FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

IoT BattleBrokers: The Best Device Broker Wins

André Fontoura de Aguiar Pinto

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Costa Aguiar

Second Supervisor: André Filipe Gomes Duarte

April 17, 2019

IoT BattleBrokers:
The Best Device Broker Wins

André Fontoura de Aguiar Pinto

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Jorge Alves da Silva
External Examiner: João Paulo Barraca
Supervisor: Ana Cristina Costa Aguiar

April 17, 2019

Abstract

The Internet of Things and Smart City industry have witnessed an incredible growth over the past decade. A crucial element of these systems is the message broker, which acts as a bridge between the sensors and the platform. Given its importance as a point of entry and processor of information, it is necessary a solution that fits each project needs (ability to process a large amount of messages per second, to have a large number of connected endpoints sending/receiving information, ...) and that can handle projected loads for possible scenarios. Moreover, problems can arise with the broker which can carry a high price when dealing with a Smart City system's scale.
Since there is a wide range of solutions which have different behaviours under different situations, it is necessary to know how different message brokers behave under our specific project. Here comes in the interest in benchmarking these applications and analyse their performance under different scenarios.
Moreover, we recognize two message broker software architectures, monolithic or as microservices. For our purposes, only the microservices architecture will be considered, and when dealing with such architecture we see that there are different components which can bottleneck the broker and to identify such bottlenecks or points of failure is a time consuming task. The question then becomes, "can we automate the process of collection and analysis of metrics, in order to benchmark the broker's performance and identify bottlenecks within its architecture?"
To answer this question we defined a set of goals: (i) Identify a set of metrics to evaluate the message brokers microservices' performance; (ii) Develop a publish/subscribe simulation platform that enables customization and is capable of simulating smart city scenarios; (iii) Develop a platform that automates the collection and processing of information relevant to the performance analysis (as defined in 1).
By the end of this work we managed to identify a set of useful metrics to the analysis of a broker's microservices performance. It was also developed a simulation platform which can simulate publishers/subscribers using multithreading, allowing some degree of customization. To the collection and analysis of information it was developed a script to start capturing information from the broker's microservices and a parsing tool which runs through this information and extracts the metrics previously mentioned. This parsing tool facilitates the analysis of a broker's inner workings and can help understanding where bottlenecks lie withing its architecture.

Acknowledgements

I'll start by thanking both my supervisor, professor Ana Aguiar and André Duarte, for the invaluable help and guidance both gave me during this dissertation work, and especially the patience and availability when the deadlines closed-in. I would also like to thank deeply to the Ubiwhere team for the warm welcoming and fun environment lived during my stay with them.

I also want to thank my parents, my brothers and my sister for the continuous support and patience during this period. A full house bursting with activity sure helps during the more frustrating development periods.

Finally, I would like to thank all my friends that were always there, being to ear me complain about my frustrations with the thesis, being to ear my speeches explaining every detail of what this work was all about. Those moments were the best to step back and organize the ideas, when everything seemed a confusion.

"I love deadlines.
I love the whooshing noise they make as they go by."

Douglas Noel Adams

Contents

List of Figures

LIST OF FIGURES

Abbreviations

AMQP	Advanced Message Queuing Protocol
CoAP	Constrained Application Protocol
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
JWT	JSON Web Token
MQTT	Message Queuing Telemetry Transport
Pcap	Packet Capture
PcapNg	PCAP Next Generation Dump File Format
Pub	Publish
REST	Representational State Transfer
SSL	Secure Socket Layer
Sub	Subscribe
TCP	Transmission Control Protocol
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

This chapter introduces an overview of this dissertation, starting by the context and motivation alongside the problem. Thereafter the main goals are presented followed by a brief description of the structure of the document.

1.1 Context

The constant growth of the Internet has greatly affected the way we live and how we perceive information. This change has also made us explore new ways to interact with the environment surrounding us, and around the beginning of this century the concept of Smart Cities rose. A Smart City consists of an urban area which deploys various different technologies in order to achieve efficient management. This involves, but is not restricted to, networks of sensors and data collected from citizens, which is then analyzed and modeled in order to optimize management processes and city assets [CNW+12b].

To this network of *things* we associate the concept of Internet of Things (IoT), which is a key support paradigm to Smart Cities. IoT can be seen as an implementation of networking technologies in physical devices and objects (e.g., sensors, machines, cars and even buildings) creating an information network of these devices, which work together to achieve a common goal [GIMA10].

This dissertation is being conducted with Ubiwhere, which is is a Portuguese company based in Aveiro and created in 2007. The company efforts are split among two sectors, Smart Cities and Telecom, and this project is integrated in the Citibrain product.

This product focuses on offering urban solutions in the domains of transportation and mobility along with waste and environmental management. These are more concretely, *Smart Parking*, *Smart Waste*, *Smart Air Quality*, and *Smart Traffic*.

1.2 Motivation & Problem

Within many IoT architectures, one of the main component is the IoT message broker, and with *Citibrain* the same applies. The message broker acts as an intermediary platform that facilitates interaction between the sensors and the platform that processes said information. Currently, Citibrain is using Meshblu for that purpose. Furthermore, Meshblu also offers authentication of sensors and gateways [Sil16].

Given the variety of sensors and the scale of the network, the message broker should accept a variety of communication protocols and should be able to handle the expected load (messages per second, average message size, number of unique connected devices, ...) for the given scenarios. Since there is a wide range of solutions which handle different situation with different behaviors, it is necessary to know, of all of them, which is the best suited for the project's needs.

Currently this is a process of manually testing alternatives with the project's setting and scenarios, and usually the project stays with the first solution which minimally satisfies its needs.

Moreover, problems can arise with the broker upon encountering certain scenarios (for example, throttling or loss of messages), which in a real-time, smart city environment, carries a high price. These broker malfunctions, or performance drops are also hard to encounter before deployment, and even when they arise, the identification of the cause must be performed manually, being time consuming.

We should also consider that there are two ways of developing these message brokers, either using microservices or using a monolithic architecture. With monolithic applications, the identification and workaround of the problems mentioned above has to consider the broker a black box, and it is very complex to try and correct the problem within the broker itself, being more useful to simply switch solutions. With microservices' architectures we can narrow down the problems faced to specific microservices, identifying where a bottleneck or point of failure lies within the broker.

Given that the original solution used by the company for brokerage is developed using microservices, this type of architecture will be the focus of this dissertation. Moreover, there are benefits to consider when looking at the microservices' architecture, and further in this work we will overview its characteristics in order to understand how message brokers benefit from such implementation.

This situation is clearly not optimal, and the fact that there are barely any objective measurement criteria for message broker's microservices' performance evaluation in Smart City's scenarios only aggravates the problem.

1.3 Objectives

Research the benchmarking of IoT message brokers is relevant, both academically and business wise, given both the lack of concrete metrics to evaluate performance within a Smart City scenario, and the rising of the investment in Smart City initiatives. By undertaking this research, we hope to

define such criteria and develop a platform to evaluate how different brokers behave on different scenarios. Therefore, with the above stated problems in mind, the goals regarding the scope of this dissertation can be segmented as follows:

$\mathscr{G}_1$. Identify a set of objective measurement criteria to evaluate the message brokers microservices' performance in a Smart City scenario;

$\mathscr{G}_2$. Develop a client/device simulation platform that enables customization and is capable of simulating smart city scenarios;

$\mathscr{G}_3$. Develop a platform that automates the collection of information relevant to the performance analysis (as defined in $\mathscr{G}_1$), and that parses it in order to facilitate said analysis;

1.4 Document Structure

The rest of the document is structured as follows:

- Chapter 2, "Literature Review": explains core concepts of the technologies at hand and provides an overview of the topics surrounding IoT message brokerage, as well as the state of the art of benchmarking solutions;
- Chapter 3, "Message Brokers' Benchmarking": attempts to further narrow down the problems at hand, define the hypothesis to be tested, and propose a set of goals to attempt to aid in solving said problems. It is also proposed a solution to achieve these goals;
- Chapter 4, "Implementation": explains how the solution was developed and how it is built, from implementation details to instructions on how to run the platforms, as well as to add compatibility with new message brokers and protocols;
- Chapter 5, "Testing and Results": presents tests and results obtained with the built platforms, analysing them in order to demonstrate the functioning of the platforms, their results and usefulness of said results to the performance analysis of a broker and its internal components;
- Chapter 6, "Conclusion": concludes the work done during this dissertation, recalling on the Goals and Hypothesis to test, wrapping up with possible future work to improve upon what was developed.

Chapter 2

Literature Review

The scope of this work focus on the benchmarking of IoT message brokers as part of specific IoT paradigms applied to Smart City's implementation. In this Chapter we explore concept of Smart City, how IoT plays a role in its implementation and finally where the IoT message broker fits within the architecture. Afterwards a revision is made of the current message broker solutions in the market, and what is currently being done regarding performance analysis and benchmarking of those solutions.

2.1 Smart City

The increase in popularity of the concept *Smart City* comes from the need to mitigate problems generated by the urban population growth. With such growth an array of management problems emerge, such as difficulties in waste management, scarcity of resources, air pollution, traffic congestions, among others [CNW+12a]. The concept is used all over the world but very inconsistently, therefore we shall start by conceptualizing a smart city.

Briefly, a Smart City is a city which strives to make itself more efficient, sustainable, equitable and livable. And we can consider such city as an urban area that connects the physical, social, business and IT infrastructures [HEH+10] in order to monitor all of its critical structures – including water, power and waste management, transport infrastructures, communication services and even major buildings – and therefore optimize resources, plan preventive maintenance activities and monitor security aspects, all in all maximizing services to its citizens [HBB+00].

2.1.1 Smart City Components

With this definition it is possible to conclude that for a city to be "smart" simply having in place some sort of integration with networking technologies is not enough. With reference to the Figure 2.1 we can visualize the framework of interconnected aspects in what it takes to develop a Smart City.

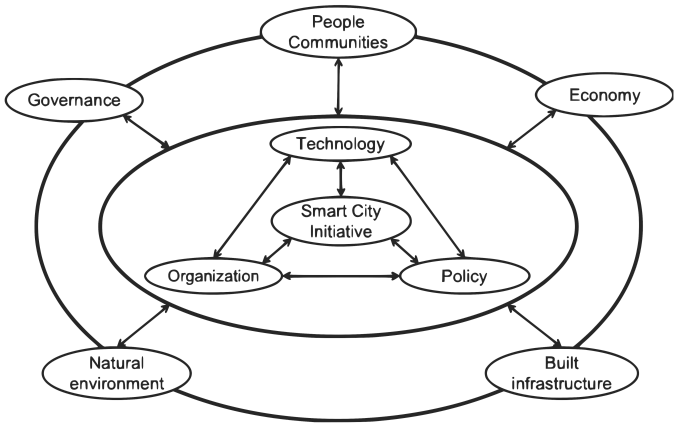

Figure 2.1: Smart City Initiatives Framework [CNW+12a]

We can use this graph to break down the components and better understand what is involved in each [CNW+12a]:

Organization Smart City projects face similar organizational and managerial as e-government initiatives, including, the project's size, the manager's attitudes, users diversity, lack of alignment of goals and conflicting goals and resistance to change. To these challenges, some strategies have been devised, as, having a skilled and experienced team, a well-respected IT leader, clear and realistic goals and planning, end-user involvement and end-user involvement [GGP05];

Technology As it was said, a Smart City relies on networking technologies in order to connect its infrastructure providing a better flow of information and therefore enhancing managing processes and resources. It also provides the platform to collect and analyze big data, which can be then used into better understanding a city's resource usage, environmental impact and bottleneck points in the natural flow of processes [RAPR16, Bat13];

Policy Being a socioeconomic development [NP11], for a Smart City project to succeed it needs also a development in the political landscape surrounding the application of said project. Policy frameworks supporting the evolution of the smart cities must be developed in order for the project to be successfully implemented, and the integration of political and institutional components is key into achieving that [Har05].

