the system. They are strongly coupled to the system, enabling analysis of specific applications, recurring to agents, tracing and dashboards, to retrieve insights to administrators. APM suites can decompose the application in it subsets, such as components or databases, creating a workflow of service invocation (AppDynamics, 2017; NewRelic, 2017). Comparing IPM and APM solutions, IPMs are less intrusive and need fewer resources, but APMs are application-oriented and allow administrators to understand users' quality of experience.

Tracing, which we detail in Subsection 2.1.4, is the ultimate white-box approach, by exploring instrumentation of the source code to generate better insights about the application, beyond simple infrastructure metrics.

Black-box monitoring

Black-box monitoring is "Testing externally visible behavior as a user would see it" (Ewaschuk, 2019). We can classify the technologies as Real User Monitoring (RUM) suites under this category. Tools like Monitis (2017) or even GoogleAnalytics (2018), provide a perception of the system from the outside, collecting information from the clients and their interactions with the system. Since black-box is independent from the internal details of the system, it does not have as much information as white-box applications. Black-box approaches usually check threshold violations and provide general dashboards and notification rules to warn system administrators.

Applications like Pingdom (2017) or the open source project Bucky (2017) are focused on the presentation layer. They rely solely on client's data, but can generate viable information that may elude system administrators control (e.g. third-party providers). They can combine information from multiple clients, using general metrics visible by the clients, such as latencies. Thus, this kind of application is mostly interested in the system metrics *per se*, not aiming to gain insights of more complex patterns happening inside the system.

2.1.2 Observability

Observing is normally defined as "to be or become aware of, especially through careful and directed attention" (TheFreeDictionary, 2019). Several definitions of observability have been introduced in the literature, some of them from the area of control theory, where this concept emerged (Brooker, 2012).

In Özveren and Willsky (1990), observability is "defined as having perfect knowledge of the current state at points in time separated by bounded numbers of transitions.", requiring knowledge of the events currently being processed in the system. In Lin and Wonham (1988) a similar definition is presented, with the knowledge of external outputs of the system, to ensure an always-observable system. Observability – a metric of how well the internal state of a system can be determined from its external outputs (Kalman, 1959) – is key to ensure responsiveness of large-scale online systems.

Newman (2015) observed that in microservices: "We cannot rely on observing the behavior of a single service instance or the status of a single machine to see if the system is functioning correctly. Instead, we need a joined-up view of what is happening. Use semantic monitoring to see if your system is behaving correctly, by injecting synthetic transactions into your system to simulate real-user behavior. Aggregate your logs, and aggregate your stats, so that when you see a problem you can drill down to the source. And when it comes to reproducing nasty issues or just seeing how your system is interacting in production, use correlation IDs to allow you to trace calls through the system.".

Taking into consideration these definitions, observability is a concept that requires techniques as the ones mentioned in Section 2.3. In a simplistic way, we can consider that monitoring is the action to get the state of a system, and observability is the property of a system to externalize the inner status.

Perfect observability would require extensive instrumentation of source code with agents dedicated to software and hardware resources. Unfortunately, an intrusive monitoring solution may not be desirable or possible due to technical constraints. Covering the entire system may also be too complex or too expensive. Hence, inferring the state of the individual components without invasive monitoring techniques can bring concrete benefits for the observability of the system.

2.1.3 Microservices

Microservices were first introduced in 2005, by Dr. Peter Rogers, with the name "micro web services" and afterwards in 2011 at a workshop of software architects, the pillars of this architecture were discussed, finalizing in 2012 with the adoption of the name "microservices" (Dragoni et al., 2017; Lewis, 2019).

Within a monolithic system, developers and administrators need to contain all code and functionalities in the same place. However, since monolithic systems are not functionally decoupled, similar functionalities may be copied in an erroneous way. Additionally, in a monolithic solution we are forced to ensure that all the system uses the same technologies. One of the consequences of this, is the evolution of the system. It is more difficult to evolve one component, without having to evolve all the system, thus, being more difficult to create new versions of some components.

Evolution of this new paradigm also benefited from a number of other factors. Microservice architectures are better suited for deployment and operation with containerization technologies such as Docker (Docker, 2018). Additionally, emergent methodologies in product development, such as Agile or Development and Operations (DevOps), with smaller teams that work independently, are well aligned with microservice architectures. Therefore, microservices bring tremendous benefits in term of development, operation, availability and scalability, and have thus become a trend for distributed large-scale systems. Finally, early adoption from major providers, such as Netflix or Amazon helped the dissemination of microservices (Atchison, 2019).

Microservices aim to be "an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities" (Dragoni et al., 2017). Each component is function-oriented or as mentioned by Robert C. Martin: "Gather together those things that change for the same reason, and separate those things that change for different reasons." (Martin, 2019). In Figure 2.1, from Lewis (2019), we can see an image describing the major differences between these two approaches.

Taking into consideration the versatility and agility of microservices, monitoring becomes a challenging task. In old monolithic systems, monitoring was restricted to a stable infrastructure. In microservice systems, administrators have to pinpoint the root cause of some anomaly in hundreds or thousands of machines. Sam Newman

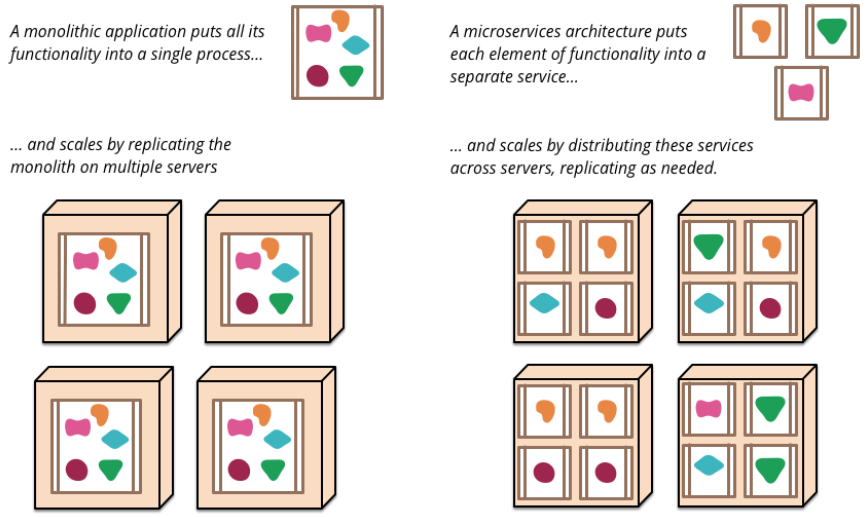


FIGURE 2.1: monolithic versus microservices (Lewis, 2019)

mentioned "breaking our system up into smaller, fine-grained microservices results in multiple benefits. It also, however, adds complexity when it comes to monitoring the system" (Newman, 2015). Some techniques are identical to monolithic solutions, such as the use of tools that retrieve information from the system as CPU or memory used — `Nagios` or `Zabbix`. However, since microservices are more complex, these tools cannot cover all the necessities of this kind of architecture.

2.1.4 Tracing

Tracing is a method that recovers causality relationships in distributed systems allowing users to make inferences concerning critical paths and relations among microservices. For OpenTracing (2019), tracing is "a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps pinpoint where failures occur and what causes poor performance". For OpenCensus (2019) tracing "tracks the progression of a single user request as it is handled by other services that make up an application. Each unit work is called a Span in a trace. Spans include metadata about the work, including the time spent in the step (latency), status, time events, attributes, links. You can use tracing to debug errors and latency issues in your applications."

Tracing is based on two components: spans and traces (Fonseca et al., 2007). Span represents a single operation in the service invocation, such as a database query or an

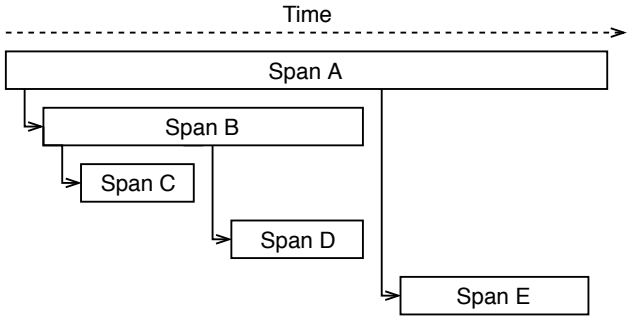


FIGURE 2.2: Sample trace over time (Bento, 2019)

HTTP request. A span may be triggered by another work unit, hence being considered a child span (e.g. a remote procedure call). A span without a parent is named parent node or root node. Each span is uniquely identified by a *spanID* and belongs to a single trace.

Traces represent the work done by a request in the system. Each trace is the sum of several spans, where a span represent an individual workload that occurs in a service of the system. All spans share a correlation identifier named *traceID* that sets relationships between spans.

Figure 2.2 represents a trace with five spans, with "span A" being the parent node (or root node), originating "span B" and "span E". The relationships can be inferred since the correlation identifier — *traceID* — , *spanID* and *parentID* are propagated to downstream spans. Another useful information, that can be seen in Figure 2.2, is the duration of each span, and the possibility to have multiple spans working at the same time.

Since tracing uses a correlation identifier to extract the workflow of a request, it is a good approach to monitor distributed systems (Sambasivan et al., 2016). The most important feature of tracing is the ability to follow a request throughout the system, giving insights about how each individual component acts to a specific request. Although very powerful, allowing to pinpoint failures in a distributed system, tracing has some disadvantages. Each component must be instrumented. Additionally, all traces must be delivered to a central point, to be processed and stored, for future analysis.

There are two major areas regarding tracing: black-box and non-black-box approaches. Concerning black-box, these methodologies rely on message-level traces of the system,

or thread and network activities, as a middleware to detect request paths (Aguilera et al., 2003; Tak et al., 2009).

Non-black-box approaches, such as in Fonseca et al. (2007), use meta-data to trace networks. Building on the same principle, Sigelman et al. (2010) created a tracing infrastructure for distributed applications. Sambasivan et al. (2016) use a similar approach to gain insight at the application level, in particular workflow-based tracing, concerning the tracing of individual requests.

OpenTracing (2019) and OpenCensus (2019) are other examples that give developers the ability to trace requests and integrate multiple languages, which can be used with state-of-the-art tracing back-end tools. These open projects also aim to create a convention in terms of semantics and annotations of the trace (e.g. span tag name such as `http.method`, to save the invoked HTTP method) (Specification, 2019).

2.1.5 Logging

Logging can be defined as "record events taking place in the execution of a system in order to provide an audit trail that can be used to understand the activity of the system and to diagnose problems. They are essential to understand the activities of complex systems" (EventLogs, 2019). Logging is a monitoring technique that "is not about prevention, but can help with detecting and recovering"(Newman, 2015). It allows to understand what is happening to a request by externalizing valuable information. Hence, it is very useful to track application errors, helping administrators and developers. In this technique, it is important to understand what information is useful and what information may be sensitive to be stored in logs (e.g. name or address of the client). Hence, logging must store all the information relevant to make debugging possible without exposing information that may conflict with some regulation (e.g. General Data Protection Regulation — GDPR). Another important requirement for logging is the capability to produce asynchronous logs to ensure that business threads will not be blocked or delayed by the generation of logging messages (Janapati, 2019).

There are plenty of logging libraries in every language to standardize processing the generated logs — e.g. according to severity of the log; info, warning, error, etc. Density of the logs (i.e. number of logs *per* lines of code) and quality of the information contained in the logs, is a very important aspect. Studies show that the majority of

TABLE 2.1: Open-source software and logging quality (Yuan et al., 2012)

log msgs	apache	openssh	postgres	squid	total
modified	605	628	3128	1106	5367
total	1838	3407	6052	3474	14771
percentage	40%	18%	52%	30%	36%

open-source software in production only logs error events. In Table 2.1 we can see a study related with open-source code (i.e. apache, openssh, postgres database and squid) and the quality of logging. We can observe in the table the number of logs presented in the application and the number of logs changes after development. We can observe that around 36% of total logs were changed after the development phase. The changes in the logging were due to log location, verbosity, availability of variables in the log message or to the contents of the text to print (Yuan et al., 2012). Hence, logging is an important aspect of an application, often requiring changes in the quality or density of the logs.

Logs from multiple sources may also be aggregated. Using aggregation of logs for distributed systems, we can enrich root-cause-analysis and detect some problems that elude single logs, or even use statistical analysis to find patterns. Aggregation and transmission of logs to a central point can be obtained with frameworks such as Logstash (2019). Technologies such as Kibana (2018) — based on ElasticSearch —, allow to query through the centralized logs to a faster analysis (Barga et al., 2004; Janapati, 2019; Johnson, 1989). In Subsection 2.1.5, we will further analyze logging tools.

2.1.6 Debugging

Debugging can be defined as "the routine process of locating and removing computer program bugs, errors or abnormalities, which is methodically handled by software programmers via debugging tools. Debugging checks, detects and corrects errors or bugs to allow proper program operation according to set specifications." (Techopedia, 2019a) Debugging provides insights about what is happening in a system.

Debugging is normally associated with the development phase, but can also be associated with the inference of performance issues — e.g. performance debugging —, in a distributed system (Aguilera et al., 2003; Khadke et al., 2012; Sambasivan et al., 2014). Debugging in this sense is the act to infer and understand what kind of bottlenecks

and performance issues are happening in a system. It is this kind of debugging that is mostly associated with this research work.

Performance debugging aims to detect anomalous situations in the distributed system, including abnormal situations, transient behaviors, profiling of the application, resource utilization in platforms such as cloud computing, and workload modelling to infer components' occupation.

2.2 Use Cases

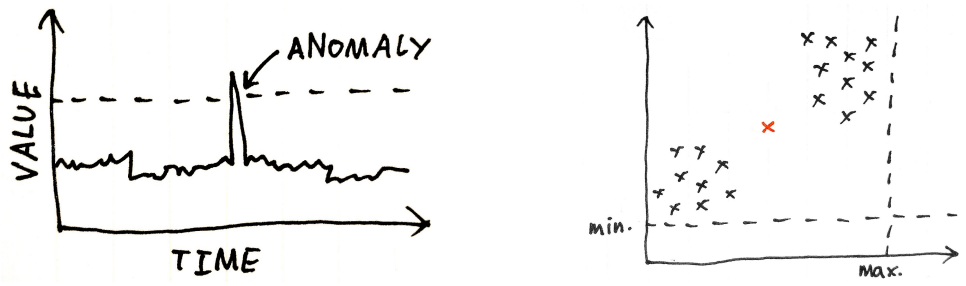
In this section we present some of the use cases associated with distributed systems monitoring. The use cases are presented as a set of specific situations that monitoring aims to detect and prevent. We consider the following situations, as mentioned in Sambasivan et al. (2014): anomaly detection — a situation that is outside of the common pattern of the system —; steady-state, i.e. a transient problem that eludes anomaly detection techniques; distributed profiling, to gain insight of the application; resource utilization on pay-as-you-go platforms and workload modelling, to predict bottlenecks or component occupation in the distributed system.

2.2.1 Anomaly detection

Anomaly Detection is defined as "the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data." (Zimek and Schubert, 2017). In anomaly detection, our main goal is to identify patterns or outliers that escape the common standard in a specific system — e.g. outliers that are in the 99^{th} percentile range.

Additionally, we can observe anomalies in time-series where observations of the system do not conform to the expected behavior, or even in a dataset where some samples do not fit in the majority of the data — see Figure 2.3.

Anomaly Detection is a use case where system administrators aim to understand behaviors that elude the common standard, like intrusion detection, fraud or abnormal financial transactions. One of the main challenges of Anomaly Detection frameworks is to correctly identify anomalous observations — e.g. maximize the true positive and true negative rates and decrease false positive and false negative events.



(a) Time Series. (b) Non-Time Series.

FIGURE 2.3: Anomaly Detection. (Mertz, 2019)

2.2.2 Diagnosing steady-state problems

Steady-state is defined as "an unchanging condition, system or physical process that remains the same even after transformation or change."(Steady-state, 2019). This concept was initially used in the field of energy consumption to determine (and monitor) in a passive way the consumption of electric appliances (Hart, 1992). However, it is easy to extended to other research fields. In Figure 2.4 we can see the illustration of what happens in steady-state series (e.g. steady-state situations) and also in transient situations.

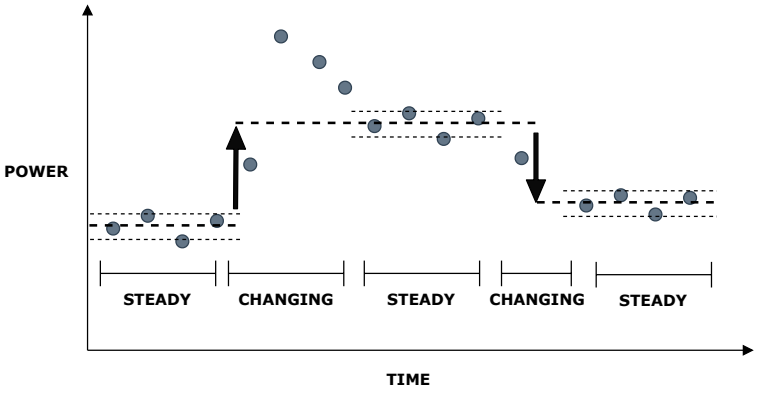


FIGURE 2.4: Detecting step-changes in sampled data. (Hart, 1992)

The issue may not be detected by common anomaly detection techniques, presented in the previous subsection, since as can be seen in Figure 2.4, the anomaly may not be transient and become the new state of the distributed system. Comparing anomaly detection with steady-state problems, an anomaly is an outlier defined as data in the 99^{th} percentile range, when we look to the overall performance of the application.

However, steady-state problems will manifest in multiple workflows, hence not being considered an anomaly — e.g. steady-state problems are below the 99^{th} percentile range.

This behavior may occur for several reasons such as misconfiguration in the application, a business configuration change, or other cause that may impact the overall performance of the system (Sambasivan et al., 2014).

2.2.3 Distributed profiling

As stated by Sambasivan et al. (2014) "The goal of distributed profiling is to identify slow components or functions. Since the time a function takes to execute may differ based on how it is invoked, profilers often maintain separate bins for every unique calling stack, so full workflow structures need not be preserved".

There are two main approaches for a distributed profiling. In monolithic solutions, we had the capability to create end-to-end profiling of a request, aiming to detect low performance functions. Distributed profiling aims to extend this kind of methodology. One of the approaches is to focus on overall low performance components (Chanda et al., 2007; Sambasivan et al., 2014).

Another possibility is to use tracing to track low performance functions. Since tracing can generate workflow structures, it may be used to create an application' profiling (Sambasivan et al., 2016; Sigelman et al., 2010).

2.2.4 Resource attribution

As stated by Sambasivan et al. (2014), resource attribution or resource utilization can be defined as "Who should be charged for this piece of work executed deep in the stack of my distributed system's components?". This sentence raises an important question for "as a service" cloud architectures. In this kind of system, the client is normally charged for the used resources in terms of computational performance and time spent. Hence, scenarios of pay as you go (PAYG), such as cloud computing, billing and resource payment need to assign resource utilization (Chen et al., 2013; Wachs et al., 2011).

Resource attribution consists of setting the correspondence between a resource in the distributed system to a work submitted by a client, having into consideration constraints like billing limitations, or energy consumption in embedded systems. Resource attribution may also be used in other scenarios, where we want to ensure that all clients have a fair resource usage of the infrastructure (Fonseca et al., 2008; Mace et al., 2015).

2.2.5 Workload modelling

A tool that performs workload modelling is a tool that collects "stand-alone events generated by operating system, middleware and application components, correlates related events to extract individual requests, expresses those requests in a canonicalized form and then finally clusters them to produce a workload model" (Barham et al., 2004).

Workload modelling normally depends on instrumentation — refer to Subsection 2.1.4 —, to collect information from the distributed system. As stated by Sambasivan et al. (2014), although instrumentation is essential to workload models, the main focus is to create a model that can explain the distributed system. Additionally, it is also very important to understand what kind of question the operator is trying to respond — e.g. "Where are requests from client A spending most of their time in the system?"(Thereska et al., 2006).

Other studies do not rely on instrumentation, but on modelling techniques, such as queuing models, but with the same goal: answering questions about the system that might not be answered using the real (i.e. in production) distributed system (Sambasivan et al., 2016).

2.3 Monitoring Techniques

In this section we present some of the most important techniques used in distributed system monitoring. They are closely coupled to the use cases of the previous section, and we can interpret these techniques as practical implementations of solutions for the aforementioned use cases.

We divide the section into four main subsections: first, we present high level monitoring dashboards; secondly, we show how alarms can be used in monitoring; thirdly, time series are presented; and finally we present modelling techniques.

TABLE 2.2: Use Case vs. Technique

Use Case	Technique
Anomaly Detection	Dashboards; Alarms; Time Series;
Steady-state problems	Dashboards; Alarms; Time Series;
Distributed Profiling	Time Series;
Resource attribution	Modelling Techniques;
WorkLoad modelling	Modelling Techniques;

Table 2.2 presents how the use cases are related with the techniques presented in the following section. Some of the techniques may respond to several use cases, such as Dashboards and Alarms. This is due to the fact that these techniques are key elements to create insights about the distributed system that may be used for different goals (e.g. anomaly detection and steady-state problems).

2.3.1 Dashboards

A dashboard is one of the most important tools for an administrator. A definition of dashboard is "A software-based control panel for one or more applications, network devices or industrial machines. Dashboards may display simulated gauges and dials that look like an automobile dashboard or a factory assembly line, or they may show business graphics such as pie charts, bar charts and graphs." (Dashboard, 2019).

Dashboards are visualization tools that merge key performance indicators (KPI), notifications and alarms triggered by the system. Their main goal is to provide a central point of information about the health of the system for administrators. There are several applications to make dashboards. Almost every business application resorts to some sort of dashboard to monitor the system. For example, in Figure 2.5, we can see an example of `kibana` (Kibana, 2018), a visualization tool for `ElasticSearch`. Other very well know tools currently used are `Grafana` (Grafana, 2018), `Prometheus` (Prometheus, 2018), `Zabbix` (Zabbix, 2019), or `Graphite` (Graphite, 2019).

Concerning industrial approaches, we have solutions owned by the infrastructuture provider (e.g. Amazon CloudWatch, Netflix or Linkedin) or third party modules such as NewRelic (2017) or DynaTrace (2017). Netflix has several modules for monitoring and instrumentation. Vector (2018), is a framework that creates dashboards with system metrics, such as CPU or network. The module requires an agent — named Performance

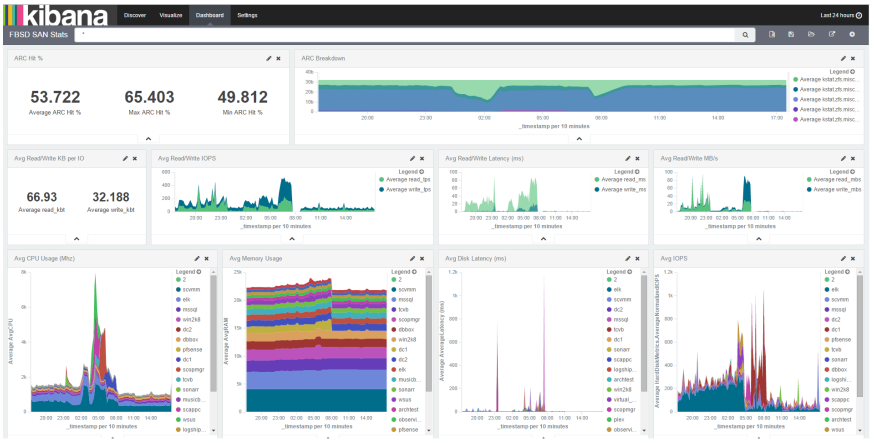


FIGURE 2.5: Kibana Dashboard (Kibana, 2018)

Co-Pilot (PCP) —, on each host or application to monitor. Another very similar approach is Prometheus (2018), an open-source monitoring solution that also requires instrumentation.

Application Performance Monitoring (APM) tools, based on instrumentation or agents, allow the creation of dashboards and the definition of notifications to administrators, when some threshold is violated. For example, DynaTrace (2017) and others (AppDynamics, 2017; Grafana, 2018; Kibana, 2018; NewRelic, 2017) have some features related to dashboards.

Cloud providers are also an interesting case to cover in this subsection, as they have their own monitoring tools, supporting configurable dashboards, like Amazon (CloudWatch, 2019) or Azure (AzureMonitor, 2019). Another interesting approach is presented in Instana (2019), where the APM presents multiple dashboards and performance analysis for microservice platforms.

2.3.2 Alarms

Alarm is normally defined " as a device or call announcing a warning or danger" (Alarm, 2019). Another definition for an alarm is "a signal signifying to an operator that an abnormal state has occurred" (Us et al., 2011). Alarms are a common piece of software in the majority of software infrastructures from aircraft cookpits, to nuclear plants, including telecommunication operators. An alarm is triggered by a source that indicates

an error, malfunction or other deviation from the normal behavior, while events alert an operator that may have to interpret and acknowledge the situation.

Concerning the source, in most cases we have three distinct origins for the alarm: infrastructural or hardware, some malfunction of the software or a violated KPI (e.g. number of requests per second below a specific threshold).

Although the definition contemplates only events with warning or critical severity, it is common for the sources to also notify events that may have lower severity (e.g. informative events, such has sporadic loss of network connectivity). These events may be handled in an alarm manager platform, capable of handling multiple sources of events to create more complex behaviors.

2.3.3 Time Series

Time Series are a way of representing data as a sequence of values sorted by time. This kind of data often arises from system monitoring, business metrics or even financial indexes. These processes are usually not random, thus having periodic behaviors. Hence, autocorrelation can be explored to extract insights from the data. Time-series can be analyzed to detect patterns in the system — i.e. outliers —, such as in Li (2019); Liu et al. (2004). Hence, anomaly detection in time series data is a common application to get details about outlier occurrence; another typical application is forecasting.

There are some essential concepts associated with time series that almost every tool uses underneath. Time series are often associated with sliding windows. This time model considers the most recent N elements of a stream, discarding older samples (and their importance), thus being a window of N elements (Das et al., 2012). This concept allows us to analyze data in a real-time manner: when it is infeasible to save all data due to limited computational resources, when the importance of data decays with time (the actual state of the system is more important than older state), and finally when quick responses with little delay in the answer are important (Muthukrishnan et al., 2005). In time series, when we want to characterize the stream, there are some common statistical metrics such as variance, quantile or frequency of an event that might help in the analysis.

Although it is a very used technique, there are some challenges in time series analysis, specially in distributed streams (from distinct origins). First, how to relate multiple

stream sources (e.g. sequentially, parallel invocations, ...) might not be trivial; secondly, we might have some streams that change the frequency of sample creation; and finally since streams may use different clocks we might have out-of-order events (Cormode et al., 2008; Das et al., 2012; Liu et al., 2010).

Concerning the tools existing in the market, Atlas (2018) — an open project from Netflix —, presents a platform to manage multidimensional time series. This platform is focused on big data. The goal is to apply prediction methods, to understand the evolution of metrics and real time analysis. Although powerful, this platform requires instrumentation of microservices. Spotify uses a similar approach. They had the need for a customized monitoring infrastructure that creates dashboards and time series. Each machine runs an agent to send information to a central point (Spotify, 2018).

A common acronym in time-series is Time-Series Database (TSDB). There are several platforms for TSDB, such as (Influx, 2018; OpenTSDB, 2018; TimescaleDB, 2018; Warp10, 2018) with multiple features such as structured query language, integration with distributed filesystem, such as Hadoop Distributed File System (HDFS), or dedicated languages, to identify patterns or anomalies through data analysis.

Another approach that relies on time series is Pip (Reynolds et al., 2006). As stated by authors "Pip is an infrastructure for comparing actual behavior and expected behavior to expose structural errors and performance problems in distributed systems.". This framework compares baseline series with the actual behavior of the system, to detect steady-state problems.

2.3.4 Modelling Techniques

Modelling Techniques are associated with the use case of workload modelling presented in Subsection 2.2.5. The main goal is to answer questions about the system, without having to change it. A model of the system is created, for example to understand what could happen when we add (or remove) components, such as virtual machines.

Regarding academic literature, Iqbal et al. (2010, 2011) propose an algorithm that processes proxy logs and, in a second phase, all CPU metrics of web servers. The purpose is to achieve elasticity concerning the number of instances of the saturated component. Liu and Wee (2009) use static performance-based rules. In this approach, if saturation is observed in a component resource, then, the user will be migrated to a

new virtual machine through IP dynamic configuration. In Battre et al. (2010), authors try to discover bottlenecks in data flow programs running in the cloud. They focus on CPU and I/O bottlenecks, and not on predicting occupation. A different approach was followed by Chi et al. (2011), where the main goal is not bottleneck detection, but optimal resource utilization using heuristic models.

Bahl et al. (2007) propose an inference Graph model from network traffic, to model and detect service degradation and failures. Their work is strongly tied to the enterprise network topology. Urgaonkar et al. (2005), propose an analytical model based on multi-tier queues for multi-tier Internet services. Work like Dilley et al. (1998); Li (2010); Yang et al. (2013) use networks or layered queues to model the distributed system. They do manual modelling at design time or modelling from deep knowledge of the system, instead of trying to extract the model from an existing system or considering other data-driven approaches. Other approaches try to model the service performance and response times. This is the case of Cao et al. (2003), who model classic web servers as $M/G/1/K * PS$ queues. However, models like $M/G/1$, $G/G/c$, with no assumption of processing time, are not amenable to closed-form solutions and cannot be easily composed.

Heinrich *et al.* (Heinrich et al., 2017) explore microservice-based systems and point out that the modelling approaches available do not fit modern microservice-based systems.

In Cao et al. (2003) the goal is to model web servers as single server queues in terms of response time and overall system performance. Van Do et al. (2008) use a similar approach for an Apache Web Server. In Shoaib and Das (2011), layered queueing networks are used to model a system with two layers – frontend and backend.

Kattepur and Nambiar (2015) present a model for Multi-tiered Web Applications using queues for individual components, such as CPU, I/O or Network. Singh et al. (2010) present an approach that uses a queuing model for each tier of the system, to predict the server capacity for a given workload.

Barham et al. (2004) present Magpie, a framework with a specific application event schema, to correlate events and extract a workload for the distributed system.

Zhao et al. (2016) present Stitch, a non-intrusive framework that creates a profiling of the application based on the unstructured logs with the assumption that the program outputs viable and enough information to create a profile.

Huang et al. (2017) present VProfiler, a tool that creates a model to identify blocks of code dominating computational latency. It uses the source code and annotations of the application, to suggest improvements in the code.

2.4 Monitoring Tools

In this section we present some of the most used tools in distributed system monitoring. They are closely coupled to the use cases of the previous Section 2.2.

We divide the section into three main subsections: first, we present logging tools that system administrators may use. Secondly, we present tracing tools that can extract relationships between service invocations. Finally, we show agents that act like a middleware between the system and the high-level techniques aforementioned, such as Dashboards or Modelling Techniques and that are essential to collect and gather system metrics.

Table 2.3: Use Case vs. Tools

Use Case	Tools
Anomaly Detection	Tracing; Logging; Agents
Steady-state problems	Tracing; Agents
Distributed Profiling	Tracing; Logging;
Resource attribution	Tracing; Agents
WorkLoad modelling	Tracing;

Table 2.3 presents how the use cases are related with the tools presented in this section. Tracing tools can be used in every use case, since they collect fine-grain information about the distributed system, necessary for several use cases. As mentioned in Subsection 2.1.5, the majority of applications on production environment only log error events. Hence, logging is useful for anomaly detection scenarios and also for profiling, since they provide hints about application behavior. Agents can give insights about the systems' components. Hence they provide information on anomaly detection, steady-state or even resource attribution. Although Table 2.3 gives relevance to some tools, with some "engineering" and effort it is possible to use tools to other use cases (e.g. logging to steady-state problems).

2.4.1 Logging Tools

As mentioned in Subsection 2.1.5, logging is an important part of observability. It is often used to ensure a fast root cause analysis when a problem occurs in a distributed system. Almost every programming language has a framework for logging and managing appenders (Janssen, 2019; Loggly, 2019). Appenders make part of the logging system

and are responsible to route the logs to a specific file or destination (Dietrich, 2019). Other functionalities for logging systems include asynchronous logging, maintenance of the logging files (e.g. create a new logging file each day), and other framework-specific functionalities — e.g. some logging frameworks on Python log events from dictionaries rather than from strings, thus structuring the logs.

There are plenty of applications to centralize logs and ensure data analysis through the logs. Some of them are Graylog (2019); LogEntries (2019); Logstash (2019) or Flume (2019). They have several aspects in common: aggregation and maintenance of historical data, possibility to query logs and correlate between several systems and some libraries for data analysis.

2.4.2 Tracing Tools

Tracing takes an important role in distributed systems. Tracing, unlike standard monitoring solutions, exposes causality relationships in the logs, allowing users to make inferences concerning critical paths and relations, e.g., among microservices. Regarding tracing, there are black-box and non-black-box approaches. Black-box approaches do not instrument application code, relying on middleware metrics. We have examples of that in Aguilera et al. (2003). In this work, authors used a tool that tracks message-level traces of the system, to debug the overall distributed system (not performing overall performance diagnosis). Another example is presented in Tak et al. (2009), where authors use threads and network activities, as a middleware to detect request paths.

Another methodology more commonly used is full instrumentation of the code (e.g. non black-box). In Fonseca et al. (2007) meta-data passing is used to trace networks. Building on the same principle, Sigelman et al. (2010) created a tracing infrastructure for distributed applications — Dapper. Sambasivan et al. (2016) use the approach to gain insight at the application level, in particular workflow-based tracing.

Linkedin (2019) has a distributed tracing system built upon Apache Samza (Samza, 2019), to detect performance issues and root cause analysis. The system uses the call paths aggregation of every 15 minutes. These call paths are used to benchmark the web page and compute the cost of invoking downstream services or web pages from the main page. Netflix (Netflix, 2018) has also created several monitoring tools using

distributed tracing, including failure injection features, to improve resilience of the overall infrastructure.

OpenTracing (2019) gives developers tracing clients in multiple languages and brings integration with state-of-the-art tracing back-end tools, such as ZipKin (2018). Google recently published a competing standard — OpenCensus (2019)—, supporting a partially overlapping set of the same back-end tracing tools. Another example of distributed tracing is Jaegger (2019).

Another solution is Pinpoint (Chen et al., 2004). This framework dynamically traces real client requests. For each request, it records the set of components used to service it. Afterwards, it performs data clustering and uses statistical techniques to correlate failures with the components most likely to have caused them. Pivot Tracing (Mace et al., 2018) uses instrumentation to correlate events in highly heterogeneous distributed systems. In Stardust (Thereska et al., 2006), authors create an infrastructure to use end-to-end tracing to create a monitoring tool in distributed systems. ETE or end-to-end response times uses instrumentation of overall components, to provide a complete picture of response times in the components of the distributed system (Hellerstein et al., 1999).

MilliScope is a monitoring framework that relies on an instrumentation tool associated with the framework — mScopeMonitors —, that gathers information in a similar way to the tracing technologies aforementioned. The main purpose of MilliScope is to identify transient bottlenecks.

While tracing (white-box methodology) gives the ability to understand the flow of individual requests, it has some disadvantages: developers have to instrument each microservice and focus not only on the business algorithms, but also on monitoring. On the other hand, black-box approaches may fail to give the insights that system administrators are looking for.

2.4.3 Agents

Agents or probes serve to monitor an application or infrastructure, having the purpose to gather data and generate performance reports. Some can even act in the case of a

module malfunction. Some agents are internal to the system and involve some intrusiveness, others are more black-box and serve to validate metrics, such as throughput. In the next subsections we will further detail both methodologies.

Internal Agents

In the literature, we can find a large body of work aiming to detect, predict and monitor distributed systems, usually in n-tier HTTP server systems (Battre et al., 2010; Bodík et al., 2009; Huber et al., 2011; Iqbal et al., 2011; Shoaib and Das, 2012; Wang et al., 2013).

Malkowski et al. (2007) aim to ensure low service response times. Authors collect many system metrics, like CPU or memory utilization, and correlate them with system performance. This should expose the metrics that best identify the performance degradation. However, this form of analysis collects more than two hundred system and application metrics. Malkowski et al. (2009), studied bottlenecks in n-tier systems even further, to expose the phenomenon of multi-bottlenecks, due to multiple resources reaching saturation. The main conclusion from this work is that lightly loaded resources may be responsible for multi-bottlenecks causing a chain reaction in the n-tier system. The framework used is very similar to their previous work, requiring full access to the infrastructure. Wang et al. (2013) followed this approach, with in-depth analysis of metrics in each component of the system. The goal was to detect transient bottlenecks with durations as low as 50 milliseconds. The problem with these approaches is that it is very hard to transpose the acquisition of such finely-grained data to different hardware and software architectures.

Battre et al. (2010) use DAG-based data flow programs running on cloud infrastructures, to detect CPU and I/O bottlenecks. Huber et al. (2011) present a dynamic allocation of VMs based on SLA restrictions. The framework consists of a continuous system introspection that monitors the cloud system and their components. This, however, requires continuous resource consumption (paid by the user) and scalability to large cloud providers.

X-ray (Attariyan et al., 2012) uses a distinct approach of internal agents. It instruments the binary code, to check what kind of process instruction was invoked. The goal is

to make a reverse-engineering of the application from the binary code level. Authors define this as as "performance summarization" of the application.

In Haselböck and Weinreich (2017), authors give some guidelines on how to build and monitor a microservice platform. The availability of instrumentation or agents to collect information from hosts and applications is assumed. In Mayer and Weinreich (2017), authors present a monitoring dashboard, but once again based on agents and service instrumentation.

In Toffetti et al. (2015), authors use a distinct approach, where each microservice is responsible for its own elasticity and scalability. Each module saves the information about their own CPU and response times. Thus, this methodology may lead to over-dimensioning the system, because each microservice has full autonomy.

In Malik and Shakshuki (2016), a tool based in the global entropy of a distributed system is presented to automatically detect anomalies. However, due to the fact that they do not rely on response time and other performance metrics their approach may lead to false positives.

In Ciuffoletti (2015), authors propose a methodology to create "monitoring as a Service", based on containers, where agents are associated with the microservice container. In this architecture, there is a one-on-one relationship between agent and container that may cause some overhead and scalability issues. Additionally, monitoring is associated with the container, in disfavor of the workflow that exists between modules. In Moradi et al. (2017), the access point of each container is changed to monitor the network. Hence, it has a kind of "man-in-the-middle" approach having no consideration about the application, or workflow of the system.

Another interesting approach is Whodunit (Chanda et al., 2007). In this paper authors aim to create a profiling tool in the shared memory of Apache and MySQL. With this information they aim to create a performance profiling of the application running on top of that infrastructure.