With these three core concepts understood it is easy to then see how the other surrounding components correlate:

Governance directly related to policy and organization, being it the component that both enables and benefits from these concepts;

People The implementation should be sensitive to the people and the communities it targets, since the key partners of a Smart City are the citizens themselves;

Economy The major driver of the initiative, being both enabler and incentive, prospering from the increase of efficiency and productivity [RFK+07];

Environment Being that one of the objectives of a Smart City is to increase sustainability, environmental concerns are to be taken into account when designing a smart city project;

Infrastructure Directly related to the Technology component, is the ICT infrastructure that enables the gathering, processing and proliferation of information.

2.2 Internet of Things

As seen in the previous section, one of the core components of a Smart City is the technology deployment. More specifically, the integration of networking technologies with other elements of the city. We define such approach as an implementation of the *Internet of Things* concept.

Semantically, Internet of Things means "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" [BH08], and although an accurate description, there are more concepts to understand in order to grasp the components and implications involved in the implementation of IoT paradigms in a Smart City scenario.

First off it should be defined what a thing is, which can be described as the participants in a process which are enabled to interact among themselves by exchanging information about the environment, while reacting autonomously to physical world events. From this we can better visualize the Internet of Things concept as a network of sensing and actuating devices which facilitate the flow of information through a unified framework [GBMP13].

2.2.1 IoT Perspectives

According to [AIM10] another approach is to split the IoT concept into three main perspectives which together build the IoT definition:

Things-oriented Which revolves around the concept of the *things* themselves and is the view that connects the physical world to the digital one (i.e. RFID, NFC, sensors & actuators);

Internet-oriented This view focuses on the adaptation of the Internet Protocol in order to connect *things*, moving from an Internet of Devices to the Internet of Things (i.e. IPSO, Internet 0);

Semantic-oriented With the predicted increase of *things*, new challenges arise with the storage, representation, interconnection and addressing of information. This perspective sees a solution to scalability with semantic technologies - as described in [TSH09], with (1) modeling *things*

descriptions, (2) reasoning over data generated, (3) semantic execution environments and (4) scalable infrastructure.

Using these definitions we can set a line of focus overlapping the Internet- & Semantic-oriented perspectives. This leads to smart semantic middleware, where this work fits in.

2.2.2 Smart Semantic Middleware

Things are heterogeneous in nature but at the same time should be seamlessly integrated into an information network. This is the role of the middleware, offering an abstraction to applications from the *things*, along with other services [BSMD11].

Based on the same work [BSMD11] it is possible to define the functional requirements of the middleware as:

Interoperability Being it network (definition of protocols to communicate between networks, independent of the message content), syntactic (format and encoding of the message) or semantic (rules to understand the information) interoperation;

Context Detection In order to be context aware - ability to characterize the situation of an entity (something relevant to the interaction user/application);

Security & Privacy Which is responsible for confidentiality, authentication and non-repudiation;

Managing Data Volumes Given the above mentioned increase in the number of *things*, it is important for the middleware to be able to manage large amounts of data exchange and storage.

These functional characteristics are handled by different modules of the middleware, as it can be seen in Figure 2.2, being one of them the message broker, the main subject of this work.

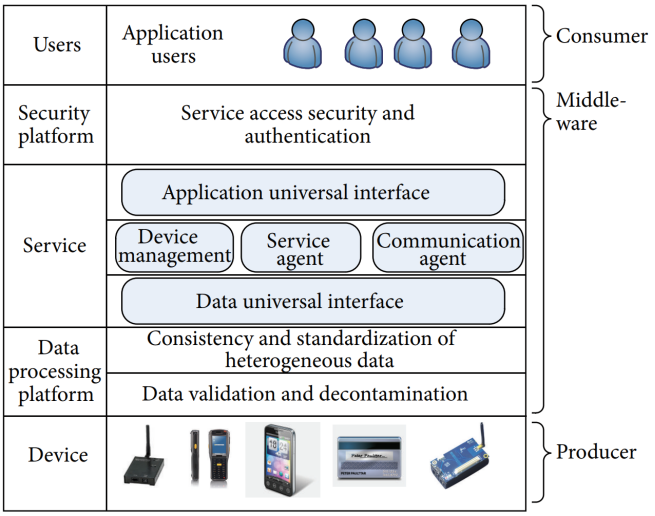

Figure 2.2: Middleware architecture, as seen in [WHZZ15]

2.3 IoT Message Broker

The message broker is a building block of the middleware. Its main function is to operate as a translator between the sensors and the rest of the middleware, taking in generated information in different formats and translating it to something the rest of the system can understand. It should be noted that in the context of this dissertation, the terms "message broker" and "device broker" are used interchangeably.

In the IoT context a broker is regularly deployed using the Publish-Subscribe messaging method, since it fits well in the IoT architecture [HKM+17].

Publish-Subscribe Messaging Flexible and asynchronous messaging system where the application that produces information publishes it on an intermediary platform and the applications that consume said informations subscribe to it. The intermediary platform has the responsibility of delivering the published messages to the subscribing entities, and usually includes the ability to transform messages, allowing different applications to communicate even if not designed to work together.

To satisfy the given definition of a message broker - and bearing in mind the IoT context -, the broker then admits some functional characteristics, as described below [Kal14]:

Routing Services Taking care of the flow of messages, to one or more destinations - given the pub/sub method, this can be seen as handling the flow between publishers and subscribers;

Message Transformation The broker can interpret and translate the messages from the varied sources enabling the subscribing applications to understand the information;

Repository Services Repository to house information on rules, logic, objects, and metadata on target or source applications;

Authentication Services For security reasons the broker should be able to filter messages based on the source, only accepting authorized publishers to communicate [GPM16].

2.3.1 Message Broker Architecture

Message brokers can be developed either using a monolithic or a microservices architecture.

Microservices Architecture Software architecture where a single, complex, application is divided into smaller, more manageable components. These components, referred to as services, are organized around capabilities bounded by context, are independently deployable, and interoperate though message-oriented communication [NMMA16]. This software design enforces modularity which allows the development of more scalable applications, more flexible technology choice, independent development and deployment of services and therefore, continuous delivery [SLM17].

Monolithic Architecture Software architecture where functionally distinguishable aspects of the application are all interwoven into a single platform. With this architecture is easier to handle cross-cutting concerns and also uses shared-memory which offers performance advantages over inter-process communications. Moreover, it offers more transparency and abstraction of the underlying mechanics of the application.

For the sake of this dissertation, it is made the assumption that the message broker is developed using a microservices architecture, since the problem was formulated from an environment where the message brokers used were developed using such architecture. Many brokers are designed in this fashion given its benefits, as described above. Figure 2.3 attempts to roughly schematize a broker's internal architecture, with the basic elements we considered relevant to benchmark.

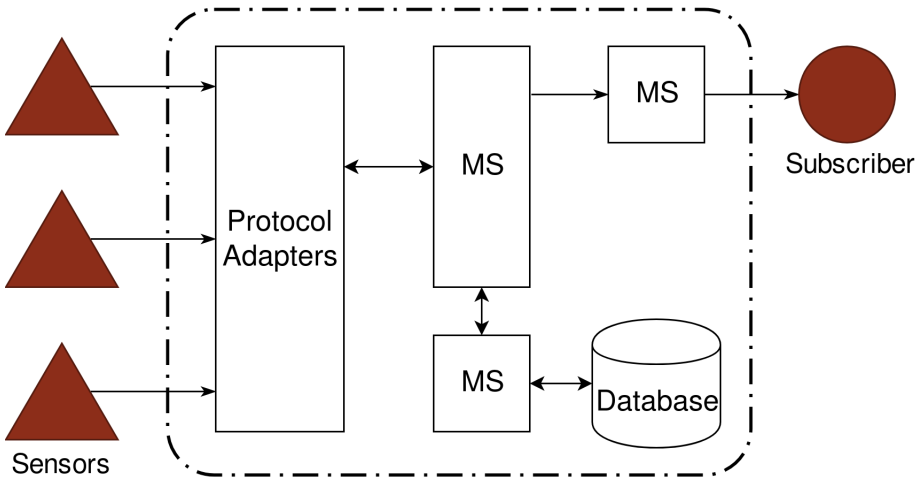

Figure 2.3: Architecture schematization of a microservices based message broker.

2.3.2 Microservice-based Brokers

If we try to analyze Meshblu's architecture into the one presented in Figure 2.4, we can say that: it makes use of a set of protocol adapters that receive the message from the gateways in a determined protocol (i.e.: MQTT, CoAP or HTTP), it then uses Redis to manage the inner communications of the broker and Mongoose to communicate with the database which is on MongoDB. The interested applications should then subscribe to the Meshblu Firehose. This analysis allows to understand how heterogeneous the implementation of the broker can be, and that analyzing its performance as a whole is not enough to understand the results measured.

For the purpose of comparison we can also consider DeviceHive's architecture, presented here in the image 2.5. Its architecture is also structured with microservices, having components dedicated to receiving the requests, processing them and storing the information.

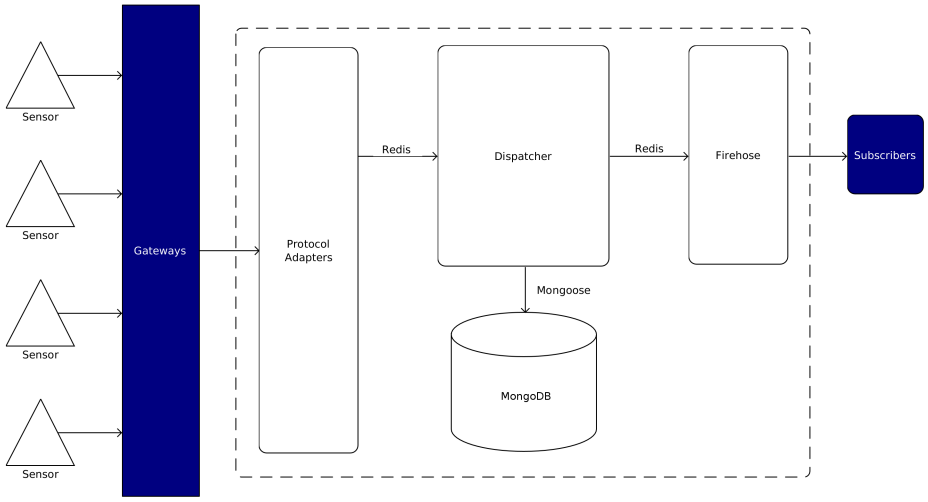

Figure 2.4: Rough schematization of Meshblu's architecture.