External Agents

This subsection reviews solutions that try to perform monitor without having access to the observed infrastructure. In Agarwal et al. (2010), authors propose a client-based collaborative approach. They use a web browser plug-in on each client that monitors all client Internet activity and gathers several network metrics. The plug-in focus is mainly the HTML initial page. It discards page resources from third-party providers, such as CDN objects and sends all information of the main site to a central point, for processing. The impact of this approach on network bandwidth and client data security is unclear. Additionally, the work of Agarwal et al. (2010) only handles network connectivity bottlenecks.

In Kreibich et al. (2010), authors present Netalyzr, a Java applet for browsers that clients can use to understand network connectivity issues. Netalyzr is mostly used when clients experience some problem. When a client wants to diagnose why some URL is slow, the tool makes several HTTP requests from other locations, gathering several network metrics. Although very powerful for connectivity issues, it does not analyze web page errors.

Dasu Sánchez et al. (2013) propose a client-based software. This has more than 90,000 installations, allowing the collection of metrics from different end users. As mentioned by the authors, it is limited by the number of hosts that are online and, consequently, cannot run continuous measurements. It only collects metrics associated with the client network point-of-view, discarding application measurements, such as HTML objects from third-party resources.

Flach et al. (2013) present a browser plugin that collects information and analyzes sites based on rules. Again, their paper focuses on network metrics and connectivity issues. Another similar approach is presented in Dhawan et al. (2012), where a Firefox extension based on Javascript was created to gather client information to diagnose network problems. Cui and Biersack (2013) is another plugin for the Firefox browser. It gathers network metrics for later evaluation of networking issues. Unlike the previous work, the collected data is not processed in real time, but transferred to a PostgreSQL database. Besides connectivity issues, authors also look at client performance problems occurring during page rendering.

In Li and Gorton (2010), authors aim to detect user-visible failures, by analyzing Web logs and users' browsing patterns to construct a Markov model. However, the fact that the client may not react to the visible failure (e.g., by leaving the page, or not refreshing it) and how would the web logs be collected from distinct users is a major concern in practice. In Li et al. (2010a), a similar idea is implemented: the interaction of a user with the web page serves to train a Bayesian model and infer if failures exist in that page. In this case, as in Li and Gorton (2010), authors focus only on AJAX requests.

Other tools (CheckMySite, 2017), allow the client to configure an URL to monitor. If something goes wrong, an alert (SMS or email) will be sent to the site owners. Unfortunately, these tools can only detect some network bottlenecks or a "slow" system, because they lack the fine-grained evaluation. Finally, some tools use a hybrid solution, with internal and external metrics (MonitoringTools, 2015). Additionally, Real User Monitoring (RUM) suites, like Pingdom (2017), Monitis (2017) or open source project like Bucky (2017) rely on clients' data, but they mostly serve to create dashboards and trigger notifications according to a set of rules.

There are also some hybrid tools that collect information from inside and outside the system. Although very powerful, they need constant maintenance, to ensure that the system is correctly monitored (MonitoringTools, 2015).

Table 2.4 illustrates the kind of resource problem detected by some of the aforementioned literature. Normally, authors aim to detect connectivity issues between the client and the distributed system, or analyze some sort of dashboards that gathers information perceived by the client.

TABLE 2.4: Monitoring specificities detection in related work

Article	Main focus
Agarwal et al. (2010)	Connectivity issues
Kreibich et al. (2010)	Connectivity issues
Vaz et al. (2011)	Connectivity issues
Sánchez et al. (2013)	Connectivity issues
Flach et al. (2013)	Connectivity issues
Cui and Biersack (2013)	Connectivity issues
Padmanabhan et al. (2006)	Connectivity issues
Li and Gorton (2010)	User-visible failures
Dhawan et al. (2012)	Connectivity issues
CheckMySite (2017)	Connectivity issues
Pingdom (2017)	Load time; connectivity issues
Monitis (2017)	Dashboards
Bucky (2017)	Dashboards

2.5 On Reliability of Web Sites

In the literature, we can find a large body of work focused on the reliability of web sites. Here, we present efforts that are of particular interest to us: Internet studies regarding top sites reliability.

In HTTPArchive (2016) — an open-source project —, the goal is to collect and understand the web page evolution over time. The project aims to understand the trends in terms of frameworks and technologies. The goal of this project is to register the evolution of the web along the years. Vaz et al. (2011) use a different approach, implementing a web crawler that gathers HTTP, DNS and TCP connection data from different locations. The goal is to understand in which layer do most of the user-visible page failures occur. The main disadvantage is the fact that authors built a customized crawler instead of using a common browser, to simulate the interaction with the web sites, something that might skew some results, e.g., related to JavaScript. Padmanabhan et al. (2006) use the PlanetLab (PlanetLab, 2015) infrastructure, to gather network information from 80 sites and analyze the source of the problems. Due to the fact that they use a specific infrastructure, they do not require any browser extension, thus being able to run out-of-the box tests. However, recent studies suggest that this pattern of concurrent accesses can significantly change the results observed (Sommers and Barford, 2007).

WebProphet (Li et al., 2010b) and Polaris (Netravali et al., 2016) are more focused on determining the dependencies between objects referred in the HTML, to decrease the page load time on the client. For this, they compute the critical path of dependencies and change the load sequence from the web page.

Palma et al. (2014), made a study to identify the lack of REST patterns in major operators. In Laranjeiro et al. (2009), authors evaluate the robustness of web services, using invalid call parameters, to observe programming or design errors. Although we also care for web services, the main concern of this thesis is on the reliability of invocations like in Ivaki et al. (2015) or Shegalov et al. (2002) and monitoring REST interfaces.

Mendes et al. (2018) present a study concerning web pages. It is focused in HTML errors presented in the web sites, and uses some techniques of web crawling to retrieve information. Their main purpose is not monitoring but to analyze the number of errors of coding presented in the site that may elude system monitoring.

2.6 Conclusion and Discussion

In this thesis, we aim to make contributions to monitoring in distribution systems. Taking into consideration this chapter, monitoring is a key aspect for distinct architectures, not only to ensure a proper system observability but also to ensure system resilience. We can clearly see that papers or industrial methodologies focused in internal — and more intrusive —, solutions tend to handle server errors, whereas outside methods tend to cover only network or client-side quality-of-service.

We are looking for bridging this gap. First, our premise is simplicity. We do not want complex solutions that generate a superabundance of dashboards or require heavy instrumentation or agents in the system.

Secondly, we aim to have methods not tied to any specific architecture, as we try to evaluate system performance from the client side. We have to resort to information that is leaked by the system to the client, such as interaction times. Additionally, by taking measurements from the perspective of the client, we can have a better insight on the quality of the response. This approach seems preferable to taking a large number of measurements from multiple vendors that compose distributed systems.

Third, although results achieved in terms of failure detection and data visualization are quite impressive, enabling system administrators to grasp crucial aspects of the system, they lack automatism. Hence, we aim to determine if it is possible to automatically determine the overall occupation of a distributed system. The fact that we have client metrics allows us to understand the influence of the client-to-server network, and also to validate the interaction of users with third party resources, thus getting useful information to improve the quality-of-experience.

Concerning web site studies, we intend to improve the previously mentioned literature in at least two aspects: first, the study that we intend to do takes into consideration a very wide range of sites and not only a handful of worldwide top sites. This offers a more realistic view of global web page error status, by taking into account, not only the big companies, but also other operations with fewer resources (that might trigger different error patterns). Secondly, we also want to align possible solutions with the real errors observed by the results of the very wide range of sites.

Chapter 3

HTTP infrastructure evaluation

In the operation of a web site, monitoring plays a major role in mitigating the negative consequences for the user quality-of-experience resulting from programming errors, network malfunctions, overloaded resources, among many other problems.

To control failures in web resources, system administrators must keep a watchful eye on a large range of system parameters, like CPU or memory occupation, network interface utilization, among an endless number of other metrics, some of them specifically related to web page performance, such as page load time. Unfortunately, even with all these metrics — that add complexity to the system and make monitoring more intrusive —, the client may experience some problems, due to web page external content, JavaScript errors, or even client specific conditions, such as network glitches or browser version.

Several studies (Sweenie, 2000, 2003) show that sites belonging to the top-50 of the most viewed web sites worldwide have errors, thus suggesting that even expensive monitoring mechanisms cannot provide a completely accurate picture of web page reliability. Another study (Oppenheimer et al., 2003) shows that an earlier detection of failures would reduce the majority of customer complaints. Customer feedback regarding web pages is a key aspect for web page trustworthiness, but metrics such as network latency or user-visible failures are difficult to get by system administrators, not to mention that some server issues might not produce the same effects in all clients.

In the previous Chapter, we analyzed distinct approaches to monitor distributed systems from white to black-box frameworks. We also presented studies related to the availability of web sites. In this Chapter, we go a step further in these studies, as we

analyze how the client identifies distinct failures. These failures might be visible to the user, such as HTML errors that do not allow a correct resource visualization (Mendes et al., 2018), or associated with a resource fetch that might degrade web page performance — e.g. load time —(Li et al., 2010b; Netravali et al., 2016).

For this we start in Section 3.1 by evaluating 3000 web pages, from the most viewed web site rankings. Additionally, and since REpresentational State Transfer (REST) interfaces are a standard for communication between systems, in Section 3.2, we evaluate some of the most used REST interfaces from global providers.

3.1 Web site Evaluation

Several studies (CMUPDL-05-109, 2005; Technology, 2005) show the significant impact for companies of users experiencing blank pages, missing items in the web page, or being unable to interact with the web page in operations such as online transactions. This may affect the company's reputation and profits.

We argue that there are still no effective means to easily detect web page problems. A successful monitoring approach needs to gather server and client-side data, e.g., through probes in the server and analytic tools in the web page, respectively. Server-side data is not enough, because other overheads and problems beyond metrics like server CPU load, memory usage, database latency, etc. can affect client observed delays.

To demonstrate that the web currently suffers from a lack of proper monitoring, we ran an experiment using 3,000 sites of the top 1,000,000 web sites from the Alexa ranking (Alexa, 2017). The samples cover 1,000 different web sites from the range 1-1,000 in the ranking, another 1,000 from the range 10,000-20,000 and a final 1,000 pages from the range 100,000-200,000. In our experiments, we used the Chrome browser (Chrome, 2017), to ensure realistic access to web pages, and simulate real user interactions. Additionally, to compare results, the experiment ran in two distinct locations: Portugal and Hungary. Several metrics were collected, such as error, network and JavaScript errors, among others.

We think that our results are noteworthy: even 16% of the top 1,000 sites have errors in their web page resources, being this value higher for less popular sites. Based on these results, we argue that there are no simple or effective tools to prevent these errors.

Taking these results into consideration, we then discuss possible client-side monitoring solutions, to complement current web page monitoring. We distinguish the different solutions, based on their level of adaptation on the client: stand-alone applications that can be periodically invoked, browser extensions to enrich client-side information and JavaScript snippets. Approaches that can collect more metrics are also much more intrusive. On the other hand, our error report leads us to the conclusion that even non-intrusive light-weight approaches for web site monitoring can cover these 16% of web errors. These light-weight approaches have the additional benefit of not compromising client security or increasing the complexity of monitoring.

3.1.1 Problem Description

In this subsection, we describe the problem, define the metrics and review the challenges associated with client-side monitoring. These concepts serve as the basis for the experiments of Sections 3.1.2 and 3.1.3.

Before rendering and displaying an HTML (HyperText Markup Language) page, browsers must first fetch the page from a server, using an HTTP (Hypertext Transfer Protocol) address that uniquely identifies the page, know as URL (Uniform Resource Locator). The browser then goes through the source code of the page, to build the DOM (Document Object Model) and the resulting render tree before displaying the page. In the process, it might need to download multiple other resources referenced in the main page through other unique URLs, like style sheets, images, scripts, etc. To fetch these resources, the browser opens several TCP (Transmission Control Protocol) (Postel, 1981) connections to their respective server. Some of these resources might be internal and reside in the same server (or at least in the same domain), other resources might be external.

Since the final result that one sees in a browser is usually the combination of many different resources, each one of these might impact the user experience, and might be affected by varying issues. These problems include network connectivity problems, server bottlenecks, HTTP errors, or even processing errors, if the resource is a script to be executed by the browser. Hence, system administrators must take performance, as perceived by the clients, into consideration, when they create and maintain the web site.

However, the crucial point here is that these metrics are external to the server providing the web page and, therefore, out of (simple) reach by the administrators, who need specific tools to get them. A typical approach is to send browsing data to an analytic tool that creates reports or triggers alarms in the presence of a violated threshold (Clicky-Analytics, 2017; Clifton, 2008; GoogleAnalytics, 2018; Internet, 2016). Normally, these tools have some ability to handle problems such as nonexistent web pages in the domain. But, since they are oriented to advertising and search engine optimization (SEO), they typically neglect correct web page display.

The purpose of this Section is to demonstrate that current web sites are not being properly monitored and to propose appropriate fixes. For this, we inspected 3,000 different sites, including the most popular ones, and looked for problems in the following very specific metrics:

- First, in the main page and its associated resources, we analyzed network errors. We decomposed network errors into DNS (Domain Name System) errors (if the main page or some resource returned an error in the name lookup phase); TCP, if the site exists, but the connection crashes or the server is unreachable; and *other* errors, because the browser we used in the experiment does not classify all the network errors as coming from DNS or TCP (ChromeErrors, 2016).

- For HTTP errors related to resources in the web page, we care for response codes in the range 400-451 (4xx) and 500-511 (5xx). These resources are referenced in the page and are necessary to render it. In our experiments, we do not count how many errors exist in a single page, but only whether any exists (one or more). For example, if, for 10 pages, we only have 1 page with HTTP errors, the statistics will return 1 web page with errors, regardless of how many resources produced an error in that specific page.

- It is also important to know how would the navigation from the user be handled. For this, we count broken links, where the server responds with a 4xx or 5xx HTTP code. Again, we only count the number of web sites that have errors in these ranges, regardless of how many such errors a single site has.

- We also care for other sources of errors related to resources in the page: fonts, style sheets, images and JavaScript. These errors might originate in the network layer, while processing the script (if applied), or in the cancelation of a resource

Table 3.1. Software used and distribution.

Component	Observations	Version
Selenium	selenium-server-standalone jar	2.45.0
Chrome	browser	48.0.2564.103
Chrome	driver	2.21.371461
Xvfb	xorg-server	1.13.3

download, e.g., because a change in the page made it unnecessary, or because a network or other error on an earlier request showed that such resource is unreachable (ChromeNetwork, 2016; StatusCanceled, 2016).

- Finally, we also care for the time needed to display a web page (after it was fully downloaded from the server) and the time that the browser needs to run asynchronous scripts, after loading the page.

In the next section, we review the monitoring approach that allowed us to collect these metrics from the 3,000 sites.

3.1.2 Experimental Settings

For the sake of doing an online analysis of the web sites, we used Selenium (2015). The Selenium framework emulates clients using browsers to access web pages. Developers might use Selenium for testing web software, but Selenium can also serve to automate repetitive tasks (e.g, administrative), as we do in this section. This framework is coupled to a browser through a *WebDriver*, which makes direct calls to browsers using their native interface for automation (ChromeDriver, 2015; Selenium, 2015). The driver can mask differences among browsers, thus providing a uniform interface for selenium. In our experiments, we used the Chrome web browser.

We used the Xvfb (Xvfb, 2015) virtual display emulator for the client machine. This display performs all graphical functions in memory, without actually needing a real screen, thus allowing Selenium to run without a terminal. We wrote a program in Java that runs in the background attached to this display emulator. This program uses Selenium and Chrome, to access the list of web sites that we previously defined. We used a Linux machine running in our department facilities in Portugal and another instance in Hungary. Table 3.1 lists the software we used and the respective versions.

Clients running from different locations have different network connectivity, thus having a distinct perspective for the same web page. This may result in disparate behaviors, like resources inaccessible from only one of the locations. Additionally, since programs run autonomously, with a time lapse of several hours, they may experience different page errors.

Algorithm 1 shows the pseudo-code of the program we wrote to monitor web pages. This program uses as input a file that we must retrieve from Alexa (Alexa, 2017), with the top one million ranking sites. Alexa keeps popularity rankings of web sites. Afterwards, we sequentially analyze 3 ranges from this file: from pages 1 to 1,000; then from rank 10,000 to 20,000 with steps of 10 (e.g. rank 10,000; 10,010; 10,020;...) and finally, from 100,000 to 200,000 with steps of 100 (e.g. 100,000; 100,100; 100,200;...). Since ranges do not include the upper limit (20,000 and 200,000), this adds up to a total of 3,000 sites analyzed.

Algorithm 1 WebPage report

Input: CSV file with the Alexa top one million sites
Output: web pages metrics and errors

```
1: Initialization :
2: Process Alexa file
3: queryrange = list(range(1,1001))
4: queryrange.extend(range(10000, 20000, 10))
5: queryrange.extend(range(100000, 200000, 100))
6: Open chrome browser
7: LOOP Process
8: for all rank in queryrange do
9:     Invoke web page;
10:    Gather web page statistics;
11:    Invoke web page links;
12:    Save metrics to File;
13: end for
```

We also used two JavaScript libraries, defined by the World Wide Web (W3) Consortium, called Navigation Timing API (NavigationTiming, 2015) and Resource Timing API (ResourceTiming, 2016). The former library gives us metrics related to the main HTML, whereas the latter relates to their resources. Used together, they provide a better understanding of network, server and processing time from the client-side point-of-view.

When the chrome browser invokes a web page, we gather several metrics from different sources: from the Navigation Timing API (NavigationTiming, 2015), we collect the time that the page takes to load, once it is received from the server (`loadEventEnd` — `responseEnd`), and the time taken by the browser to execute JavaScript for the `window.load` event (`loadEventEnd` — `loadEventStart`). From the Chrome driver and Selenium, we collect network, driver and browser logs (in JSON format). With these logs, we are able to detect network failures (at main HTML or resources), such as the ones associated with DNS, TCP or HTTP; error codes such as the ones in the ranges 4xx or 5xx[1]. We do also capture errors related to JavaScript and other components normally listed on the browser's console, such as Images, Fonts, among others. We could also collect information related to resources available in the Resource Timing API (ResourceTiming, 2016), but the web logs from the Chrome driver make these metrics redundant.

Additionally, and one of the most relevant aspects of our work is that we parse the main HTML page, to get all links accessible to the users through web page interaction. We follow and invoke these links, to check if any HTTP error occurs (with error codes 4xx or 5xx, related to client or server errors, respectively). This information is important, because the availability of the links is tightly connected to the utility of the web page. As we shall see the number of broken links is high, even in top web sites. To make this process run faster, and since we only need the error codes, the URLs in the web page links were invoked using the java HttpURLConnection, instead of using a full-blown browser.

3.1.3 Results

In this subsection, we present the results of our experiment. The experiment took several days to finish, mostly due to the time consumed in the invocation of all links associated with each web page. Tables 3.2, 3.3, 3.4 and 3.5 present the most significant results we got. For the sake of keeping the discussion brief, we only include metrics that have interesting results.

In Table 3.2, for the client in Portugal, we analyze the number of web pages with network or HTTP errors. This table contains three classes of problems: problems with

[1]The list of all network errors displayed by the browser can be retrieved from the Chromium open source project (ChromeErrors, 2016)

page resources (HTTP 4xx and 5xx), e.g., some image; connection errors (DNS, TCP and other) and broken links, i.e., links that point to resources outside the page that exhibits some problem. Connection errors are all related to the main HTML page or one of its resources. As we mentioned before, the numbers in the table refer to the total number of sites where we could observe the problem. This means that, for example, in the first line, first column of Table 3.2, the number of HTTP 4xx errors in the top 1,000 sites is 161. I.e., 161 sites have one or more resources that are not accessible and return a 4xx error code. Taking into consideration the results from Table 3.4 for the client in Hungary, we realize that the values are very similar, but not equal. This is mainly due to the fact that the two instances of the program run independently and with a time interval, for the same list of websites.

The number of errors is quite high in general, especially in lower raking sites. Differences between the first and the other two rows of the table are high, for most metrics. This is true for internal problems and for external links, including network error conditions, which are also much less frequent in the top ranking sites. Most problems come from the external links that tend to break quite often, either with a 4xx or a 5xx error code (right side of the table). However, internal problems (left-side of the table) are arguably more important, as they might result in visible problems in the page layout. As much as 16% of top-tier sites may suffer from some form of internal problem. This number is even higher for the lower rankings. The same is true for connectivity errors (center of the table). DNS, TCP and other forms of errors are less frequent in major sites. The HTTP 5xx error codes are the only ones where the frequency of problems seems stable across all rankings. We can speculate this might be due to an inverse relation between complexity and ranking positions (i.e., more complex pages correspond to lower ranking numbers), but a clear demonstration of such hypothesis requires further study.

Overall, these results suggest that top-tier sites either have better network connections, or more server resources, or both. We might say the same about the contents themselves, most likely due to significant advantages in the lifecycle of the web pages (one or more among design, development, testing, deployment, and maintenance).

Table 3.2. Number of sites with Network and HTTP errors - Portugal

	HTTP 4xx errors	HTTP 5xx errors	DNS errors	TCP errors	Other Network errors	Broken Links 4xx	Broken Links 5xx
$range_{1000}$	161	62	68	27	96	115	65
$range_{10000}$	251	47	122	38	111	182	42
$range_{100000}$	291	51	113	37	114	193	43

Table 3.3. Number of sites with resource errors and event averages - Portugal

	Resource Font errors	Resource style sheet error	Resource image error	Resource JavaScript error	Resource JavaScript External	Resource JavaScript Internal	Resource JavaScript Both
$range_{1000}$	11	15	131	153	136	13	4
$range_{10000}$	16	16	134	189	136	39	14
$range_{100000}$	27	27	143	174	103	62	9

Table 3.4. Number of sites with Network and HTTP errors - Hungary

	HTTP 4xx errors	HTTP 5xx errors	DNS errors	TCP errors	Other Network errors	Broken Links 4xx	Broken Links 5xx
$range_{1000}$	154	50	58	4	78	129	24
$range_{10000}$	242	50	143	8	96	223	54
$range_{100000}$	267	48	132	21	88	291	43

Table 3.5. Number of sites with resource errors and event averages - Hungary

	Resource Font errors	Resource style sheet error	Resource image error	Resource JavaScript error	Resource JavaScript External	Resource JavaScript Internal	Resource JavaScript Both
$range_{1000}$	10	11	100	163	148	13	2
$range_{10000}$	17	16	119	172	119	38	15
$range_{100000}$	26	25	133	172	104	60	8

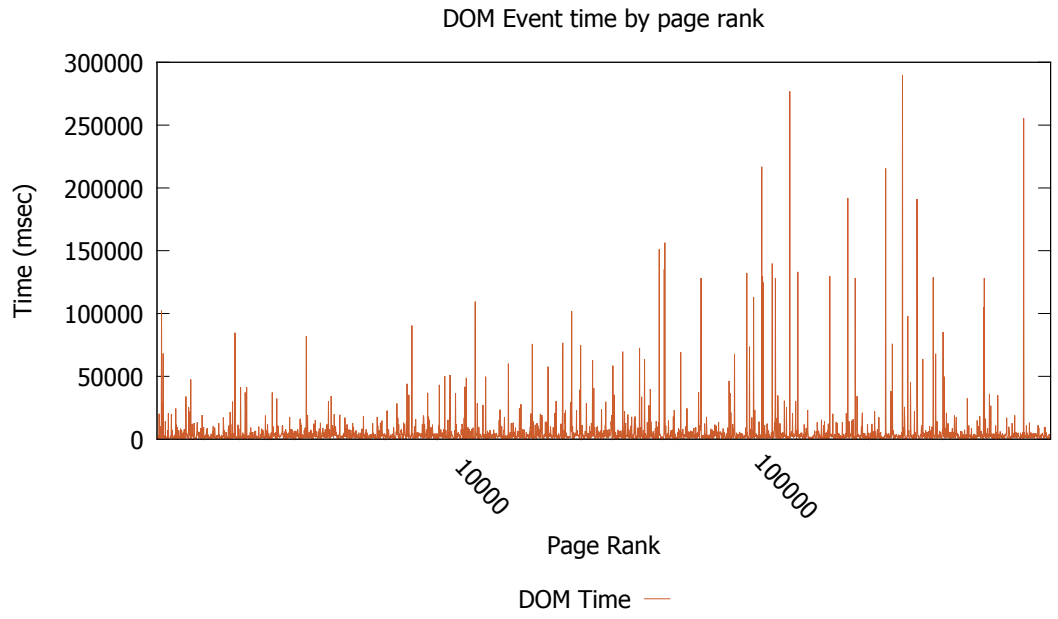


FIGURE 3.1. DOM event by page rank

In Table 3.3, we show the number of sites that returned at least one resource error, for different types of resources. Resource problems include error getting a font, an image or processing JavaScript (left side). Regarding JavaScript errors, we split data into external or internal to the web page domain, or both, if errors exist in internal and external resources. We can see that fonts and style sheet resources cause much fewer errors than images and JavaScript. Another interesting result is that top-tier sites have more errors in external JavaScript resources. This is an indication that top pages rely more on standard libraries, normally hosted in another domain.

Concerning the results we got with the Navigation Timing API, in Figure 3.1, we can see the difference between the `domLoading` event and the `domComplete` event times, for every web page. We only show the graphic for the Portuguese experiments; results were very similar in the Hungarian case. In this time interval, the browser starts parsing the HTML document and loads all resources of the page (images, css, etc). At `domComplete`, the document readiness is set to "complete", which means that the next event, called `onLoad` can be fired. To the user, this means that the loading spinner on the browser has stopped. Figure 3.2 shows the time interval between the `loadEventStart` and the `loadEventEnd`. This interval measures the time taken to do additional logic, such as JavaScript. In both figures, we sort the pages according to their rank in the x-axis.

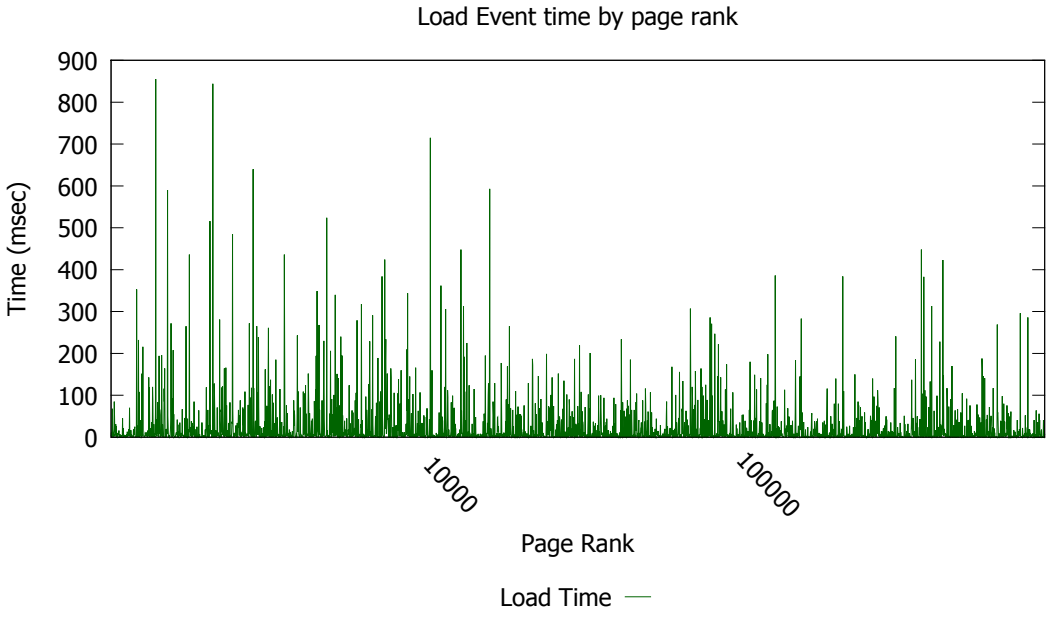


FIGURE 3.2. Load event by page rank

TABLE 3.6. THREE PAGE RANK RANGES

Event	$range_{1000}$	$range_{10000}$	$range_{100000}$
Average DOM Event	4517	6067	6963
Average Load Event	33	24	19

Hence, the first 1,000 ranks correspond to the top 1,000 pages, the following 1,000 ranks correspond to pages in the interval 10,000 to 20,000 and so on, as we described before. The maximum time for Figure 3.1 is very high because the timeout was not enabled at the browser or selenium. Therefore the web page could take some minutes to be fully loaded.

From the first row of Table 3.6, and from Figure 3.1, it is quite clear that the browser tends to take a *considerably* longer time to parse and get the resources of less popular web pages. We might say that this is not surprising: first, big companies with more resources may have country-specific versions of the web pages, thus ensuring better performance. Secondly, major sites use Content Delivery Networks (CDNs), to put the files closer to the users, thus granting faster download times. In fact, we observed that some of the peaks observed in Figure 3.1 correspond to sites hosted in China, which are geographically distant from our clients.

Regarding the second row of Table 3.6 and Figure 3.2, we observe a completely distinct pattern. As mentioned earlier, these data concerns the time taken to handle additional JavaScript, after the document is fully parsed and the resources are loaded. We can see that some higher ranking pages are JavaScript intensive, although the table shows a very marginal decrease of this time with page rank.

3.1.4 Client-Side Monitoring

Taking into consideration the results of previous subsection, we discuss some solutions that might serve to improve web page reliability. We suggest three different options involving different levels of transparency to the client: a stand-alone approach, a browser extension and a JavaScript snippet.

Stand-Alone Application

First, using a stand-alone application (similar to the one we showed in subsection 3.1.2), provides us the most options. By having total control of the browser and resorting to a testing framework such as Selenium, developers and system administrators could invoke periodically the tool with a simple scheduler such as Crontab (2015). Afterwards, the metrics could be centrally collected to create useful statistics and even generate alarms. The major disadvantage of this approach is that a customized stand-alone application is not practical to install on the clients. Monitoring a site in this way would therefore be limited to a handful probes controlled by the site owners.

Browser Extension

A second approach would be to install a browser extension, to get the most important metrics from the web page interaction and, again, send them to a central monitoring site (we refer to some work using browser plugins in Chapter 2). Extensions could bypass some of the security constraints associated with JavaScript. For example, with extensions it would be possible to have access to the browser APIs and therefore to network logs. This would enable network error collection (DNS, TCP and others). Additionally, the extension could invoke links associated with a web page. Unfortunately, extensions have two setbacks: first, it does not look feasible to convince hundreds or

thousands of users to install some browser extension, which could actually raise issues concerning security and privacy; secondly, different extensions should be developed for each different browser, thus entailing a great effort and cost.

JavaScript Code

Site owners might use JavaScript and AJAX in the web pages they serve, to collect error information. By precluding the need for special software, this would rule out the shortcomings of the previous approaches. Furthermore, this would allow for a very simple integration with analytic tools, like GoogleAnalytics (2018). Naturally, this can only work for resources inside the main page, once the browser loads the JavaScript.We now present some of the solutions possible to be achieved to collect networking errors, internal 4xx errors, internal 5xx errors, external broken links and resource errors, using Javascript code.

Resource Errors Regarding JavaScript exceptions and console logs, it is possible to use the `window.addEventListener` for *error* events with the `useCapture` argument set to `true` or use the `window.onerror` event. This will retrieve the element or script that originated the error, and not the specific error message. This approach can also generate an alert to system administrators, in case of problems. As an example, we can see Listing 3.1 for JavaScript errors.

Networking Errors With JavaScript, one cannot know which resources returned DNS or TCP errors, although this could be inferred using the Resource Timing API (ResourceTiming, 2016). Internet Explorer and Firefox add entries in the PerformanceResourceTiming array for resources with network issues[2]. A `domainLookup` or connection time of zero associated to a resource indicates a network error.

Internal 4xx Errors It is possible to customize an HTTP 4xx page for this range of errors. As the user is redirected to this page, administrators will receive an alert.

[2]The Chrome browser does not provide resource information in the Resource Timing array, when the resource does not return a response.

Table 3.7. Comparison of methodologies

	Stand-alone Application	Browser Extension	JavaScript Code
Network Problems	Y	Y	Y Indirectly for resources
Broken Links	Y	Y	Y Proxy
JavaScript Errors	Y	Y	Y
Real-world application	hard to deploy	security constraints	easier to scale and deploy

Internal 5xx Errors Detecting resources with an HTTP 5xx error could be done by analyzing the difference between the `responseEnd` and the `responseStart` (time taken to retrieve the resource). Having a connection time different from zero and a response time equal to zero is an indicator that the browser could not retrieve the resource from the server. Another way to detect a 5xx error on a resource, in the Firefox browser, is to check if the `duration` (available in the Resource Timing API) is zero for this resource. As an example, we can see Listings 3.2.

External Broken Links Although JavaScript might invoke internal links, it may be prevented from accessing any link outside of the server domain, unless cross-domain communication is active, or if a proxy is used with AJAX (HTML5, 2016). This proxy will then invoke the URL and return the request in JSON, thus not breaking cross-domain security. One web service that can be used to this effect is the Yahoo Query Language Yahoo (2016), which allows the JavaScript code to invoke the external URL and get JSON content in return. As an example, we can see Listings 3.3.

```
// test javascript errors
window.addEventListener('error', function(e) {
    //print error
    console.log('error on ' + e.target.src);
    //send error to server
    dataLayer.push({
       'event'  : 'javascriptError',
       'target' : e.target.src
    });
}, true);
```

Listing 3.1. JavaScript code for error logging

```
window.addEventListener("load", checkResult);
function checkResult() {
     // test network resources problems
```

```
    if (navigator.userAgent.indexOf("Firefox")!=-1 ){
        // get resources timing metrics
        var e = window.performance.getEntries();
        for (var i in e) {
            // for the document we need to use window.performance.timing
            if (e[i].name != "document") {
                // duration equal to zero means that the resource could not be fetched
                if (e[i].duration==0) {
                    if (window.console) console.log("Network error in resource: " + e[i].name + " type:" +
    e[i].initiatorType);
                    //send event to analytics
                    dataLayer.push({
                        'event' : 'networkError',
                        'type' : e[i].initiatorType,
                        'resource' : e[i].name
                    });
                }
            }
        }
    }
}
```

Listing 3.2. JavaScript to collect resource fetching errors

```
window.addEventListener("load", checkResult);
function checkResult() {

    // test broken links. Validate if the links were tested in the last 60 minutes...
    if (typeof {{Session alive}} === 'undefined') {
        try {
            // Get page links and make proxy querys...
            var allLinks = document.links;
            for (var i=0; i<allLinks.length; i++) {
                // Set up variables for the call
                var httpcode;
                var link = encodeURIComponent(allLinks[i]);
                var endpoint = 'http://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20'+
                'html%20where%20url%3D%22' + link ;

                // Make the API call
                jQuery.ajax({
                  type : 'POST',
                  dataType : 'jsonp',
                  url : endpoint + '%22&format=json'+
                  '&diagnostics=true&callback=?',
                  async : true,
                  success : function(data) {
                    httpcode = data['query']['diagnostics']
                    ['url']['http-status-code'];
                  },
                  error : function(xhr, textStatus, errorThrown) {
                    console.log('Error while fetching link :: ' + errorThrown);
                  },
                  complete : function() {
                    if (typeof httpcode !== 'undefined' && (httpcode.startsWith("4") ||
    httpcode.startsWith("5") )) {
                        // send broken link
                        console.log('Broken link: ' + link);
                        dataLayer.push({
                          'event' : 'linkError',
                          'error' : httpcode,
                          'link'  : link
                        });
                    }
                  }
                });
              }
```

```
        } catch(e) {
            console.log('Error collecting broken links... ' + e.message);
            dataLayer.push({
              'event' : 'APIError',
              'APIErrorMSG' : e.message
            });
        }

        // Set cookie to 60 minute expire date
        var d = new Date();
        d.setTime(d.getTime()+3600000);
        var expires = "expires="+d.toGMTString();
        document.cookie = "session=1; "+expires+"; path=/";
    }
}
```

Listing 3.3. JavaScript to collect broken link information

Client Side Monitoring methods — discussion In Table 3.7, we make a summary of the different methods mentioned in this section. In Listings 3.1 to 3.3, we illustrate how to analyze web page errors using JavaScript. The browser invokes Listing 3.1 on error events. This code logs the error and collects the object that generates the error. To ensure a more realistic scenario, we include code necessary to send these errors to be displayed in Google Analytics (GoogleAnalytics, 2018). Listing 3.2 is associated with the load event and Listing 3.3 is associated with broken links. For brevity we omit the inclusion of external jQuery scripts. We check for broken links and resource fetching errors. We also send the collected metrics to Google Analytics. This allows us, in a relatively fast way, to understand the capabilities of this well know tool. Furthermore, regarding the test for broken links, in a production scenario, this test should only be made occasionally for each client. Therefore, we created a session cookie (using Google's tag manager) with an expiration time of one hour. In this way, links are not tested more than once per hour.

3.2 Evaluating REpresentational State Transfer Evaluation Services

REpresentational State Transfer (REST), first defined by Fielding in his PhD thesis in 2000 (Fielding, 2000), has become a major standard for services provided over the Internet. The power of REST comes from the versatility of interfaces that simultaneously support the presentation layer of web pages developed in JavaScript, mobile applications that are native for Android or iOS, and interaction with third-party providers,

most notably for the sake of authentication, but by no means limited to it. The modern computing infrastructure, including the cloud, heavily relies on REST services as these became ubiquitous. The number and names of web site providers using REST and making their REST interfaces available for programmers is huge: Facebook, Twitter, Instagram, Amazon, Microsoft, Google, YouTube and many others. In spite of this success, critical web interactions still pose a major challenge for developers, whenever operations fail to execute properly. Services involving reservation of some resource, payments, or other business-oriented interactions cannot easily tolerate requests with incorrect responses or no responses at all. However, the popularity of the running REST interfaces, the number of available operations and the large number of users they support, provide us new opportunities to evaluate the reliability of real, highly relevant distributed systems. In particular, we would like to know exactly how likely is it for an operation to have success or fail without providing adequate information to the client.

To achieve this goal, we ran an experiment using successive HTTP invocations to REST services of three major providers of video and file storing and sharing. We did this for several days, keeping notice of response times, unanswered requests, HTTP status of the responses and HTTP headers. To avoid disclosing sensitive information regarding these services, we keep them anonymous and refer to them by the letters A, B and C. The results are noteworthy: even major providers cannot avoid a significant number of errors in the access to their REST resources and a wide disparity of response times. While some of these problems come from the network, the provider itself is also to blame in many cases. Although small in relative numbers, errors do not only exist, but exist in large amounts, when we consider the vast utilization of these services. Hence, we can conclude that, to build reliable distributed systems, developers must use mechanisms that take connectivity, availability and performance problems into account. The results of this section aim to motivate developers for the central problem of achieving reliable invocation of services over the network and also to understand how a client can monitor a REST interface.

3.2.1 Experimental Settings and Results

To evaluate the reliability of online REST services, we ran a client that regularly requested operations from three well-known service providers. All three providers reside

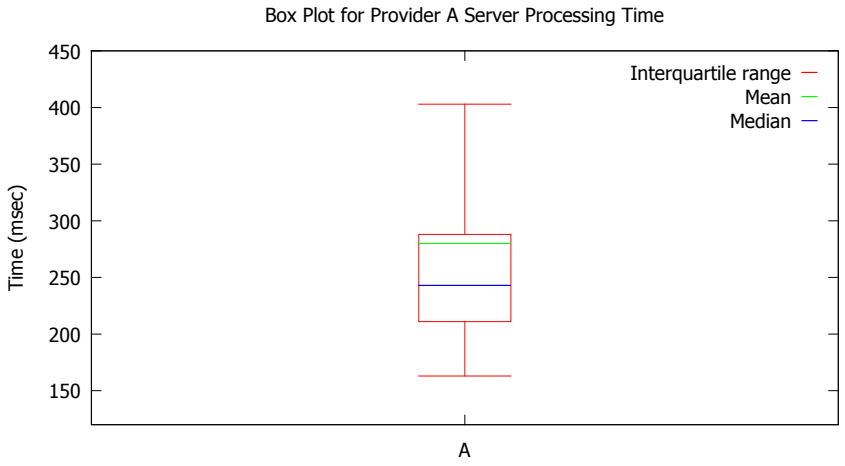


FIGURE 3.3. Provider A server time statistics - boxplot

in the west coast of the United States, whereas the client is in Portugal, around 9,000 kilometres away. We used the crontab scheduler on a Linux machine, to run the cURL command, which invoked the same REST resource during several days, with intervals of 4 seconds. We accessed two file sharing providers (A and B) and a video sharing provider (C). In the file sharing providers, we did a REST request to get the contents of the home directory of a user account; in the video sharing provider, we did a REST request to get the list of the 50 most popular videos. To access data from provider A, we had to use the HTTP POST method; from the other two providers, we used the GET method. Note that according to the HTTP protocol (Fielding and Reschke, 2014), the POST method is non-idempotent. Repeating an invocation might thus have side effects.

We also gathered a specific HTTP extension header from provider A, which stated the time that the server took to answer the request. This extension allowed us to determine server-side times, regardless of network performance. In Figure 3.3, we show the box plot associated with the server response time, while in Figure 3.4, we show the histogram of the same samples, distributed by the time it took to answer the request (x axis) and the frequency of each time interval (y axis). Looking at Figure 3.3 and Figure 3.4, we can observe that the servers of provider A take around 250 *ms* to reply to the majority of requests, once they arrive. However, as we can see in the box plot, the actual times vary a lot. Additionally, if we take a closer look at the histogram, we verify that sometimes the server responses take as much as 6000 *ms*. These response

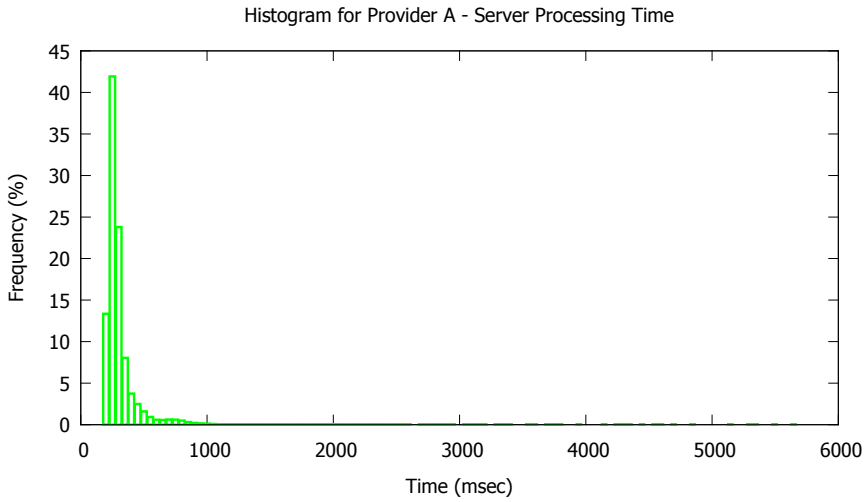


FIGURE 3.4. Provider A server time statistics - histogram

times (plus the network latency) may be treated as a timeout by the client software or users, which might reload or resubmit forms in response during this interval.

Concerning the network time between our client and the servers of providers A, B and C, in Figure 3.5 we show a box plot for the server+network times of our samples. Service invocations will necessarily involve large network latencies, as light takes around 60 *ms* to cover the round-trip (in vacuum). Interestingly, provider C presents a very stable connectivity, having very short interquartile ranges and a mean similar to the median. The results of provider C are truly remarkable, suggesting, not only, that providers A and B take long internal times to fulfill the service, but also that such times have a large variance (i.e., they cannot be attributed to the network). These differences might result from the nature of the services involved. It might be simpler to prepare the response to a request for the most popular 50 videos in store, than it is to provide the list of files in the home directory of a specific user. The precise reasons, however, are difficult to discern, without knowing the internal details of these providers. Furthermore, as far as we could tell, the requests we did to provider C were not served by any cache. The comparison of graphics of Figures 3.3 and Figure 3.4 also suggests that the total internal server time of provider A is larger than what we see in Figure 3.4. Furthermore, since variances for providers A and B are somewhat large, they might pose some problems for client-side software, especially if developers want to wait for (outlier) responses taking more than the 75 percentile to arrive.

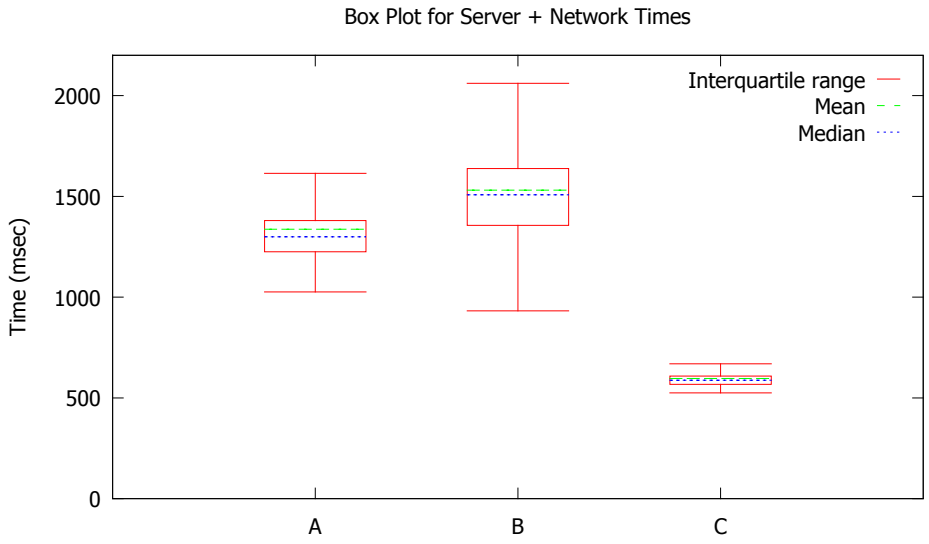


FIGURE 3.5. Server + Network time statistics for the distinct providers

TABLE 3.8. NUMBER OF ERRORS

	Provider A	Provider B	Provider C
Connection setup errors	104(0.057%)	43(0.018%)	6(0.005%)
Connection crashes	5(0.003%)	28(0.012%)	7(0.005%)
Application errors	38(0.02%)	3(0.0012%)	0(0%)

Our next step was to count the errors in response to our requests (refer to Table 3.8). We divide errors as seen by clients in three different classes: network errors related to the connection setup phase (first line), network errors occurring *after* the connection is setup (second line), and application errors (third line). Network errors include connection timeouts, connection crashes, read timeouts, or unreachable hosts. Application errors occur when the client gets an HTTP error in the ranges 4xx or 5xx (we only got 5xx, since we kept using a limited list of addresses in our requests). We can see that the number of failed requests is quite relevant, especially when one accounts for the intensive utilization of these services. Most errors are due to the network interaction, whereas only a few occur in the application, caused by some temporary server overload, for example. While service providers can invest in resources and software to improve their network and the response of the application, connection setup errors and connection crashes might be out of their control, as they mostly depend on third-party providers. On the other hand, for developers, the most difficult problems to overcome are connection crashes and provider errors, because applications can hardly

be sure about the outcome of the REST invocation, when these happen. This situation might be even worse for mobile clients, as intermittent connections will certainly cause more errors, thus raising the importance of tolerating ambiguous, absent or erroneous responses.

3.3 Conclusion

Ensuring proper quality-of-service to users of a web site raises great and largely unsolved challenges to system developers and administrators. The evidences we collected in this chapter support exactly this point-of-view. This chapter focused on monitoring of HTTP infrastructures, such as web pages and REST interfaces. We wanted to monitor and extract useful information from major web sites and REST providers.

From Section 3.1 we conclude that even large companies with vast resources can fail to provide impeccable, failure-free, web sites. As much as 16% of top-tier sites have some sort of relevant error, such as a missing resource. To mitigate this problem, we argue that web site providers must include client-side observations into their monitoring tools. Unfortunately, collecting metrics in clients raises complex challenges, ranging from web page performance degradation to security constraints. Hence, we discuss three different options to take the client-side point-of-view, from an impractical stand-alone client to a more limited, but feasible, JavaScript approach. As we saw in Section 3.1.4, even this latter option can cover most problems referenced in Tables 3.2 to 3.5, thus paving the way to enriching currently available monitoring mechanisms.

Concerning Section 3.2, we are by no means the first ones to evaluate or benchmark online services. However, taking into consideration Chapter 2, we are not aware of other work comparing distinct providers in terms of error counting, request response times, and their distribution. The importance of knowing the normal behaviour and the frequency of errors in online services comes from the importance of ensuring that non-idempotent REST requests execute at most once. However, when the user receives an error and risks to resubmit the request, s(he) will be left without knowing whether the request was executed zero, one or more times. This is an inevitable source of mistrust about the service provider and ultimately contributes to the degradation of the provider's reputation.

The evidence we collected so far strongly supports the idea that developers need to employ fault-tolerance techniques, if they use non-idempotent services. Depending on the provider, the fraction of invocations that fail with ambiguous results might be as large as 0.02 percent. Despite being a well-known problem, to achieve an implementation of at-most-once or (in certain conditions) exactly-once execution semantics in HTTP, developers must recur to customized software that handles operation identifiers. An approach to ensure idempotent operations is to use the HTTP Header `If-Unmodified-Since` with the date of the last (supposed) update. The server would either respond with success, if the resource was not modified after the date, or respond with an `HTTP 412 PreCondition Failed` otherwise. However, this mechanism is not ensured by most providers; furthermore, this approach would force the server to associate a date to each resource. As a final remark, we can say that 1) we showed that problems in non-idempotent REST invocations occur frequently, 2) there is no ready-to-use reliable invocation solution for web developers, and 3) the monitoring tools can be enriched with client-side data.

For future work we would like to tackle a number of challenges. First, concerning Section 3.1, a particularly interesting concern for future work will be the integration of such monitoring JavaScript snippets with analytic tools from large providers, such as Google Analytics, in a real production scenario. Additionally, a wider experiment that covers the entire one million top ranking sites might produce some interesting results. Secondly, for Section 3.2, we identify the need for collecting additional information from other top web providers, to ensure a more complete overview.

Chapter 4

Client-Side Bottleneck Identification of HTTP Infrastructures

In the operation of an Hypertext Transfer Protocol (HTTP, 1999) server, bottlenecks may emerge at different points of the system often with negative consequences for the quality of interaction with users. To identify this kind of problem, system administrators must keep a watchful eye on a large range of system parameters, like CPU, disk and memory occupation, network interface utilization, among other metrics, some of them specifically related to HTTP, such as response times or sizes of waiting queues. Despite being powerful, these mechanisms cannot provide a completely accurate picture of the HTTP protocol performance. Indeed, the network latency and transfer times can only be seen from the client, not to mention that some server metrics might not translate easily to the quality of the interaction with users. Moreover, increasing the number of server-side metrics involved in monitoring adds complexity to the system and makes monitoring more intrusive.

We argue that a simpler mechanism, based on client-side monitoring, can fulfil the task of detecting and identifying an HTTP server bottleneck. The arguments in favour of this idea are quite powerful, as client-side monitoring provides the most relevant performance numbers, while, at the same time, requiring no modifications to the server,

which, additionally, can run on any technology. This approach can provide a very effective option to complement available monitoring tools. Moreover, HTTP administrators lack a perfect picture of the service as seen by the client, from request submission to downloading the last byte of the response.

To achieve this goal, we used several distinct approaches, that will be further detailed in the next sections. The overall goal was to understand the limits of a full black-box monitoring in the context of HTTP Servers. We used several methods that allowed us to retrieve server information from the client. The main idea is that some bottlenecks expose themselves with a different signature in the request and response time series.

With the experiments from this chapter, we managed to discover patterns and identify bottlenecks in a lightweight fashion. We believe that these simple mechanisms can improve monitoring tools, by providing HTTP administrators with qualitative results that add to the purely quantitative metrics they already own.

The remainder of this chapter is organized as follows. In Section 4.1, we analyze distinct patterns that leak from a controlled laboratory experience. In this experiment, we intended to observe the server infrastructure from the outside and gather the smallest possible number of metrics from the inside. We undertook several experiments in a controlled server, to identify the patterns that correspond to bottlenecks. These experiments clearly show that one can actually diagnose different bottlenecks, by analysing response times as seen by browsers. Secondly, in Section 4.2, we evaluate how we could infer the location of HTTP bottlenecks in real-world web pages, using time series from the raw data. To overcome this problem, we used the times collected and foreseen by users, aiming to interpret and distinguish patterns from the request and response times. Finally, Section 4.3 concludes this chapter.

4.1 On Identifying Bottlenecks Distinct Signatures

Improved monitoring mechanisms should be independent from the server technology, should require little to no configuration and should provide information of the real quality of service offered to clients.

To reach these goals, we intend to observe the server infrastructure from the outside and gather the smallest possible number of metrics from the inside. We undertook

several experiments in a controlled server, to identify the patterns that correspond to bottlenecks. These experiments clearly show that one can actually diagnose different bottlenecks, by analyzing response times on browsers. These results pave the way to future monitoring mechanisms, mostly based on quality of service evidence, supported by user data.

To prevent service disruption, the owner of the service must promptly identify and remove bottlenecks, by launching extra resources, such as more bandwidth, CPUs or disk space. Unfortunately, this is not simple in practice, because the providing side lacks a perfect picture of the service as seen by the client, from mouse clicking to downloading the last byte. Additionally, having precise metrics of a running system is expensive and causes a lot of interference with the system itself.

We focus on this exact problem: detecting bottlenecks using the minimum and simplest possible metrics. We aim to perform this bottleneck detection in three-tier web sites using client-side data, because clients have a better perspective of the offered quality of service than providers. To evaluate this possibility, we perform batch submission of requests to the service and collect timing responses on the client. This is not unlike the current paradigm of HTTP performance tools, like HTTPerf (Perf, 2013) or JMeter (JMeter, 2013). With one of these tools, the system administrator controls the invocation of large numbers of requests to observe the response of the system, usually for the sake of tuning performance.

However, since these tools impose a heavy load to the service (and thus are usually run offline), we aim to perform a similar evaluation online, while standard users are running the service. Instead of generating artificial requests, our goal is to use real requests for the same purpose, by collecting and uploading browser data to the system administrators. Our long-term goal is to identify as many problems as possible. In this section, we restrict our effort to the three resources that are very important for performance: CPU, server I/O access and client-server I/O bandwidth.

We aim to demonstrate the feasibility of identifying specific bottlenecks (CPU, I/O or network) using browser metrics plus an internal server metric. We describe this process and our main contribution in Section 4.1.1.

Table 4.1. Measured metrics

Metric	Description
Request Time	Time between connection initialization and first response byte received from server
Response Time	Time between first and last response byte received from server
Latency	Time delay experienced in client-server communication
Query processing time	Time that an HTTP request spends on the database
Total Time	Request + Response Time

Table 4.2. Metrics Required to Detect Bottleneck (Idealized Results)

	Bandwidth	Database	Threads
Request Time	F	T	T
Response Time	T	F	F

4.1.1 The Client-Side Tool

We follow a very simple approach to detect possible bottleneck causes of 3-tier Web systems. We only consider three possible causes: processing, database or bandwidth bottlenecks. Processing bottlenecks are related to CPU limitations, which may be due to HTTP thread pool limitations of the Web Server, or CPU machine exhaustion, e.g., due to bad code design that causes unnecessary processing. Database bottlenecks are part of the Input/Output operations, which clearly depend on query complexity, database configuration and database access patterns. Bandwidth bottlenecks are related to network congestion, significantly affecting client-server communication times.

We assume the point of view of an IaaS cloud, although our method can also partly apply to a PaaS or a hybrid scheme. The client does not own the resources, but has some control on the source code of a site he or she wants to make publicly available. However, the client does not have root access to the target system's nodes. I.e, the client can deploy, but cannot change machine configuration. Hence, we do not want to capture internal system metrics that strongly depend on the specific cloud, but we can collect timestamps during the HTTP processing, doing only slight modifications to the application source code.

We initially considered a number of metrics that should allow us to understand if the system has some bottleneck and where. These metrics contain a mix of network and database times. We do not need the CPU time, because we can infer this time based on the query processing time. The CPU time, in our system, is the sum of the processing time plus the time to access the database. In Table 4.1 we list these metrics and their corresponding description. In an ideal closed setting, we only need the Total Time (request time plus the response time) to distinguish the aforementioned three bottlenecks, if they occur separately — i.e. CPU, Database (I/O) and Network bottleneck. In a bandwidth bottleneck between client and server, the difference between request and response times will tend to grow with the load of the system (specially if the HTTP answer is considerably bigger than the request). In a database bottleneck, the time to get the first byte of the response will tend to grow with the load of the database. A processing bottleneck will have a similar effect. To distinguish these two cases, we can submit multiple equal jobs at once using more jobs than threads available to run them: responses will come in groups. To give an example, assume that the server has a pool of 5 threads. If we submit 20 jobs at once that take around 0.5 seconds of CPU time each, every 0.5 seconds (plus a few other delays) we will get 5 different responses. This is a clear indication of a CPU bottleneck. In reality getting such a well defined pattern is not so easy, because requests will not occur in clearly defined batches. Hence, we need one additional measurement to distinguish between a CPU and a database bottleneck. We use the query processing time. Unlike the previous metrics of Table 4.1, this one is internal to the server. We could also consider the processing time, but this is reciprocal to the former. Fortunately, we may eliminate the latency, because this is constant and should not grow on a server bottleneck. In Table 4.2, we display the relation between the bottlenecks that we are observing and the variables necessary to detect them. We use a "T" (true) and "F" (false) to express the necessity or lack of it to use a variable to observe a given bottleneck. These relations are ideal. A practical setting may be more complex, as we show in Section 4.1.2.

A simple way to take measurements from the client side is to use a performance evaluation tool, like Apache's JMeter. The inconvenient of this approach is that it may only work if the client machine is powerful enough to stress the server and the server is disconnected from real users in a testbed. We intend to follow a different approach. Since our goal is to tackle more generic settings, we want to use data from real clients. For this, we collect the required metrics directly from the user's browser, using the

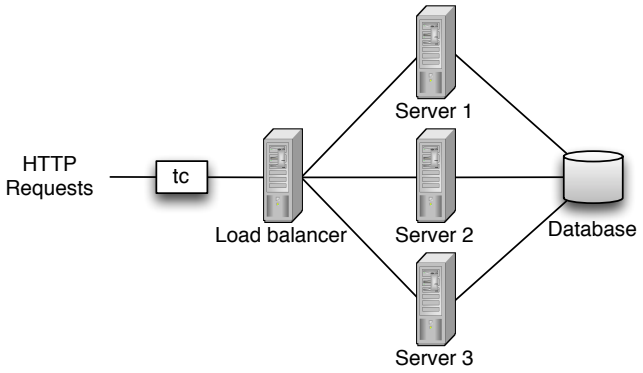


FIGURE 4.1. Experimental setup

JavaScript Navigation Timing API (NavigationTiming, 2015). This JavaScript library can read the request and the response times of a given HTTP interaction. When the requested resource loading finishes, we send performance indicators (via AJAX) to a web service, which stores timing data on a database. The remaining times of the process might be kept directly on the server side[1]. Putting in simple terms, a final implementation should work like this: the owner of the site should add a few JavaScript lines to the web page, to instruct the browser to collect the necessary metrics. After collecting these metrics, the browser should send them to the server. This will enable the owners of the site to analyze performance results as seen by the clients. This analysis, however, might be more complex than with JMeter, because it is full of real life noisy data. In the next section, we observe and tackle this precise problem.

4.1.2 Experimental evaluation

The goal of our experiment was to observe the feasibility of taking client-side measurements to detect performance bottlenecks. For this, in this subsection, we first describe the setup, before detailing the experimental results.

Experimental Setup

To run our experiments, we deployed the "Java Petstore" (PetStore, 2013) application, including the Petstore schematic tables. In the front-end of the server, we have a load

[1]Nevertheless, for the sake of simplicity, in the tests we performed, we first sent the server data to the client, which then uploads all the metrics in a single operation.

TABLE 4.3. Software used and distribution

Component	Observations	Version
Load Balancer	HTTPD with AJP Connector	2.2
Cluster	VMs with GlassFish	3.1.2
Database	MySQL	5.1.69

balancer that directs user requests to a group of GlassFish Application Servers (GlassFish, 2013) running in different Virtual Machines (VMs). These Application Servers take care of the presentation (first) and business (second) tiers. The business tier is stateless, because the Java application keeps all its data in the back-end database (the third tier). This architecture is illustrated in Figure 4.1. The load balancer machine runs an Apache HTTP Server and an AJP Connector for the load balancing. We also installed a Traffic Control (tc) tool (TrafficControl, 2017) on the system entry point to simulate a congested network. To ensure that some page requests took longer and to avoid cache utilization that would change the bottlenecks, we increased the contents of the database tables, relatively to the original petstore, with synthesized data.

Table 4.3 summarizes the most important software components of this experiment. We evaluate the performance of this system using two different methods. The first injects requests through a standard performance evaluation tool (Apache JMeter). This approach enables us to run a finely controlled experience, although limitations of the client machine running the Apache JMeter may cause requests to slide in time. To simulate an intensive utilization of the web site by browsers, in the second method, we submitted requests to our infrastructure using the Firefox web browser from 15 machines. The requests were submitted using a script that started the browser on every machine. The purpose of this experience is to analyze the difference between a tool like JMeter and a simulation that is closer to the real-life Internet utilization with different browsers accessing the same infrastructure.

We injected three different bottlenecks on the server: a database bottleneck, a network bottleneck and a CPU (threads) bottleneck. The first one corresponds to requests that read a large amount of data from the database, to inject the second one we used the traffic control tool and, to delay the responses in the CPU, we reduced the overall number of threads in the cluster to 5 and put them to sleep 1 second per request.

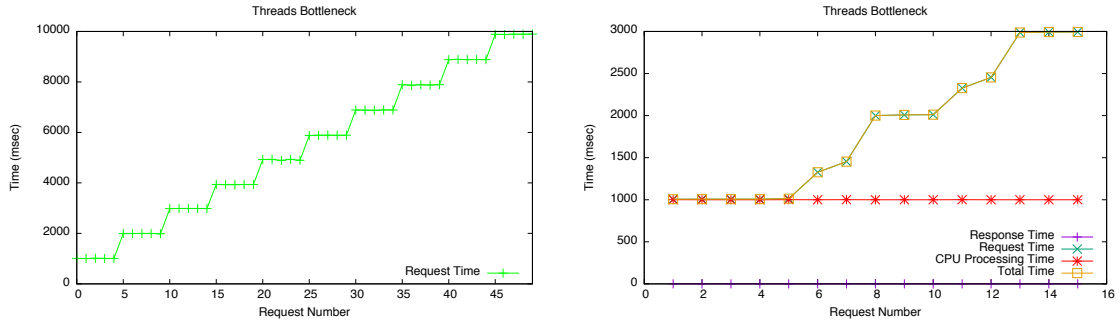


FIGURE 4.2. CPU bottleneck

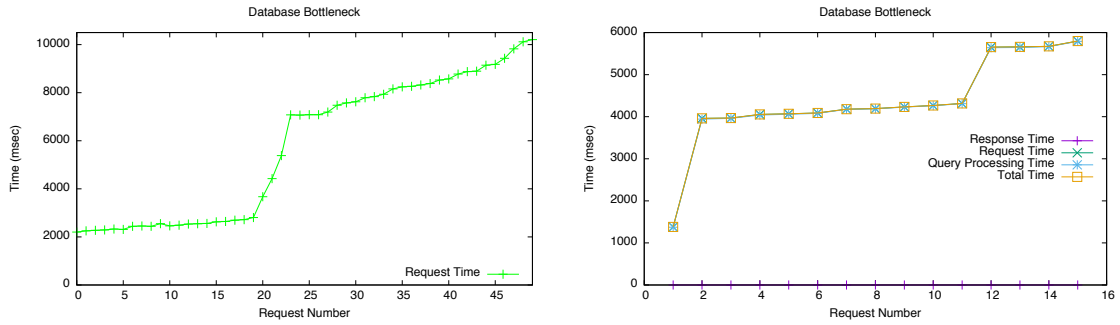


FIGURE 4.3. Database (I/O) bottleneck

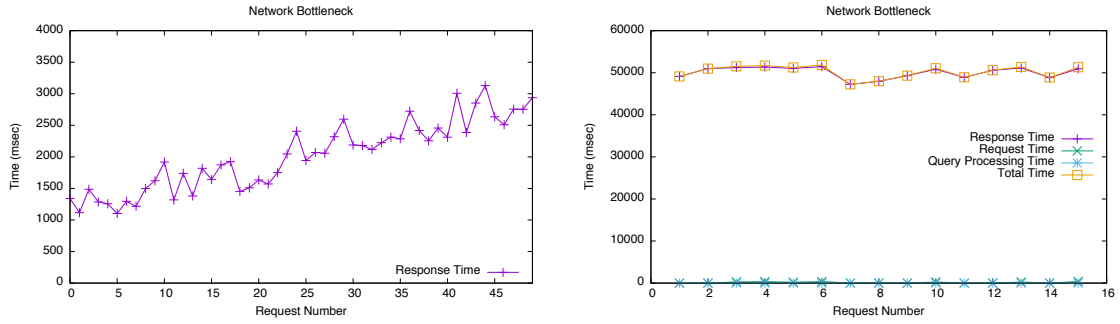


FIGURE 4.4. Network bottleneck

Results

We show the results of our experiments in Figures 4.2, 4.3, and 4.4. On the left-side of the figures we have the response to the JMeter tool. On the right-side, we have the response to the browsers. The x-axis shows the number of the request, from 1 to 50 with JMeter, and from 1 to 15 with the browsers, whereas the y-axis shows the time of each response in milliseconds.

Using JMeter, we managed to minimize the number of external factors interfering with our measurements. For example, network latency is almost always the same, because all the clients are run from the same node, in the same local area network as the server. Furthermore, the tool will spawn multiple threads at once that will send the same request to the server within the shortest possible time frame. This will contribute to a nearly simultaneous arrival of all the requests. In the case of the browsers test, due to the distributed nature of the simulation environment, the results are noisier. We noticed that the delay of starting a new browser sometimes goes to several hundreds milliseconds, making it very hard to perform simultaneous requests, not to mention the differences in the local clocks, as each machine decides when to launch the browser. Unlike this, in the JMeter case, the delay of the requests was within the few milliseconds range. The browser experiment is therefore more akin to a possible real utilization of the site, but, as we shall see, results become harder to interpret.

After plotting the results, we identified several patterns resulting from Request and Response Times. Firstly, for the JMeter tests, we noticed the ladder-type behavior, when the server reaches its HTTP thread pool limit (in our case, the GlassFish cluster was limited to serve 5 simultaneous requests). Since all clients were launched almost simultaneously and given the significant processing time, the steps of the ladder are easily identifiable on the left side of Figure 4.2. We repeated the tests, changing the number of threads in the HTTP thread pool of cluster instances and we were always able to observe this behavior.

When a database bottleneck is present in the system, the time taken to generate the page is dominant in the overall process. This behavior turns Request Time into the largest factor in communication, because the time it takes for the server to send the first byte of the response is mostly consumed on the database query. When the bandwidth is the source of the performance problem, the time taken to transfer the page from the server becomes dominant in the total communication time. We can ignore the time it takes to transfer the data from the client to the server, because in most cases the client request size is very small in comparison to the server response. The Response Time is therefore crucial to identify a bandwidth bottleneck in the infrastructure. Additionally, we expect the pattern of bandwidth bottlenecks to be less regular, because the request and response packets have to go through a congested channel. We can see these differences on the left sides of Figures 4.3 and 4.4. One should notice that these plots display different metrics. In the case of the network bottleneck, we display the Response Time,

Table 4.4. Metrics Required to Detect Bottleneck(Practical Results)

	Bandwidth	Database	Threads
Request Time	F	T	T
Response Time	T	F	F
Database Query Time	F	T	T

whereas in the other two cases we display the Request Time (both metrics are easily available in JMeter and browsers). This observation agrees with Table 4.2, i.e., these times suffice to identify different types of bottlenecks.

Most of the patterns we observed on the previous experiment occur again in the browsers experiment, but it is harder to distinguish between different bottlenecks using only client-side measurements. For example, the CPU bottleneck loses its ladder-like aspect that was characteristic in the more controlled environment (refer to the right side of Figure 4.2). Despite still being there, the effect is much less visible in the browsers experiment, and, we believe that, in general, one might be completely unable to identify this specific kind of bottleneck from the Request Time alone. A clear separation requires an extra variable to distinguish the time the request spends on the CPU from the time it spends on the database. We show the CPU processing time (Figure 4.2 right) and query processing time for this (Figure 4.3 right). With these metrics, the component responsible for the delay is immediately identified. In fact, although the difference in patterns between the Request Times of Figures 4.2 and 4.3 (which are nearly the same as the Total time) might be unclear, the query processing time is negligible in the case of the CPU bottleneck, whereas the CPU processing time is negligible in the other case. Furthermore, the query processing time we measured includes the waiting time to access the database and thus makes the evaluation simpler, when compared to the CPU processing time, which we can see as a constant in Figure 4.2 right. In fact, this difference in behavior could be eliminated, if one includes the waiting time for the CPU in the CPU processing time, and, therefore, we can consider the query processing time and the CPU processing time to be pretty much equivalent for our needs.

In light of these results, we are now able to review Table 4.2, to consider the evaluation of bottlenecks under more realistic settings. As a result, we created Table 4.4, which adds the database query time. This table is a step towards identifying bottlenecks using timing measurements from real web clients.

4.1.3 Conclusion

The monitoring of system resources poses a challenge to system architects and administrators. To achieve this goal, we proposed to detect three types of bottlenecks: processor, bandwidth and I/O. Unlike previous work, we mostly aim to use client-side metrics for detailed observation of the server. The point is to strongly reduce the intrusiveness of monitoring. While other approaches analyze dozens or hundreds of server-side metrics, in this Section, our evaluation suggests that we need only one such metric, to distinguish CPU time from database query times in a "black box" fashion.

While this section demonstrates that we can identify the source of a bottleneck with only a handful of metrics, the great challenge is to do such detection in real time with actual client requests distributed over time, instead of using a single burst of requests over an offline system. The ability to do so might turn out to be an excellent way of improving the existing monitoring tools, by introducing the client perspective of performance and still distinguishing different types of bottlenecks. In the next section we will analyze this kind of problem.

4.2 Black-Box Bottleneck Identification on Multi Component HTTP Infrastructures

In this section, we extend the methodology presented before in Section 4.1, with an additional experiment. To achieve the goal of detecting bottlenecks in real time using clients, we require two metrics taken from the web browser: *i*) the time it takes from requesting an object to receiving the first byte (request time), and *ii*) the time it takes from the first byte of the response, to the last byte of data (response time). We need to collect time series of these metrics for, at least, one or two carefully chosen URLs. These URLs should be selected according to aspects such as popularity. As we describe in Subsection 4.2.1, the main idea is that different signatures in the request and response time series may imply different bottlenecks. For now, we aim to have a binary decision: whether a bottleneck is present or not.

To create such time series, in Subsection 4.2.2, we resort to experiments on real web sites, by automatically requesting one or two URLs with a browser every minute, and collecting the correspondent request and response times. With these experiments, we managed to discover cases of bottlenecks. We believe that this simple mechanism can improve the web browsing experience, by providing web site developers with qualitative results that add to the purely quantitative metrics they already own.

In Subsection 4.2.3, we fetch pages from the same server using two synchronized clients. This enables separation between client-side network problems and server-side problems. However, the main goal of this experiment is to verify whether observations from one of the clients takes us into a set of conclusions that fits the observations of the second one. To avoid any bias, in Subsection 4.2.4, we introduce a simple algorithm that evaluates request and response times from both clients, before outputting the cause of the problem.

Surprisingly, we noted that, occasionally, the two clients disagree about the quality of the interaction with the server. One of them suffers from an isolated problem, which does not occur again, while the other client does not suffer from any problem at all. This suggests that some requests get a very unfair treatment along their way. Even a network and server that seem to be lightly loaded can exhibit this sort of delay at times. Determining exactly where and how frequently does this happen is, we believe, an interesting practical open concern.

4.2.1 Creation of Time Series Through Client-Side Metrics

In this section, we evaluate the possibility of detecting bottlenecks, based on the download times of web pages, as seen by a client. CPU bottlenecks may be due to thread pool constraints of the HTTP Server (specially the front-end machines), or CPU machine exhaustion, e.g., due to bad code design that causes unnecessary processing or waiting. I/O bottlenecks will probably be related to database (DB) operations, which clearly depend on query complexity, DB configuration and DB access patterns. We also consider network bottlenecks.

Detecting bottlenecks from the client's point-of-view is very appealing. It allows to reduce instrumentation of the server and also to identify bottlenecks that might elude administrators, such as network problems between the client and the server. Since pinpointing the exact zone of the bottleneck might be impossible, our goal is to use the web page profiling to understand if a bottleneck is present.

We propose to systematically collect timing information of one or two web pages from a given server, using the browser-side JavaScript Navigation Timing API (NavigationTiming, 2015). Figure 4.5 depicts the different metrics that are available to this JavaScript library, as defined by the World Wide Web (W3) Consortium. Of these, we will use the most relevant ones for network and server performance: the request time (computed as the time that goes from the request start to the response start) and the response time (which is the time that goes from the response start to the response end):

- Request Time: client-to-server network transfer time + server processing time + server-to-client network latency.
- Response Time: server-to-client network transfer time.

We chose these, because the request and response times are directly related to the request *and* involve server actions, which is not the case of browser processing times, occurring afterwards, or TCP connection times, happening before. Although these metrics can give us some insights of where the time is being consumed, our goal is to understand if the client can check the system health.

We argue that identifying bottlenecks and their cause with time series of these two metrics is actually possible. In this section, we improve our work from Section 4.1.

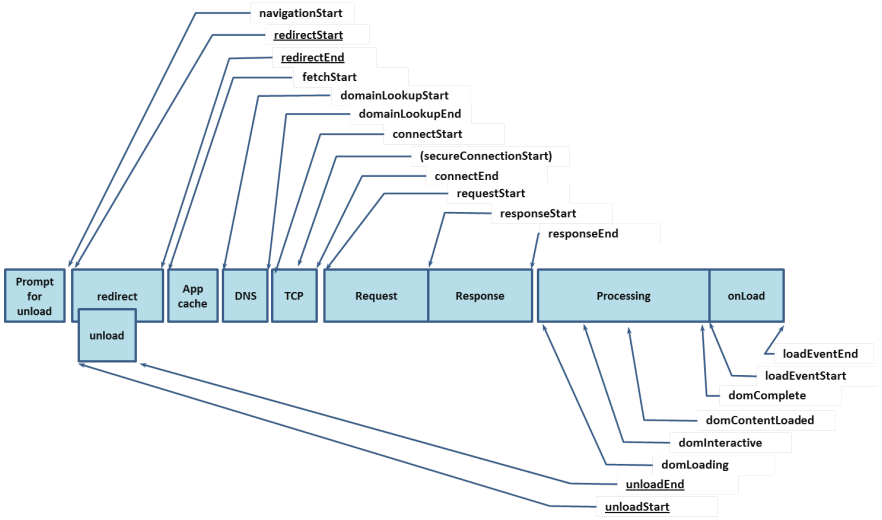


FIGURE 4.5. Navigation Timing metrics (figure from NavigationTiming (2015))

TABLE 4.5. SOFTWARE USED AND DISTRIBUTION.

Component	Observations	Version
Selenium	selenium-server-standalone jar	2.43.0
Firefox	browser	23.0
Xvfb	xorg-server	1.13.3

When a bottleneck is present in the system, a modification occurs in the patterns of the request and response times. One should notice that responses must occupy more than a single TCP (Postel, 1981) segment; otherwise, other variables may emerge in the patterns.

4.2.2 Experimental Evaluation

In this Subsection, we present the results of our experimental evaluation. We present the setup before showing the most important results obtained with the experiments.

Experimental Setup

For the sake of doing an online analysis, we used a software testing framework for web applications, called Selenium (Selenium, 2015). The Selenium framework emulates clients accessing web pages using the Firefox browser. It allows to mimic client behaviors when interacting with a web page, thus simulating real clients. Additionally, it give

us access to the JavaScript Navigation Timing API (NavigationTiming, 2015). We use this API to read the request and response times from the visited web pages. We used a UNIX client machine, with a crontab process, to request a page each minute (Crontab, 2015), using Selenium and the Firefox browser. We emulated a virtual display for the client machine using Xvfb (Xvfb, 2015). Table 4.5 lists the software and versions used.