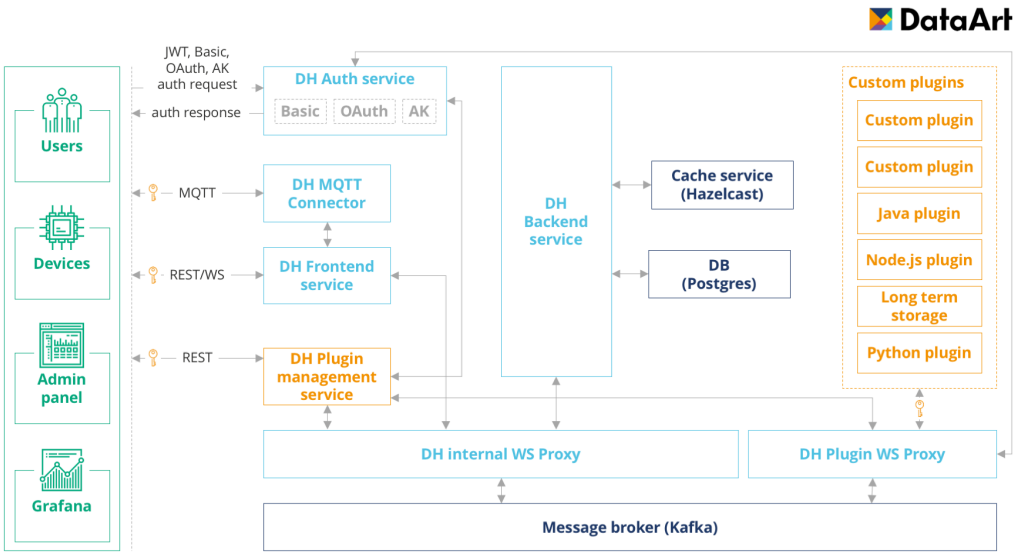

Figure 2.5: DeviceHive architecture's diagram, as made available in its documentation.

2.4 IoT Message Broker Benchmarking

With the growth of IoT systems new attention is increasing towards optimizing said systems. The message broker, being such key element of many IoT systems, is then under attentive analysis, and many works are starting to come out focusing on the metrics for evaluating its performance.

2.4.1 Meeting IoT platform requirements with open pub/sub solutions

In the work [HKM+17] the writers start by considering the main requirements of a IoT platform when deploying a Pub/Sub solution and proceed to benchmark available middleware which satisfies those requirements.

The writers start by setting three reference scenarios - Social Weather service, Smart car sharing and Traffic monitor - then, deriving from generic IoT applications requirements illustrated by those scenarios, the writer present a set of functional requirements as follows:

Messaging Pattern Motivation to use the pub/sub pattern and also support of a point-to-point messaging pattern;

Filtering Interested parties want to receive only a subset of all information. A topic-based filtering is mandatory and a content-based scheme is highly desirable;

QoS Semantics The middleware should enable annotating subscriptions and messages with QoS requirements and provide subscribers with the latest value while waiting for the next sensor reading;

Topology It is assumed that most cloud-based solutions use a hybrid approach which matches producers and consumers using a centralized broker;

Message Format The Pub/Sub solution must be payload agnostic and support binary payloads.

In addition to these requirements, the writers also highlight two non-functional requirements: (i) the system should be scalable and (ii) messaging should have low latency. To evaluate these two requirements, the writers also defined two performance metrics, starting with the throughput of a single broker (number of messages per second a broker can process) and then the end-to-end delay between gateways and the subscribed nodes.

This work is obviously broader, and we can use the Figure 2.6 to better visualize the architecture and understand how the objectives differ relative to the subject of the performance evaluation. The work cited considers the whole *Cloud Tier* when performing the benchmarking, and in this work the broker is discriminated against other middleware components, only being evaluated the interaction between the *Gateway Tier* and the *Cloud Tier*.

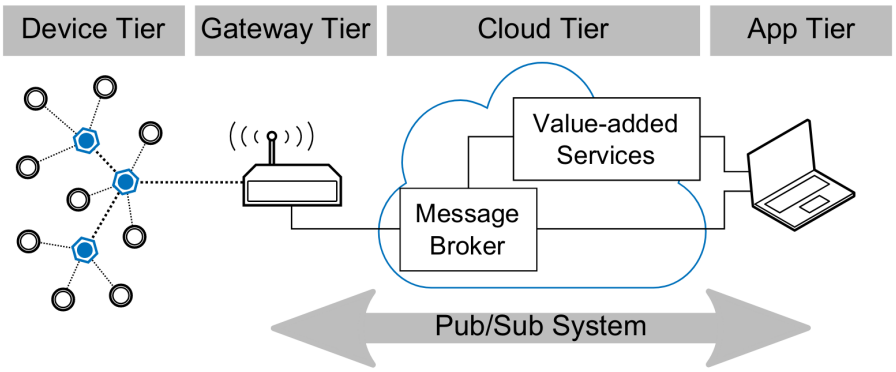

Figure 2.6: General architecture of cloud-centric IoT systems [HKM+17]

It is nonetheless useful, as it defines some general requirements for the middleware, which can be considered in the evaluation of the broker itself.

2.4.2 Benchmarking Pub/Sub IoT middleware platforms for smart services

Another work intimately related to performance and benchmarking evaluation of IoT middlewhere is [PCAM18]. In this work the writers focus more than the previous on the benchmarking of the middleware on a Smart City context and focus more on the performance side of the problem, developing a set of attractive metrics in regards to the purpose of this dissertation.

During this work the writers defined a set of both qualitative and quantitative dimensions to compare middleware solutions, having in mind a smart city context. Starting with the qualitative dimensions for requirements analysis and ease of use:

- Support for the desired communication model (pub/sub is the typical in such context);
- IoT application requirements (as in [B+12]);
- Viability and limitations in each scenario;
- Quality of the documentation and support.

And to evaluate the performance, the writers measure the speed and efficiency of the middleware, with:

- **Publish time:** time between sending the publish request and receiving the response;
- **Subscribe time:** time between sending the publish request and receiving the subscribe notification;
- **Total time:** time between sending the first publish request and receiving the last publish response;
- **Size of marshalled data:** the content-length header of the application protocol;
- **Size of publication:** size of the payload of the transport protocol of the publish packet;
- **Total amount of data used to publish a resource:** sum of the network level sizes of all packets exchanged;
- **Goodput:** size of the serialized data divided by the publish time.

Some auxiliary metrics were also defined in order to infer if an increase in the pub/sub times were related to an increase of network congestion or to reduced broker performance, and to relate the increase in the total time to publish all data to an increase in the number of retries, packet re-transmissions, or both. These metrics are:

- Number of request retries in cases when the publish fails due to the broker;
- Round trip time to broker;

- Number of re-transmissions of transport and application transport packets and the delay verified for that.

In this work the writers then benchmarked FIWARE and OneM2M according to these metrics. One of the conclusions that is interesting pointing out is that the researchers were able to find that the broker performance can depend on different components like the database and the protocol used.

2.4.3 A Modular Tool for Benchmarking IoT Publish-Subscribe Middleware

Lastly we should take a look at Luís Zilhão work *A Modular Tool for Benchmarking IoT Publish-Subscribe Middleware*. This work also rose from similar motivations, and it is the implementation of the metrics defined in [PCAM18] as a benchmarking platform for IoT middleware. Given that this work's objective also lays with the evaluation of brokers' performance, it is correct to see the current dissertation as a more deepened approach, expanding the broker and considering its internal implementation and microservices.

2.5 Conclusions

Although becoming a focus topic on IoT development, broker benchmarking is still a problem when opting for different brokers for implementing a Smart City system. Despite the shortcomings in [ZMA18], the author designs a platform that can be used as a base for automated benchmarking of various brokers. It describes a modular platform, as to facilitate integration of new brokers or different protocols, and collects relevant metrics, as proposed in [PCAM18]. However, such work considers the broker as a blackbox and is oblivious to its internal implementation. This allows a comparison between different brokers and the detection of problems with a broker under certain situations, but falls short on identifying the cause of such problems and where, in the broker's implementations (considering a microservice architecture, as previously mentioned), the problem originated.

Information such as this can, and is, useful in cases where changing the current solution for brokerage is too expensive. Moreover, such information is also useful to the broker's developers themselves, as it allows to see witch service bottlenecks the application, or where more load on the network is being produced.

With this work we hope that a platform can be designed and deployed where the benchmark is made at the level of those internal components, aiding to answer not only which broker performs better but why and what can be adapted to improve the broker's results.

Chapter 3

Message Brokers' Benchmarking

As it was seen on the last chapter, despite the current lack of works on the field of benchmarking and automated benchmarking of IoT message brokers, it is a topic of growing interest and works are being conducted on the the subjects of benchmarking criteria and metrics and of manual benchmarking specific brokers with scenarios considered useful for the researchers. It was also investigated the possibility of having an unified platform evaluating some of the defined metrics, therefore having comparable benchmarking results, obtained through an automated process.

Based on what was concluded during the Literature Review, in this Chapter we attempt to thoroughly describe the problems in question, as well to define the scope of the dissertation concluding on the hypothesis to be tested and goals to achieve.

3.1 Thesis Framing and Statement

First it should be stated the set of beliefs that the author holds. These will frame the research goal of this dissertation and serve as boundaries of the research topics.

The base belief is that the microservice's architecture is the most used and best suited for the development of IoT message brokers, which, given its modular nature, facilitates scalability and allows continuous delivery. The author also believes that there is added value to the developers in the analysis of the communications between microservices within a broker, not only when using a specific message broker while developing an IoT product, but also when developing a message broker itself. Furthermore, the author believes that the metrics described in [PCAM18] are valid and appropriate for benchmarking IoT message brokers. Finally, it is assumed that IoT

message brokers are heterogeneous among themselves, and that their usage and architecture is greatly diverse.

The above statements won't, therefore, be proven or analyzed within this dissertation work. Instead, the author assumes the usefulness to developers in the analysis of some of the metrics stated in [PCAM18] adjusted to the microservices that make up a message broker. It is also assumed that the IoT message brokers are very diverse and therefore the benchmarking platform that is to be developed will not need to accept all message brokers, instead aiming at facilitating the addition of more brokers and communication protocols.

With the above premises in mind the fundamental question to be approached with this dissertation becomes:

"Can a platform be developed that partially automates the performance analysis of an IoT Message Broker's microservices while being more efficient than a manual analysis?"

This question can then be decomposed into a set of more specific problems, in order to facilitate reaching an answer. It should also be defined a base of comparison, as to understand how a developed solution may compare to another methods of reaching the same results.

3.1.1 Current Challenges

First we should overview some challenges in IoT message brokers' Benchmarking, and how these challenges will be considered during this dissertation, framing the hypotheses and objectives formulated. The main challenges identified in this topic can be split into theoretical, relative to the criteria to be used, and technical, relative to technological constraints, and from this we have the following issues, relevant to this work:

1. **What criteria should be used to evaluate the performance of a broker?**
   This is the base challenge in the subject of IoT message broker's benchmarking. Specifically in the context of this dissertation, what criteria should be used to evaluate the performance of a broker's microservices?

2. **Which metrics should be collected and analyzed, and how?**
   Based on the criteria identified, what metrics do we need to measure? What metrics would be relevant in the comparison of solutions and in the identification of bottlenecks within said solutions? And after identifying those metrics, how should they be measured and collected, with as little impact as possible on the performance of the broker?

3. **What conditions should be met to fairly compare brokers?**
   Since one of the main goals is to be able to identify which is the best brokerage solution to a given scenario, how can a fair comparison be assured? More specifically, what can be done around a platform to assure equal circumstance?

4. **How should the benchmarking scenario be simulated?**
   After defining the criteria and the relevant metrics, the question becomes how to simulate the scenarios. This encompasses not only the technologies to employ, but also the options

open to customization as well as the method to validate a scenario as an accurate simulation.

5. **How can the collection and representation of metrics be automated?**
   Finally, can a platform be built in order to facilitate this analysis and comparison? Does said platform produce results close enough to reality that can be considered useful? And does said platform provide results in less time, or with less expense, than testing s solution manually?

3.1.2 Validation Methodology

To validate the results obtained and how we answer the fundamental question of this dissertation, we must first define the dimensions which require validation.

First the chosen metrics and criteria to use must be considered useful. This is only briefly covered, since it is already considered that the criteria described by [PCAM18] are valid for the benchmarking of message brokers, and need only to be adapted to cover a more in depth analysis that include its microservices.