One of the criteria we used to choose the pages to monitor was their popularity. However, to conserve space, we only show results of pages that provided interesting results, thus omitting sites that displayed excellent performance during the entire course of the days we tested (e.g., CNN (CNN, 2015) or Amazon (Amazon, 2015)) — these latter experiments would have little to show regarding bottlenecks. On the other hand, we could find some bottlenecks in a number of other web sites:

- **Photo repository** — We kept downloading the same 46 KiloBytes (KiB) Facebook photo, which was actually delivered by a third-party provider Content Delivery Network (CDN). During the time of this test, the CDN was retrieving the photo from Ireland. This experiment displays several performance problems.
- **Portuguese News Site** — this web page is the 5^{th} most used portal in Portugal (only behind Google – domain .pt and .com, Facebook and Youtube) and the 1^{st} page of Portuguese language in Portugal (Alexa, 2015). This web page shows considerable performance perturbations, especially during the wake up hours.
- **Portuguese Sports News** — This is an online sports newspaper. We downloaded an old 129 KiB news item and hit an inexistent resource for several days. The old news item certainly involves I/O, to retrieve the item from a DB or other repository, whereas the inexistent may or may not use I/O, we cannot tell for sure. We ensured a separation of 10 seconds between both requests.
- **Social Network Site** — We used the most popular social network and the largest social network worldwide. The technology demands are enormous to ensure a good quality of experience to the users and, therefore, preventing bottleneck occurrences. However, recent blackouts in the system have shown the potential of our tool to detect system anomalies and predict web page disruptions.

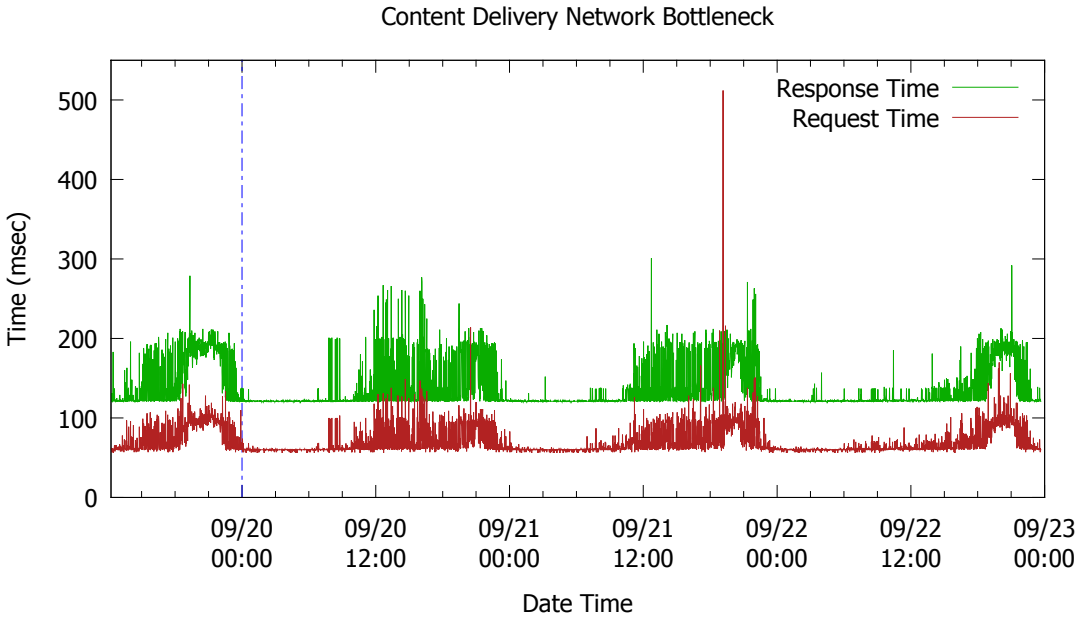


FIGURE 4.6. CDN bottleneck.

Results

In Figures 4.6, 4.7, and 4.8, we start by analyzing the results from the Content Delivery Network and from the Portuguese News site. These figures show the normal behavior of the systems and enable us to identify periods where the response times fall out of the ordinary.

Figure 4.6 shows the response of the CDN site for a lapse of several days. We can clearly observe a pattern in the response that is directly associated to the hour of the day. During working hours and evening in Europe, we observed a degradation in the request and response times (see, for example, the left area of the dashed line on September 19, 2014, a Friday). The green and the red lines (respectively, the response on top and the request times on bottom), clearly follow similar patterns, a sign that they are strongly correlated. Computing the *correlation coefficient* of these variables, $r(Req, Res)$, for the left side of the dashed line we have $r(Req, Res) = 0.89881$, this showing that the correlation exists indeed. However, for the period where the platform is more "stable" (after the dashed line and before the second peak period) we have $r(Req, Res) = -0.06728$. In normal conditions the correlation between these two parameters is low. This leads us to conclude that in the former (peak) period we found a bottleneck

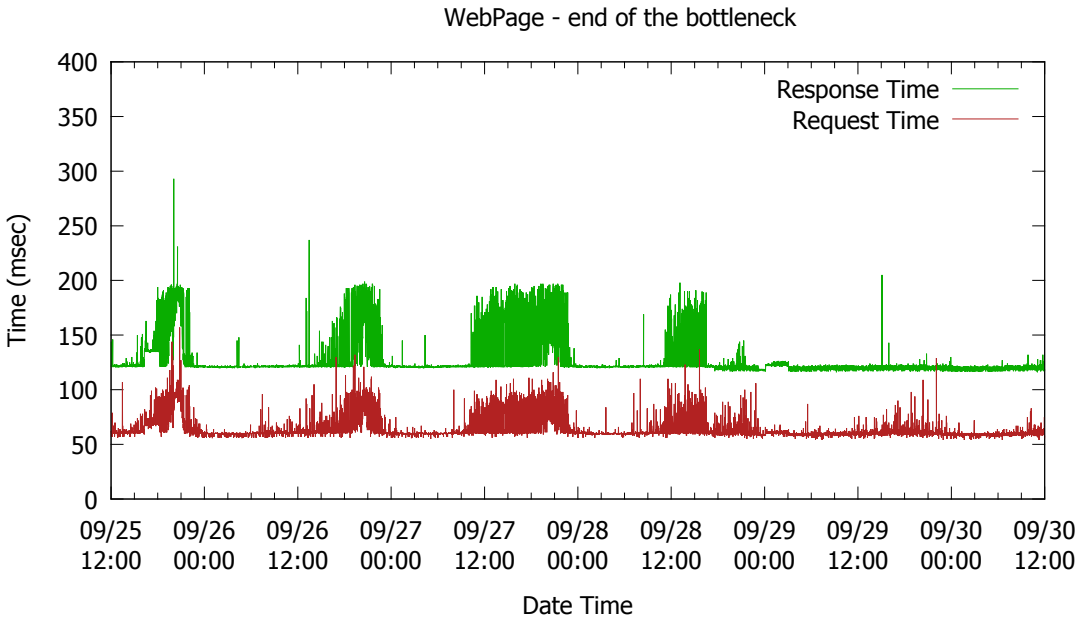


FIGURE 4.7. CDN - end of the bottleneck.

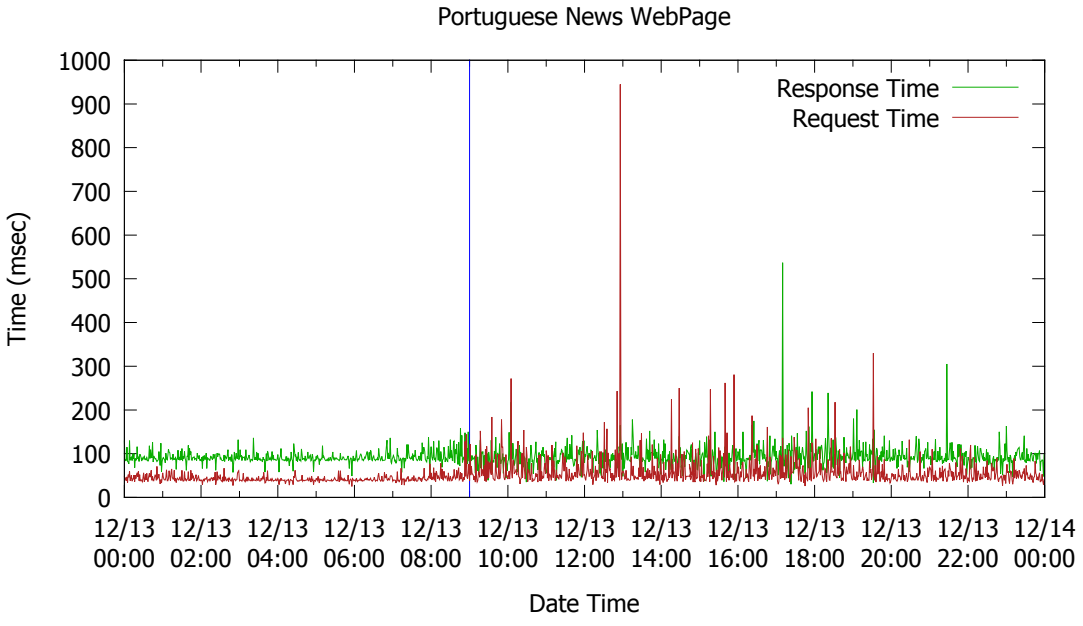


FIGURE 4.8. Portuguese News Site bottleneck.

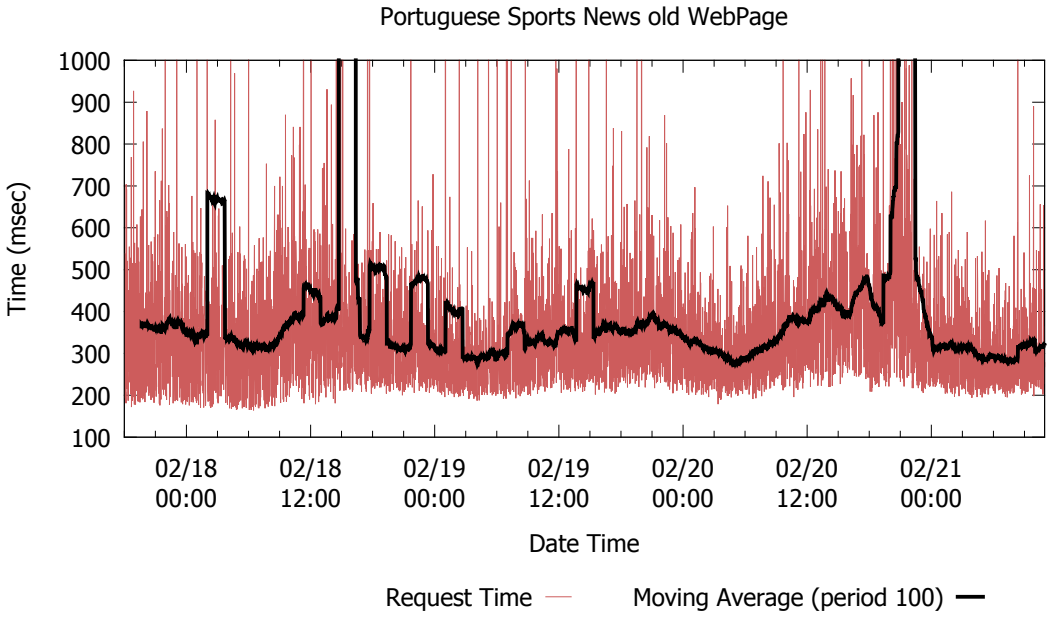


FIGURE 4.9. Portuguese Sports News old page — request times.

that does not exist in the latter. However, our method cannot determine where in the system is the bottleneck. Interestingly, in Figure 4.7, we can observe that the bottleneck disappeared after a few days. On September 29^{th}, we can no longer see any sign of it.

Regarding Figure 4.8, which shows request and response times of the main page of a news site, we can make the same analysis for two distinct periods: before and after 9 AM (consider the blue solid vertical line) of December 13, 2013 (also a Friday). Visually, we can easily see the different profiles of the two areas. Their correlations are:

- $r(Req, Res)_{before9AM} = 0.36621$
- $r(Req, Res)_{after9AM} = 0.08887$

The correlation is low, especially during the peak period, where the response time is more irregular. This case is therefore quite different from the previous one, and suggests that a different bottleneck exists in the system, during periods of intense usage.

Figures 4.9, 4.10, 4.11, and 4.12 show time series starting on February 18^{th}, up to February 21^{st} 2015. We do not show the response times of the non-existent page as these are always 0 or 1, thus having very little information of interest for us. In all

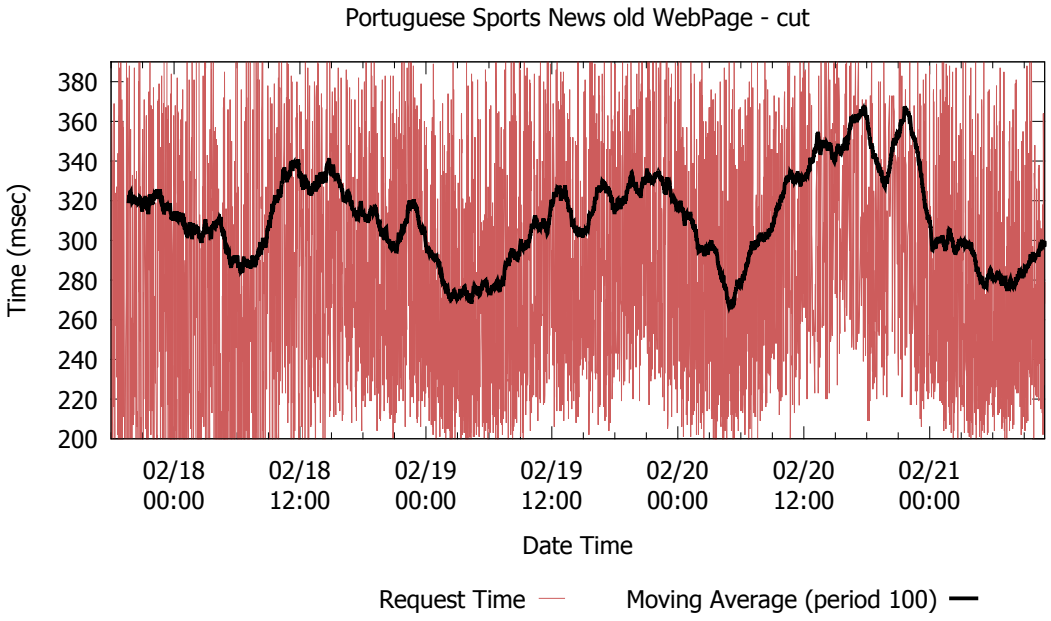


FIGURE 4.10. Portuguese Sports News old page — request times with peaks cut.

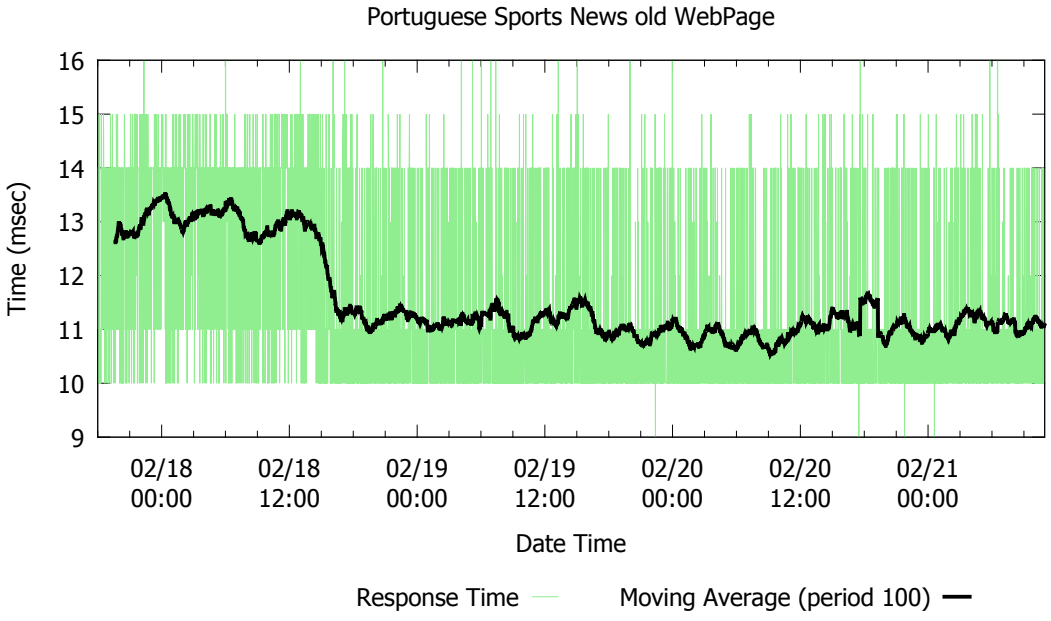


FIGURE 4.11. Portuguese Sports News old page — response times.

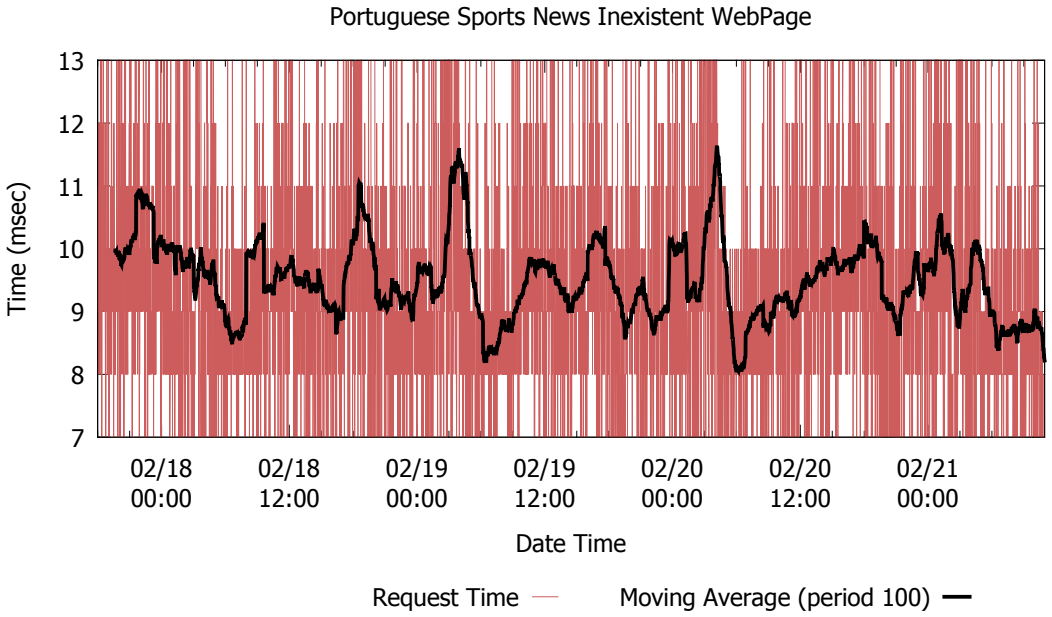


FIGURE 4.12. Portuguese Sports News inexistent page — request times.

these figures, we add a plot of the moving average with a period of 100 samples, as the moving average helps us identifying tendencies.

Figures 4.9 and 4.10 show the request time of the old 129 KiB page request. The former figure shows the actual times we got, whereas in the latter we deleted the highest peaks (those above average), to get a clearer picture of the request times. A daily pattern emerges in these figures, as daytime hours have longer delays in the response than night hours. We can visually see that the response times of Figure 4.11 do not exhibit this pattern, which suggests a low correlation between request and response times (which is indeed low). Next, we observe that the request times of the existent and non-existent pages (refer to Figure 4.12) are out of sync. The latter seems to have much smaller cycles along the day, although (different) daily patterns seem to exist as well.

Figures 4.13 and 4.14 show a period when the social network web page was down in the entire world, due to a system misconfiguration. Figure 4.13 shows how the page behaved, regarding request and response times — before, during and after the system resumed responding correctly. Figure 4.14 gives a closer look of the period before the web page failure. Time periods without request or response times occurred when the client reached the configured timeout and aborted the web page request. Currently, the timeout is configured to be 60 seconds. Analyzing Figure 4.13, we can identify the

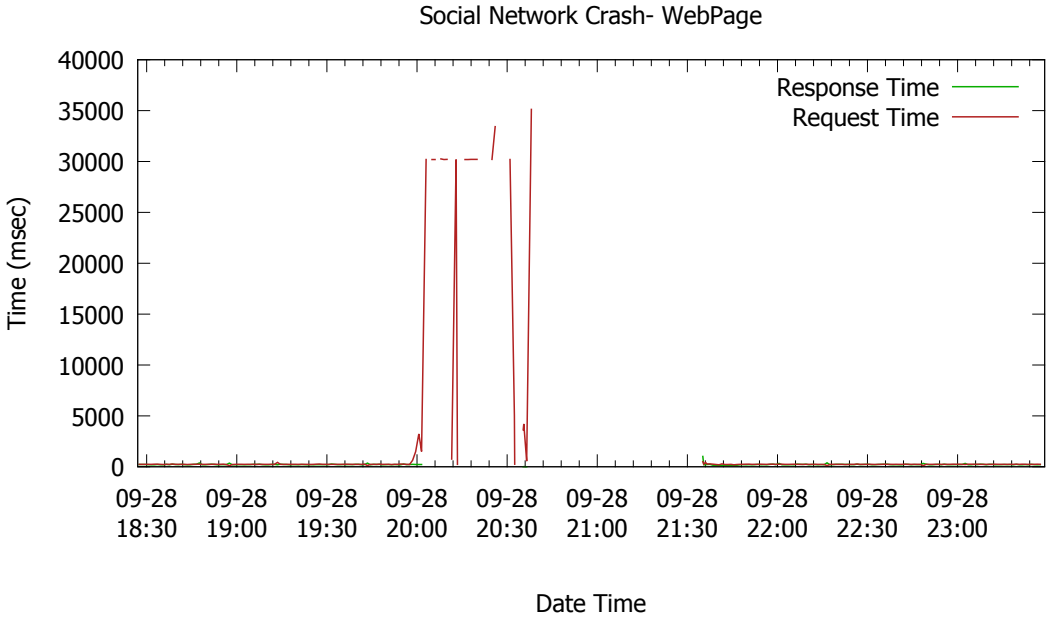


FIGURE 4.13. Social Network Web Page crash.

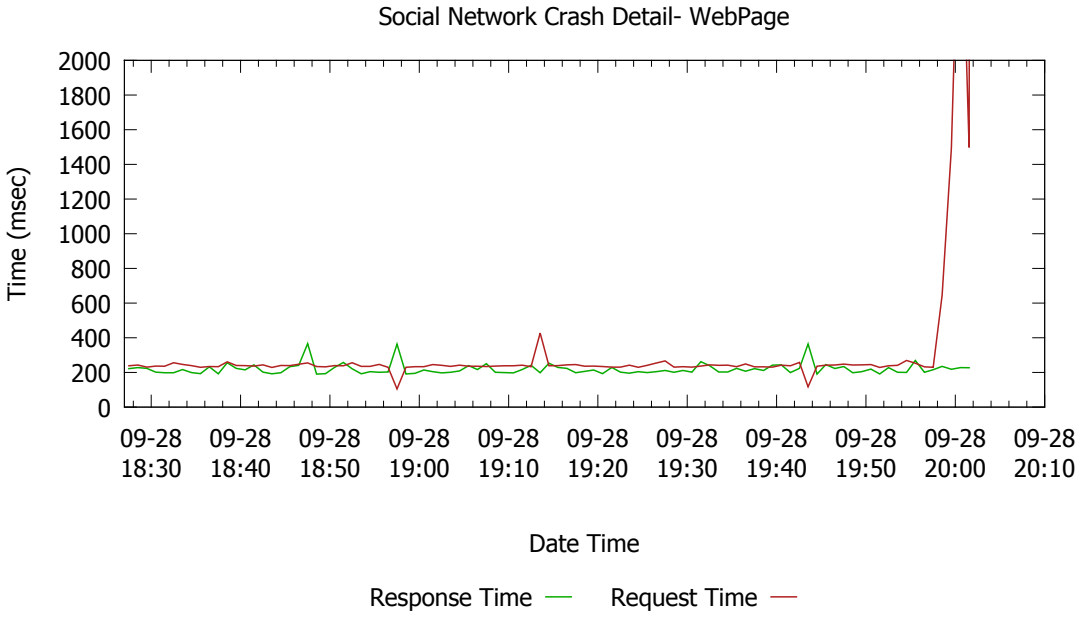


FIGURE 4.14. Social Network Web Page crash detail.

period of time when the web page was down or responding incorrectly. This might be important, if the web page is hosted in a third-party provider that might be held responsible for the failure and the user wants to complain for a refund (Azure, 2015; EC2, 2015)

Figure 4.14 gives a closer look of the minutes before the failure. The request time increased significantly, while the response time remained unchanged — that points to a bottleneck affecting access to the server, and not the response. Facebook publicly said that "configuration issue" made to Facebook service did not allow millions of users to log in the social network (DailyMail, 2015). Prior to the complete failure, we can observe a 5-minute window, where the problem was starting to become apparent, and that could be used to fix the problem or alert system administrators.

4.2.3 Client-Side Monitoring Using Two Distinct Networks

In Subsection 4.2.1, we focused on detecting bottlenecks using only client metrics, using a time series approach. We now resort to two distinct clients. The observation from two different clients should be coherent; clients should agree on whether a bottleneck exists, and where does it come from. It they do not agree a transient bottleneck may be present, or a bottleneck in a different path to the server.

Experimental Settings

We used the same technologies mentioned in Table 4.5. Each client was running a Selenium instance to invoke a specific web page, through Firefox. However, having two unsynchronized clients would invalidate the results, because we would not know if the requests were made at the same time. To eliminate this limitation we created a communication protocol between the clients in Java RMI (Downing, 1998). Before fetching a page, the two clients communicate with each other, to determine which page to get and to ensure some degree of synchronicity. The process consists in 3 steps — first, client A notifies client B to invoke a determined URL; second, both clients invoke the URL and save the request and response time; and finally, client A receives the data from the web page invocation from client B. One of the clients (client A in this description) has the role of "master", triggering the web page invocation and the collection of data from both requests. Clients should be in different locations to have

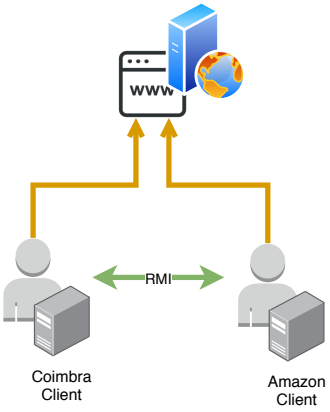


FIGURE 4.15. Experimental Setup - 2 clients

different network connectivities. We picked our own department facilities in Coimbra, and a virtual machine in the Amazon Web Service cloud in the Northern Virginia Region AWS (2015). Figure 4.15 illustrates this interaction — the RMI connection is identified in green, whereas the HTTP connections to the web page are depicted in orange.

One of the criteria we used to choose the pages to monitor was their popularity. Additionally, the web page should have the same location regardless of the origin of the client. To ensure this, we compared the IP given by the DNS to each client, to ensure that they were monitoring the same server. A second criterion was to monitor web pages from different geographical locations.

To conserve space, we only show results of pages that provided interesting results. Among these, we could find some bottlenecks in the following web sites:

- **American electronic commerce and cloud computing company** — We kept downloading the main page from this popular web page hosted in the United States from our two clients. This experiment displays a significant performance improvement, at some point in time. This improvement was observed in both clients.
- **Chinese Search Engine** — this web page is the front-end for one of the most popular search engines in the world (Alexa, 2015). It is hosted in China. This web page shows some network perturbations during a specific time in both clients.

- **Portuguese Sports News** — This is an online sports newspaper already used in Subsection 4.2.1. The web page is located in Portugal (Europe). We downloaded the main page during several days. We verified several pattern changes associated with system bottlenecks in both clients.

Results

Since we now have two clients invoking the same URL, at the same time, we expect similar server response patterns in both clients. One would assume that whenever the response time pattern in only one of the clients changes, the difference should result from some bottleneck in the client-server network path that is specific to the client observing the change. However, if both clients observe a modification in the response patterns (in terms of request and response time series), we can conclude that this is the result of a component that is common to both clients. Hence, this is the result of a system bottleneck or a common network path.

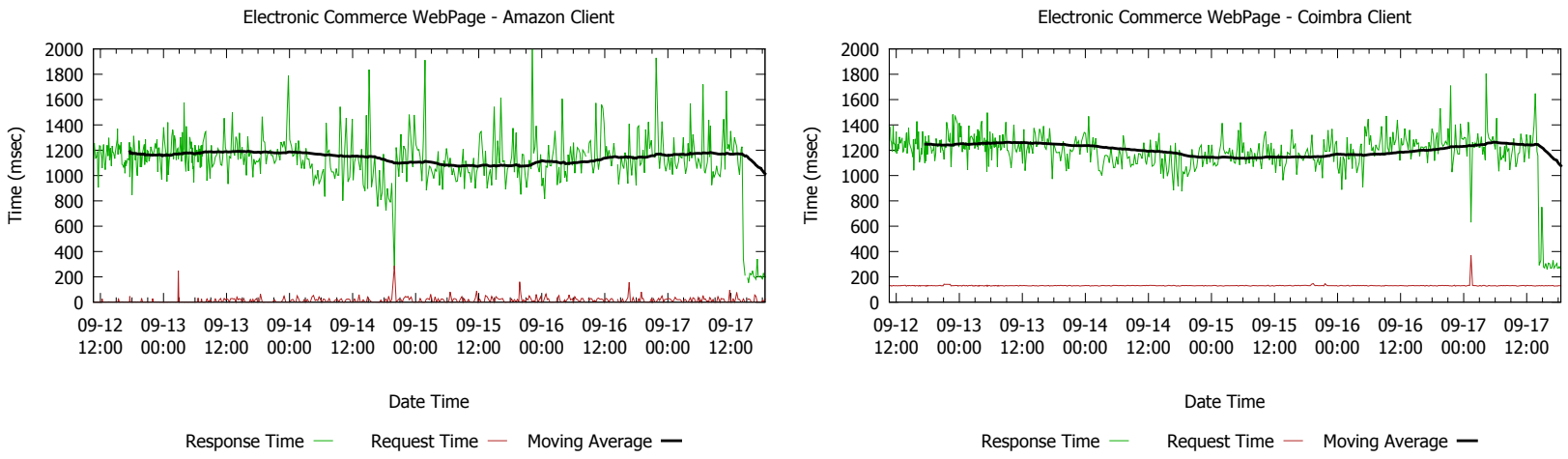


FIGURE 4.16. American electronic commerce web page

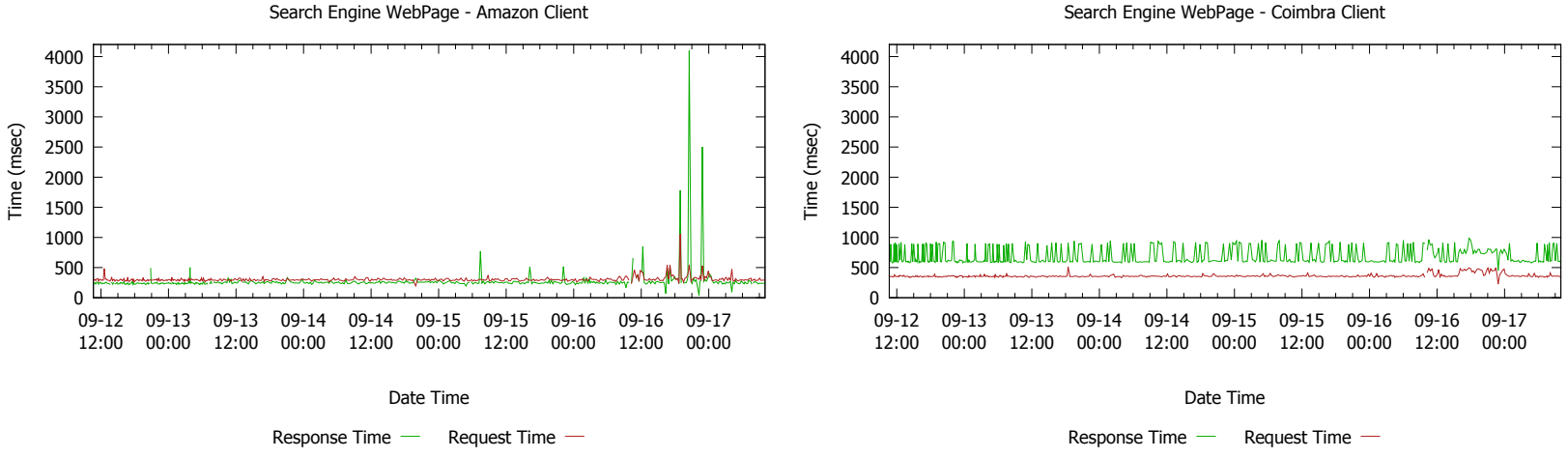


FIGURE 4.17. Chinese search engine web page

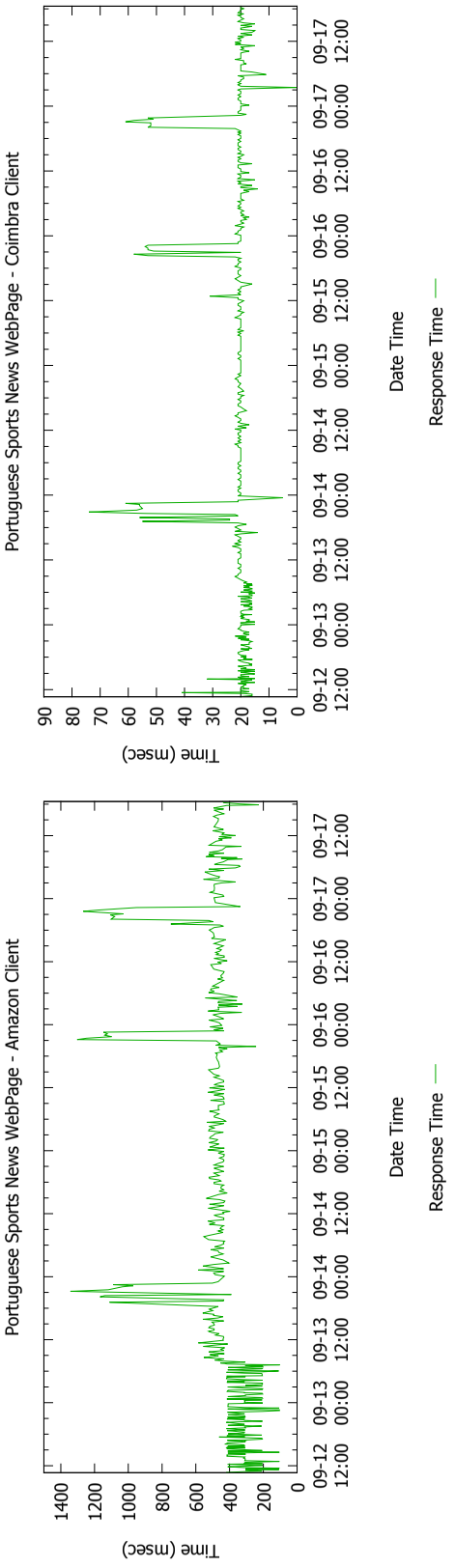


FIGURE 4.18. Portuguese sports news web page

We start by analyzing the results of the American electronic commerce web page in Figure 4.16, which shows the response of the main page for a lapse of several days. The pair of figures mostly shows normal behavior seen in Coimbra and in the AWS, thus allowing us to identify periods when the response times fell out of the ordinary for one or both clients. We can clearly observe a pattern in the response that is directly associated to the hour of the day. Additionally, the pattern exists for both clients, this meaning that both were experiencing the same constraints (system or network) from the web page. To have a better understanding of the trends, we also show a moving average of the last 100 samples of response time, in black. Computing the *correlation coefficient* for the response and request times for both clients, $r(Req, Res)$, for the interval between September 12^{th} 13:00 and September 13^{th} 13:00 2015, we get a correlation of $r(Req, Res)_{Coimbra} = 0.13896$ and $r(Req, Res)_{AWS} = -0.07370$. Since none of the clients observed significant congestion conditions, this suggest a normal behavior of the system. Near the end of the experiment, still in Figure 4.16, we can see an improvement in the response times of both clients. We can infer that the improvement experienced by both clients seems to be a consequence of a change in the web page. An improvement in the system network is less likely, because the request time rested unchanged for this period.

Figure 4.17 shows the request and response times of the Chinese Search Engine web page. The web page presents a relatively stable pattern during most of the days. During this period (before September 16^{th}), the *correlation coefficient* was $r(Req, Res)_{Coimbra} = 0.04532$ and $r(Req, Res)_{AWS} = 0.16566$, both low values. However, for the period after September 16^{th}, there was a significant change observed by the AWS client, and although less significant, also by the Coimbra client. This is even clearer when we calculate the *correlation coefficient* of both clients for this period. The correlation in Coimbra was $r(Req, Res)_{Coimbra} = 0.69707$ and in AWS $r(Req, Res)_{Aws} = 0.57794$. This means that the correlation for request and response times in both clients increased significantly, when compared to the normal pattern. We are, most likely, observing a network bottleneck in a common path between the server and the clients, since both request and response times were affected.

Figure 4.18 shows a degradation of the response time in 3 distinct moments. This degradation was observed in both clients at the same time. When we calculate the correlation for a stable period of good performance (e.g., between the first and second peak), we

Table 4.6. Correlation for the 3 peaks of Portuguese Sports News site

Correlation	Coimbra	AWS
$r(Req, Res)_{1^{st}peak}$	-0.32036	0.35263
$r(Req, Res)_{2^{nd}peak}$	0.33789	0.35815
$r(Req, Res)_{3^{rd}peak}$	0.28955	0.15749

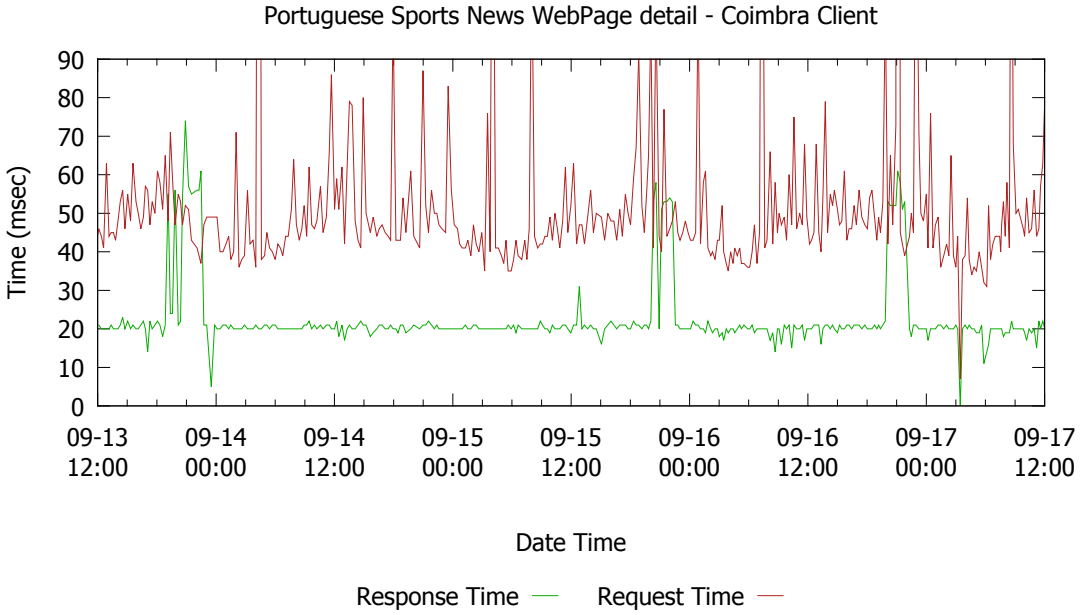


Figure 4.19. Portuguese Sports News Web Page - Detail

get $r(Req, Res)_{Coimbra} = -0.02381$ in Coimbra and $r(Req, Res)_{AWS} = 0.17699$ in the AWS. Then, for the three observed peaks, we have the correlation values of Table 4.6.