Second, it should be evaluated if the developed platform precisely simulates the intended testing scenarios. Such analysis encompasses not only how the simulation is designed but also what technologies are used and deployed to mimic the devices and the sending of messages. Therefore, this validation will be made alongside the developers at Ubiwhere with an analysis of the end product and how it behaves, comparing to a real system.

3.2 Hypothesis and Thesis Goals

Taking into consideration the above questions and the final objective of this thesis, we can then decompose the dissertation into the following hypothesis to be tested:

$\mathscr{H}_1$. It is possible to automate the process of collection and parsing of information in order to obtain the defined benchmarking metrics;

$\mathscr{H}_2$. A simulation platform can be developed in order to simulate smart city scenarios;

$\mathscr{H}_3$. The addition of compatibility with new message brokers or communication protocols can be made simple.

In order to test these hypothesis, during the conduction of this study a platform should be designed and developed which will encompass these problems and attempt to solve them. Therefore the following goals were defined (as previously mentioned):

$\mathscr{G}_1$. **Identify a set of objective measurement criteria to evaluate the message brokers microservices' performance in a Smart City scenario.** Which will use as a base the criteria in [PCAM18] and adapt to the evaluation of the communications between the microservices;

$\mathscr{G}_2$. **Develop a platform able to simulate smart city scenarios.** Said platform must be a simple client simulation platform that enables customization to the point where an user can simulate a smart city scenario. It must also enable simple integration of new brokers or new communication protocols.

$\mathscr{G}_3$. **Develop a platform that automates the collection of information relevant to the performance analysis, and that parses it in order to facilitate said analysis.** Such platform should bear low impact on the performance of the message broker, be generic enough as to cope with as many brokers as possible, and should (as the simulation platform) have an easy integration of new communication protocols.

$\mathscr{G}_4$. **Provide a solution to the case at hand, *Citibrain*.** Finally, we should be able to help providing a solution to the *Citibrain* problem, being a new broker alternative or an adaptation of the current solution in order to remedy the situation.

3.3 Proposed Solution

With the above goals in mind, the dissertation will now focus on proposing a solution to achieve them, and hopefully answer weather the raised hypothesis are valid or not.

3.3.1 Measurement Criteria

Starting with Goal 1 of this thesis, as it is the base of the rest of the objectives, the criteria to measure the performance of the broker should be defined. For this, we will analyze the proposed metrics in [PCAM18] and see how they suit the evaluation of the microservices.

The review of the criteria and metrics defined by the authors of said article has already been covered in Chapter 2 with some detail. From there we can state that a qualitative analysis of the broker won't represent much weight in this evaluation, since the focus will be on the internal implementation of the broker. That doesn't mean this dimension will be completely ignored along this work, but conclusions about a broker's characteristics relative to these criteria won't be tested or proven as an objective of the thesis.

That said, the solution will focus on the quantitative dimensions to evaluate the performance of a given broker. As a review, the authors of the article propose these metrics:

- Publish time
- Subscribe time
- Total time
- Size of marshalled data
- Size of publication

- Total amount of data used to publish a resource
- Goodput
- Number of retries in case of publish failure
- Round trip time
- Number of re-transmissions of transport and application transport packets and the delay verified for that

The analysis platform is planned to take into account all of these metrics, and relative to its microservices it is planned to measure:

Response time: the time it takes, after sending a request, to receive the response from a microservice;

Request and response sizes: the packet sizes of some relevant requests and responses transmitted between microservices;

Failed Requests and Responses: the number of requests that didn't receive a response or lost messages between the microservices.

Transmitted bytes: the total number of bytes transmitted from and to a microservice along a simulation;

CPU and RAM usage: the computational resources used by a microservice, also along a simulation run.

It is believed that these metrics will enable identification and analysis of possible bottlenecks within a broker's implementation, and test points of failure.

3.3.2 Scenario Simulation

Focusing now on Goal 2 of this thesis, developing a platform able to simulate smart city scenarios, it was planned to develop a platform that allows customization up to a point where a Smart City scenario can be recreated by the user.

For the purpose of this thesis, we consider a Smart City scenario, a scenario where there are many publishers connected, publishing notifications in random intervals of time. These publishers, or sensors, are organized into categories and the message size varies according to the type of sensor, up to around 1MiB including headers.

Since the objective involves having different protocols and brokers, while also facilitating the addition of new ones, the planned platform for the scenario simulation consists of an interface which then connects the available communication protocols. This way, the user just needs to code a new protocol in, and the interface automatically integrates it to the platform.

In regard to different brokers being compatible with the simulation platform, it is planned to have the necessary information about a broker in configuration files, and the main interface will fetch them, allowing the user to select which is the target broker and the protocol implementation to fetch the necessary information in order to simulate the scenario. This configuration file should have, about the broker, information such as: message structure, needed information from responses and general information about the broker as to establish the connection.

About the simulation itself, in order to approach a more fair comparison and to more thoroughly test the broker, each run shall start by creating all the needed devices and clients, fetch them, and then start the sending the notifications. This way the starting point for the comparison is similar and it will also be testing authentication features and the registering of new devices and clients.

It is planned to allow the user to customize the number of clients, the number of subscriptions per client, the number of devices sending notifications, the time interval between notifications and the payload size of the notifications. The simulation platform will also output some information right away, such as the number of sent notifications, the number of received notifications, the time between sending a notification and receiving it on the client and finally the number of lost or failed notifications.

3.3.3 Metrics' Collection and Analysis

For the Goal 3, being the development of a platform that automates the collection and analysis of relevant information for the benchmarking of a broker, the choice has been made to first capture the data relevant for the analysis and parse it only after the simulation. This choice was made in order to lower the impact that the analysis tool has on the broker's performance.

This choice divides the platform into two components: the capturing component, that starts everything necessary for the capturing of information, and the parsing component, that should extract the information needed from the captured data.

The capturing component should be straightforward. Since there are already available different solutions for network analysis and packet sniffing, the only thing it should be able to do is to start capture instances for each microservice and note relevant information for identification of captures. It should also attempt to extract the logs produced during the run, since these logs usually have useful information about what is going on with each component, specially if a problem arises. It is also planned to collect data through already available monitoring tools, which measure the CPU and RAM usage of different components, as well as data transmitted from and to microservices.

The parsing component will then be responsible for extracting the relevant information from the captured data and display it in a more readable manner - either by a graphic representation or simply highlighting a summary of the observed values, such as averages, maximums and minimums. Similar to the simulation tool, this component is planned to have a main interface which then links parsers for different protocols, and just like the simulation platform it will also rely on configuration files for broker specific information. This way it is hoped to have the addition of compatibility with new protocols or brokers facilitated to the user.

This parsed data will then be represented alongside the data collected with the monitoring tools mentioned above, this way facilitating the identification of bottlenecks and points of failure. The additional information provided by the monitoring tools not only complements the parsed information from the captures but is also used when the communications between the microservices are not easily associated with requests/responses and therefore the platform will rely mostly on analyzing the amount of data transmitted between them.

3.3.4 Citibrain's Use Case

With this information, and the analysis made by the platform it is hoped that a user can, given a set of brokers and a specific scenario, reach a conclusion about what broker is best suited to handle the target scenario. The user must also be able to understand where, in the internal implementation of the broker, a problem might arise and identify possible bottlenecks relative to the microservices of the broker.

This will also be tested by attempting to aid in answering Citibrain's question as to what is the best brokerage solution for the project, and where might be a problem when the current solution, Meshblu, fails to handle a given situation.

For reference, we can use Figure 3.1, where the Citibrain platform is structured, to visualize what will be the focus of this thesis within a more composite project.

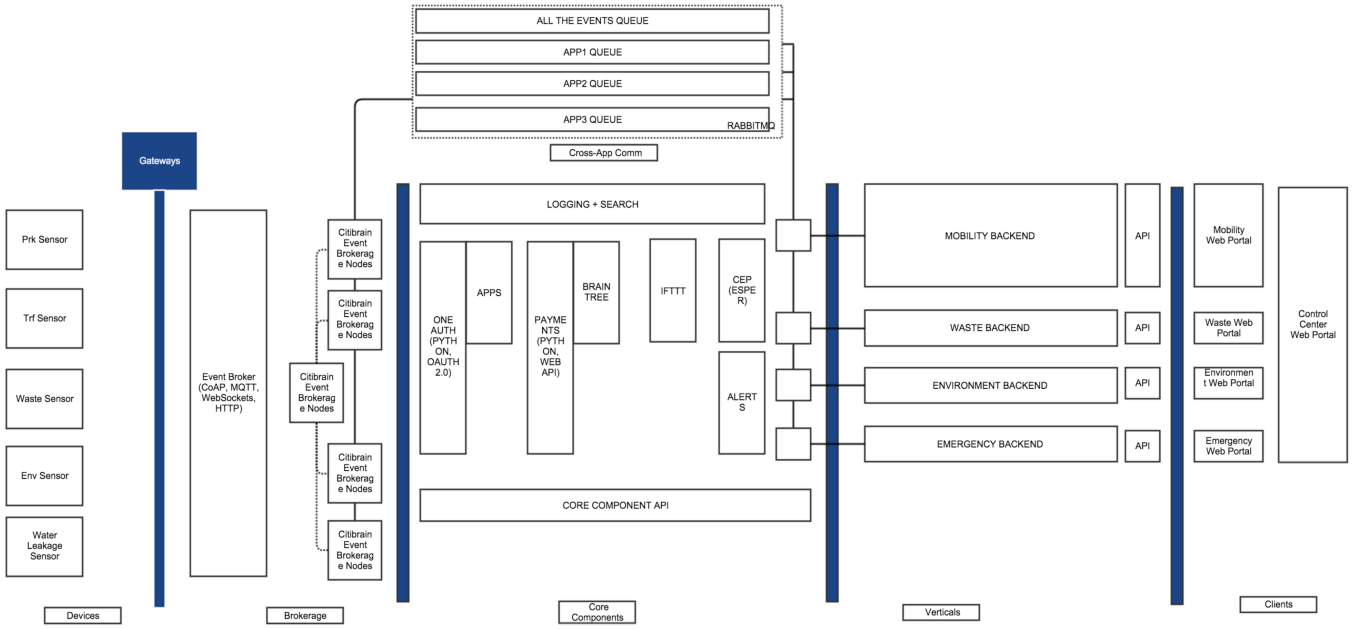

Figure 3.1: Architecture of the Citibrain platform as depicted in [Sil16]

3.3.5 Message Brokers

During the development of the platforms two brokers were used as test examples, Meshblu and DeviceHive. Two brokers were used in order to test the interoperability of the platform during the

development, and after making a feature function with a broker it was then tested with the other broker. This way the platform was developed not specifically for a broker but it was also taken into account how the addition of compatibility with new brokers was impacted.

3.3.5.1 Meshblu

Meshblu was selected mainly because it was the solution Citibrain was using for its brokerage. The overall architecture of this broker was already exposed in Chapter 2. The way Meshblu is organized to handle different communication protocols is by providing a microservice for each different protocol, to which it calls *protocol adapters*. Right now there are available adapters for HTTP, Websocket, MQTT, CoAP, XMPP and AMQP. Security-wise, Meshblu uses an *UUID* and a *Token* for authentication. It should also be noted that Meshblu uses MongoDB as database, which communicates using the *MongoDB Wire Protocol*.

HTTP The API endpoints require the security credentials mentioned above passed in the HTTP headers. These credentials are obtained by registering a node (device or user) via a POST to /devices, afterwards, a device can publish notifications with a POST request. When a client subscribes via HTTP, an HTTP Stream is opened and the API returns the relevant messages as they are sent/received.