This *correlation coefficient* for both clients in the three peaks differs considerably, especially in Coimbra. We show with finer detail the request and response times of this client in Figure 4.19. The correlation is never very high, thus pointing to a problem that is not affecting in the same way the request and response time, especially in the first and third peaks.

4.2.4 Automatic Detection of Bottlenecks Using Two Distinct Networks

Our next step was to do a simple automated mechanism to detect bottlenecks, using information coming from the pair of clients. This is beneficial, not only because

automation may eventually lead to a quicker detection of performance problems, but mainly because a visual inspection as we did in the previous subsections is error-prone and is subject to all sort of biases. Indeed, with this new scheme we were able to achieve new conclusions and get a deeper insight of performance bottleneck problems.

Overview

Table 4.7. All possible combinations of congestion and correlation

Client 1		Client 2		Cause
Congestion?	Correlation	Congestion?	Correlation	
No	Irrelevant	No	Irrelevant	No Bottleneck
No	Irrelevant	Yes	Irrelevant	Client 2's Network
Yes	Similar Client 2	Yes	Similar Client 1	Server or Common Network
Yes	Distinct from Client 2	Yes	Distinct from Client 1	Multiple bottlenecks

The pair of clients provides four different variables that we can feed into an algorithm: a boolean value per client telling whether or not the client sees a congestion; and the correlation between the request and the response times, for both clients. Determining if the client is observing a congestion in the service is not trivial, in the sense that different algorithms may respond differently. In our experiments, we used an algorithm based on a moving average, described on Algorithm 2, and further detailed in the next subsection. Unlike congestion, the correlation is easier to determine. As before, we use the last 100 metrics of request and response times. Note that for the correlation to tell us something, we must be careful enough to request pages that are relatively large, but still go in a single non-chunked HTTP message. Considering high and low correlations, we get 8 possible combinations of the four variables, of which we arrange the cases of interest in Table 4.7. We omit the redundant cases, where client 1 and client 2 would simply swap their variables. For example, since we already have "No, Irrelevant, Yes, Irrelevant", in the line before last, it would be pointless to include a "Yes, Irrelevant, No, Irrelevant".

Line 1 of the table is pretty much trivial: none of the clients observes a bottleneck, therefore, looking for correlation is not relevant. In line 2, the client observing the congestion can tell that the network is congested. Since client 1 observes no congestion, the network of client 2 is the culprit. The following case is equally straightforward: when both clients observe a congestion and a similar correlation, the server or common network is the culprit. In the following case, more than one bottleneck exists: client 1 can tell that the server or common network is the most likely cause for the delay in

the request and response times, because responses are taking quite a long time. On the other hand, client 2 is also observing a congestion, but it can see a different correlation in the request and response times, thus concluding that the problem lies in other place. This is possible, if both the server and client 2 network are sources for delays. We will now try to confirm to what extent do real observations actually fit into this model.

Algorithm to detect bottlenecks

In this subsection, we present an algorithm that evaluates the variables of Table 4.7 and outputs the cause of the problem. The algorithm combines the request and response time series retrieved at the same time, by two distinct clients for the same web page, collected in our "synchronous request" experiments. We wrote the algorithm in Python and we describe its high-level details in pseudo-code in Algorithm 2. The expression CongestionClient (1 or 2) $\geq$ CongestionThreshold becomes true when the moving average of the request plus receive time of the last 100 values grows above 15% of the average of all samples, for that client. We consider the threshold that splits low from high correlation to be 0.2. The correlation also takes into account the last 100 measurements.

Results

In this subsection, we present some of the results we obtained with Algorithm 2 and compare them to the data previously analyzed in Subsection 4.2.3. Although we ran our algorithm under varied conditions, we focus on its responses for the inputs depicted in Figures 4.16, 4.17, and 4.18. We can say that all the bottlenecks we identified by visual inspection in these figures were also identified by the algorithm, which pointed out the same sources for problems. Although this might result from the specific thresholds we selected and from the size of the sliding window in the moving average and network correlation, the effort of tuning the algorithm and making it run under real data allowed us to reach two results:

- The algorithm cannot cope with request or response times that are too low. For instance, if the request or response times fall to values near the millisecond range, as in one case where the client and server were very close to each other, any small increase in the response time, no matter how small it is, will look as a congestion.

Algorithm 2 Identify Bottleneck

```
if CongestionClient1 ≥CongestionThreshold then
    if CongestionClient2 ≥CongestionThreshold then
        if CorrelClient1 ≥HighCorrelationThreshold then
            if CorrelClient2 ≥HighCorrelationThreshold then
                Server or Common Network
            else
                Client 1's Network and Server ( Multiple bottlenecks presented )
            end if
        else if CorrelClient2 ≥ HighCorrelationThreshold then
            Client 2's Network and Server ( Multiple bottlenecks presented )
        else
            Server or Common Network
        end if
    else
        Client 1's Network
    end if
else if CongestionClient2 ≥ CongestionThreshold then
    Client 2's Network
else
    No Bottleneck
end if
```

- The algorithm wrongly identified some "outliers" while handling requests that take some time to be answered. This happens because some requests take so long to get an answer that they are able to push the moving average above the congestion threshold. The interesting thing is that this sort of delay, which happens rarely, is usually not seen by the peer client, which is fetching pages from the same server; and it is not seen neither before, nor after, by the same client. This suggests that some requests get an unfair amount of wait. If this is really the case, what is the source of the problem and how often does this happen remains as an open question.

4.2.5 Discussion and Conclusion

We proposed to detect bottlenecks on HTTP servers using client-side observations of request and response times. A comparison of these signals, either over the same, or a small number of resources, enables the identification of bottlenecks. We did this work having no access to internal server data and mostly resorting to visual inspection of

the request and response times. If run by the owners of the site, we see a number of additional options:

- Simply follow our approach of periodically, invoking URLs in one or more clients, as a means to complement current server-side monitoring tools. This may help to reply to questions such as "what is the impact of a CPU occupation of 80% for interactivity?".
- A hybrid approach, with client-side and server-side data is also possible. I.e., the server may add some internal data to *each* request, like the time the request takes on the CPU or waiting for the database. Although much more elaborate and dependent on the architecture, instrumenting the client and the server sides is, indeed, the only way to achieve a full decomposition of request timings.
- To improve the quality of the analysis we did in Subsection 4.2.2, site owners could also add a number of very specific resources, like a page that has known access time to the DB, or known computation time.
- It is also possible to automatically collect timing information from real user browsers, as in Google Analytics (Clifton, 2008), to do subsequent analysis of the system performance. In other words, instead of setting up clients for monitoring, site owners might use their real clients, with the help of some JavaScript.

In summary, we collected evidence in support of the idea of identifying bottlenecks from the client side. We managed to run several experiments with a second client. Although mostly concurring with our initial observations, the second client opened an entirely new perspective: sometimes one of the clients observes a delay in a single very concrete request, which is neither observed by the other client, nor by the client itself, either before or after. I.e., even when the server seems to be delivering a normal service, clients may occasionally fail to receive a response in reasonable time.

Let us consider the unique route taken by each request: the TCP connection takes the request up to the server, where some thread reads it, processes it, and (most likely) forwards it to another layer of the system, where some thread will eventually fetch several items from the database, before enqueuing or sending the response back to the client. Each request might follow slightly different routes, depending on the threads that get it. This suggests a simple, but significant conclusion: a few unlucky requests

get blocked at some point inside the server. While there was never any guarantee that *all* requests would get a fair treatment, or that they would all get a quick response, observing such cases in a moderate number of samples is, we think, interesting. This observation raises the question of determining the exact mechanism behind starvation of some specific requests, and how likely is such mechanism to come into play.

4.3 Conclusion

This chapter focused on monitoring HTTP infrastructures. To achieve this task we used distinct approaches: first, we analyzed a laboratory experiment to understand the patterns for requests and response times that are leaked from a system that is exposed to several bottlenecks. Secondly, using this knowledge, we gathered statistics from several major web pages through weeks, to understand what kind of information the client would get.

Then we analyzed the limits of black-box monitor using only client-side data. Taking into consideration the bottleneck injection from Section 4.1 and the analysis of distinct patterns in time series of Section 4.2, we think that there is a path to improve current monitoring solutions. Having a black-box solution may be too ambitious, but a hybrid solution where the white-box standard tools are enriched with valuable data from the client-side might be promising.

We would also like to understand if a black-box algorithm — such as a machine learning algorithm —, can present good results detecting bottlenecks in a distributed system. This will be one of the research question for the next chapter: identify the possibility to have machine learning techniques to predict system components' health.

Chapter 5

Machine Learning Monitoring Techniques for Web Sites

System monitoring tools, Application Performance Monitoring (APM) or Real User monitoring (RUM) can help to ensure that users accessing services enjoy a good quality of experience (QoE). Unix boxes usually ship with a plethora of monitoring tools, such as `top`, `lsof`, or `netstat`. Together with `Nagios` (Nagios, 2019) or `Zabbix` (Zabbix, 2019) these may help system administrators to analyze a large array of raw metrics out-of-the-box, such as memory or CPU occupation by process, open files, network connections, and many others.

APM suites, such as New Relic (NewRelic, 2017), AppDynamics (2017) or DynaTrace (2017) enable analysis of specific applications, at the cost of being more intrusive. Normally, these tools launch agents on systems, which can, under certain conditions, decompose application running times in components, say database time, or service invocation. While system monitoring and Unix tools may lack application context, they are less intrusive than APMs and need fewer resources. However, Unix tools cannot grasp the users' quality of experience. On the other hand, commercial APM suites can amass much more information on the application, for some programming languages, usually at the expense of significant resources and configuration.

White-box suites, APM or simple monitoring tools, have two important disadvantages: they are closely coupled to the infrastructure, and they ignore the fact that the client, not the server, is the most important piece of the business application. The client

has some specific conditions, such as network and third-party resources that cannot be controlled (and monitored) by the system itself.

Therefore, to gain a system-wide perspective, monitoring tools should include client-side data. We have followed this path in the previous chapters. In Chapter 3, we focused on using JavaScript snippets to gather client-side information with an approach similar to Google Analytics (GoogleAnalytics, 2018). In this Chapter, we extend our previous work of determining the conditions of operation of an infrastructure and the corresponding network, from a limited amount of data. This is simple and brings the benefit of a monitoring suite that is minimally tied to the architecture and software solutions used in the system.

In previous chapters, we analyzed distinct approaches to visualize and track anomalies and detect bottlenecks in HTTP infrastructures. In this chapter we go a step further and use machine learning techniques.

Machine Learning was first introduced by Arthur Samuel. The first published work in this scientific field presented an algorithm that learned and created tactics playing checkers (Samuel, 1959). Since then, countless algorithms were introduced. A major division exists between algorithms with labelled data — supervised techniques —, and others with unlabelled data that extract insights from raw data. Some of the algorithms are demanding in terms of computational resources (e.g. deep learning), thus making deployment difficult in real environments. This was one of the reasons for the limitations of machine learning in real world applications — resources to be able to train complex models. After some technology breakthroughs, machine learning started to gain visibility once again with a popular game — chess. Machine victories over reputable world chess players, amazed the public and attracted attention to the potential of machine learning. In the words of the Danish grandmaster Peter Nielsen: "I always wondered how it would be if a superior species landed on Earth and showed us how they played chess" (Nielsen, 2019).

It was not only in chess that machine learning and artificial intelligence, disrupted current trends. E-commerce and streaming applications, recommendation systems based on clustering algorithms have recently evolved to improve services and profits of companies such as Amazon or Netflix (Gomez-Uribe and Hunt, 2015; Linden et al., 2003).

Fast forwarding 60 years from the article of Arthur Samuel, machine learning (ML) and artificial intelligence is revolutionizing almost every field of human life, and creating

new business models. In this chapter we explore the potential of ML to do client-side monitoring of a distributed system. Machine Learning might help us in many aspects of our work, such as analysing raw data, or predicting resource occupation.

The remainder of this chapter is organized as follows. First, in Section 5.1, we evaluate how we can understand if a bottleneck is occurring in a laboratory experiment. Then, in Section 5.2 we go a step further, and implement a more complex and realistic experiment analyzing the information that leaks from the distributed system. In Section 5.3 we conclude this chapter.

5.1 Black-Box Monitoring of a Single Machine

The most intrusive monitoring suites tend to ignore the actual clients' waiting times, often a result of their real locations and network conditions. Furthermore, some resources on a web page are served from third-party providers, thus slipping under the radar of server-side instrumentation. To gather a complete picture of the system, white-box monitoring solutions should include client-side data. The simplicity of installing and changing as little as possible and *automatically* observing beyond administration frontiers is extremely appealing.

In this section, we follow this path and evaluate to what extent can black-box monitoring identify different bottlenecks, using only clients' data. Real user monitoring suites, like Pingdom (Pingdom, 2017), Monitis (Monitis, 2017) or open source project Bucky (Bucky, 2017) do this to some extent, by relying on clients' data, but they mostly serve to create dashboards and trigger notifications according to a set of rules. In simple terms, our goal in this section differs from these previous approaches, because we want to automatically infer more information about the internal state of the server, using only readings from clients.

To extract information from the server, we resort to two types of timings, known as *request* and *response* times. Shortly, the request time is the time that elapses from the request to the first byte of the response, on the client, whereas the response time is the time to transfer the entire information following this byte. Based on these times, we aim at pinpointing the precise cause for internal and external bottlenecks. We created a laboratory experiment with the Mediadrop open source video service (Mediadrop, 2017), to identify two possible sources of bottlenecks: CPU and network. We ran a set

of clients under a combination of 100 different types of CPU and network performance restrictions. For each of these combinations, we collected the request and response times, for the batch of client requests. Using these data, we trained two algorithms, a linear and a non-linear one, to identify the state of the server.

Our results demonstrate that client-side data enable CPU and network bottleneck identification, thus showing not only the utility, but the necessity of considering client metrics for performance monitoring. The increase in complexity to collect the additional data is also fairly small, because JavaScript snippets can do the trick of uploading limited amounts of performance data to the server. The trade-off involved thus seems to be quite favorable and independent of the operating system, server-side programming language, or platform.

The rest of this section is organized as follows. Subsection 5.1.1 describes the problem we tackle in this section and the method we used to solve it. Subsection 5.1.2 describes the experimental settings. Subsection 5.1.3 discusses distinct machine learning alternatives to solve our problem. In Subsection 5.1.4 we show the results of our experiment and evaluate the meaning of these results, the strengths of this approach and the limitations.

5.1.1 Problem Description

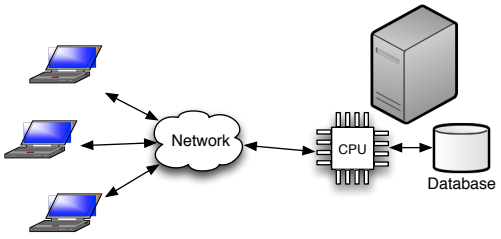


FIGURE 5.1. Representation of the considered server-side bottlenecks

As we show in Figure 5.1, we consider a stand-alone HTTP server, including a relational database, and a set of clients requesting objects available on the server. We collect the request and response times seen by each client, and, based on these times, we aim at determining the level of utilization of network and CPU on the HTTP infrastructure, including the client-server network. We assign a real number in the interval $[0, 1]$ to the availability of each of these resources, where 1 stands for a completely available resource and 0 for a completely occupied one.

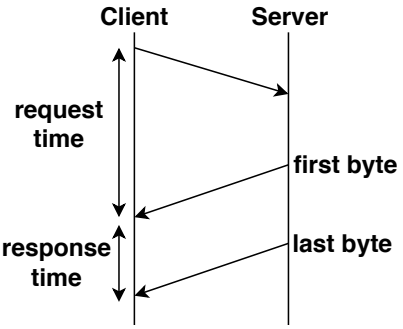


FIGURE 5.2. Request and response times

We use two metrics that are available on web browsers, via JavaScript: the request time and the response time. In Figure 5.2, we can see that the request time is the time it takes from the client's request to the first byte of the response. This time includes the network round-trip-time, plus all the server processing delays. The response time is the time from the first byte to the last byte of the response. To some extent, the former time mostly involves latencies, whereas the latter mostly concerns network transfer throughput, although in some cases the server might also need to perform extensive computation and disk operations to produce the first byte, or even the remaining bytes of the response.

To automatically determine the kind of bottleneck affecting the server, clients perform object requests to the server and measure the request and response times. As we discuss in Subsection 5.1.4, we eventually discarded the response time, and used the request time alone. Based on a sequence of request times, the goal is to determine the level of availability of CPU and network in the $[0, 1]$ scale. Note that in real conditions, system administrators can follow an approach similar to Google Analytics (GoogleAnalytics, 2018), to upload client data to their facilities and analyze such data, to determine server operating bottlenecks.

In our experiments, we run the server under a wide range of controlled conditions, knowing beforehand the exact level of availability of each resource. Each of these conditions produces different request times on the clients. We then get the times directly from the clients and add the availability numbers for the two resources, in the $[0, 1]$ interval. We train machine learning algorithms using these data, to perform an offline analysis of the overall performance, to understand the possibility of pinpointing the server status using only client data.

5.1.2 Experimental Setup

To experiment our method, we used the Mediadrop (Mediadrop, 2017) open source video platform. Mediadrop is one of the components of BenchLab (Cecchet et al., 2011). BenchLab addresses the limitations of older benchmark tools like TCP-W (TPC-W, 2017), or RUBIS (Rubis, 2017), which are outdated and do not include Web 2.0 features.Ceehet *et al.* (Cecchet et al., 2011) list some more possibilities, but their adoption is somewhat more complicated, because these benchmarks lack open source implementations.

We installed Mediadrop on a Ubuntu 14.04.1 LTS Server x64, running on a Citrix XenServer virtualization platform. The virtual machine has 2 single-core Intel Xeon CPU E5-2650 0 @ 2.00GHz virtual processors, and 2 GiB of RAM. We used a server thread pool with size 10, along with other Mediadrop default settings and components. The storage is accessed through a storage area network, via 10 Gbps fiber channel. Mediadrop supports HTML5, Flash, and includes features such as video statistics or popularity, social network integration with Facebook and Twitter, content management and the ability to import videos from Youtube. Mediadrop has an off-the-shell front-end that can be accessed through a browser to let users see, import or comment videos. It was written in Python and can be extended via plugins.

To simulate CPU bottlenecks, we used the cpulimit tool (Cpulimit, 2017), which limits the CPU usage of a process. To restrain the network available to the Mediadrop server, we used the traffic control tool (TrafficControl, 2017). We varied the available CPU from 10% to 100% in steps of 10% (for 10 different values overall) plus the network from 100 kbps to 1000 kbps in steps of 100 (another set of 10 values). In total, with these two tools, we operated the server under 100 different conditions (10×10), by doing requests from the client. The goal is simple: understand if, from the client's point-of-view, we can recognize these distinct patterns of network and CPU usage.

To observe the effect of the aforementioned tools, and ensure the correct outcome of the experiments, we resorted to the `/proc` virtual file system. For the CPU, we got data directly from `/proc/stat`; for the network utilization, we used `bmw-ng` (BwmNg, 2017).

We ran client processes on the same virtualization infrastructure as the server, using a similar hardware configuration, with two identical single-core virtual CPUs, in the

Table 5.1. Software used and distribution.

Component	Observations	Version
Mediadrop	open source video platform	0.10.3
Selenium	selenium-server-standalone jar	2.53.1
Firefox	browser	45.4.0
JMeter	performance application	3.0
Xvfb	xorg-server	1.13.3
cpulimit	binary	0.2
traffic control	change network bandwidth	1.0.2

same local area network. The client's operating system is CentOS 6.7 x64. To perform client-side operations, we used a test tool called Selenium (Selenium, 2015). Selenium is a framework that emulates clients accessing the Internet using browsers. Normally, this tool is used for front-end tests, but it can be used to automate tasks, such as accessing a Uniform Resource Locator (URL) and collect the responses, as we do in this Section. To avoid being tied to a specific browser, Selenium uses a *WebDriver*, which allows the framework to use pretty much any option. In our experiments we used Firefox. To run multiple browsers without real screens, we used Xvfb (2015), to emulate a display and perform the graphical operations. To control the clients, we injected the browser requests through Apache JMeter (JMeter, 2013), which is a standard performance evaluation tool. JMeter triggered the clients, each one of them using the Selenium framework, coupled with Firefox. We resorted to the Navigation Timing Application Programming Interface (API) (NavigationTiming, 2015), to collect performance times. This API, is a JavaScript-based mechanism that runs on the browser and enables collection of several performance times, including interaction times with the server, as well as rendering and processing times of the browser itself. Figure 5.3 depicts the different metrics that are available to this library, as defined by the World Wide Web (W3) Consortium.

This API collects information like DNS or TCP times, as well as the request and response times we need for our algorithms. Table 5.1 summarizes the software and respective versions of the most important components of the setup. Our utilization of standard tools reproduces production site conditions, something that would not be possible with customized clients.

In our experiments, we used 4 browsers, each one of them triggered by a distinct JMeter thread. Each browser requested the entry page of Mediadrop 25 times. With

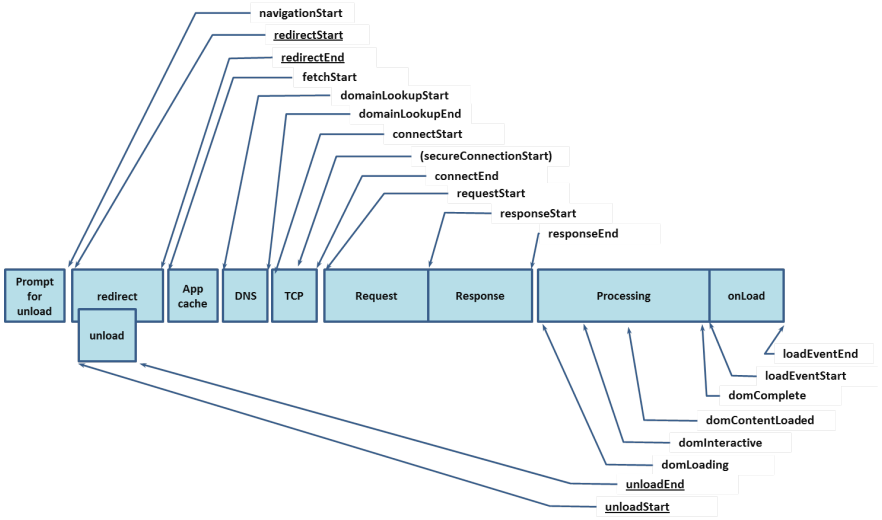


FIGURE 5.3. Navigation Timing metrics (figure from NavigationTiming (2015))

the combination of the two metrics, plus three iterations, our infrastructure generated 300 distinct results (10 steps for CPU $\times$ 10 steps for network $\times$ 3 iterations). From these 300 we collected 100 distinct results using the median of the three iterations that each client did for the 25 requests. These 100 values, together with the operating server conditions, will be the input of the machine learning algorithms of Section 5.1.3.

The program we used in the experiment is summarized in Algorithm 3.

5.1.3 Machine Learning Approach

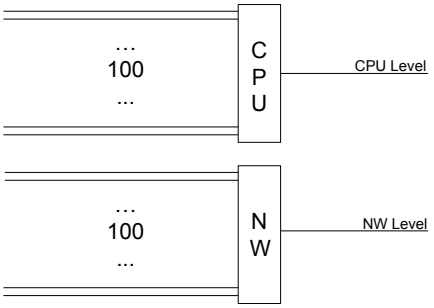


FIGURE 5.4. Machine learning regression models for CPU and network prediction of availabilities

We followed a machine learning approach to predict CPU and network availabilities from the clients' input data. We created a regression model (rather than a classification)

Algorithm 3 WebPage report

Input: Range of metrics to measure
Output: web pages metrics

```
1: Initialization :
2: cpu_range = list(range(10,100,10))
3: network_range = list(range(100kbit, 1000kbit, 100))
4: iteration_range = list(range(1, 3, 1))
5: Open Mediadrop Application
6: LOOP Process
7: for all cpu_rank in cpu_range do
8:     Limit Mediadrop Application to cpu_rank
9:     for all nw_rank in network_range do
10:        Limit NW Bandwith to nw_rank
11:        for all it_rank in iteration_range do
12:            Run it_rank iteration
13:            Invoke JMeter from the client machine;
14:            Create 4 threads with respective Firefox Browsers;
15:            Invoke web page link;
16:            Save metrics to File;
17:        end for
18:    end for
19: end for
20: Parse Data and create final output with the median from all experiments.
```

for each of the two problems, as illustrated in Figure 5.4, since the output variables take continuous values (rather than class labels). The idea behind regression is to predict a real value, based on a previous set of training examples (Sen and Srivastava, 2012). We provide 100 different lines to each regression model, where each of the lines has 100 inputs: 25 request times seen by the first client, another 25 by the second, etc.. There is a 101^{st} value in the line, which is the availability of the resource (i.e., the actual output value, e.g., CPU). This reference is necessary for training, but it is not available in the test cases.

A wide range of regression methods are available in the literature (Witten et al., 2016). In the context of the present study, two of them assume particular relevance: simple linear regression (SLR) and Support Vector Regression (SVR). SLR is a simple algorithm where input-output linearity is assumed. This method was selected for its simplicity, speed and adequacy as a baseline approach, following the principle of Occam's razor or law of parsimony ("when you have two competing theories that make exactly the same predictions, the simpler one is the better") (Thorburn, 1915). However, since

TABLE 5.2. Regression results for CPU and network availabilities

Method	CPU		Network	
	MAE	CC	MAE	CC
SLR	0.21 ± 0.03	0.59 ± 0.12	0.28 ± 0.05	0.29 ± 0.17
SVR	0.12 ± 0.02	0.82 ± 0.05	0.14 ± 0.02	0.79 ± 0.05

the linearity assumption may not be valid, we also evaluate a non-linear solution, in this case, SVR. SVR is a particular case of Support Vector Machines, where the outputs are real-valued and input-output non-linearity is typically assumed, by using a non-linear kernel, e.g., polynomial or radial basis function kernels. SVR's have proved to outperform other regression approaches in a wide range of problems (Panda et al., 2013).

These algorithms were run using the Weka framework (Hall et al., 2009). For SVR, the normalized polynomial kernel was selected based on experimental results. Both input and output data were normalized to the $[0, 1]$ interval, to attain better numerical behavior in SVR training. As for algorithmic parameterization, both algorithms were employed with default parameters. Finally, all experiments were performed using 10-fold cross validation with 20 repetitions.

5.1.4 Results

As we referred in Section 5.1.2, we collected a set of 100 request and response times for the Selenium and JMeter client invocations. As we went through the data, we observed that the request time completely dominated the overall time necessary to retrieve the webpage, whereas the response time only contributed with an insignificant offset, thus providing little or no information at all. For this reason, we used the request times alone as the input to the machine learning algorithms.

The results obtained for the CPU and network availabilities regression models are summarized in Table 5.2. We report average results for two evaluation metrics: the mean absolute error (MAE) and the Pearson correlation coefficient (CC), between the predicted and actual values. Since we performed 20 repetitions of 10-fold cross-validation, we present average and standard deviation results.

As one can observe, good results were attained for CPU and network availability prediction. Average 0.12 MAE (with 0.02 standard deviation) and 0.14 (with 0.02 standard

deviation) were attained using SVR, respectively. On a $[0, 1]$ range this is a good result. Regarding correlation, high correlation coefficient values (0.82 and 0.79, respectively) also denote that the proposed SVR approaches adequately model CPU and network availabilities.

Both SLR and SVR work better with the CPU than with the network, because CPU bottlenecks tend to dominate the request time. When the availability of the CPU is only 0.1 the average request time of the 100 requests (4 clients times 25 requests) can grow as large as $10,000$ ms, whereas a very occupied network can only raise request times to around 400 ms. Hence, very large request times point to low CPU availability, but can make it quite difficult to identify the state of the network, because this component contributes relatively less to the overall delay. Indeed, when CPU availability is only 10%, the correlation coefficient (CC) of the network decreases to 0.4, for SVR, as we try all network availability levels. Finally, we observed that when the CPU and the network contribute with delays of the same magnitude, in the order of $200 - 300$ ms, the correlation coefficient of the network grows to 0.75. Put in other words, SVR could easily identify the availability of the resource that is responsible for the longer delays (CPU), and also did a good job for a second resource (network), as long as the relative magnitudes of both delays are more or less the same. In the extreme case where one of the resources (CPU) dominates the delay, the other (network) becomes almost invisible.

Comparing SVR and SLR, the former clearly outperforms the latter in all cases and in both metrics (MAE and CC). To evaluate the significance of these results, statistical significance tests were performed using the MAE results obtained for SLR and SVR. As both MAE distributions were found to be Gaussian using the Kolmogorov–Smirnov test, the paired T-test was carried out. These results proved statistically significant (p-value < 0.001). Hence, the improved performance obtained from the non-linear SVR model shows that the input-output relationship cannot be adequately captured using a simple linear regression model.

5.2 Black-Box Monitoring of HTTP Clusters

In the previous Section, we focused on the detection of bottlenecks resorting to multiple clients and a single HTTP server, with the goal of analyzing the feasibility of a black-box approach.

We now extend the previous section to a more complex scenario, with a cluster of server machines. Our goal is to determine the conditions of operation of the cluster of servers and network, using a limited amount of data. We aim to experimentally evaluate whether a client can tell if a specific machine of the cluster is underperforming. This is simple and brings the benefit of a monitoring suite that is minimally tied to the architecture and software used in the system.

To get information from the server, we will continue to use the request time. Again, we resorted to the Navigation Timing API framework (NavigationTiming, 2015), available in the most common browsers, to get this time.

Based on client data, we analyze the possible causes associated with internal or external bottlenecks. We ran several clients that accessed the Mediadrop application, under a combinations of different CPU and network operating conditions. For each combination, we collected the request times, for the batch of clients accessing the web page. The data was used to train two distinct machine learning algorithms, a linear and a non-linear one.

Our results demonstrate that it is possible to have CPU and network bottleneck detection using client-side data even for a cluster of servers. Since data is gathered using a JavaScript snippet, it would be easy to upload this information to a central point, with the convenience of this approach being operating system and platform independent. This solution is clearly more limited than in a single-server machine, because other machines of the cluster can now compensate for problems in one of their peers. Nevertheless, we can still identify in coarse terms the operating conditions of each machine. Given the simplicity and the complete scope of this form of monitoring, we argue that this approach should be a complement to standard monitoring solutions, with the goal to improve the clients' quality of experience.

The rest of the Section is organized as follows. Subsection 5.2.1 describes the method. Subsection 5.2.2 describes the experimental settings for the problem we tackle in this

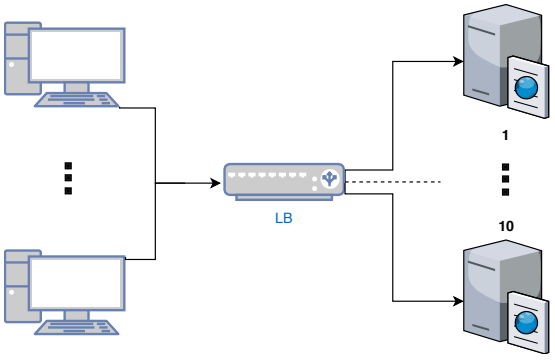


FIGURE 5.5. Representation of the considered infrastructure

section. Subsection 5.2.3 discusses distinct machine learning techniques to solve our problem. In Subsection 5.2.4 we show the results of our experiment, evaluate the meaning, the strengths of this approach, and the limitations.

5.2.1 Problem Description

In our experimental evaluation, as illustrated in Figure 5.5, we consider an HTTP infrastructure comprised of 10 server machines running a social media application (Mediadrop, 2017). In the front of the server machines we have a load balancer (LB) and a set of clients requesting HTTP objects.

As in Section 5.1, we use a metric that is available in all modern web browsers, via the framework Navigation Timing API: the request time. Figure 5.2 depicts this metric, including the server processing time and the network round-trip-time from the beginning of the request, until the first byte of the response has arrived. As before we do not need a complementary metric, the response time, which includes the time needed to get all the response. Since we were always able to infer network occupation, from request time alone, we simply did not use the response time.

Besides the request times observed by each client, we added an indication of which back-end server sent the reply in the response. Clients should then upload their data to a central point, from where we aim to automatically infer two metrics: 1) the level of occupation of the server's CPU and 2) the client-server network occupation. This information will enable system administrators to know the real operating conditions that clients experiment, and to react to bottlenecks in parts they control.

In Section 5.1, we directly used raw data collected by the clients without any kind of processing. Therefore, the order of arrival of the clients' responses had an impact in the method, i.e., having client A's response before client B's response was not the same as having the responses in reverse order. The reason for this is that we fed the request times to the machine learning algorithms in some specific order. In this section, we wanted to avoid this dependency. Hence, we used the averages and median times of all the clients' raw data, in a rolling window approach. These values were then mapped with the vacancy of the resources in the interval $[0, 1]$ (0 is entirely occupied, 1 is entirely vacant).

As we analyze on Subsection 5.2.4 to achieve better results in the machine learning algorithms, we also added information of where the request was processed, having this way a more fine grain information concerning the system. In a production environment, to collect this information, an approach similar to GoogleAnalytics (2018) could be made, to collect and determine the operation status of the server.

The server was run in different operation sets, knowing *a priori* the vacancy of each resource. The clients for each set ran several requests and all the data was collected. Using this data, we trained the machine learning algorithms.

5.2.2 Experimental Setup

In our experimental setup, we used as before, the open source platform for video contents, named Mediadrop (Mediadrop, 2017). The Mediadrop software was installed in ten machines running Ubuntu 16.04, running on a virtualized platform. Each virtual machine has 2 single-core Intel Xeon CPU E5-2650 0 @ 2.00GHz virtual processors, and 1 GiB of RAM. We used the Mediadrop default settings.

To simulate the CPU and network loads, we used two distinct tools. In one specific machine of the cluster, we used the cpulimit tool (Cpulimit, 2017). This tool limits the CPU of the process running Mediadrop. To limit the network, we used the traffic control tool (TrafficControl, 2017). CPU vacancy for the Mediadrop processes was limited from 10% (almost entirely occupied) to 100% (entirely vacant) in steps of 10%. For the network, we used 5 levels, as we did not notice much performance differences in having additional levels. The network levels used were $50, 100, 250, 500$ and 1000 kbps. This gives a total of $10 \times 5 = 50$ different server conditions.

As before, to control the actual effect produced on the servers, we used standard tools, such as the information obtained from `/proc/stat` or `top`, for the CPU utilization, and `bmw-ng` (BwmNg, 2017) for the network usage.

The load balancer used to distribute requests among the 10 back-end servers was an `NGINX` proxy server. The load balancing algorithm used was round-robin. It is relevant to mention that the load balancer includes a health-check mechanism that cuts the load on heavily stressed machines, thus having some interference on the experiments we did.

Client applications use a similar hardware configuration, with two virtual CPUs. The operating system is CentOS 6.7 x64. Since we wanted to invoke the Mediadrop webpage, we used Selenium (Selenium, 2015). In this scenario, we used this tool to access the Mediadrop webpage in an autonomous way, and collected the request times.

To control the several clients that access the infrastructure, with minimal effort, we integrate the Selenium clients with Apache JMeter (JMeter, 2013), which is a standard performance tool. Hence, our experimental setup concerning the clients, consisted of JMeter invoking our clients emulated by Selenium and Firefox, which collected the information from the Navigation Timing API (NavigationTiming, 2015). The components used in the experiment are identical to Table 5.1 with the inclusion of NGINX to balance the load throughout the 10 machines.

We used a total of 4 browsers, each one of them associated with a JMeter thread. We configured each thread (i.e. each client) to invoke the entry page of Mediadrop application 50 times - for a total of 200 requests. To provide more data to the machine learning algorithms, we made 20 iterations for each combination of CPU and network occupations. This means that for the initial 200 requests, we generated $4,000$ raw results, for each of the 50 configurations — a grand total of $200,000$ requests of total raw data for all the experiment.

The $4,000$ requests of each setup were aggregated to produce results independent of the invocation order. For each of the $4,000$ requests concerning one configuration, we aggregated request times by computing the following data: the number of requests processed by the back-end server machine (as we said, the server leaks the machine ID to the client), the minimal request time, the maximum request time, the median of the request time and the mean request time observed. With this information, we added

the CPU or network level, depending on the regression case we are trying and fed the data to the algorithms of Section 5.2.3.

5.2.3 Machine Learning Approach

FIGURE 5.6. Example of Machine Learning regression models for CPU prediction of availabilities

With the information collected in our experiment, we used two machine learning algorithms to predict the CPU and network occupation, based on the clients' data. The method used in this section is similar to the one used in previous section, however some difference exists. We aggregated the raw data for the several iterations, where each line had the target value of the configuration of CPU or network. Each line has the features represented in Figure 5.6 and the output to be predict, which is the vacancy of the resource (i.e., the actual output value, e.g., CPU in the figure). This latter value is available in the test scenario, but unavailable in the validation cases. The vacancy of resources is normalized in the interval $[0, 1]$, being 1 an entirely free, and 0 a totally occupied resource.

There is a wide body of knowledge regarding regression models, e.g. Witten et al. (2016). In this context, we opted for the two previous used algorithms: Simple Linear Regression (SLR) and Support Vector Regression (SVR).

Both algorithms were run using the Weka framework (Hall et al., 2009). For the SVR, we used the normalized polynomial kernel, with normalized data in the interval $[0, 1]$, to achieve better behaviour in the training set. For SVR and SLR we used default values for the remaining parameters. To analyze the data, we used 10-fold cross validations, with 20 repetitions.