Websocket To establish a connection via Websocket, the node should first have the credentials mentioned above, which can be obtained with a "register" message, afterwards, the node must use those credentials by sending an "identity" message containing the UUID and the Token. Messages sent and received by a Websocket connection are JSON arrays with the following structure: [type, data], where "type" is a string with the topic of the message (like the mentioned "register" and "identity" topics) and data is a JSON Object with the payload.

CoAP Very much like the HTTP API, the device must register with a POST to /devices, and use the credentials obtained in the header of the messages sent. To publish messages, the node must also use a POST request, and when subscribing, the CoAP adapter has various streaming APIs under /subscribe.

MQTT Meshblu's MQTT adapter is incomplete and doesn't support the registration of new devices into the platform. Because of this another protocol must be used for that purpose (like Websocket or HTTP) and with the returned UUID plus the Token, we can start publishing messages or subscribe to topics. Another particularity of Meshblu is that when subscribed to a device, the client will not receive messages sent by that device, instead receiving the messages received by the device. Because of this, the publishing of new messages by a device are made to itself. The publish topic for identification of a node is "identify" and to publish a messages the topic is "message", moreover, to subscribe to a device, a node must send a subscription request with the target device's UUID as topic.

Meshblu also has a microservice named Firehose, that supports AMQP, Socket.io or XMPP, and is responsible for delivering messages to the devices. As it states in Meshblu's documentation: "In order to receive messages, a device must connect and authenticate with the Firehose. Once connected, the Firehose will immediately start streaming down all messages that the device is subscribed to.". In this thesis, though, this component won't be taken into consideration since none of protocols mention were yet included in the platform.

Although manageable to understand, as of right now, Meshblu lacks on a good documentation. What is provided is not complete, and for some protocols the API is not very well described nor well established, making this broker hard to pick up and test on.

3.3.5.2 DeviceHive

DeviceHive was selected as a test Message Broker because of its simple deployment and extensive documentation, which made it a good broker to use as subject. Its architecture was also mentioned earlier, and also uses protocol adapters to be compatible with different protocols, and although offering a set of plugins and a plugin manager, those components will not be taken into consideration for this work.

Although not represented in the architecture above, when deployed, DeviceHive has a proxy microservice responsible for receiving all communications which it then redirects to the appropriate adapter. Currently, DeviceHive supports HTTP, Websocket and MQTT communications, and for security uses JSON Web Tokens (JWT). Authentication is handled by a separate microservice, and requests get redirected to the "DH Auth service".

HTTP Before starting, a node must first either login with a *username* and *password* or have a JWT created on its behalf. The node must then use that JWT along with the messages it sends to the broker. Unlike Meshblu, the client must poll the broker in order to get new messages, and there is no streaming connection available via HTTP. For this, the API offers a "poll" method which returns all notifications past a specified timestamp. In case no notifications are found, the method blocks until it receives a new notification or until a timeout is reached, in which case it returns an empty response. To publish a new notification, the node must send a POST request to /device/deviceId/notification and it should be noted that notifications can be posted on behalf of a node.

Websocket To establish a connection via Websocket, a node must, after connecting, authenticate using a JWT. This can be done by sending an "authenticate" request with the token. Again, notifications can be made on behalf of the devices, and follow the structure in Listing 3.1. The "action" parameter in the message dictates the type of message, in this example, "notification/insert" to publish a notification. A node can also subscribe to notifications by sending a "notification/subscribe" message, and if successful, the server will start streaming publishes regarding that topic.

```
{
    "action": "notification/insert",
    "requestId": {object},
    "deviceId": {string},
    "notification": {
        "notification": {string},
        "timestamp": {datatime},
        "parameters": {object}
    }
}
```

Listing 3.1: DeviceHive's publish request using a Websocket connection.

MQTT DeviceHive also offers compatibility with MQTT, with which a node can connect, and after authenticating with a JWT, send publish or subscription requests. To make a request to DeviceHive, a node should publish the request to the "dh/request" topic with the target "action" (first line in Listing 3.2). To receive responses, the node must subscribe to the response topic, following the structure "dh/response/request-action@client-id". In the second line of Listing 3.2 we can see this adapted to the "device/list" example, where the "request-action" is "device/list" and the "client-id" is the MQTT client own id. Finally, we can see in the last two lines of Listing 3.2 an example of a notification publish and a subscription to that notification's topic.

```
mqttClient.publish('dh/request', '{"action":"device/list"}');
mqttClient.subscribe('dh/response/device/list@mqttClientId');
mqttClient.publish('dh/notification/network1/devicetype1/device1/
    temperature', notificationObject);
mqttClient.subscribe('dh/notification/network1/devicetype1/device1/
    temperature');
```

Listing 3.2: DeviceHive's MQTT publish and subscribe examples.

3.4 Support Technologies

3.4.1 Docker

cker is a computer program that performs containerization, which is a feature where the kernel allows the existence of multiple isolated user-space instances. These instances, also called containers, bundle their own applications, libraries and configuration files and are created from "images" which specify their exact contents. With figure 3.2 we can more easily understand how a container runs in a host, and compare it to a virtual machine. Instead of deploying a complete operating system running with the hosts resources, a container shares the kernel of the host machine and only

isolates its user-space. Containers can also be replicated and deployed across machines, making a containerized application easily scalable.

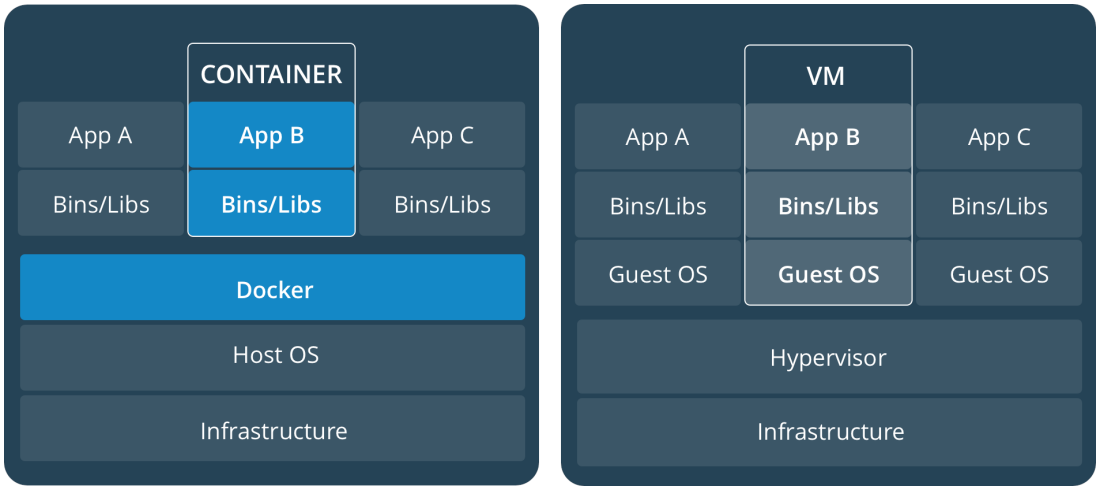

Figure 3.2: Docker containers compared to Virtual Machines (as in Docker's documentation)

These characteristics make docker containers lightweight, and not only that, facilitate the deployment and scalability of applications. Since a container packages all application's code and dependencies, it is a stand-alone executable that is very independent of the infrastructure making it portable and easy to deploy and run. Besides, even if an application is not built with docker, one can create an "image" to run said application in a container as long as the code is available.

Docker also facilitates when an aplication runs on multiple containers (in the case of a microservice architecture, for example), offering "Compose" which uses a YAML file to configure the applications services (usually refered to as "docker-compose.yml"). Then, with a command similar to "docker-compose up" all the containers are built and ran.

All this makes containerization a good strategy when deploying message brokers developed using a microservice architecture, which usually are applications that require flexibility and scalibility, depending on the scenario.

3.4.2 Dumpcap & Pyshark

Both these tools are used in network analysis. Dumpcap is a tool included with the packet analyzer *Wireshark* and according to its documentation: "Dumpcap is a network traffic dump tool. It lets you capture packet data from a live network and write the packets to a file.", the file it writes to being in the *pcapng* format by default. Dumpcap allows to select a network interface to use for the live capture with the "-i <capture interface>" option, the default snapshot length (maximum size of a network packet that is saved to disk) with "-s <capture snaplen>" and the file to save the data to with "-w <outfile>". For the purpose of this work, these three options are the only ones that are relevant.

Pyshark is a wrapper for tshark which allows the parse of a capture file using Python. Tshark is a terminal version of Wireshark, designed to both capture and display packets, while also being

able to parse saved capture files. As an example of how Pyshark accesses and displays the packets we can refer to the code in 3.3. *FileCapture* is the method used to read from a saved capture, and the parameter *decode_as* is a way to tell tshark to decode protocols in situations it usually wouldn't. The last three commands show how data in the packet is accessed.

3.4.3 cAdvisor + Prometheus

cAdvisor is a tool that collects and exports information about running containers on an host and for each container it exports historical resource usage and network statistics. It runs on the background and for the purpose of this thesis it can export CPU and RAM usage for each container as well as the amount of data transmitted in and out of a container.

Prometheus is a toolkit that records real-time metrics in a time series database, and cAdvisor exposes container statistics as Prometheus metrics out of the box. This bundle allows us to collect metrics related to the containerized microservices using an HTTP API, reachable under /api/v1.

```
>>> import pyshark
>>> cap = pyshark.FileCapture('mycapture.cap', decode_as={'tcp.port==27017':'mongo'
    })
>>> cap
<FileCapture /tmp/mycapture.cap>
>>> print cap[0]
Packet (Length: 698)
Layer ETH:
        Destination: aa:bb:cc:dd:ee:ff
        Source: 00:de:ad:be:ef:00
        Type: IP (0x0800)
Layer IP:
        Version: 4
        Header Length: 20 bytes
        Total Length: 684
        Identification: 0x254f (9551)
        Flags: 0x00
  ...

>>> packet = cap[1]
>>> packet.ip.dst # By protocol attribute
192.168.0.2
>>> packet[2].src # By layer index
192.168.0.100
```

Listing 3.3: Example of using Pyshark to read a capture file.

3.5 Conclusion

After deepening the understanding of the problem at hand we identified the main questions to answer with this dissertation: (i) what metrics to use in the benchmarking, (ii) how to collect and analyze these metrics, (iii) what are the conditions for a fair comparison, (iv) how should the scenario be simulated, and (v) how can this process be automated. With these questions in mind, a set of goals was set and a solution designed based on those goals.

This analysis built a foundation of the work to develop and slices the main question into more manageable problems. Namely, (i) the identification of the metrics to use, which we already defined in this Chapter, (ii) the development of a customizable simulation platform that can be used to recreate Smart City scenarios, which was presented an overview of a solution to develop, (iii) the development of a platform to collect and aid in the analysis of the metrics defined, which should bear low impact on the test and produce better results than manually testing the broker, and finally (iv) use the platform to benchmark different brokers in order to test the results obtained and attempt to help reaching a solution to the Citibrain's problem.

Chapter 4

Implementation

In this Chapter the solution developed to answer the problems described in Chapter 3 is further described as well as the development process of said solution. This description will also aid in understanding how the addition of compatibility with new Message Brokers and communication protocols is achieved.

As to understand some of the design decisions made during the development of the solution, we will overview the development process, and how the difficulties and challenges that arose were dealt with.

4.1 Simulation Platform

The main broker used for the development of the platform was DeviceHive, given its comprehensive documentation and ease of use, and the work for the simulation platform began with developing a simple Websocket client that created both a device and an user. The device would send notifications and the client would subscribe to the device's notifications. This was initially used mainly to understand how the broker communicates internally and to generate basic material upon which the development of the Parsing Tool would work on.

After having the bare-bones of the Websocket client, it was adjusted to instead of having the data to transmit hard-coded, have a configuration file which would describe the messages to send, variables to collect from responses, and overall set-up actions to have the simulation running. In Listing 4.5 we can see an example of said configuration file, in this case for the Meshblu broker.

4.1.1 Interface

When the Websocket client was completely adapted and working on both DeviceHive and Meshblu depending on the configuration file used, a graphic interface was developed. This interface allows the user to customize the:

- Number of clients
- Number of devices
- Size of the payload
- Interval between publishes
- Number of subscriptions by client
- Maximum number of notifications to send, per protocol and device

Moreover, the interface also measures the time between the publishing of a notification and it's receipt by the clients, and presents, when the simulation ends, the number of sent notifications, received notification, lost or failed notifications and the average time between publishing and receiving.

When running this interface, the user must specify a path to where the brokers' configuration files are, and have the appropriate protocol files in the same directory as the interface script. When ran, the script will search the configurations directory for *.cfg* files, and display their name on a dropdown menu, for the user to select which broker is to be tested. It will also scan the current directory for files ending with *_client.py*, this being the naming defined for the protocol files (i.e.: *ws_client.py, mqtt_client.py, ...*).

The interface will also allow the user to start and stop the simulation of a specific protocol, or start and stop the simulation of all protocols with the given parameters.

4.1.2 Communication Scripts

The script that establishes the connection with a broker defines three different classes and the methods *start* and *stop* which should have the form defined in Listing 4.1.

```
1    start(config_param, metrics_param, n_cli, n_dev, pl_size, msg_int, subs_cli)
2    stop()
```

Listing 4.1: Two main functions to define in the protocol connection script.

These two functions are called by the interface component, with the *config_param* being the configuration file with the broker information, the *metrics_param* the array where the metrics mentioned above are stored, *n_cli* the number of clients to simulate, *n_dev* the number of devices

to simulate, *pl_size* the size of the message content, *msg_int* the interval between messages, and *subs_cli* the number of subscriptions per client.

The classes defined by the script are:

Controller Responsible for setting up the scenario, it establishes a connection with the broker and creates the necessary devices for the simulation, saving the necessary information. It was decided to have this initial step because the login and registering of devices varies greatly between brokers, for example: in DeviceHive, there is a login step and the sending of notifications is made on behalf of a device; with Meshblu, the device is registered, receiving a *uuid* and *token* pair, with which it can then authenticate and start sending messages.

Client Each being a client to simulate, receiving the information from both the interface and the controller (devices created and respective identifiers to subscribe to). Each time it receives a notification it updates the *metrics* variable, and it notes how long it took between the publishing and the receipt.

Device Simulates a device: receives the necessary information from the controller as to authenticate, and then starts publishing messages, recording a message id along with a timestamp, which the Client will use to know the time interval between sending and receiving notifications.

Each Client and Device runs on its own thread, and when a Client receives a message it starts a thread responsible for processing the information, while the main one returns to listening for more notifications.

When the simulation is stopped, the *stop* method terminates all Device and Client threads and calculates the average time between sending and receiving a notification.

4.1.3 Data Capturing

For the capturing of information relevant to the analysis it was first considered the already available logs from each microservice, but given the heterogeneity of the log file formats, and the brokers themselves, it was not a viable solution. Some microservices barely produce logs while others produce too much irrelevant information, for this context, about the state of the microservice.

While studying both brokers, Meshblu and DeviceHive, it was noticed that both of these brokers had a Docker repository in order to easily deploy the brokers with the microservices containerized. Because of this, the starting scenario for the development involved having the brokers in a containerized circumstance, where each microservice resides in a Docker container. It not only made it easier to deploy and run the brokers but also facilitated the capturing of data from each microservice.

Not only that but it was also noticed that DeviceHive already had a Grafana plug-in which made use of cAdvisor plus Prometheus to collect and display some container-related metrics, allowing a fast and easy representation of some data relevant for our analysis. And the company where this work was conducted, Ubiwhere, also made use of a similar set-up to monitor its own system, running Meshblu.

For the collection of more data additional to the mentioned above, it was used a packet capturing tool, namely Dumpcap, which should be running while the simulation is taking place. For this it was developed a simple script that checks running containers on the current system, and for each it runs an instance of Dumpcap, capturing on the virtual interfaces created by Docker with the option "-s 0", which taken from the dumpcap's documentation: "specifies a snapshot length of 262144, so that the full packet is captured". When terminated, the script stops the capturing and saves the containers' logs.

4.2 Parsing Tool

The parsing of the data consists of a script that runs through the capture files and tries to extract relevant information from them.

It was an incremental process, first being developed code to run through the pcapng files produced during the simulation. Afterwards, a simple simulation of clients was run and for each new communication protocol detected a way to parse it was developed, sometimes being simply ignored if no useful information could be extracted from packets of that particular type of protocol. Some captures were also ignored simply because there was too much information in them with little relevance to the benchmarking process. Because of that, the parsing of said captures would consume too much time in relation to the additional information produced.

The main objective of this process is to associate requests with responses and where, in the architecture, a message is taking longer to process or even where a message was lost. For this, the main script splits the capture into TCP streams, and runs through them. For each communication protocol, a separate file must be included with the appropriate code to parse that type of protocol, this way being easier to add compatibility with new protocols, if necessary.

To achieve this, a new process is started for the parsing of each capture file, where various arrays are created. These arrays are mainly: (i) tcp streams' array, (ii) malformed packets and retransmissions, (iii) received messages of given protocol, (iv) received requests of given protocol, (v) timestamps of request/response pairs of given protocol.

For Websocket there is also an array of unidentified messages - this is because to associate a response to a given request we must look into the message contents, and in case there is no specific identifier, we just fallback to associate the next response with the oldest unanswered request that has the same *action* or *topic* as the response.

4.2.1 Adding Protocols

The parsing of the messages is handled by different modules depending on the protocol, which should be imported to the main parser. As an example of these modules we can see in Listing 4.2 the parser of HTTP packets, where it associates responses with requests and takes note of its timestamps. In this case, since pyshark already displays to what an HTTP request is an HTTP message responding to, the association is straight-forward (using the "request_in" attribute).

```
1  def http_parse(**args):
2    args['http_received'].append(int(args['packet'].number))
3
4    req = -1
5    try:
6      req = int(args['packet'].http.request_in)
7    except AttributeError:
8      args['http_req'].append(int(args['packet'].number))
9
10   if req in args['http_req']:
11     args['http_times'][(req, int(args['packet'].number))] = (float(args['capture'][
           req-1].sniff_timestamp),float(args['packet'].sniff_timestamp))
12     args['http_req'].remove(req)
13
14 def data_parse(**args):
15   for layer in args['packet'].layers:
16     if layer.layer_name == 'http':
17       http_parse(args)
18     else:
19       return
```

Listing 4.2: Example of a parser module for HTTP packets.

In the "args" dictionary we have references to the arrays mentioned above, and the "<protocol>_parse()" method should use these references to update the arrays. First by adding the packet number to the received packets array relative to that protocol (line 2), then it should check if the packet is a request or a response. In case it is a request, it should add the packet number to the respective array (line 8), otherwise it should check to which request the message is responding to (lines 10-12), removing the request packet number from the requests array and adding the key (request_number, response_number) to the dictionary "<protocol>_times", with the value being (req_timestamp, res_timestamp).

After the module is complete, some modifications to the main script must be made. In Listing 4.3 we can see an excerpt from the parsing script, modified to just highlight the handling of different protocols. To add a new protocol (HTTP as an example) the modifications to the main script go as follow:

- Line 1-2: import the functions.
- Line 4-8: dictionary that associates a protocol name (as appears on the packet object) to a parsing function.
- Line 12-14: arrays that will hold an array each for each TCP stream. In the example, we have an array for received HTTP messages, HTTP requests waiting response and HTTP request/response pairs with respective timestamps, as mentioned above.
- Line 18-20: in case a new TCP stream is detected, add an empty array to the arrays created on lines 12-14.

- Line 22: use the attribute "highest_layer" of the pyshark packet object in order to call the correct parsing function and pass the new arrays as parameters of that function.

```
1  from my_http import http_parse
2  from my_http import data_parse
3
4  packet_type = {
5    'HTTP'       : http_parse,
6    'JSON'       : http_parse,
7    'DATA'       : data_parse
8  }
9
10 def parse_capture(capture, capture_raw, config_param):
11   tcp_streams = []
12   http_received = []
13   http_req = []
14   http_times = []
15   for packet in capture:
16     if len(tcp_streams) <= int(packet.tcp.stream):
17       tcp_streams.append((str(packet.ip.addr) + ':' + str(packet.tcp.srcport), str(
             packet.ip.dst) + ':' + str(packet.tcp.dstport)))
18       http_received.append([])
19       http_req.append([])
20       http_times.append({})
21
22     packet_type[packet.highest_layer](capture=capture, capture_raw=capture_raw,
         config=config_param, packet=packet, http_received=http_received[int(packet.
         tcp.stream)], http_req=http_req[int(packet.tcp.stream)], http_times=
         http_times[int(packet.tcp.stream)])
```

Listing 4.3: Excerpt of the main parsing script.

4.2.2 Parser Output

After the parsing tool runs through all the captures and fetches the information from Prometheus, the output will be a text file along with two HTML files, with graphic representation of some information, for reach capture. In Listing 4.4 we can see part of an output file produced by the parser, where, for each TCP stream identified on the capture it records the packets of each of the parsed protocols, as well as which ones were requests, which were answered and which are still pending a response. For some of the groups, as it can be seen in line 3, for example, it also splits the packets according to their source IP address. Moreover, it also registers some of the information used to produced the graphics displayed further.

The HTML files produced as result present the times between receiving/sending a request and sending/receiving a response, and in Figure 4.1 the example of said graphic can be seen. Each box

```
TCP STREAM: ('meshblu_mqtt_1:39880', 'meshblu_redis_1:6379')
  HTTP Packets: []
  HTTP Pending Requests: {'172.18.0.2': {}, '172.18.0.10': {}}
  HTTP Times: {'172.18.0.2': {}, '172.18.0.10': {}}
  WS Packets: []
  WS Pending Requests: {'172.18.0.2': [], '172.18.0.10': []}
  WS Times: {'172.18.0.2': {}, '172.18.0.10': {}}
  WS from Server: {}
  PGSQL Packets: []
  PGSQL Pending Requests: deque([])
  PGSQL Times: {}
  MONGO Packets: []
  MONGO Pending Requests: {}
  MONGO Times: {}
  MQTT Packets: []
  MQTT Publishes: {'172.18.0.2': [], '172.18.0.10': []}
  MQTT Pending Requests: {'172.18.0.2': {}, '172.18.0.10': {}}
  MQTT Times: {'172.18.0.2': {}, '172.18.0.10': {}}
  REDIS Notifications: {'172.18.0.2': {}, '172.18.0.10': {}}
  MALFORMED Packets: []
  RETRANSMISSIONS: []
Total Malformed Packets: 0
Total Retransmited Packets: 0
Graphic Recreation Info:
redis:{}
traces:[]
```

Listing 4.4: Excerpt of the text file produced by the parser.

plot is labeled with the target of the communications along with the note weather the request is made to or received from said target. The Y-axis indicate the time, in seconds.