5.2.4 Results

The results obtained for CPU and network occupation for the two distinct models are presented in Table 5.3 and in Table 5.4. Table 5.3 presents the values for the 10 levels of CPU and 5 levels of network and in Table 5.4 we present the values with a

TABLE 5.3. Regression results for CPU and network availabilities — all occupation levels

Method	CPU		Network	
	MAE	CC	MAE	CC
SLR	0.18 ± 0.03	0.43 ± 0.11	0.19 ± 0.01	0.56 ± 0.11
SVR	0.16 ± 0.06	0.45 ± 0.17	0.21 ± 0.13	0.51 ± 0.19

TABLE 5.4. Regression results for CPU and network availabilities — 3 ranges of levels (small, medium, large)

Method	CPU 3 levels		Network 3 levels	
	MAE	CC	MAE	CC
SLR	0.12 ± 0.03	0.70 ± 0.11	0.18 ± 0.08	0.71 ± 0.16
SVR	0.15 ± 0.10	0.85 ± 0.12	0.15 ± 0.13	0.76 ± 0.17

subset of the collected values. This subset was chosen taking into consideration the small, medium and large values of the interval of both CPU and network occupation mentioned in Section 5.2.2. The problem of having the complete range of availabilities is that similar levels of CPU and network are very difficult to separate due to the influence of the remaining machines in the cluster. We thus restricted our goal to check if the methods can actually do a good job of telling the big picture regarding the resource state (resource available, unavailable or halfway).

In Table 5.3 and Table 5.4 we show the MAE results between the estimated and the real value. Since we make a 10-fold cross-validation, with 20 repetitions, we present the values associated with the average and standard deviation of all the iterations.

While we get acceptable results in Table 5.3, results in Table 5.4 are much better. We can see that the former results are only average, but if we only consider 3 ranges of vacancy (small, medium, large), as in the latter table, the results are very good. We get a particularly high correlation, as well as low MAE, this meaning that both machine learning methods, specially SVR can do an excellent regression.

Results are better for the CPU bottlenecks, because, as before, the CPU bottleneck tends to dominate the overall request time. Specially in lower CPU vacancy (10%) the overall request time grows considerably (in the order of $10,000$ ms), whereas a very occupied network bottleneck only produces a delay of 400 ms. Therefore in the extreme case of low CPU and network resources, the time is largely dominated by the CPU starvation.

When we compare the results of this section with the work of Section 5.1, we verify that results are not so good now. Since the infrastructure is more complex, with a load balancer and 9 extra servers, patterns in the request time become more elaborate. This is true even knowing the machine sending the response. In fact, if we did not know this, we would not be able to get any information at all regarding the state of one particular server (the other 9 would compensate for any trouble).

5.3 Conclusion

Monitoring of web pages is a major challenge, due to the complexity, third-party resources and the need to achieve excellent quality-of-experience. The standard approach is to rely on white-box tools that collect performance metrics from the server. However, this tends to be intrinsically associated with the architecture and software used, thus excluding the clients' point-of-view. Since the goal is to understand if the quality of experience is good, it is important to collect metrics from the client, especially due to the client-to-server network and external resources that are outside of the administrators' control.

The main objective of this chapter was to perform automatic detection of bottlenecks. The evidence we collected supports the idea that a black-box monitoring system, using client metrics alone, can be achieved with limited effort. In particular, results demonstrate that it is possible to accurately separate an internal server bottleneck, such as CPU, from external network bottlenecks. Furthermore, since this method is not tied to any operating system or programming language, it is compatible with current monitoring systems, thus being able to complement standard white-box monitoring tools.

In the process, we realized that there are limits to what we can infer from the client perspective, or more precisely, and in converse terms, we identified some of the information that the server and networks leak about its internal condition, during their operation (in a process that bears some similarities to black-box attacks that aim at gaining information from the physical implementation of crypto systems (Vateva-Gurova et al., 2015; Wang et al., 2017)).

As future work, we want to mitigate some of the disadvantages of using a supervised machine learning technique. Using a trained data set may be unfeasible to do in large production systems. Next we do an attempt to build a model of complex systems from

the client's observations, and use that model to infer the internal state of the system. This will be one of the research question for the next chapter — identify the possibility to simulate and predict system components' occupation in microservice architectures.

Chapter 6

Microservice Monitoring Techniques

Microservice systems are a modern approach used by technological companies to create highly available, elastic and dynamic systems. This kind of architecture enables teams to work independently on different services and ensures that modules are oblivious to changes in the surrounding system.

There are some challenges concerning monitoring and observability of microservice production systems, as these become more complex compared to monolithic solutions. Dynamic scalability, network distribution, fragmented resources, or dynamic architectures, make microservice systems difficult to observe and control. When compared to monolithic solutions, microservices have more possible points of failure and can severely decrease the quality of service. In monolithic systems, monitoring is restricted to the system, in a stable infrastructure with little elasticity. In microservice systems, administrators have to pinpoint the root cause of the anomaly in hundreds or thousands of machines, with services that have high elasticity and communicate with each other. The increased complexity creates a difficult task for administrators.

More traditional approaches based on adding instrumentation to the source code, or generic metrics such as CPU or network occupation, agents, logging, watch-dogs, dashboards, etc., have serious shortcomings for modern distributed systems. Although they can indeed create a very good image of the infrastructure, two problems subsist: first, they only supply an extended set of tools to react to already in-place incidents, and

secondly they lack "intelligence", if the system is not observable, i.e., if we cannot access metrics in precise spots that are hidden inside complex black boxes.

We need better tools and methods to understand the overall status of the system. Actual monitoring tools are built on the premise of observable systems, with agents, instrumentation, or some sort of module heartbeats. Furthermore, some resources may elude administrators control, such as objects located in third-party providers - e.g. content delivery networks. While some frameworks aim to gather information from the client-side point-of-view such as Pingdom (2017), Bucky (2017), or enterprise solutions, such as DynaTrace (2017), they are basically aimed at creating simple dashboards and insights of the platform to trigger alerts to administrators based on a set of custom rules. We want to go beyond this, and automatically infer service occupation, using as little data as possible from the systems, possibly because such data is unavailable. Additionally, perfect observability would require extensive instrumentation of source code with agents dedicated to software and hardware resources. Furthermore, a centralized point in the system would be required to gather, store, process and display data in dashboards.

In Chapter 4, we gathered clients' data — collected by JavaScript snippets —, to improve monitoring using the client-side point-of-view, as a complement to traditional monitoring applications. In Chapter 5, we used machine learning techniques to pinpoint two sources of system bottlenecks — CPU and network —, using only the raw data visible by clients. In this chapter we aim to understand if these methods can be extrapolated to more complex microservice scenarios.

Hence, the purpose of this chapter is to propose new techniques to monitor microservice architectures to ensure proper system observability. The remainder of this chapter is organized as follows. First, in Section 6.1 we study if it is possible to have a non-intrusive monitoring system, in a microservice ecosystem. This approach is based on the assumption that the system needs a gateway for service discovery and routing to other microservices. Secondly, we try to understand the feasibility to monitor a two component system resorting only to the total request time that the client receives. This study is presented in Section 6.2. In Section 6.3, we go a step further and improve our method, using more advanced techniques, such as neural networks to, not only, monitor but have a precise and fine-grained occupation level of each layer. Finally, Section 6.4 concludes this chapter.

6.1 Nonintrusive Monitoring of Microservice-based Systems

Microservices have become a trend in the development of distributed systems. This new paradigm evolved due to a number of factors. First, standard monolithic systems were difficult to maintain, deploy, develop and scale. Hence, there was the need to decompose these vertical systems in modules that are function-oriented and that could be handled separately, in terms of development and management. Secondly, microservice architectures are better suited for deployment and operation in Docker (Docker, 2018) or other containers. Finally, methodologies in product development, such as Agile or DevOps, with smaller independent teams, are more aligned with microservice architectures. Therefore, microservices have tremendous benefits in terms of development, operation, availability and scalability, and have thus become a standard for large-scale systems.

Since there is communication between several microservices, a powerful monitoring technique consists of instrumenting all the modules, creating traceability for a particular request. Tracing normally propagates a correlation identifier that can be used to determine the flow through several microservices. In other words, tracing allows system administrators to determine the entire workflow of applications. There are some frameworks that help to implement tracing, such as ZipKin (2018), OpenTracing (2019) or the one presented on Sigelman et al. (2010).

Despite the benefits, tracing brings two major drawbacks. First, all microservices must have tracing implemented and be responsible to send the data to a central point. This platform gathers, processes and aggregates the raw data. Therefore, developers have to focus, not only on the business algorithm, but also on monitoring and operation of the microservice. Secondly, the central point may be a system bottleneck, due to the large number of records. In fact, tracing systems normally purge older samples or save only a small percentage of data.

Bearing in mind the aforementioned solutions, one could think that administrators have all the tools to monitor systems. However, in reality, operators use a plethora of platforms and frameworks, some of them adopted from monolithic systems. These tools only give insights of what is happening in the system, and it is the administrators' responsibility to endure the hard task of navigating through several dashboards and

notifications to identify the problem. Hence, microservices have introduced a new paradigm to develop distributed systems, with well defined functions and boundaries, but still use monitoring techniques similar to what we could find in older architectures.

The monitoring tool we propose decouples monitoring functionalities from function-oriented microservices. It is a solution that is neither invasive, nor disruptive, as it requires no adaptations on the microservice level. As a consequence, it is a good solution to already implemented production systems. To achieve this, we used and adapted a gateway from Netflix, named Zuul (2018), to collect metrics from the requests made by the microservices. Based in metrics such as response time, origin and destination of the requests, we aggregated the raw data in a concise output with relevant information, such as average response time, topology and overall service characterization.

Our results show that we can obtain relevant and useful information to system administration, even though we use a non intrusive approach methodology. We do not need to instrument microservices or add agents to the infrastructure, resorting only to components already needed by microservice modules. Therefore, the solution presented is useful, viable — specially in very dynamic and elastic systems —, and aligned with microservice methodology.

The rest of the Section is organized as follows. Subsection 6.1.1 describes the problem we tackle and the method we used to solve it. Subsection 6.1.2 describes the experimental settings. In Subsection 6.1.3 we present and evaluate the meaning of the results, the strengths of this approach and its limitations.

6.1.1 Proposed Methodology

In this Chapter, we tackle the problem of monitoring microservice architectures. In vertical solutions, monitoring is easier, because the application does not change that much over time. Microservice systems evolved from new development methodologies, such as Agile or DevOps and new deployment techniques, such as containers. System monitoring did not follow this evolution and is still based on the applications and techniques for monolithic systems. In Subection 2.3.1, we discussed how major worldwide technological companies are struggling with this fact, being forced to create customized platforms for their needs. Indeed, monitoring is a complex and difficult problem in highly dynamic systems.

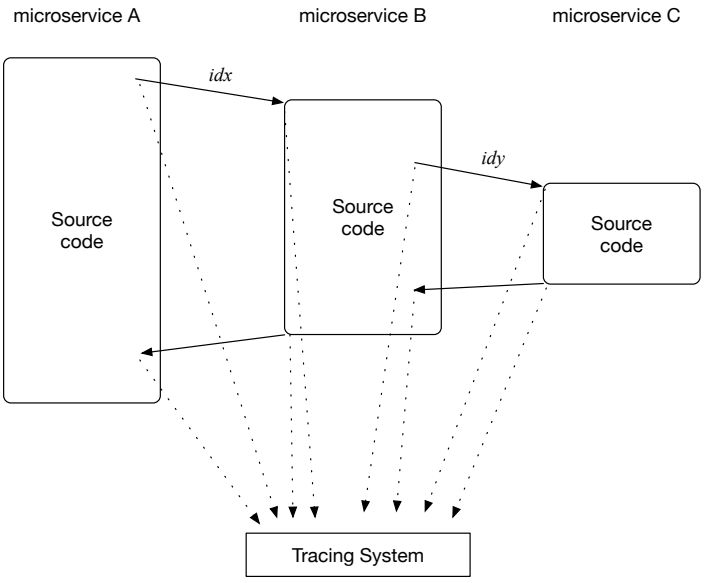


FIGURE 6.1. Tracing of microservices application (optimizations to reduce tracing messages omitted).

We analyzed the monitoring problem from a different perspective. A typical approach for monitoring would consist of instrumenting or adding agents in as much layers as possible, from hosts to middleware, up to the application layer. Refer to Figure 6.1, which shows a sequence of three microservices, where some function in A invokes a function in B, which invokes a function in C. To bring the information of the interdependent invocations to the monitoring system, messages must carry some identifier that allows their correlation, for example, an HTTP header with the same identifier ($idx = idy$). Unfortunately, this involves changing the source code of the application. While this technique creates several monitoring points, these are also additional points of failure and maintenance that couple monitoring with business logic. This goes against the microservice methodology, which follows the premise of function-oriented fine-grained modules. To eliminate the need for instrumentation, we follow a non-traditional approach. Knowing that microservices resort to a gateway to make service discovery and redirect requests, we added the capability to collect some monitoring metrics to this gateway. The idea is to make the gateway gather information, such as response time, IP and port of origin and destination, and the identification of the function that was invoked. This approach brings advantages, such as decoupling the monitoring system from the application without hindering system scalability, because the gateway and associated services are horizontally scalable.

In the next subsections, we describe our methodology in more detail. First, we present the architecture, and how we incorporate our solution in Netflix modules. Secondly, we go through the metrics we can collect in the gateway and discuss the information dashboards we can build with standard tools. Finally, we present how we implemented and distributed the tool.

Architecture

In microservice infrastructures, gateways are often used to solve the service discovery problem. Hence, it is very appealing to take advantage of this module to observe the system. We resorted to three Netflix components: Zuul, Ribbon and Eureka, responsible for gateway, load balancing and registry of scalable service, respectively. These services allow us to gather metrics data, being an advantage to monitor microservice systems. In Figure 6.2, we present the high level architecture view of our tool.

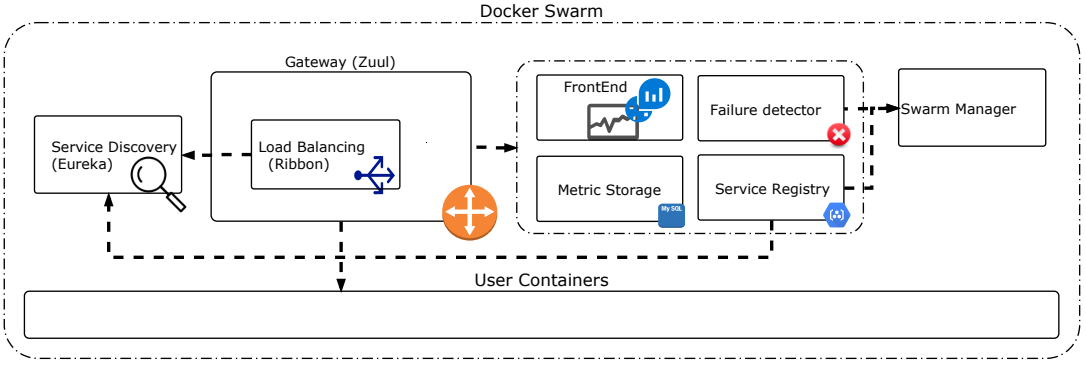


FIGURE 6.2. System components

We use four components that are aligned with microservice best practices, such as service discovery or containerization. First, the module "Metric Storage" gathers metrics collected by the customized Zuul application. These metrics are response time — in requests between services or directly from the client —, IP and port of origin and destination of the request, and function invoked on the destination microservice. This module, also acts as a backend to the "Frontend" module, where we display relevant information such as response times, topology and characterization of services.

The other two modules are associated with "Service Registry" and "Failure Detection". We focused on docker containers (Docker, 2018), because this is perhaps the most important framework in this technological field. By analyzing containerization, data

Table 6.1. Collected Metrics

Metric	Type
Start Time	Long
End Time	Long
Duration	Long
Origin IP	String
Origin Port	Integer
Destination Service	String
Destination Instance	String
Destination IP	String
Destination Function	String

description of containers and how the container manager works, it is possible to fully automate the process of registry and failure detection. This give us a tremendous advantage, since we do not need container instrumentation. In this case, our module is notified when a change occurs in the containers, such as creation, destruction or state change, through an agent associated to the container manager. Bearing in mind that we also observe HTTP results, it is also possible to implement a module responsible for failure detection. This module combines information from the HTTP results with container status, to add capabilities to our system of autonomous maintenance and recovery. When an instance fails, it is possible to remove or restart this instance, without needing administration supervision or microservice instrumentation. It is also relevant to mention that the components involved in monitoring are horizontally scalable, and therefore do not harm performance or availability of the application. Since we remove the instrumentation necessary of systems like the one of Figure 6.1, the processes responsible for extraction and processing of the metrics are outside of the critical path, and consequently do not create any sort of overhead.

Collected Metrics

We present in Table 6.1 the metrics collected in our "Metric Storage" module. For each request, regardless of the origin (either another microservice or a client), we save metrics associated to the origin and destination of the request.

Beside standard plots with averages and quartiles, e.g., as in box-plots, this raw information allows us to create high-level information about the system. For example, it is possible to dynamically extract topological information and characterize the level of

interaction among different microservices. Additionally, we can also calculate response times and load of each microservice, inferring maximum capacity and quality-of-service of each module, to ensure correct dimensioning.

For the frontend layer of our monitoring system, we used `Grafana` (Grafana, 2018), an open platform for analytics and monitoring, highly flexible and customizable. To display a few more complex plots, we used as a complement, graphics generated using the `R` language — a common standard in the academic field, for simulation and analysis, incorporating the output on `Grafana`.

Implementation

To validate our tool, we made a fully non-intrusive implementation for the `docker swarm` container management platform. The source-code and deployment instructions are available on GitHub (2018) as open-source. Additionally, it also contains the sample microservice application used in our experimental validation, of Section 6.1.2. The tool is easily deployable in a system with `Docker` and `Swarm Manager` installed. Since the monitoring tool needs an overlay network (DockerOverlay, 2018), the system must have this network created and configured, to ensure the correct operation of our approach.

Afterwards, the only parametrization needed is the name of the `overlay` network. The remaining parameters may be defined with the default values without loss of functionality. To use our monitoring solution, one could download the repository, define the `overlay` network in our configuration file and run the installation script that will automatically generate and deploy a `docker-compose` manifest file. The Service Registry component, described in Subsection 6.1.1, will subscribe to the `docker` event API and be notified of container creation, destruction and state change. As such, service registration on the gateway will be done automatically, requiring no collaboration from the services themselves. This is possible because each container already carries the relevant metadata, such as name and service port.

The monitoring solution includes the user-customizable frontend module, with `Grafana`. Furthermore, we developed and included a custom plug-in, written in `R`, to generate more complex visualizations, such as Chord diagrams (Gu et al., 2014). Our raw data storage module, uses influxDB and MySQL databases. In Table 6.2 we present the overall containers associated (and deployed) with the tool. Once installed in a `Docker`

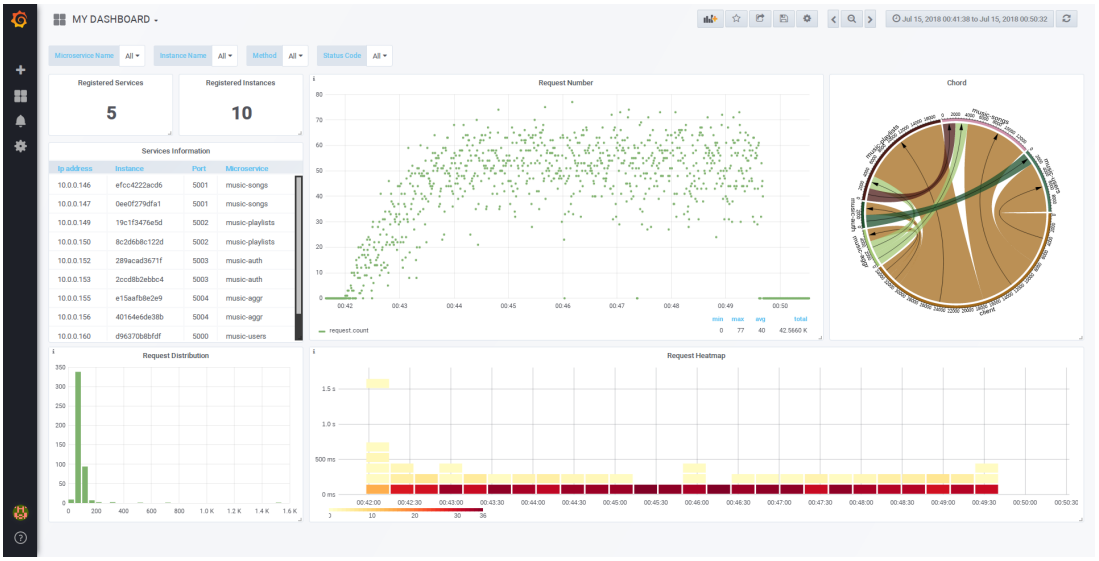


FIGURE 6.3. Sample user-customizable frontend.

`Swarm` container manager, all other applications deployed on it will automatically use our gateway for service discovery and monitoring, as long as they are in the same `overlay` network.

Figure 6.3 shows an example dashboard, extracted during the experimental stage. In this case, we show the charts described in Section 6.2.3, such as histograms and chord diagrams.

TABLE 6.2. Tool Containers

Container	Description
Eureka	Service Discovery
Zuul	Gateway
Service Registry	Manages containers life-cycle, in association with Eureka
InfluxDB	Time-series scalable DB
MySql	DB
Grafana	Frontend
Chord Plugin	Generates chord visualizations

6.1.2 Experimental Setup

In this subsection, we present the experimental setup used, the changes made to Netflix modules, and our microservice application. First, concerning the infrastructural modules, used by the test application, we rely on modules that are stable. To do load

Table 6.3. Microservice and functions available

microservice	functionality	request type	description
Authentication_MS	/	GET	System Healthcheck
	/login	POST	Validate user credentials and create token
Users_MS	/	GET	System Healthcheck
	/login	POST	Validate user credentials
	/users	GET	Get user info
	/users	POST	Create user
	/users/{id}	DELETE	Remove user
	/users/{id}	PUT	Update user
Playlists_MS	/	GET	System Healthcheck
	/playlists	GET	Get playlists associated to a user
	/playlists	POST	Create playlist
	/playlists/{id}	GET	Get playlist
	/playlists/{id}	PUT	Update playlist
	/playlists/songs/{id}	DELETE	Remove a specific music from a playlist
	/playlists/songs/{id}	GET	Get music info associated to a playlist
	/playlists/songs/{id}	POST	Add music to playlist
Songs_MS	/	GET	System Healthcheck
	/songs	GET	Get music info
	/songs	POST	Create music info
	/songs/convert/{id}	GET	Convert music from `mp3` extension to `wav`
	/songs/criteria	GET	Get music list based on some criteria
	/songs/{id}	DELETE	Remove music
	/songs/{id}	PUT	Update music
Aggregator_MS	/	GET	System Healthcheck
	/playlists/songs/{id}	GET	Get all music info associated to a playlist

balancing, we used Ribbon. This module give us several advantages, such as the available load balancing algorithms, the use of REST interfaces, but most importantly, an off-the-shelf integration with the remaining support modules from Netflix. Hence, integration with the discovery and registry module — Eureka —, is made, allowing a more agile instantiation and implementation of our methodology. Aligned with Ribbon and Eureka, we also used the Netflix gateway — Zuul —, that uses Ribbon internally. Zuul gets service location through a query to Eureka, and then routes requests to the correct service. Since requests have to pass through Zuul, this module allows us to have a clear vision regarding traffic between microservices and to gather monitoring information to a central point.

The other component of our experimental setup is the application that allows us to test the monitoring method. The application that we implemented is related to music and has five microservices with well defined functions. The application allows its clients to manage users, playlists and songs. On Table 6.3, we identify the overall endpoints associated to each microservice, respective invocation methods and a brief description.

Since we wanted to collect raw information about the requests, but without instrumenting microservices, we changed the Zuul source code to register information concerning

origin and destination of each request. We save the following information: microservice that made the request, start time, end time, IP and Port of the request origin, microservice instance that processed the request, and function that was invoked. With this information, we were able to extract relevant information about the system, such as topology or average response times, decomposed by microservice and function. As mentioned, we did not need any kind of instrumentation on the source code of the application (i.e., we only changed the infrastructure). The raw data is then pre-processed and redirected to a MySQL database that is part of our "system metric" module.

The software was installed in a virtual environment running Ubuntu 16.04. The virtual machine had 8 vCores, with 22 GiB of RAM. All components were installed with standard parametrization, except the Zuul parameter `sensitive-headers`. This configuration allows us to propagate the authentication token through all microservices without any kind of manipulation from the gateway.

To simulate load on the system, we used Apache JMeter (JMeter, 2013). We configured this load tool with 10 threads, and a launching period of 120 seconds. Each thread ran during 10 minutes with the following loop:

1. Create User;
2. Authenticate;
3. Get user;
4. Update user;
5. Add song;
6. Get song;
7. Update song;
8. Convert song;
9. Add playlist;
10. Get playlist;
11. Update playlist;
12. Add music to playlist;

Table 6.4. Software used

Component	Observations	Version
Zuul	Gateway	1.4.4
Eureka	Service discovery	1.4.4
Ribbon	Load balancer	1.4.4
MariaDB	DB used by microservices	10.3.7
MySQL	DB used by the frontend	8.0
JMeter	Load testing tool	4.0

13. Get music from playlist;
14. Get all musics from playlist;
15. Delete music from playlist;
16. Delete playlist;
17. Delete song;
18. Delete user.

In Table 6.4 we present the open-source components used in the experiment and respective versions.

Our ultimate goal with this experiment is very simple: understand the limits, benefits and disadvantages of our "black-box" nonintrusive monitoring tool. Figure 6.4 summarizes the entire system, with application, infrastructure, including the monitoring tool and a load generator.

6.1.3 Results

In this section, we present the results of gathering monitoring data from the API gateway. This technique allows us to extract raw data from microservice interactions and, therefore, create a set of metrics and charts with relevant information for administrators, without the need of instrumentation or agents at hosts level. In this Section, we present 5 visualizations that combined, give us a clear vision of the system.

Concerning the frontend application, we divided visualization into 3 distinct charts. First, we need to understand which microservices have a higher variance in the response

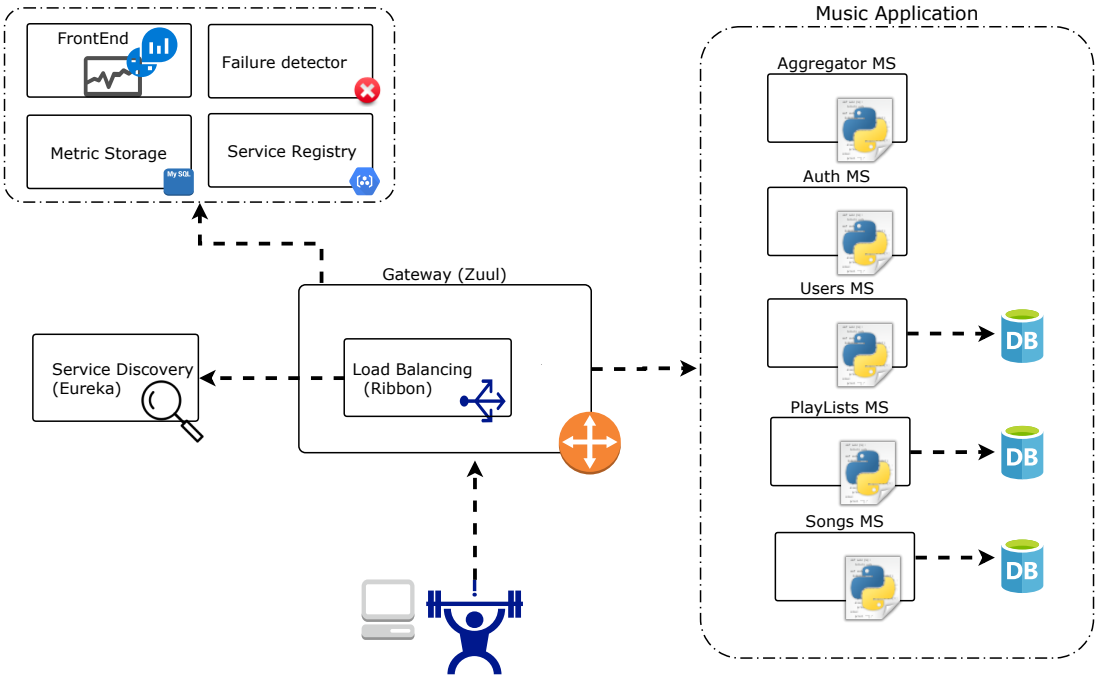


FIGURE 6.4. System architecture

time. To get this data — see Figure 6.5 —, we opted for a boxplot chart. This kind of graphic allows us to have compressed information in only one visualization. Figure 6.5 was created based on data extracted from our MySQL database. It is relevant to mention that although we are presenting response time distributions of microservices, it is possible for the user to drill-down, and visualize the same distribution by destination function inside each microservice.

Regarding dependencies between microservices, we resort to a graph. This representation allows us to present topology and dependencies between modules. In Figure 6.6, we can easily see the relations between the different microservices, and the direct accesses from clients.

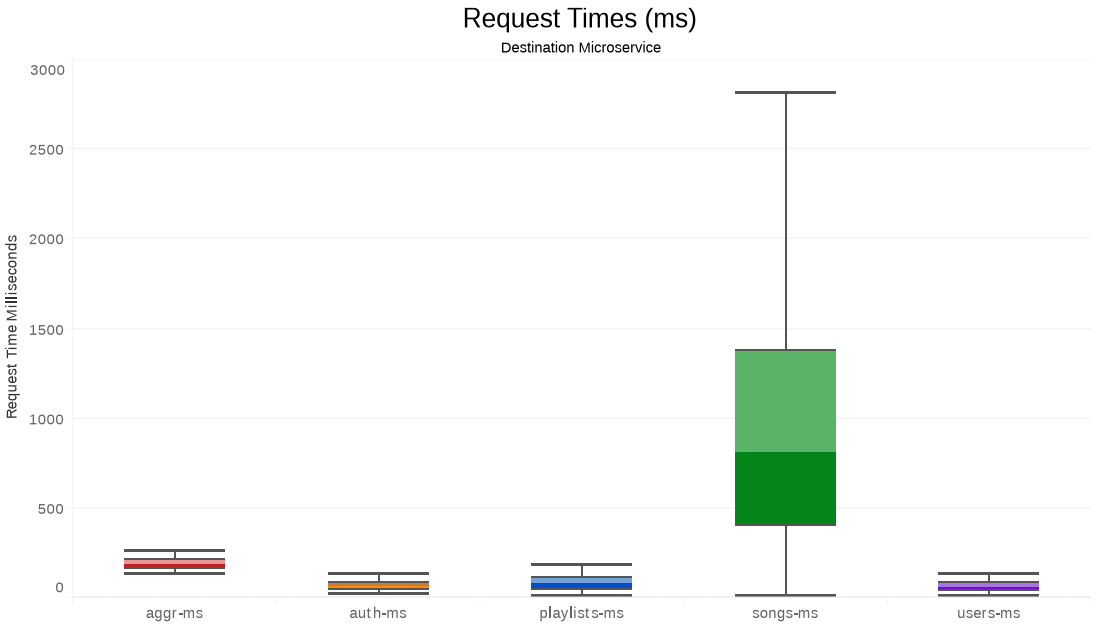


FIGURE 6.5. Boxplot of response time by microservice

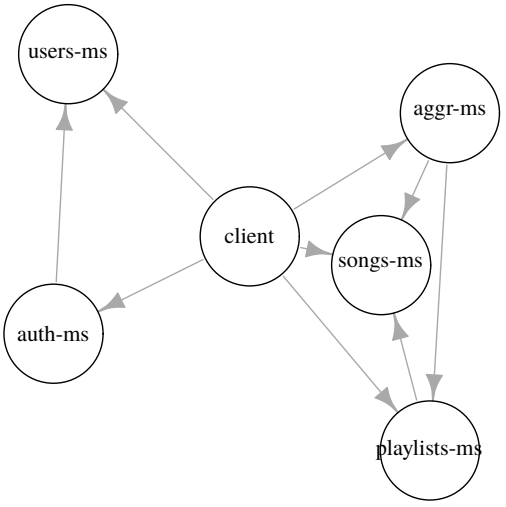


FIGURE 6.6. Application graph

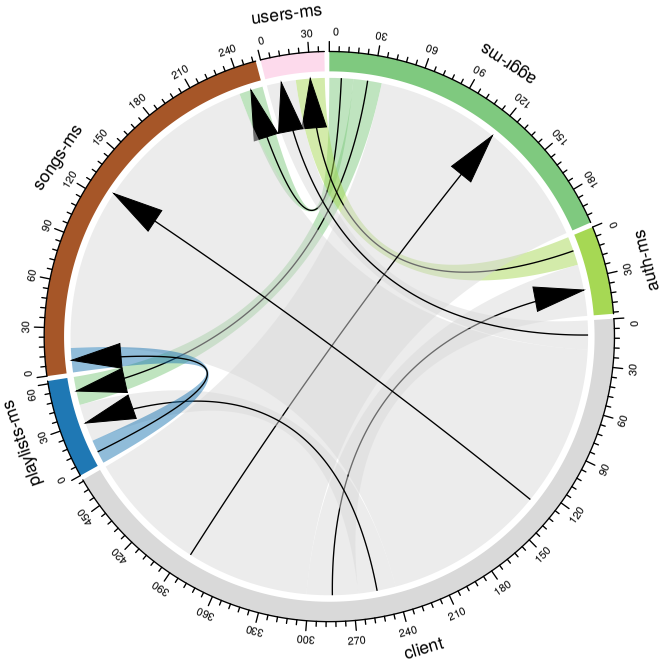


FIGURE 6.8. Chord diagrams — Latency

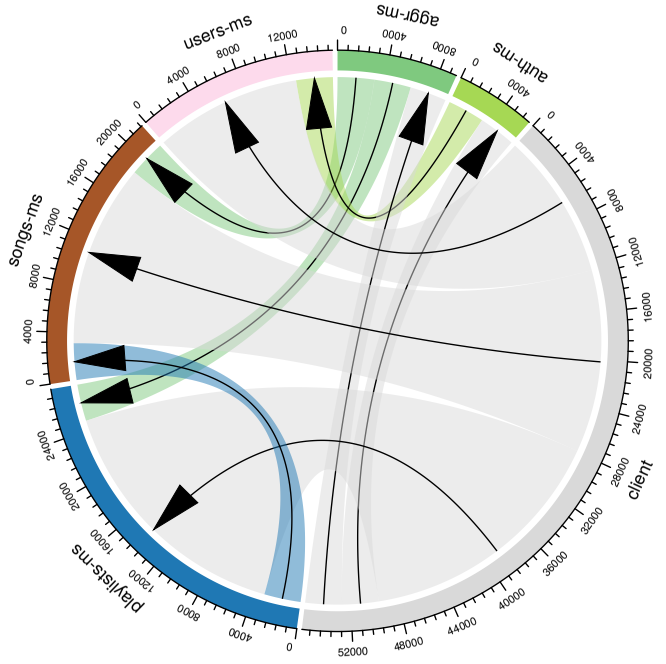


FIGURE 6.7. Chord diagrams — Frequency

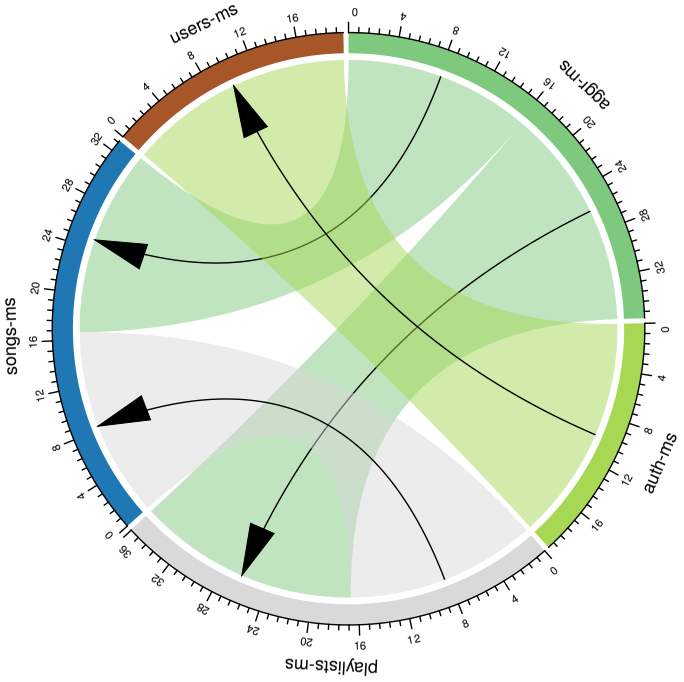


Figure 6.9. Chord diagrams — Latency without Client

Having response times distribution in boxplot charts for each microservice (and function) and dependencies between microservices in graph visualization, there is still a crucial aspect missing, to understand the health of the system: the importance of each microservice and function in the system. To achieve this, we resort to chord diagrams, based on the work of Gu et al. (2014). This kind of graphic allows us to see more complex relations between entities. Graph nodes are arranged along a circle, and the importance of their interactions is proportional to the width of the connecting arcs. We use arrows to provide information of which side receives the call, and colors to simplify interpretation. For instance, in Figure 6.7, we can see the number of requests, and in Figure 6.8 latency. In a very large system, a chord graph comprising everything would probably be difficult to read. Hence, to improve diagram interpretation, administrators can select which microservices to display, as seen in Figure 6.3. For instance, in Figure 6.9 we see latency, without requests made by the clients, since these requests can have a huge impact on the graph and make other interactions less visible.

Taking into consideration Figure 6.7, an administrator can verify that microservice `playlist-ms` was the origin or destination of around $28,000$ requests. From these, around $3,000$ were requests from `playlist-ms` to `songs-ms`, $21,000$ requests were made directly by clients and around $3,000$ from the `aggr-ms` microservice. This way, we have a vision of the relevance of the `playlist-ms` microservice in the overall system. Additionally, the same analysis could be made for latency. The box-plots, combined with the dependency graph and the chord diagram, give us a good idea of the system capacity, module importance and response time distribution by microservice or function.

An interaction of a system administrator with the monitoring system could go this way: the administrator would first look to Figure 6.5. This box-plot, provides a clear understanding of what services have higher response times. It is easily observable that the service `songs-ms` has the highest response times of all modules. The second module with higher response times is `aggr-ms`. Nevertheless, an administrator would try to understand the importance that the `songs-ms` microservice has in the system. Although it has a higher response time compared with other microservices, an administrator should look to the remaining Figures. With Figure 6.6, he can see that `songs-ms` receives invocations directly from the client, `aggr-ms` and `playlists-ms`, so, there is a large system dependency on the `songs-ms` microservice. Furthermore, we notice that the latency of `songs-ms` depends on who is invoking it, presenting a much higher latency for client-initiated invocations. This would show that either they are invoking

different functions or there is some anomaly. An administrator - using our application - can then drill down and see granular invocation data for further analysis.

The last information that an administrator needs is the number of requests and latency in the calls between microservices. Even though `songs-ms` has a high response time, and is a key module in terms of dependencies, we need to understand if the number of requests that go through `songs-ms` is relevant in the overall number of requests processed by the system. To have this information, we can look to Figure 6.7 and 6.9. We can see that `songs-ms` is an important destination of requests, specially from client and `aggr-ms`. In fact, looking to Figure 6.9, we can see that `aggr-ms` dependencies (`songs-ms` and `playlists-ms`) have low latencies, so they are not a bottleneck.

Given that Figure 6.7 shows the `aggr-ms` service makes roughly twice as many invocations as it gets, it leads to the conclusion that latency is a result of multiple requests, possibly serially or with low parallelism. Therefore, an administrator with these visualizations would have two possible solutions: drill-down the boxplot of `songs-ms` by function, to check if there is any offending function, and/or try to improve the way how `aggr-ms` invokes dependencies. It is important to remember that all this information is achievable with no instrumentation or agents in the infrastructure.

When we compare our approach to current monitoring tools for microservices, we can see some benefits, as well as disadvantages. One of the disadvantages, is related to tracing. We do not have the granularity that tracing offers, to understand the workflow of specific requests. Hence, we may miss some information concerning causality between microservices. Nevertheless, if we have a widespread distribution of requests, we can still estimate the workflow. On the other hand, our module is far less intrusive, as it does not have the overhead to develop instrumentation or deploy agents in the system. Additionally, our solution could be implemented in legacy systems in a very agile way, something that is probably beyond reach of tracing-based solutions.

6.2 Black-box monitoring techniques for multi-component services

Actual monitoring tools are built on the premise of observable systems, with agents, instrumentation, or some sort of module heartbeats. Furthermore, some resources may elude administrators control, such as objects located in third-party providers — e.g. content delivery networks. While some frameworks aim to gather information from the client-side point-of-view, they are basically aimed at creating simple dashboards and insights of the platform to trigger alerts to administrators based on a set of custom rules. We want to go beyond this and automatically infer service occupation, using as little data as possible from the systems, because such data is unavailable.

In this section we further extend our previous work — specially from Chapter 5, where internal observability may not apply —, with two generic layers that simulate two components of a system. This system could be a microservice that is complex and contains several queues inside, or a couple of microservices, one after the other, if intermediate timings of requests are not available, i.e., if we cannot relate the times at which the first and second services interact in response to an initial request to the first service. To get insights about the two-layer system, we only used the total time seen by the caller. This time aggregates the overall invocation time that sums up the two services. Based on this time, our goal is to determine the overall occupation of both services. We created a simulation using a two queue system, with one goal: infer the service occupation of each layer resorting to the collection of the response times. We then applied two methodologies to extract occupation of each layer (component) and extract error metrics: first, we used a similar approach as in Chapter 5, with supervised machine learning algorithms to identify service capability. Secondly, we used another method that tries to decompose the overall response time into the components' times to identify the occupation of each.

Our results demonstrate that a methodology that extrapolates information about a non-observable system of two layers is feasible and can improve performance monitoring. There is no overhead associated with these methodologies, and both methods — machine learning and division of the signal —, present advantages and complementary properties. These methods can improve monitoring when instrumentation or observability is difficult or unable to be achieved by administrators of the system.

The rest of the section is organized as follows. Subsection 6.2.1 describes the monitoring problem we tackle in this section as well as the methods we propose. Subsection 6.2.2 describes our experiment. In Subsection 6.2.3 we present and evaluate the meaning of the results, discussing the strengths of both approaches and its limitations.

6.2.1 Proposed Methodology

In this Subsection, we describe the problems and challenges associated with observability of a system and the definition of the metrics used.

With the increase of complexity of applications, and the need to reduce time-to-market, monitoring becomes more important than ever to ensure that component failures and bottlenecks do not affect user's quality-of-experience. Traditional monitoring approaches use a large set of tools and methods, such as tracing, logging, or correlation identifiers between services or system tools, such as `Nagios` or `Zabbix`. Although functional, these approaches require the system to be prepared to give away information about its current internal status. In legacy systems, or systems without some form of instrumentation or agents, this might be difficult. Additionally, the effort to create logging or tracing in a production system may be too high.

We present an approach that neither requires instrumentation, nor disperse logging tools over the system. In Chapter 5, we followed a similar approach to pinpoint bottlenecks in two distinct layers: external network and internal system. In this Section, we further evolve this methodology, by modelling a system in two components, and determining each component's occupation, without using instrumentation in the middle. Understanding each component's occupation may give a huge advantage for administrators, whenever monitoring is impracticable, either because it is too costly or because the internal details of the system are unknown, e.g., because the source code is unavailable or too complex. The only metric that we used was the total time observed by the entity that invoked the system. The total time represents the time that the request spent in the two components, from the beginning of the request until the end. In fact, our method of decomposing a system of two layers is very generic and can be used in different scenarios, like a system with a database and network, or two modules interconnected, or even a chain of microservices.

In the next subsections, we present our methods to evaluate a "black-box" approach of a two-layer system. First, we present an approach based on a machine learning supervised algorithm. Then, we describe our method based on the split of the signal. Both methods use the data collected from our experiment.

Machine Learning Algorithm

We followed a machine learning approach to predict each layer's occupation from the experiment input data. We used a regression model — instead of a classifier — since our output is a continuous value, instead of a set of class labels (Sen and Srivastava, 2012). We created a regression model for each layer, as illustrated in Figure 6.10.

In addition to the regression models, we also wanted to evaluate the possibility of having decision tree classifiers. This is appealing, because decision tree models are highly interpretable and, therefore, an advantage to system administrators and operators.

We provide to our algorithm $3,000$ lines to each model. Each line has $2,000$ requests made to the system, meaning that our classifier has $2,000$ features – e.g. inputs. There is a 2001^{st} value in each line, that corresponds to the layer 1 or layer 2 occupation (i.e., the real occupation of each layer). This value is mandatory, since we are training a supervised machine learning algorithm. However, this output is not available in the test cases, because we want to predict the occupation level, as illustrated in Figure 6.10.

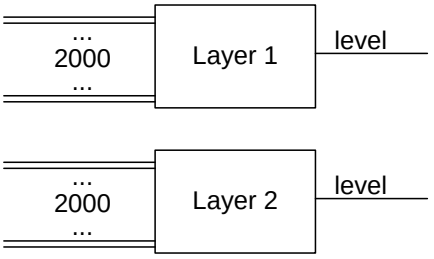


FIGURE 6.10. Machine learning regression models for the two layers

Among the wide range of supervision machine learning methods available in the literature (Witten et al., 2016), we focused on the two previously used models: Simple Linear Regression (SLR) and Support Vector Machine (SVM).

The literature (Witten et al., 2016) also includes a wide range of decision tree models, but since we wanted to predict a real value, we opted for a decision tree regression

model. This kind of model allows us to have more insight about how it works, thus making it more interpretable, a great advantage for system operators.

To run the algorithms under our training set, we used the `scikit` learn framework (SciKit, 2018). This framework, written in python, is a common standard for data scientists and people that want to generate models based on input data. For SVR, we used RBF and a polynomial kernel to evaluate the differences, with normalized data (input and output), in the $[0, 1]$ interval, to achieve better predictions. The decision tree regression model was evaluated with default parameters and data. Concerning the evaluation, all experiments were performed using a 10-fold cross validation with 20 repetitions.

Exponential Variable Sum Algorithm

The problem of determining the occupation of two sequential components, which we call layers, be they resources or distributed services, observing only global response times, can be modeled as determining the two tandem random processes responsible for generating an observed request time distribution. In other words, given a statistical distribution, since we know *a priori* that the distribution is the sum of two factors, what we want is to separate these factors. Particularly, in this Section we explore this approach under the assumptions that the service times are approximately exponentially distributed and the service is under relatively low load (occupation $\rho < 0.3$), at the time of observation. As long as the load and parallelism of each layer is known, the results can be used to extrapolate the total capacity of each layer. It should be noted however that this does not reduce the generality of the solution. The same approach could be applied under different distributions and loads, as long as some conditions are met (the resulting distribution having enough information to extrapolate the original processes). Furthermore, the two processes under study do not necessarily have to be tandem, they may, for example, be interleaved, as long as the sum of the components follows the previously described assumptions. A good example of where interleaving is the more adequate model, would be resource time-slicing.

To define it more precisely, given $f(x)$ and $g(y)$, which are the density functions of the service times of two layers, known to be approximately exponentially distributed, of rates λ and μ, we know that the total service time will be described by the sum

of two random variables, $h(z)$, obtained from the convolution of the two, as shown in Equation 6.1.

$$h(z) = (f * g)(z) = \int_{-\infty}^{+\infty} f(z-y)g(y)\ dy \tag{6.1}$$

By substituting for the particular case of the exponential distribution, we get the result for $h(z)$ in Equation 6.2, which we integrate in Equation 6.3 to obtain the cumulative distribution function $H(z)$.

$$\begin{aligned} h(\lambda, \mu, z) &= \int_0^z \lambda e^{-\lambda t} \mu e^{-\mu t}\ dt \\ &= \lambda\mu \left(\frac{1}{e^{z\lambda}\mu - \lambda e^{z\lambda}} - \frac{e^{-z\mu}}{\mu - \lambda} \right) \end{aligned} \tag{6.2}$$

$$\begin{aligned} H(\lambda, \mu, z) &= \int_0^z h(t)\ dt \\ &= \int_0^z \lambda\mu \left(\frac{1}{e^{t\lambda}\mu - \lambda e^{t\lambda}} - \frac{e^{-t\mu}}{\mu - \lambda} \right)\ dt \\ &= \frac{\lambda\mu e^{-z\mu} \left(\lambda e^{z\lambda} + \left(e^{z\lambda} - 1 \right) \mu - \lambda e^{z\lambda} \right) e^{z\mu} \right)}{\lambda e^{z\lambda}\mu^2 - \lambda^2 e^{z\lambda}\mu} \end{aligned} \tag{6.3}$$

Armed with the cumulative distribution function, and the empirical cumulative distribution function, $ecdf(x)$, from the observed sample S of total service time, we made an `R` script to determine the variables λ and μ, by solving the optimization model in Equation 6.4. The objective is selecting the variables to minimize the mean square error between the empirical cumulative function of the sample and $H(z)$, effectively fitting it to the data.

$$
\begin{aligned}
\underset{\widehat{\lambda},\widehat{\mu}}{\text{Minimize}} \quad & \frac{1}{n}\sum_{i=1}^{n}(\eta_i - \epsilon_i)^2 \\
\text{subject to} \quad & \tau = max(S_i),\ \forall i \\
& \delta = \frac{\tau}{1000} \\
& \eta_i = H(\widehat{\lambda}, \widehat{\mu}, i\delta),\ i = 1, ..., 1000 \\
& \epsilon_i = ecdf(i\delta),\ i = 1, ..., 1000 \\
& \widehat{\lambda} > 0,\ \widehat{\lambda} \in \mathbb{R} \\
& \widehat{\mu} > 0,\ \widehat{\mu} \in \mathbb{R}
\end{aligned}
\tag{6.4}
$$

At the end of this step, we are left with a $\widehat{\lambda}$ and $\widehat{\mu}$, which are estimators of the service rate of Layers 1 and 2 respectively. Given the load under which the measurements were taken, which we denote as W, as well as the parallelism levels, denoted c_1 and c_2, the occupation of each layer, ρ_1 and ρ_2 can be determined as shown in Equation 6.5 where S denotes the service rate of a layer ($\widehat{\lambda}$ or $\widehat{\mu}$).

$$\rho = \frac{W}{c \cdot S} \tag{6.5}$$

The assumption that the parallelism level c is known might seem limiting. However, in practice, this parameter is easy to monitor using classic tools, and even if not directly measurable, heuristics can be used to estimate it. One such example is exploiting the relationships between maximum throughput T and parallelism given by Equation 6.6.

$$T = c \cdot S \tag{6.6}$$

6.2.2 Evaluation

We validated and benchmarked the two methods with total service time samples extracted from a simulated service with two layers. The described system was simulated as two sequential single server queue systems ($M/M/1$) in `R` using the `qcomputer` (Ebert et al., 2017) package (shown in Figure 6.11).

To simulate different occupation rates, we used a fixed global request arrival rate W of 30 requests per unit of time and varied the service rates S of each layer according

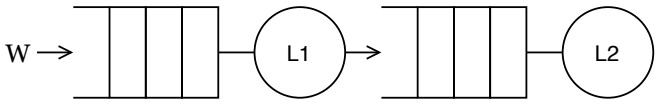


FIGURE 6.11. Two sequential single server queue systems.

to Equation 6.5. A set of 30 samples of 2000 observations each was then obtained for all the permutations of the two layers with occupations in the range $[0.1, 0.9]$ in 0.1 increments (e.g. $(0.1, 0.1), (0.1, 0.2), ...$). The data was saved in a comma-separated file (`CSV`), each sample containing 2000 features and the 2 targets, in this case the occupation parameters.

Using cross-validation we trained the machine learning algorithms and evaluated the quality of the predictions. For the exponential variable sum algorithm we used the proposed optimization approach to predict the targets for each sample.

To compare the two general approaches we calculated the Mean Square Error (MSE) for each layer grouped by occupation range in 0.1 increments. Additionally, as we tried multiple machine learning approaches, these metrics were also used to pick the best algorithm.

6.2.3 Results

As we referred in Subsection 6.2.1, we collected a set of 2000 total response times for the clients invocations. We analyze the results obtained with the machine learning algorithms and also with the exponential decomposition.

TABLE 6.5. Regression model results for Layer 1 and Layer 2 occupation

	Layer 1	Layer 2
Method	MSE	MSE
Decision Tree	0.12 ± 0.08	0.09 ± 0.06
SLR	0.81 ± 0.61	0.93 ± 0.44
SVR	0.05 ± 0.04	0.05 ± 0.03

The results obtained for the Layer 1 and 2 occupation models are summarized in Tables 6.5, 6.6 and 6.7. We report average results for the mean square error (MSE), between the predicted and actual values. Since we performed 20 repetitions of 10-fold cross-validation, we present average and standard deviation results.

TABLE 6.6. Machine Learning Decomposition

Range	Layer 1 - MSE	Layer 2 - MSE
0.1	0.12	0.12
0.2	0.07	0.08
0.3	0.04	0.03
0.4	0.03	0.02
0.5	0.02	0.02
0.6	0.03	0.02
0.7	0.03	0.03
0.8	0.03	0.03
0.9	0.03	0.03

TABLE 6.7. Exponential Decomposition

Range	Layer 1 - MSE	Layer 2 - MSE
0.1	0.08	0.08
0.2	0.12	0.26
0.3	0.25	0.51
0.4	0.40	0.68
0.5	0.93	0.74
0.6	1.79	1.76
0.7	3.76	4.95
0.8	11.38	11.43
0.9	66.48	74.87

Table 6.5 presents the results for three distinct classifiers: decision tree regressor, SLR and SVR algorithms. As expected, SVR outperforms the others algorithms, since it uses a non-linear kernel that fits better the raw data that is handled in this use-case scenario.

The SVR algorithm attained an average of MSE 0.05 (with 0.04 standard deviation) for Layer 1 and 0.05 (with 0.03 standard deviation) for Layer 2. On a $[0, 1]$ range this is a good result.

SVR algorithm outperforms SLR and Decision tree model, for both layers. Therefore, we compared SVR algorithm with a non-linear kernel with the exponential decomposition method. To do this, we trained the same SVR algorithm for both Layer 1 and Layer 2. We submitted to the fitted model only values for a specific range (e.g. $0.1, 0.2$) and registered the MSE for the predicted values. In Table 6.6 we evaluate the machine learning accuracy for distinct Layer levels. We can observe that for low occupied layers the algorithm has a worse MSE, compared to a more occupied Layer.

In Table 6.7 we show results for exponential decomposition method. We evaluated the MSE for each layer 1 and 2 occupation level. One can notice that this algorithm outperforms SVR on lower occupation levels. However, for high occupation levels, the exponential algorithm, does not do a good predict on, being outperformed by the machine learning approach.

Another difference is that the Machine Learning algorithm requires supervised training and the exponential decomposition does not. On the other hand, the latter can only be used when the system load during the sampling period is known. Given their complementary properties and ranges of effectiveness, they are obvious candidates for hybrid application.

6.3 Towards Occupation Inference in Non-Instrumented Services

Inferring occupation of individual components without help from instrumentation or external agents can bring concrete benefits for the observability of the system. A clear use case would be extracting information at sub-instrumentation granularity as well as improving the visibility over legacy parts of a system that are not or cannot be adequately instrumented. Given this goal, in this section, we aim to determine whether we can perform such separation using a neural network. It has been shown that any system can be decomposed into an arbitrary number of queues as a result of the properties of sums of Markovian processes (Horváth and Telek, 2002). Furthermore, request rates for modern use cases are known to be well modeled by Poisson processes for large numbers of clients (Shahin, 2017).

To evaluate this possibility, we created a laboratory experiment, where we use a system with two sequential $M/M/1$ queues. We ran a set of several combinations for layer occupations, from lightly occupied to heavily busy. For each combination, we collected the response time for a batch of client requests. Using this data, we trained a neural network, which we eventually set to three hidden layers of 100 neurons each, with two outputs representing the level of occupation of each component of the system. The point is to understand if the neural network could predict the occupation of each layer without expert understanding of the system. We evaluated our trained neural network against a baseline optimization method. To extract the occupation, this method explicitly uses

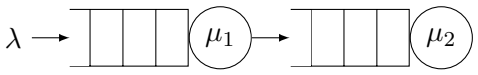


FIGURE 6.12. Tandem $M/M/1$ queues.

underlying knowledge of the system, to fit the observed data with a tandem queue model.

Our experiments show that the neural network can accurately infer the occupation of each layer. With the exception of the case where one of the layers is extremely busy and dominates the response time of the system, both methods, the neural network and the tandem queue model, achieve satisfactory results. These results show the feasibility of using machine learning to do black-box monitoring of systems. Furthermore, it reinforces that black-box observations contain enough information to reason about the structure of the system that generated it. The assumption that a generic system can be decomposed into queues with exponential service times is supported by existing research (Shahin, 2017), theoretical results (Philippe, 1998) and from our previous Section 6.2. The intuition is that a computer, as a discrete system, can be seen as an arrangement of buffers (queues) representing the resources.

The rest of the section is organized as follows. Subsection 6.3.1 describes the methods we used for the problem we tackle in this section. Subsection 6.3.2 describes the settings for our experiment. In Subsection 6.3.3 we show and evaluate the meaning of the results, the strengths of this approach and its limitations for both methods.

6.3.1 Proposed Methodology

In this Subsection, we describe our approach to infer the occupation of each component in the system of two queues depicted in Figure 6.12.

Using a sample of response times, we determine the occupation of each component (which we refer to as layer), both with model fitting and a deep neural network. While the model fitting algorithm explores the underlying structure of the system and serves to prove that it is indeed possible to do the black-box prediction, our goal is to create a neural network that precludes the need for any assumption or knowledge about the system.

Tandem Queue Model Fitting

To determine the occupation of each layer of a tandem, two-component system, we first need to model their response to load. In a generic way, this response function is defined by the time it takes to service each request and the level of parallelism. Since we know the response time is the sum of the two random processes representing the service time portions happening at each layer, as a naive approach, we could attempt to model the data, by fitting the sum of two random variables (exponential for example). This would shed light on the time spent on each layer and indirectly their capacity. However, this approach fails to capture the variation in response time in response to load/occupation, as a result of not considering the time spent waiting for the service.

Queuing theory gives us a theoretical framework to predict response time variation in function of occupation. As such, we modelled the system as a network of two tandem single-server queues ($M/M/1$), shown in Figure 6.12. This model assumes Markovian properties for both inter-arrival times as well as service times. It further assumes no parallelism. We forewent more general models, for which approximate or numerical solutions are known, as the objective is not generally solving the problem, but to prove its feasibility and establish a performance baseline for a relatively simple case. Each queue is defined by its arrival rate λ and service time μ, and the occupation ρ is $\frac{\lambda}{\mu}$. The probability density function (PDF) for the response time distribution is given in Equation 6.7 by $r(\lambda, \mu, x)$.

$$r(\lambda, \mu, x) = \begin{cases} (1-\rho)\mu e^{-x(1-\rho)\mu} & x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

The model resulting of the composition of two tandem $M/M/1$ queues is defined by the arrival rate (λ), and the service rate of each queue (μ_1, μ_2), and has a response time distribution given in Equation 6.8 as $t(\lambda, \mu_1, \mu_2, x)$ and a respective cumulative distribution function (CDF) given in Equation 6.9 as $T(\lambda, \mu_1, \mu_2, x)$. Note that due to space restrictions, the numerator on $\tau(\lambda, \mu_1, \mu_2, x)$ is split in two lines.

$$
\begin{aligned}
t(\lambda, \mu_1, \mu_2, x) &= (r * r)(x) \\
&= \int_0^x r(\lambda, \mu_1, z) r(\lambda, \mu_2, x - z)\ dz \\
&= \mu_1 \mu_2 \left(1 - \frac{\lambda}{\mu_1}\right)\left(1 - \frac{\lambda}{\mu_2}\right)\left(\frac{e^{x\lambda - \mu_1 x}}{\mu_2 - \mu_1} - \frac{e^{x\lambda - \mu_2 x}}{\mu_2 - \mu_1}\right)
\end{aligned}
\tag{6.8}
$$

$$
\begin{aligned}
T(\lambda, \mu_1, \mu_2, x) &= \int_0^x t(\lambda, \mu_1, \mu_2, z)\ dz \\
&= \begin{cases} \left(1 - \frac{\lambda}{\mu_1}\right)^2 {\mu_1}^2 \left(\frac{1 - ((\mu_1 - \lambda)x + 1)e^{\lambda x - \mu_1 x}}{{\mu_1}^2 - 2\lambda\mu_1 + \lambda^2}\right) & \mu_1 = \mu_2 \\ \tau(\lambda, \mu_1, \mu_2, x) & \text{otherwise} \end{cases}
\end{aligned}
\tag{6.9}
$$

where,

$$
\tau(\lambda, \mu_1, \mu_2, x) = \frac{\left[\begin{array}{l} \left(1 - \frac{\lambda}{\mu_1}\right)\mu_1\left(1 - \frac{\lambda}{\mu_2}\right)\mu_2 e^{-\mu_2 x - \mu_1 x} \\ \quad \left((\mu_1 - \lambda)\, e^{\mu_1 x + \lambda x} + \left((\mu_2 - \mu_1)e^{\mu_1 x} + (\lambda - \mu_2)\, e^{\lambda x}\right) e^{\mu_2 x}\right) \end{array}\right]}{(\mu_1 - \lambda){\mu_2}^2 + (\lambda^2 - {\mu_1}^2)\mu_2 + \lambda{\mu_1}^2 - \lambda^2\mu_1}
$$

As we want to find the occupations ρ of each layer, and $\rho \in]0, 1[$, we rewrite the model in terms of occupation, as shown in Equation 6.10.

$$
\Theta(\lambda, \rho_1, \rho_2, x) = T(\lambda, \frac{\mu_1}{\lambda}, \frac{\mu_2}{\lambda}, x) \tag{6.10}
$$

To fit it to the data, we use optimization to find the values ρ_1, ρ_2 that minimize the mean square error (MSE) between Θ and the empirical cumulative distribution function ($eCDF$) of the samples. We assume λ is known for the time interval when the samples where taken. Figure 6.13 shows an example of how the model Θ fits the $eCDF$ after the optmization step. This particular sample was generated from a system with a $(0.7, 0.2)$ occupation, and the optimization determined parameters $(0.16, 0.73)$.

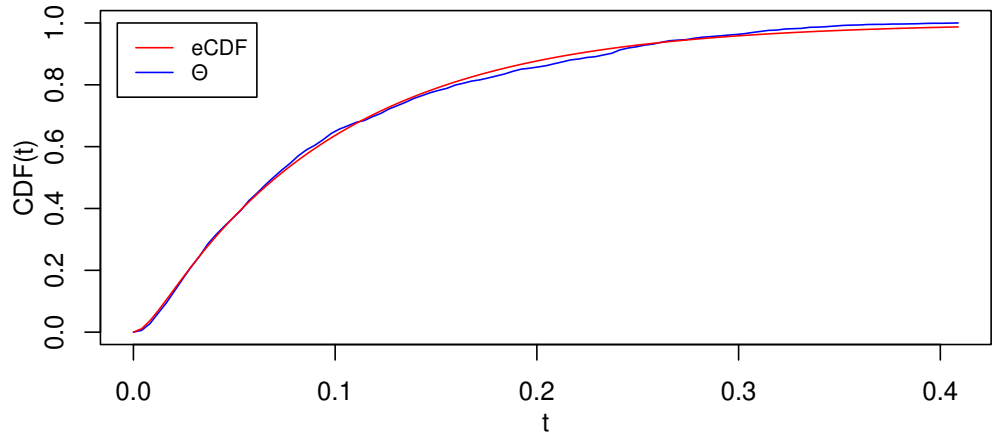


FIGURE 6.13. Empirical and predicted Cumulative Distribution Functions (*CDF*).

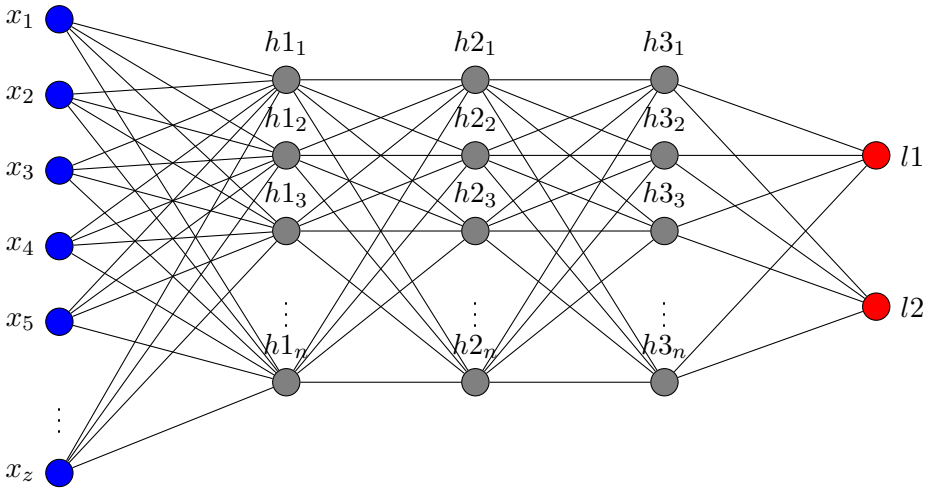


FIGURE 6.14. Representation of the network for the two components

Machine Learning Approach

We used a neural network that predicts each system layer's occupation from raw response times, as clients observe them. Since we wanted to predict the output of a continuous value, our neural network would have to solve a regression problem, i.e. we want to predict our output value as accurately as possible — contrasting with a classification problem.

We made several tests with distinct algorithms and opted for a deep neural network. The rational being that we wanted to correlate both layers' occupation, since the output visible to the client is associated with both layers and has a complex non-linear relation. Furthermore, current software frameworks make deployment and use simple, as well as

production-ready. In addition, we experimented several setups for the neural network — distinct number of layers, nodes, and activation functions. Our final configuration for the neural network consisted of one input layer with 2000 nodes, three hidden layers, each with 100 nodes, with the activation function being the `relu` function (TensorFlow, 2018) and, in the 2 node final layer, a linear activation function. Since the network is shared by both outputs, we were able to have a multi-output regression model to predict each layer's occupation. Hence, both occupations are correlated and influence the hidden layers, having an impact on the outcome. Figure 6.14 illustrates our model. The network receives z input values, which are the response times seen by z client requests.

We provide to our network around $10,000$ lines of raw data from our experiment. Since we are training our neural network, and therefore using a supervised machine learning approach, we need labeled data. Each line has $2,002$ values: $2,000$ response times as seen by clients and the labels: the $2,001^{st}$ value is the occupation of one layer, the $2,002^{nd}$ value is the occupation of the other layer. In the test scenario this output is not available, serving only to validate the accuracy of the prediction.

6.3.2 Evaluation

To validate the tandem queue model fitting method and the neural network, we used response time data generated with a simulated two-component system. We modeled a two component system as two sequential single server queueing systems since it elegantly expresses the variation of response time with occupation. The simulation was made using the `qcomputer` (Ebert et al., 2017) package written in `R`.

The occupation levels were defined as all the combinations of 0.1 increments in the range $[0.1, 0.9]$ - e.g. Layer 1 at 0.1 and Layer 2 at 0.5. The arrival rate was fixed at 30 requests per unit of time and the service rate of each layer varied to express the desired occupation. For each combination of occupations, we collected 150 samples. Since we had 9 occupation levels per components, we collected $9^2 * 150 = 12150$ samples of $2,000$ request observations each. These observations correspond to overall response times of each request.

To evaluate and create the neural network model, we used `Tensorflow` with `Keras` (TensorFlow, 2018). This framework, allow us to rapidly generate and save our model and

is a common standard for the generation of complex networks in the industry and academy. Of the 150 samples, 80%, or 120 per occupation combination, were used to train the neural network, and the remaining (30 per combination) for the validation of both methods — test set. Furthermore, of the portion allocated to the neural network training, 30% were used for model validation in `Tensorflow`. We used a `Sequential` model and the Mean Absolute Error — for the optimization score function —, and a total of 100 iterations over the dataset — i.e. "epochs".

For validation and comparison of the two approaches (neural network and model fitting), we calculated the following error metrics: Mean Square Error (MSE), Mean Absolute Error (MAE), as well as the mean Euclidean distance, because we wanted to understand the distance between the two-dimensional predictions *versus* the real value. These metrics were calculated for the whole set of predictions, by range for each layer occupation as well as for each combination of occupations (Euclidean distance).

6.3.3 Results

Having generated predictions for a test set of $2,430$ samples (30 samples for each of the 81 combinations of occupations), we calculated their respective errors. Besides the global and per range error for each individual layer predictor, for which we used MSE and MAE, we calculated the error as the Euclidean distance for the prediction pair. This latter metric gives an absolute error value that more intuitively shows the quality of the prediction and better exposes issues such as the prediction regressing to the mean of the two occupation values.

Table 6.8. Global results for both methods.

Method	Layer 1		Layer 2	
	MAE	MSE	MAE	MSE
Queue Model	0.05 ± 0.07	0.01 ± 0.03	0.05 ± 0.08	0.01 ± 0.03
Neural Network	0.09 ± 0.11	0.02 ± 0.04	0.08 ± 0.10	0.02 ± 0.04

Table 6.9. Error metrics for each method and layer, grouped by range.

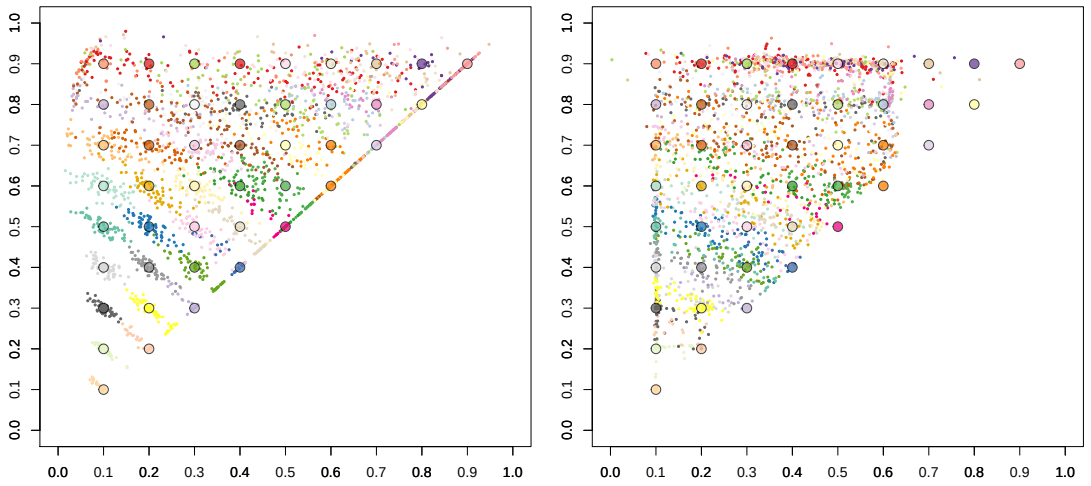
	Queue model				Neural network			
	Layer 1		Layer 2		Layer 1		Layer 2	
ρ	MAE	MSE	MAE	MSE	MAE	MSE	MAE	MSE
0.1	$.05 \pm .09$	$.01 \pm .04$	$.05 \pm .09$	$.01 \pm .04$	$.12 \pm .14$	$.04 \pm .06$	$.11 \pm .13$	$.03 \pm .05$
0.2	$.05 \pm .08$	$.01 \pm .03$	$.06 \pm .11$	$.02 \pm .05$	$.1 \pm .1$	$.02 \pm .04$	$.11 \pm .11$	$.02 \pm .04$
0.3	$.06 \pm .09$	$.01 \pm .03$	$.06 \pm .09$	$.01 \pm .04$	$.1 \pm .08$	$.02 \pm .02$	$.08 \pm .07$	$.01 \pm .02$
0.4	$.06 \pm .08$	$.01 \pm .03$	$.07 \pm .09$	$.01 \pm .03$	$.09 \pm .08$	$.02 \pm .02$	$.08 \pm .07$	$.01 \pm .02$
0.5	$.06 \pm .07$	$.01 \pm .02$	$.06 \pm .08$	$.01 \pm .03$	$.09 \pm .09$	$.02 \pm .03$	$.08 \pm .08$	$.01 \pm .03$
0.6	$.06 \pm .07$	$.01 \pm .03$	$.05 \pm .07$	$.01 \pm .03$	$.09 \pm .1$	$.02 \pm .04$	$.09 \pm .1$	$.02 \pm .04$
0.7	$.05 \pm .05$	$.01 \pm .02$	$.05 \pm .06$	$.01 \pm .03$	$.1 \pm .1$	$.02 \pm .04$	$.09 \pm .1$	$.02 \pm .05$
0.8	$.04 \pm .05$	$0 \pm .03$	$.04 \pm .05$	$0 \pm .2$	$.1 \pm .13$	$.03 \pm .06$	$.08 \pm .11$	$.02 \pm .05$
0.9	$.04 \pm .04$	$0 \pm .01$	$.04 \pm .03$	0 ± 0	$.06 \pm .13$	$.02 \pm .06$	$.01 \pm .02$	0 ± 0

We focus on the MAE and mean Euclidean distance since they have the same unit. Table 6.8 shows the MAE and MSE for each layer, as well as the respective standard deviations. The Mean Euclidean distances for the Tandem Queue Model Fitting and the Deep Neural Network are 0.09 ± 0.10 and 0.15 ± 0.12, respectively. Euclidean distances are further detailed in Figures 6.16(a) and 6.16(b).

The best method in Section 6.2, has a MSE of 0.05 ± 0.04 and 0.05 ± 0.03, for layer 1 and 2 respectively. The new methods both show significant improvement over those same metrics, as shown on Table 6.8. Moreover, both methods show improvement over all ranges of occupation. Table 6.9 shows error metrics for each method and layer, grouped by range. For example $\rho = 0.1$ shows the results for the pairs $(0.1, *)$ and $(*, 0.1)$, where the asterisk stands for all values. As neither model could distinguish the order of the occupation values — $(0.1, 0.5)$ is indistinguishable from $(0.5, 0.1)$ — the errors were calculated after sorting the layer values. The results are much more consistent over the whole range, compared to our previous work from previous Section, where there was clear degradation, especially on the model fitting approach.

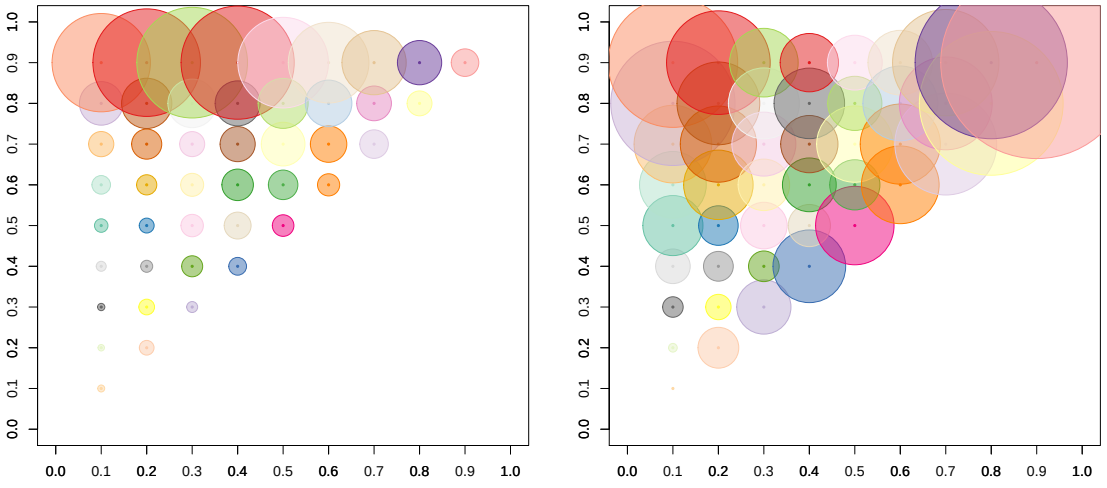
However, the error of each individual layer does not convey an important aspect of the prediction quality. We wish to preserve the relationship between the two occupation values, meaning that $(0.1, 0.5)$ should be predicted as such, instead of averaging the values to $(0.3, 0.3)$. To understand if this relationships exists, we measure the error as the Euclidean distance between the target and prediction pairs. Figure 6.16 shows the error in a way that preserves that relation. We present two visualizations for each method.

Firstly, we show the predictions as a cloud of points color-coded by expected occupation pair (Figures 6.15(a) and 6.15(b)). The true occupations have a black circumference; predictions are dots with the same color as the disk inside the circumference. For example, in Figure 6.15(a), we see that for occupations $(0.1, 0.1)$ all the predictions are clustered around the real value. As we get farther from the origin, they get increasingly more disperse. On Figure 6.15(b), representing the neural network results, we see there is a particular pattern of dispersion for the extremes. On high occupations, predictions are shifted along one of the axes. This means that the neural network is able to accurately predict the higher occupation value, but the second one will tend towards central values. This is expected, since the highest occupied component dominates the overall response time experienced by the client.