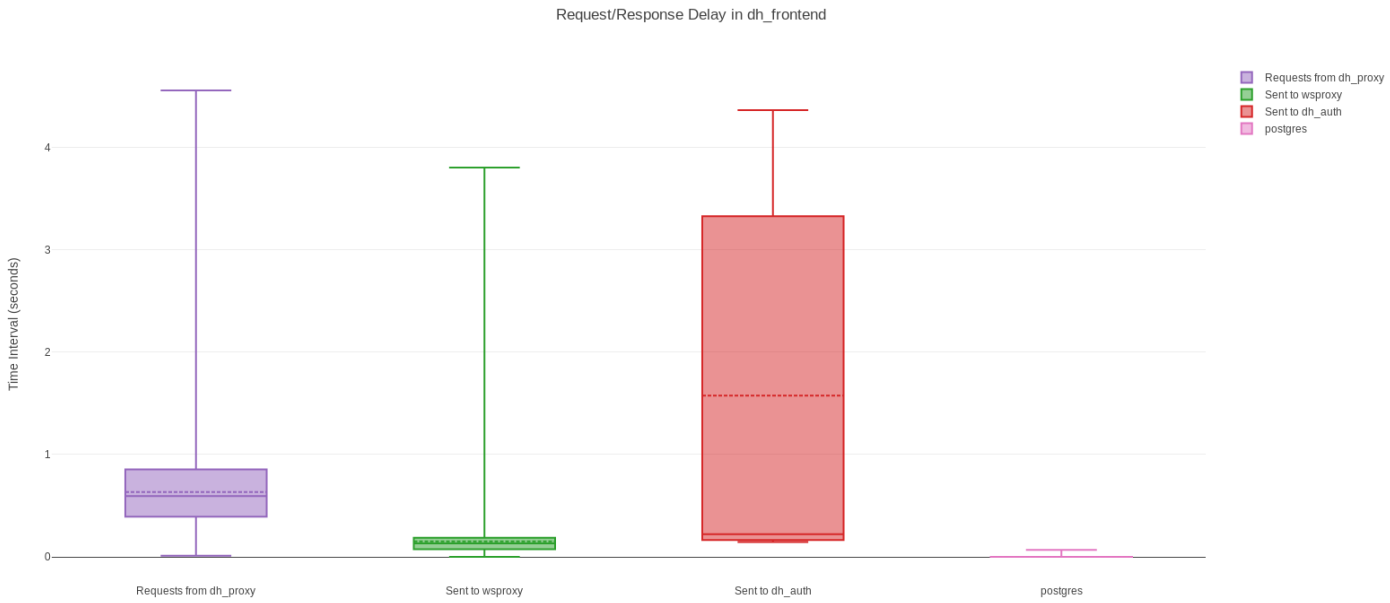

Figure 4.1: Graph generated by the parsing tool, representing request/response intervals.

Another graphic that is produced is a line chart, and this is used to represent communications with the Redis and Websocket protocol, as it can be seen in Figure 4.2. Along the X-axis we have the epoch time in increments of one second, and the Y-axis indicates the number of bytes transmitted. Just as in the box plots mentioned above, this graph distinguishes between sent and received packets. Also about Redis, it is attempted to find notification IDs on its packets. This was implemented because of the internal communications of Meshblu, and is used to track notifications inside the broker.

Finally, the parser also fetches information from Prometheus (obtained with cAdvisor) about each container running, namely, the CPU and RAM usage, and the quantity of transmitted and received bytes. In Figure 4.3 we can see an example of the resulting graphs, the percentage of CPU usage by each microservice of the broker, which should be taken into consideration that is the cumulative usage of the available CPUs in the machine, hence not topping at 100%. All the information used to plot the mentioned graphs is written to text files, in case the user wants to use it at a later time.

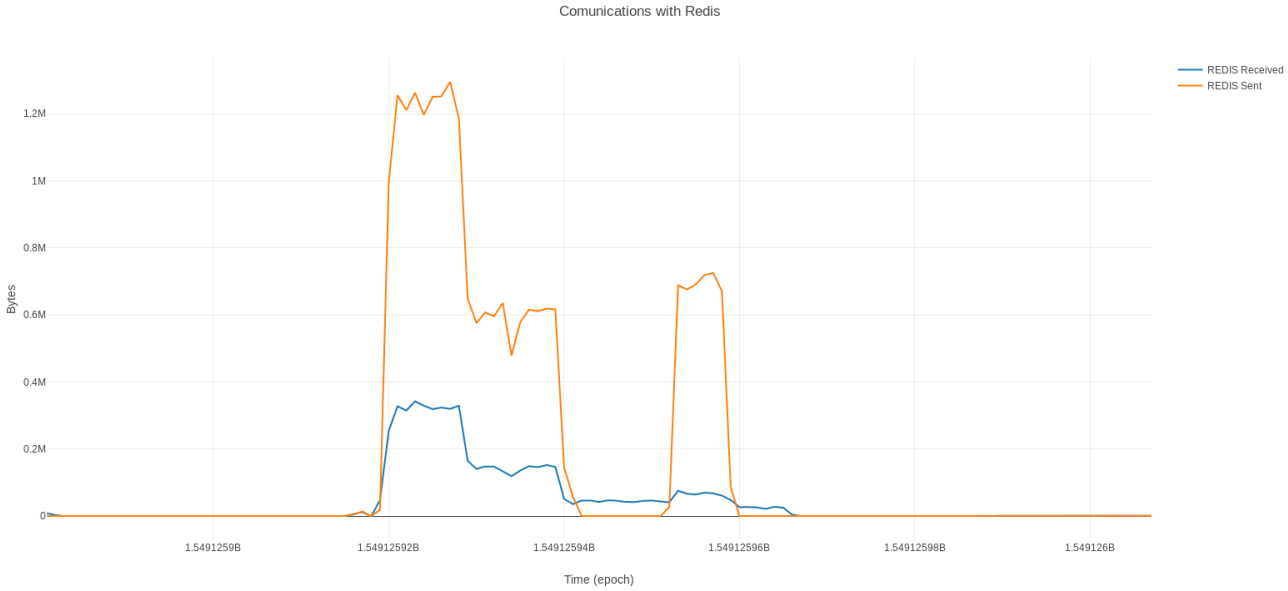

Figure 4.2: Graph generated by the parsing tool, representing Redis communications.

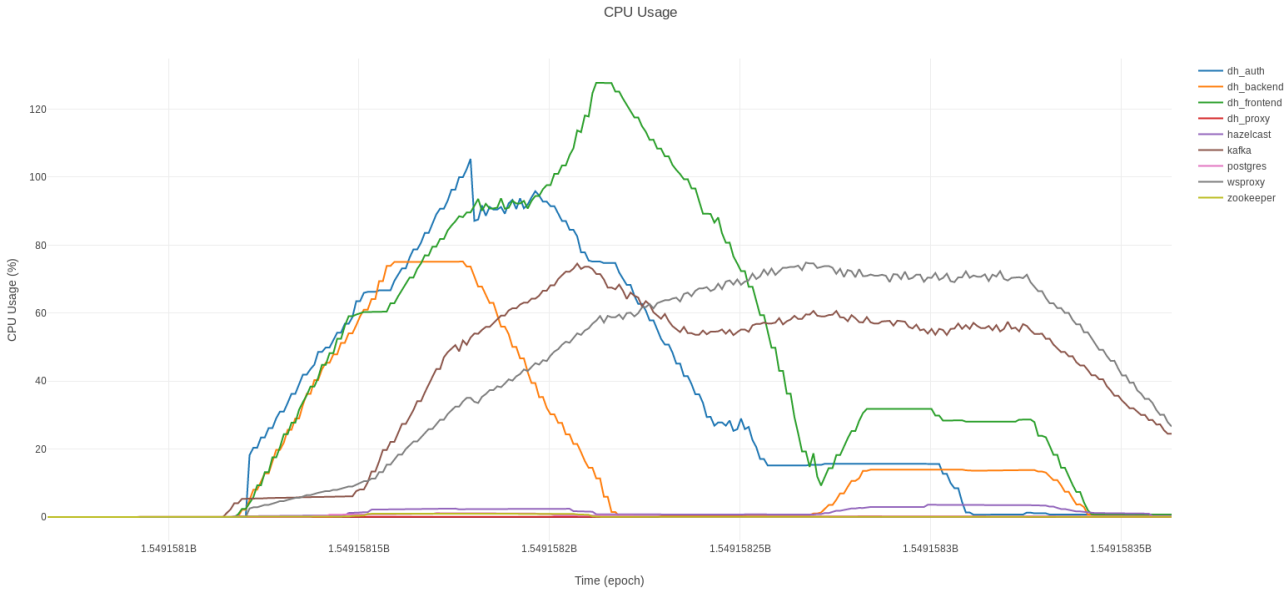

Figure 4.3: Graph generated by the parsing tool from information given by Prometheus.

4.3 Configuration File

As it was mentioned above, for the tools described in this chapter to know how to communicate with the broker and parse its messages, a configuration file is needed. An example of the configuration file excerpt used by the simulation tool can be seen in Listing 4.5.

In regards to the simulation there are five groups of variables:

General This section is dedicated to general connection variables or variables that are shared by the components;

<role>_vars Section dedicated to variables used by the <role> as well as variables needed from the received messages - the simulation will run through the response messages and store any variables whose key matches a key under <role>_vars;

<role>_msg Section with the description of the messages to send as well as the order by which they should be sent;

Publish_topics Topics to be used with the MQTT protocol to publish notifications;

Subscription_topics Topics to be used with the MQTT protocol for subscriptions.

In regards to the parser tool, there is one more section as seen in Listing 4.6. It contains some information that the parser needs to parse Websocket packets, since it needs to be content-aware in order to determine what message is answering to what request, being:

msg_type_position Position of the type of the message.

msg_unique_id Position where the parser may hope to find a message ID, if none is found it uses the message type as fallback

Lines 4-8 In case the type of the response message differs from the request, it should be added as <request_type> = <response_type>. If a type doesn't have a match it assumes both types are equal.

no_response Array with the types of the messages that are not typically responded to.

```
1 [Parser]
2 msg_type_position = 0
3 msg_unique_id = requestId
4 identity = ready
5 devices = devices
6 device = device
7 register = registered
8 subscribe = subscribe
9 no_response = ["message"]
```

Listing 4.6: Section of the configuration file dedicated to the parser's information.

```
[General]
ws_url = ws://172.18.0.3
seq_key = 0
device_name_system = uuid

[Client_vars]
msg_id =
uuid =
token =
deviceId =
sender_id_name = fromUuid
notification_name = notification_id
notification_resp = message
subscriptions_per_client =
device_id_name = uuid
setup_ops = 2

[Client_msg]
setup_op_0 = ["register", {"type":"sample_user"}]
setup_op_1 = ["identity", {"uuid":${Client_vars:uuid}, "token":${Client_vars:token}}]
subscribe = ["subscribe", {"uuid":${Client_vars:deviceId}}]

[Device_vars]
msg_id =
uuid =
token =
notification_id =
payload_size =
message_interval =
setup_ops = 1

[Device_msg]
setup_op_0 = ["identity", {"uuid":${Device_vars:uuid}, "token":${Device_vars:token}}]
send_notification = ["message", {"devices":["*"],"topic":"status", "payload":{"notification_id":${Device_vars:notification_id}}}]

[Controller_vars]
msg_id =
uuid =
token =
notification_id =
device_list_name = 1
setup_ops = 2
on_response = True

[Controller_msg]
setup_op_0 = ["register", {"type":"sample_device"}]
setup_op_1 = ["identity", {"uuid":${Controller_vars:uuid}, "token":${Controller_vars:token}}]
activity_op_0 = ["register", {"type":"sample_device"}]
activity_op_1 = ["devices", {}]
cleanup_op_0 = ["unregister", {"uuid":${Controller_vars:uuid}, "token": ${Controller_vars:token}}]
```

Listing 4.5: Example of a configuration file used by the simulation platform.

4.4 Challenges

During the development some difficulties arose that shaped the final product, and required more attention.