(a) Queue model. (b) Neural network.

FIGURE 6.15. Model predictions.



(a) Queue model (b) Neural network

FIGURE 6.16. Mean Euclidean distance

Secondly, we show bubble charts, where the radius of each bubble is the mean Euclidean distance of the predictions from the real value. Here it is visible that the queue model, in Figure 6.16(a), shows stability along the range, except on the highest occupation values. Figure 6.16(b), relative to the Neural Network, presents a different pattern, having lower accuracy when the difference in occupations is highest ($(0.1, 0.9)$) or as it get closer to $(0.9, 0.9)$.

Results show that we can estimate the internal occupations from the system's response time with small error. When at least one of the components is very busy, only one of

the high occupations is correctly approximated. We do not regard this as problematic, because a very busy component will dominate the response time anyway.

6.4 Conclusion

Monitoring and observability of systems is a challenge for administrators, due to increased application elasticity, complexity, granularity and dynamics. New software releases, agents, tracing, and a plethora of system monitoring tools and dashboards creates a complex environment for administrators to ensure correctness and proper quality of service. Additionally, current tools put the burden of analysis, such as detecting bottlenecks and performance issues on the operators. As the number of services and technologies in a system increases, so does the complexity and time spent in operational tasks, such as root cause analysis.

In this chapter, we made several endeavours for new approaches for monitoring, without any instrumentation or probes in the system. First, we aimed to analyze the limits of a "black-box" approach, using only some of the infrastructural modules already deployed in a microservice architecture. We resorted to Netflix modules, customizing the gateway, to gather raw data from microservice invocations. Results show that our solution involves minimal configuration efforts to be integrated in the system and to produce relevant information to administrators.

Secondly, we aim to identify the occupation of each component on a two-layer system. The evidence collected shows that it is possible to identify the capacity and occupation of each layer, solely from the overall response time, as measured by a client. Since the proposed method is not coupled to any infrastructure or language, it can be used in any kind of system where observability is a major concern. The two methods – supervised machine learning and exponential decomposition – combined can complement each other's shortcoming and provide insights about the overall system performance, although with some shortcomings, such as the non-adaptive behaviour with distinct layer occupation, resulting in a median value for both layers.

Third, we improve our methods to identify occupation for a two-layer system. More specifically, we wanted to compare two distinct methods: first, an optimized algorithm specifically designed to our scenario, and secondly, a neural network trained with the data collected from our experiment. Our results have improved, showing that it is viable

to infer the load of each layer collecting only the overall response time. Hence, these two methodologies – neural network and tandem queue model – are able to improve current monitoring tools, and ensure a more fine-grained knowledge about the system.

For future work, there are some directions that could be further investigated. First, the open-source tool designed in the first section could be improved for automated analysis, such as in critical path enumeration and anomaly detection, to give administrators more information about the high-level behavior. Secondly, we could use the knowledge related to layer occupation, extending this methodology for more generic topological inference.

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the thesis and explain the main conclusions, contributions and achievements of this research work. Additionally, we pave the way for future research directions.

7.1 Summary of the Thesis

Distributed systems take an important role in society. Almost every single system of our digital life has, in its genesis, a distributed system. These systems have thus become an essential part of the daily life for billions of people worldwide. Unavailability and quality-of-service degradation is something that no one is willing to accept. Maintaining the healthiness of a distributed system requires not only the proper people — administrators and operators —, but also effective monitoring tools. However, monitoring distributed systems involves multiple challenges.

This thesis started by presenting a survey on monitoring in distributed systems. It is observable that this research field is highly dynamic with a lot of contributions in the industry and also in the academia. There is a plethora of solutions concerning this kind of system, from more intrusive to some that are more black-box. An explanation for this abundance is that some companies lack viable monitoring solutions to comply with their demands, thus having to create customized tools. Selecting the right tool and acquiring the best possible information to mitigate and even predict system performance issues is not a trivial task. Each monitoring solution supports a specific feature or enables some

characteristic that is useful for the development or operator team. Hence, the need for the aforementioned personalized solutions implemented in their systems.

We analyzed several academic studies and industrial solutions to monitor distributed systems. This analysis helped us to find out the gaps between what we wanted to achieve and the current state of the art. As a result, we observed that the majority of monitoring solutions implement dashboards and notification systems; furthermore, they are intrusive and they are intertwined with the system to monitor. Another key aspect is that although they create dashboards and present the errors, it is always the responsibility of administrators and operators to interpret the data. This could be mitigated with some autonomous performance issue detection.

This thesis aimed to answer the research question of what are the limits of a full black-box solution, using client-side data. It also tries to identify what kind of information is useful as a complement to current monitoring tools, to reduce time to detect any kind of issues in the system. The client-side method is applied to several infrastructures, such as websites, REST interfaces, common applications supported by HTTP clusters, and larger microservice architectures.

We proposed several distinct approaches to analyze client-side data: firstly, collecting client information retrieved by standard frameworks, like the NavigationTiming (2015), before processing the data using time series and machine learning. We extended this approach to other more generic solutions that did not require JavaScript instrumentation. Secondly, we investigate how exactly do clients perceive bottlenecks that are occurring in the distributed system. We were interested in understanding when bottlenecks arise in the infrastructure and to determine whether clients (as opposed to infrastructure owners) could pinpoint their possible cause. Thirdly, we had the goal of understanding what valuable information may emerge when we combine information from multiple clients using the same distributed system. For this we correlated data retrieved by distinct clients. Finally, with the previously acquired know-how we analyzed a microservice infrastructure, to understand if a microservice could monitor other microservices that are invoked.

As a result of the work of this thesis, we conclude that:

- Clients can indeed perceive problems in complex distributed systems and, in some cases, they can pinpoint the location of such problems. We demonstrate in Chapters 3, 4, 5 and 6 the following results:
  - Some errors may elude system's administrators control, specially with third-party providers, with direct impact on the client, as we showed in Chapter 3.
  - Clients with few information about the system can create a profile of the system, thus understanding its normal behavior. This was demonstrated in the experiments of Chapter 4.
  - Even when the client has only a coarse-grained view of the overall health of the system, thus being unable to identify the root cause of a problem, administrators may still enrich current monitoring solutions with client's data, to have better insights about the quality of service. This was demonstrated with the experiments from Chapter 5 and 6.
- The previous conclusions are quite promising for microservice architectures, because each microservice may monitor its downstream services. This "auto-monitoring" solution requires little to no instrumentation of the infrastructure as demonstrated on Chapter 6.

As a final conclusion, administrators cannot be locked in their own shell, hoping, from their instruments that clients are enjoying the best experience from their systems. This thesis shows that clients have valuable information to diagnose system problems and provide timely alerts to operators.

7.2 Future Work

We believe that the work presented in this thesis can improve the state of the art on distributed systems monitoring. However, due to time constraints, we could not explore a large number of topics, including the following:

- Monitoring solutions presented in this thesis were mostly experimented in laboratory. The use in a real-world scenario with multiple variables of uncertainty is the next step to have more robust methods.

- A collection of previously labelled bottlenecks could increase the quality of system monitoring. However, we could not follow this path, due to the absence of online data sets or real-world scenarios, as far as we could tell.
- In this thesis, we mostly designed black-box monitoring approaches that may be interconnected with current white-box (and more intrusive) solutions. However, we could not explore the technical details of how to combine these mechanisms.
- In Chapter 3, we presented a study over $3,000$ web sites. These websites were a sample from the top 1 million most visited web sites. At this time, we are working on a more complex experiment, where we are going to analyze all the ranking (i.e. all the 1 million websites). Taking into consideration 1 million web sites instead of only $3,000$ may bring new conclusions and more research questions.
- In Chapter 6, we monitored microservices. As stated in that chapter, there is the possibility of inferring the system components and extracting topology from an outside point-of-view.

Bibliography

Agarwal, S., Liogkas, N., Mohan, P., and Padmanabhan, V. (2010). Webprofiler: Cooperative diagnosis of web failures. In *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, pages 1–11.

Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. (2003). Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74.

Alarm (2019 (accessed June, 2019)). Alarm definition. `https://www.yourdictionary.com/alarm`.

Alexa (2015 (accessed May, 2015)). Alexa — Top Sites in Portugal. `http://www.alexa.com/topsites/countries/PT`.

Alexa (2017 (accessed May, 2017)). Alexa — top-ranked websites. `https://support.alexa.com/hc/en-us/articles/200449834-Does-Alexa-have-a-list-of-its-top-ranked-websites-`.

Amazon (2015 (accessed May, 2015). Amazon.com: Online shopping for electronics, apparel, computers, books, dvds & more. `http://www.amazon.com`.

AppDynamics (2017 (accessed May, 2017)). Appdynamics. `https://www.appdynamics.com/product/application-performance-management/`.

Atchison, L. (2018 (accessed September, 2019)). Microservice architectures: What they are and why you should use them. `https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/`.

Atlas (2018 (accessed May, 2018)). Atlas. `https://github.com/Netflix/atlas`.

Attariyan, M., Chow, M., and Flinn, J. (2012). X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, Hollywood, CA. USENIX.

AWS (2015 (accessed May, 2015)). Papers — Amazon Web Services. `http://aws.amazon.com/`.

Azure (2015 (accessed May, 2015)). Papers — Windows Azure Service Level Agreement. `http://www.windowsazure.com/en-us/support/legal/sla/`.

AzureMonitor (2019 (accessed May, 2019)). Azure Monitor. `https://azure.microsoft.com/en-us/services/monitor/`.

Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M. (2007). Towards highly reliable enterprise network services via inference of multi-level dependencies. *SIGCOMM Comput. Commun. Rev.*, 37(4):13–24.

Barga, R., Chen, S., and Lomet, D. (2004). Improving logging and recovery performance in phoenix/app. In *Proceedings of the 20th International Conference on Data Engineering*, pages 486–497. IEEE.

Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. (2004). Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA. USENIX Association.

Battre, D., Hovestadt, M., Lohrmann, B., Stanik, A., and Warneke, D. (2010). Detecting bottlenecks in parallel dag-based data flow programs. In *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, pages 1–10.

Bento, A. P. (2019). Observing and controlling performance in microservices. Master's thesis.

Bodic, P., Friedman, G., Biewald, L., Levine, H., Candea, G., Patel, K., Tolle, G., Hui, J., Fox, A., Jordan, M. I., and Patterson, D. (2005). Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 89–100.

Bodík, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., and Patterson, D. (2009). Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA. USENIX Association.

Brooker, G. M. (2012). Feedback and control systems. pages 159–205.

Bucky (2017 (accessed June, 2017)). Bucky — performance measurement of your app's actual users. `http://github.hubspot.com/bucky/`.

BwmNg (2017 (accessed May, 2017)). Bandwidth Monitor. `https://github.com/vgropp/bwm-ng`.

Cao, J., Andersson, M., Nyberg, C., and Kihl, M. (2003). Web Server Performance Modeling using an M/G/1/K*PS Queue. In *10th International Conference on Telecommunications, ICT 2003*, volume 2, pages 1501–1506.

Cecchet, E., Udayabhanu, V., Wood, T., and Shenoy, P. (2011). Benchlab: an open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 4–4. USENIX Association.

Chanda, A., Cox, A. L., and Zwaenepoel, W. (2007). Whodunit: Transactional profiling for multi-tier applications. *SIGOPS Oper. Syst. Rev.*, 41(3):17–30.

CheckMySite (2017 (accessed May, 2017)). Check my website. `https://checkmy.ws/en/features/`.

Chen, C., Maniatis, P., Perrig, A., Vasudevan, A., and Sekar, V. (2013). Towards verifiable resource accounting for outsourced computation. In *ACM Sigplan Notices*, volume 48, pages 167–178. ACM.

Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E., Brewer, E., and Brewer, E. (2004). Path-based faliure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 23–23, Berkeley, CA, USA. USENIX Association.

Chi, R., Qian, Z., and Lu, S. (2011). A heuristic approach for scalability of multi-tiers web application in clouds. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pages 28–35.

Chrome (2017 (accessed April, 2017)). Chrome browser. `https://www.google.com/chrome/browser/desktop/`.

ChromeDriver (2015 (accessed April, 2015)). Chrome driver - webdriver for chrome. `https://sites.google.com/a/chromium.org/chromedriver/`.

ChromeErrors (2016 (accessed April, 2016)). Papers — Chrome List of Network Errors. `https://src.chromium.org/svn/trunk/src/net/base/net_error_list.h`.

ChromeNetwork (2016 (accessed June, 2016)). Papers — Network - Google Chrome. `https://developer.chrome.com/devtools/docs/protocol/1.0/network`.

Ciuffoletti, A. (2015). Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163 – 172. 1st International Conference on Cloud Forward: From Distributed to Complete Computing.

ClickyAnalytics (2017 (accessed May, 2017)). Clicky web analytics. `https://clicky.com/`.

Clifton, B. (2008). *Advanced Web Metrics with Google Analytics*. SYBEX Inc., Alameda, CA, USA.

CloudWatch (2019 (accessed May, 2019)). Amazon CloudWatch. `https://aws.amazon.com/cloudwatch/`.

CMUPDL-05-109 (2005). Causes of failure in web applications (cmupdl-05-109), dec 2005. `http://repository.cmu.edu/cgi/viewcontent.cgi?article=1047&context=pdl`. Retrieved: April, 2016.

CNBC (2019). Target says cash registers back online and customers can make purchases again after systems outage. https://www.cnbc.com/2019/06/15/targets-in-store-payment-is-system-down-impacting-stores-nationwide.html. Retrieved June 2019, from CNBC.

CNN ("2015 (accessed May, 2015)"). Breaking news, u.s., world, weather, entertainment & video news - cnn.com. `http://edition.cnn.com`.

Cormode, G., Korn, F., and Tirthapura, S. (2008). Time-decaying aggregates in out-of-order streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 89–98, New York, NY, USA. ACM.

Cpulimit (2017 (accessed May, 2017)). Cpulimit - cpu usage limiter for linux. https://github.com/opsengine/cpulimit.

Crontab (2015 (accessed May, 2015)). Crontab - quick reference | admin's choice - choice of unix and linux administrators. http://www.adminschoice.com/crontab-quick-reference.

Cui, H. and Biersack, E. (2013). Troubleshooting slow webpage downloads. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 405–410.

DailyMail (2015 (accessed Nov, 2015)). Facebook crash. http://www.dailymail.co.uk/sciencetech/article-3252603/Facebook-goes-Social-network-crashes-time-month-leaving-users-panic.html.

Das, S. K., Kant, K., and Zhang, N. (2012). *Handbook on Securing Cyber-Physical Critical Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Dashboard (2019 (accessed June, 2019)). Dashboard definition. https://www.pcmag.com/encyclopedia/term/40726/dashboard.

Dhawan, M., Samuel, J., Teixeira, R., Kreibich, C., Allman, M., Weaver, N., and Paxson, V. (2012). Fathom: A browser-based network measurement platform. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 73–86, New York, NY, USA. ACM.

Dietrich, E. (2018 (accessed September, 2019)). Log appender: What is it and why would you use it? https://dzone.com/articles/log-appender-what-is-it-and-why-would-you-use-it.

Dilley, J., Friedrich, R., Jin, T., and Rolia, J. (1998). Web server performance measurement and modeling techniques. *Performance Evaluation*, 33(1):5–26.

Docker (2018 (accessed June, 2018)). Docker. https://www.docker.com/what-docker.

DockerOverlay (2018 (accessed June, 2018)). Docker overlay network. https://docs.docker.com/network/overlay/.

Downing, T. B. (1998). *Java RMI: remote method invocation*. IDG Books Worldwide, Inc.

Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. In Mazzara, M. and Meyer, B., editors, *Present and Ulterior Software Engineering*. Springer.

DynaTrace (2017 (accessed May, 2017)). Dynatrace. `https://www.dynatrace.com/platform/`.

Ebert, A., Wu, P., Mengersen, K., and Ruggeri, F. (2017). Computationally Efficient Simulation of Queues: The R Package queuecomputer.

EC2, A. (2015 (accessed May, 2015)). Papers — Amazon EC2 Service Level Agreement. `http://aws.amazon.com/ec2-sla/`.

Engadget (2019). Facebook's massive outage was the result of a server configuration change. `https://www.engadget.com/2019/03/14/facebook-instagram-outage-server-configuration-change/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAFSw-ONCiV29uxow2-gAHKfhDFQulfYoUsuT-BmA7WPHiN_vdikgn6uMA8SXuvl1Bqz6Jn8XMVPM5iuOB7AUPgf1tXck3goKw_aJnjciAwQal4-VFts2pWm_x9wX9wl4A_dH_i9cK1Y4X7pbAYantN_3xH_e_eVC0PUg3p6Ixf7W`. Retrieved: Mar, 2019.

EventLogs (2019 (accessed May, 2019)). Event logs. `https://en.m.wikipedia.org/wiki/Log_file`.

Ewaschuk, R. (2019 (accessed September, 2019)). Monitoring distributed systems. `https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/`.

Ewaschuk, R. and Beyer, B. (2016). *Monitoring Distributed Systems: Case Studies from Google's SRE Teams*. O'Reilly Media, Incorporated.

Fielding, R. and Reschke, J. (2014). Hypertext transfer protocol (http/1.1): Message syntax and routing. Technical report.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, USA.

Flach, T., Katz-Bassett, E., and Govindan, R. (2013). Diagnosing slow web page access at the client side. In *Proceedings of the 2013 Workshop on Student Workhop*, CoNEXT Student Workhop '13, pages 59–62, New York, NY, USA. ACM.

Flume, A. (2019 (accessed September, 2019)). Apache flume. `https://flume.apache.org/`.

Fonseca, R., Dutta, P., Levis, P., and Stoica, I. (2008). Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 323–338, Berkeley, CA, USA. USENIX Association.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. (2007). X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation (NSDI'07)*, number April, page 20. USENIX Association.

Gartner (2013 (accessed May, 2013)). Papers — gartner says 60 percent of virtualized servers will be less secure than the physical servers they replace through 2012 [online]. `www.gartner.com/it/page.jsp?id=1322414`.

GitHub (2018 (accessed May, 2018)). Github – monitoring_ms. `https://github.com/fabiopina/monitoring_ms`.

GlassFish (2019 (accessed August, 2013)). Technical white papers — GlassFish Application Server. `http://glassfish.java.net/`.

Gomez-Uribe, C. A. and Hunt, N. (2015). The *Netflix* recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4):13:1–13:19.

GoogleAnalytics (2018 (accessed May, 2018)). Google Analytics. `https://analytics.google.com/analytics/web/`. Retrieved May, 2018.

Grafana (2018 (accessed May, 2018)). Grafana. `https://grafana.com/`.

Graphite (2019 (accessed June, 2019)). Graphite. `https://graphiteapp.org/`.

Gray, J. Why do computers stop and what can be done about it?

Graylog (2019 (accessed September, 2019)). Graylog correlation & event management engine. `https://www.graylog.org/`.

Gu, Z., Gu, L., Eils, R., Schlesner, M., and Brors, B. (2014). circlize implements and enhances circular visualization in r. *Bioinformatics*, 30(19):2811–2812.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.

Halpern, J. Y. (1987). Using reasoning about knowledge to analyze distributed systems. *Annual Review of Computer Science*, 2(1):37–68.

Hart, G. W. (1992). Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 80(12):1870–1891.

Haselböck, S. and Weinreich, R. (2017). Decision guidance models for microservice monitoring. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 54–61.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S., and Wettinger, J. (2017). Performance Engineering for Microservices. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE '17 Companion*, pages 223–226, New York, New York, USA. ACM Press.

Hellerstein, J. L., Maccabee, M. M., Mills, W. N., and Turek, J. J. (1999). Ete: a customizable approach to measuring end-to-end response times and their components in distributed systems. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 152–162.

Herbst, N. R., Kounev, S., and Reussner, R. (2013). Elasticity in Cloud Computing: What It Is, and What It Is Not. *Presented as part of the 10th International Conference on Autonomic Computing*, pages 23–27.

Horváth, A. and Telek, M. (2002). Phfit: A general phase-type fitting tool. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 82–91. Springer.

HTML5 (2016 (accessed April, 2016)). Papers — Cross-Origin Resource Sharing. `https://www.w3.org/TR/cors/`.

HTTP (1999). Rfc 2616 - Hypertext Transfer Protocol — HTTP / 1.1. Internet Engineering Task Force (IETF).

HTTPArchive (2016 (accessed April, 2016)). Http archive. http://httparchive.org/.

Huang, J., Mozafari, B., and Wenisch, T. F. (2017). Statistical analysis of latency through semantic profiling. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 64–79. ACM.

Huber, N., Brosig, F., and Kounev, S. (2011). Model-based self-adaptive resource allocation in virtualized environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 90–99, New York, NY, USA. ACM.

Influx (2018 (accessed May, 2018)). Influxdata. https://www.influxdata.com/modern-time-series-platform/.

Instana (2019 (accessed May, 2019)). Instana. https://www.instana.com/.

Internet, A. (2016 (accessed May, 2016)). At internet. http://www.atinternet.com/.

Iqbal, W., Dailey, M. N., Carrera, D., and Janecek, P. (2010). Sla-driven automatic bottleneck detection and resolution for read intensive multi-tier applications hosted on a cloud. In *Advances in Grid and Pervasive Computing*, pages 37–46. Springer.

Iqbal, W., Dailey, M. N., Carrera, D., and Janecek, P. (2011). Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879.

Ivaki, N., Laranjeiro, N., and Araujo, F. (2015). A taxonomy of reliable request-response protocols. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 456–463, New York, NY, USA. ACM.

Jaegger (2019 (accessed Jun, 2019)). Jaegger. https://www.jaegertracing.io/. Retrieved Jun, 2019.

Janapati, S. P. R. (2017 (accessed May, 2019)). Dzone - distributed logging architecture for microservices. https://www.elastic.co/products/kibana/.

Janssen, T. (2018 (accessed September, 2019)). Java logging frameworks: log4j vs logback vs log4j2. https://stackify.com/compare-java-logging-frameworks/.

JMeter (2013 (accessed June, 2013)). Performance tools — Apache JMeter^{TM}. http://jmeter.apache.org/.

Johnson, D. B. (1989). Distributed system fault tolerance using message logging and checkpointing. Technical report, DTIC Document.

Joyce, J., Lomow, G., Slind, K., and Unger, B. (1987). Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150.

Kalman, R. (1959). On the general theory of control systems. *IRE Transactions on Automatic Control*, 4(3):110–110.

Kattepur, A. and Nambiar, M. (2015). Performance modeling of multi-tiered web applications with varying service demands. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 415–424.

Khadke, N., Kasick, M. P., Kavulya, S. P., Tan, J., and Narasimhan, P. (2012). Transparent system call based performance debugging for cloud computing. In *Presented as part of the 2012 Workshop on Managing Systems Automatically and Dynamically*, Hollywood, CA. USENIX.

Kibana (2018 (accessed May, 2018)). Kibana. `https://www.elastic.co/products/kibana/`.

Kiciman, E. and Fox, A. (2005). Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041.

Kreibich, C., Weaver, N., Nechaev, B., and Paxson, V. (2010). Netalyzr: Illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 246–259, New York, NY, USA. ACM.

Lamport, L. (1987). Leslie lamport's home page. `http://research.microsoft.com/en-us/um/people/lamport/`.

Laprie, J. C. (2008). From dependability to resilience. In *In 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks.*

Laranjeiro, N., Vieira, M., and Madeira, H. (2009). Improving web services robustness. In *2009 IEEE International Conference on Web Services*, pages 397–404.

Lewis, J. (2019 (accessed June, 2019)). Microservices - a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`.

Li, H. (2010). A Queue Theory Based Response Time Model for Web Services Chain. *2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4.

Li, S. (2019). Time Series of Price Anomaly Detection. `https://towardsdatascience.com/time-series-of-price-anomaly-detection-13586cd5ff46`. 2019 (accessed Mar, 2019).

Li, W. and Gorton, I. (2010). Analyzing web logs to detect user-visible failures. In *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML'10, pages 6–6, Berkeley, CA, USA. USENIX Association.

Li, W., Harrold, M. J., and Görg, C. (2010a). Detecting user-visible failures in ajax web applications by analyzing users' interaction behaviors. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 155–158, New York, NY, USA. ACM.

Li, Z., Zhang, M., Zhu, Z., Chen, Y., Greenberg, A. G., and Wang, Y.-M. (2010b). Webprophet: Automating performance prediction for web services. In *NSDI*, volume 10, pages 143–158.

Lin, F. and Wonham, W. (1988). On observability of discrete-event systems. *Information Sciences*, 44(3):173 – 198.

Linden, G., Smith, B., and York, J. (2003). Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80.

Linkedin (2019 (accessed May, 2019)). Linkedin - tracing. `https://engineering.linkedin.com/distributed-service-call-graph/real-time-distributed-tracing-website-\protect\discretionary{\char\hyphenchar\font}{}{}performance-and-efficiency`.

Liu, H., Shah, S., and Jiang, W. (2004). On-line outlier detection and data cleaning. *Computers and Chemical Engineering*, 28(9):1635–1647.

Liu, H. and Wee, S. (2009). Web server farm in the cloud: Performance evaluation and dynamic architecture. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 369–380, Berlin, Heidelberg. Springer-Verlag.

Liu, Y., Zhang, L., and Guan, Y. (2010). Sketch-based streaming pca algorithm for network-wide traffic anomaly detection. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 807–816.

LogEntries (2019 (accessed September, 2019)). The fastest way to analyze your log data. `https://logentries.com/`.

Loggly (2019 (accessed September, 2019)). Python logging libraries and frameworks. `https://www.loggly.com/ultimate-guide/python-logging-libraries-frameworks/`.

Logstash (2019 (accessed September, 2019)). Logstash. `https://www.elastic.co/products/logstash`.

Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. (2015). Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 589–603, Berkeley, CA, USA. USENIX Association.

Mace, J., Roelke, R., and Fonseca, R. (2018). Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst.*, 35(4):11:1–11:28.

Malik, H. and Shakshuki, E. M. (2016). Towards identifying performance anomalies. *Procedia Computer Science*, 83(Supplement C):621 – 627. The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops.

Malkowski, S., Hedwig, M., Parekh, J., Pu, C., and Sahai, A. (2007). Bottleneck detection using statistical intervention analysis. In *Managing Virtualization of Networks and Services*, pages 122–134. Springer.

Malkowski, S., Hedwig, M., and Pu, C. (2009). Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 118–127. IEEE.

Manifesto, R. (2019 (accessed June, 2019)). Reactive Manifesto. `https://www.reactivemanifesto.org`.

Martin, R. C. (2019 (accessed June, 2019)). The Single Responsibility Principle. https://www.oreilly.com/library/view/97-things-every/9780596809515/ch76.html.

Mayer, B. and Weinreich, R. (2017). A dashboard for microservice monitoring and management. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 66–69.

Mediadrop (2017 (accessed June, 2017)). Mediadrop - mediadrop open source project. http://mediadrop.video/.

Mendes, J., Laranjeiro, N., and Vieira, M. (2018). Toward characterizing html defects on the web. *Software: Practice and Experience*, 48(3):750–757.

Mertz, N. B. (2019 (accessed June, 2019)). Anomaly Detection in Google Analytics — A New Kind of Alerting. https://medium.com/the-data-dynasty/anomaly-detection-in-google-analytics-a-new-kind-of-alerting-9c31c13e5237.

Monitis (2017 (accessed June, 2017)). Real user monitoring (rum) - monitis. http://www.monitis.com/real-user-monitoring/.

MonitoringTools (2015 (accessed June, 2015)). Papers — External Site Monitoring Services. http://softwareqatest.com/qatweb1.html#MONITORING.

Moradi, F., Flinta, C., Johnsson, A., and Meirosu, C. (2017). Conmon: An automated container based network performance monitoring system. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 54–62.

Muthukrishnan, S. et al. (2005). Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236.

Nagios (2019 (accessed June, 2019)). Nagios. https://www.nagios.org/.

NavigationTiming (2015 (accessed May, 2015)). Papers — Navigation Timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.

Netflix (2018 (accessed May, 2018)). Netflix. https://speakerdeck.com/adriancole/distributed-tracing-and-zipkin-at-netflixoss-barcelona.

Netravali, R., Goyal, A., Mickens, J., and Balakrishnan, H. (2016). Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, pages 123–136.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition.

NewRelic (2017 (accessed May, 2017)). New Relic. `https://newrelic.com/application-monitoring/features`.

Nielsen, P. (2017 (accessed September, 2019)). Google's 'superhuman' deepmind ai claims chess crown. `https://www.bbc.com/news/technology-42251535`.

OpenCensus (2019 (accessed May, 2019)). Opencensus. `https://opencensus.io/`.

OpenTracing (2019 (accessed September, 2019)). Opentracing. `http://opentracing.io/`.

OpenTSDB (2018 (accessed May, 2018)). Opentsdb. `http://opentsdb.net/`.

Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA. USENIX Association.

Owezarski, P., Lobo, J., and Medhi, D. (2013). Network and service management for cloud computing and data centers: A report on cnsm 2012. *Journal of Network and Systems Management*, 21.

Özveren, C. M. and Willsky, A. S. (1990). Observability of discrete event dynamic systems.

Padmanabhan, V. N., Ramabhadran, S., Agarwal, S., and Padhye, J. (2006). A study of end-to-end web access failures. In *Proceedings of CoNEXT*, Lisboa, Portugal.

Palma, F., Dubois, J., Moha, N., and Guéhéneuc, Y.-G. (2014). Detection of rest patterns and antipatterns: A heuristics-based approach. In Franch, X., Ghose, A. K., Lewis, G. A., and Bhiri, S., editors, *Service-Oriented Computing*, pages 230–244, Berlin, Heidelberg. Springer Berlin Heidelberg.

Panda, R., Rocha, B., and Paiva, R. P. (2013). Dimensional music emotion recognition: Combining standard and melodic audio features. In *Proceedings of the 10th International Symposium on Computer Music Multidisciplinary Research (CMMR)*, pages 583–593.

Perf, H. (2013 (accessed June, 2013)). Papers — HP Web Server Performance Tool. http://www.hpl.hp.com/research/linux/httperf/.

PetStore (2013 (accessed August , 2013)). Technical white papers — Java Petstore 2.0. http://www.oracle.com/technetwork/java/index-136650.html.

Philippe, N. (1998). Basic elements of queueing theory application to the modelling of computer systems. *Le Chesney, France: INRIA*.

Pingdom (2017 (accessed May, 2017)). Website performance monitoring - pingdom. https://www.pingdom.com/.

PlanetLab (2015 (accessed June, 2015)). Papers — Planet Lab. https://www.planet-lab.org/.

Postel, J. (1981). Transmission Control Protocol. RFC 793 (Standard). Updated by RFCs 1122, 3168.

Prometheus (2018 (accessed June, 2018)). Prometheus. https://prometheus.io/.

ResourceTiming (2016 (accessed March, 2016)). Papers — Resource Timing. https://www.w3.org/TR/2016/WD-resource-timing-20160225/.

Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. (2006). Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 9–9, Berkeley, CA, USA. USENIX Association.

Rimal, B., Choi, E., and Lumb, I. (2009). A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM 09. Fifth International Joint Conference on*, pages 44–51.

Rubis (2017 (accessed June, 2017)). Rubis home page. http://rubis.ow2.org/.

Sambasivan, R. R., Fonseca, R., Shafer, I., and Ganger, G. R. (2014). So, you want to trace your distributed system? key design insights from years of practical experience.

Sambasivan, R. R., Shafer, I., Mace, J., Sigelman, B. H., Fonseca, R., and Ganger, G. R. (2016). Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 401–414, New York, NY, USA. ACM.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

Samza (2019 (accessed May, 2019)). Apache Samza.

Sánchez, M. A., Otto, J. S., Bischof, Z. S., Choffnes, D. R., Bustamante, F. E., Krishnamurthy, B., and Willinger, W. (2013). Dasu: Pushing experiments to the internet's edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 487–499, Lombard, IL. USENIX.

SciKit (2018 (accessed June, 2018)). Scikit learn. `http://scikit-learn.org/stable/index.html`.

Selenium (2015 (accessed May, 2015)). Papers — Selenium Browser automation. `http://www.seleniumhq.org/`. accessed: May, 2015.

SemaText (2019 (accessed July, 2019)). Papers — Rum vs APM. `https://sematext.com/blog/rum-vs-apm/`.

Sen, A. and Srivastava, M. (2012). *Regression analysis: theory, methods, and applications*. Springer Science & Business Media.

Shahin, A. A. (2017). Enhancing Elasticity of SaaS Applications using Queuing Theory. *IJACSA) International Journal of Advanced Computer Science and Applications*, 8(1):279–285.

Shegalov, G., Weikum, G., Barga, R., and Lomet, D. (2002). Eos: exactly-once e-service middleware. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 1043–1046. VLDB Endowment.

Shoaib, Y. and Das, O. (2011). Web application performance modeling using layered queueing networks. *Electron. Notes Theor. Comput. Sci.*, 275:123–142.

Shoaib, Y. and Das, O. (2012). Using layered bottlenecks for virtual machine provisioning in the clouds. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 109–116.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

Singh, R., Sharma, U., Cecchet, E., and Shenoy, P. (2010). Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 21–30, New York, NY, USA. ACM.

Sommers, J. and Barford, P. (2007). An active measurement system for shared environments. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 303–314, New York, NY, USA. ACM.

Specification, O. (2019 (accessed September, 2019)). Opentracing specification. `https://github.com/opentracing/specification`.

Spotify (2018 (accessed June, 2018)). Spotify. `https://labs.spotify.com/2015/11/17/monitoring-at-spotify-introducing-heroic/`.

StatusCanceled (2016 (accessed April, 2016)). Papers — What means status canceled. `http://stackoverflow.com/questions/12009423/what-does-status-canceled-for-a-resource-mean-in-chrome-developer-tools`.

Steady-state (2019 (accessed May, 2019)). Steady-state. `https://www.yourdictionary.com/steady-state`.

Sweenie, T. (2000). No time for downtime: It managers feel the heat to prevent outages that can cost millions of dollars. Internet Week, N.807, 3.

Sweenie, T. (2003). The black friday report on web application integrity. Business Internet Group, San Francisco, CA.

Tak, B. C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B., and Chang, R. N. (2009). vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 19–19, Berkeley, CA, USA. USENIX Association.

Technology, T. (2005). A study about online transactions, prepared for tealeaf technology inc, oct 2005. `http://www-01.ibm.com/software/info/tealeaf/`. Retrieved: April, 2016.

Techopedia (2019 (accessed May, 2019)a). Debugging. `https://www.techopedia.com/definition/16373/debugging`.

Techopedia (2019 (accessed September, 2019)b). Monitoring software. `https://www.techopedia.com/definition/4313/monitoring-software`.

TensorFlow (2019 (accessed Feb, 2018)). Tensorflow. `https://www.tensorflow.org/guide/keras`.

TheFreeDictionary (2019 (accessed September, 2019)). Observing. `https://www.thefreedictionary.com/observing`.

Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. (2006). Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM.

Thorburn, W. M. (1915). Occam's razor. *Mind*, 24(2):287–288.

TimescaleDB (2018 (accessed May, 2018)). Timescaledb. `https://www.timescale.com/`.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., and Edmonds, A. (2015). An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, pages 19–24, New York, NY, USA. ACM.

TPC-W (2017 (accessed May, 2017)). Tpc-w benchmark, objectweb implementation. `http://jmob.ow2.org/tpcw.html`.

TrafficControl (2017 (accessed May, 2017)). Traffic control. `http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html`.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.*, 33(1):291–302.

Us, T., Jensen, N., Lind, M., and Jørgensen, S. B. (2011). Fundamental principles of alarm design. *International Journal of Nuclear Safety and Simulation*, 2(1):44–51.

Van Do, T., Krieger, U. R., and Chakka, R. (2008). Performance modeling of an apache web server with a dynamic pool of service processes. *Telecommunication Systems*, 39(2):117–129.

Vateva-Gurova, T., Suri, N., and Mendelson, A. (2015). The impact of hypervisor scheduling on compromising virtualized environments. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1910–1917.

Vaz, C., Silva, L., and Dourado, A. (2011). Detecting user-visible failures in web-sites by using end-to-end fine-grained monitoring: An experimental study. In *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pages 338–341.

Vector (2018 (accessed May, 2018)). Vector. `https://github.com/Netflix/vector`.

Verge (2019). Facebook, instagram, and whatsapp were down for more than two hours. `https://www.theverge.com/2019/4/14/18310069/facebook-instagram-whatsapp-down-outage-issues`. Retrieved: Mar, 2019.

Verizon (2013 (accessed May, 2013)). Papers – 2013 data breach investigations report. `http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2013_en_xg.pdf`.

Wachs, M., Xu, L., Kanevsky, A., and Ganger, G. R. (2011). Exertion-based billing for cloud storage access. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 7–7, Berkeley, CA, USA. USENIX Association.

Wang, Q., Kanemasa, Y., Li, J., Jayasinghe, D., Shimizu, T., Matsubara, M., Kawaba, M., and Pu, C. (2013). Detecting transient bottlenecks in n-tier applications through fine-grained analysis. *ICDCS'13*.

Wang, X., Qi, Y., Zhang, C., Qi, S., and Wang, P. (2017). Secretsafe: A lightweight approach against heap buffer over-read attack. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 628–636.

Warp10 (2018 (accessed May, 2018)). Warp10. `https://www.warp10.io/`.

Whittaker, Z. (2012). O2 scraps ericsson database after second major network outage. www.zdnet.com/o2-scraps-ericsson-database-after-second-major-network-outage-7000005972. Retrieved Nov. 2012, from Northrop Grumman Information Systems.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Burlington, MA, 4 edition.

Xvfb (2015 (accessed May, 2015)). Xvfb. `http://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml`.

Yahoo (2019 (accessed April, 2016)). Webpage — Yahoo Query Language (YQL). `https://developer.yahoo.com/yql/`.

Yang, W.-p., Wang, L.-c., and Wen, H.-p. (2013). A queueing analytical model for service mashup in mobile cloud computing. *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 2096–2101.

Yuan, D., Park, S., and Zhou, Y. (2012). Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 102–112.

Zabbix (2019 (accessed June, 2019)). Zabbix. `https://www.zabbix.com/`.

Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. (2016). Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 603–618.

Zimek, A. and Schubert, E. (2017). Outlier detection. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 1–5, New York, NY. Springer New York.

ZipKin (2018 (accessed June, 2018)). Zipkin. `http://zipkin.io/`.

Zuul (2018 (accessed May, 2018)). Zuul. `https://github.com/Netflix/zuul`.