4.4.1 Simmulation Platform

Starting with the simmulation platform, brokers are very heterogeneous in the way the communications are implemented, and although there is common ground, some particular situations are easily encountered when attempting to develop a generalized platform.

In the particular case of DeviceHive and Meshblu, we can start by the fact that the publishing of notifications differs greatly, as it was mentioned above. In DeviceHive, notifications are posted on behalf of the devices, meaning, a node must login and after authenticating it can send notifications indicating the device ID from where the notification is supposedly sent (provided that it has the correct permissions). This can also be handled by generating specific JWT with the desired permissions and passing them to the nodes, but given its complexity, this scenario is not contemplated on the platform. On the other hand, Meshblu gives credentials upon registering a device, and a node must authenticate using those credentials to send notifications as that device. To solve these differences, it was implemented an initial step (mentioned above as the "controller") which starts by registering the needed devices for the simulation and then passes that information to the "Device" and "Client" components. In the case of Meshblu, it uses said information to authenticate (UUID and Token), while DeviceHive simply authenticated as "admin" and uses the IDs returned by the controller to publish notifications on behalf of those devices. The platform knows how to handle this step according to the information provided in the configuration file, therefore, adding a new broker does not interfere with these process, as long as all the information needed is specified in the configuration file.

Another difficulty that arose during the development of the benchmarking platform was the underdevelopment of the Meshblu's MQTT API, which doesn't provide methods to register new devices. To avoid this problem the controller now checks with the configuration file if it is needed to use a different protocol than the current one to register devices, and if so it uses Websocket (as of now) for that step. Another particularity of Meshblu is the fact that nodes subscribed to a device won't receive the notifications published by the device, instead receiving notifications the device receives. To function properly, the Meshblu's configuration file must describe in the publish message that the UUID of the recipient is the same as the sender.

Finally, the simulation platform relies on information received by the brokers. Since the variable names are different depending on the broker, these must be included in the configuration file, and how to handle them. The platform itself simply searches through the messages searching for the specified keywords, and if their found, it sets the value of the variable within the configuration file. It should be noted that these changes don't affect the file itself, only being changed during the simulation execution.

4.4.2 Data Capturing

The main problem faced with the capturing of the communications between the containers was that a container must be already started in order to have a virtual interface created for them. This led to loss of information since the capturing would only start after the setup operations of the broker were already taking place. The main setback was with DeviceHive, since it uses Kafka with a wrapper named "wsproxy" to handle communications between microservices. This wrapper starts by establishing websocket connections with the other microservices, and since the capture starts after the connection is established, it couldn't decode the websocket packets exchanged.

Further down into development a solution was found, which involves stating the containers and right afterward issuing a "docker-compose pause". From there the capture script can be ran, and the containers "unpaused" ("docker-compose unpause"), this way reducing the amount of information lost and being enough to capture the establishing of the websocket connections with the "wsproxy" microservice.

This leads to another obstacle encountered which is the association between virtual interfaces and the respective container. For this, the capture script must access the host's "/sys/-class/net/veth*/iflink" files to get the interfaces IDs, and afterwards does the same from within the containers but with the "/sys/class/net/eth0/iflink" file. This second operation requires a running container, therefore the capture script must pause after starting the captures, the containers must be "unpaused", and the script can then resume, executing the necessary commands to get said information from within the containers.

4.4.3 Parsing Tool

Relative to the parsing of the capture files, the most challenging aspect was the need for content awareness in websocket messages. This lead to having the parser read the content of the websocket packets and finding ways to use available information to associate requests to responses. The issue here is that Tshark (used by Pyshark to read capture files) truncates the content of websocket messages, and to read the full content, we must specify Pyshark to include the raw packet. Moreover, captured packets may include more than one websocket layers. To solve this it was developed of an extraction method that ran through the packet, breaking the websocket payloads, unmasking them and decoding, since no apparent dissection was already in place for raw packets. Latter in the development it was discovered that some of this information was already included and that Tshark already reassembles Websocket packets and unmasks them, and the extraction process only needed to iterate over the Websocket layers and decode the payloads. After having the payload readable, the tool would use the information made available in the configuration file to know what to look for when associating responses to requests.

Another problem faced was the memory consumption of running Pyshark on large capture files. This was not identified right away, so the platform was developed without taking it into consideration, requiring a later restructure to store the necessary information while running though the captures, not keeping the whole capture in memory.

4.5 Summary and Running Instructions

In this Chapter it was described how the solutions were developed and the final result of the development. It was also explained how a new communication protocol or message broker can be added to the platforms. To conclude, it will be explained how to run both the simulation platform and the parsing tool. But first it should be noted how the platform directory is structured.

In Figure 4.4 we can see that structure represented, where the "/clients" directory holds the simulation tool, the "parser.py" and "capture.py" are the parsing tool and the capturing tool respectively, with the "docker-compose.yml" file being used to setup cAdvisor+Prometheus. The "my_<protocol>.py" files describe the parsing of specific protocols and the "<protocol>_client.py" are the implementation of different communication protocols used by the simulation tool. The "<broker_name>.cfg" files are the configuration files that hold information about the brokers. Finally, the "client/logs" directory hold information produced by the simulation platform, the "logs/captures" and "logs/info.txt" hold the captures and information produced by the capturing script, and the "logs/parsed_data" directory hold the output of the parser tool.

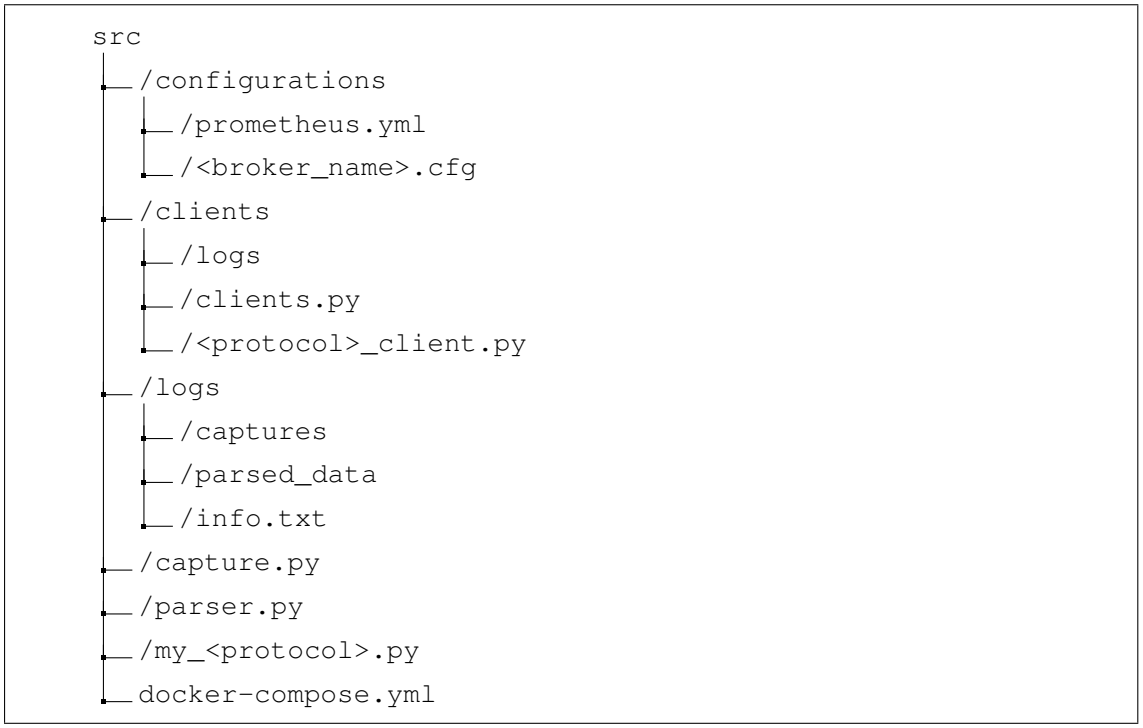

Figure 4.4: Directory structure of the developed platform.

Before even running the broker it should be noted that the capturing script is dependant on the fact that the message broker uses Docker for deployment and has its microservices containerized. This is because the capture is made on the virtual interfaces created by Docker for each container. Moreover, the bundle cAdvisor with Prometheus is also container dependant, and cAdvisor is build specifically to monitor containers.

That said, the host machine needs to be running cAdvisor along with Prometheus, and this can be done with a "docker-compose.yml" file as in Listing 4.7, and, as it is mentioned in Prometheus' documentation, a Prometheus configuration file must also be present. An example of said file can be seen in Listing 4.8.

```
version '3.2'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
    - 9090:9090
    command:
    - --config.file=/etc/prometheus/prometheus.yml
    volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
    depends_on:
    - cadvisor
  cadvisor:
    image: google/cadvisor:latest
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
    depends_on:
      - dispatcher
```

Listing 4.7: "docker-compose.yml" file to build and run cAdvisor+Prometheus

```
scrape_configs:
- job_name: cadvisor
  scrape_interval: 1s
  static_configs:
  - targets:
    - cadvisor:8080
```

Listing 4.8: Contents of file used to configure Prometheus

Having a "docker-compose.yml" file already written to deploy the message broker, the code in Listing 4.7 can be included in that file, after which the user must run "docker-compose up" to start up the message broker and the monitoring tools.

When everything is running, the capturing script must be ran next, to start capturing on all the containers' interfaces. Afterwards, the user just needs to run the simulation interface. This program takes as an argument the configurations directory, where the configuration files for each broker are located. After the simulation is over, the user can terminate the capturing script, and afterwards run the parser. This parser also takes as argument the configuration file of the used broker and will take the captures (saved in ./logs/captures, relative to the capturing script directory) and parse them to display the results.

Chapter 5

Testing and Results

In this Chapter the platform will be tested and its output will be analysed in order to ascertain the usefulness of the developed tools.

5.1 Scenario Platform and Setup

For the benchmarking experiments described in this Chapter, two machines were used: one to simulate the nodes, with an Intel Core i5-3750K - which is composed by four cores and four threads, with a base frequency of 3.40 GHz - and 8GiB, 1333MHz, of RAM; and another to run the broker and the parser afterwards, which has an Intel Core i5-2520M - composed by two cores and four threads, and a base frequency of 2.5GHz - and 4GiB, 1333MHz, of RAM. Moreover, the simulation platform was running over a virtual machine with 4GiB of RAM allocated, using the Manjaro 18.0.2 operating system running Linux 4.19.16 and the host machine running the Windows 10 operating system. As for the broker and parser, they are running on the same operating system as the virtual machine mentioned, Manjaro 18.0.2.

Relative to the platform, the user must change the addresses specified in the configuration files of each broker, which was set to "192.168.1.8", since the two machines are in the same network, and that is the address of the machine running the broker. Moreover, all the information must be deleted from the broker before running the simulation.

As to facilitate the understanding of the produced data by the parser, the user should also modify the parser script to include the IP address of the machine running the simulation tool. That can be done by adding a (key, value) pair to the "ip_name" dictionary, which in these runs will be "'192.168.1.7': 'client'". This step is not necessary, being used only in the output to replace that IP address by the name 'client'.

After that is done, the broker is started using Docker and the method described in the previous Chapter: "docker-compose up –no-start" to create the containers, "docker-compose start;docker-compose pause" to start and pause the containers right after, run the "capture.py" script, unpause the containers and start both Prometheus and cAdvisor - "docker-compose up -d" using the "docker-compose.yml" file made available with the platform - and finally press the "Enter" key on the "capture.py" script to save information relative to the virtual interfaces. At this point the broker is running along with Dumpcap and Prometheus+cAdvisor.

With that, the user can then run the simulation tool with the command "python ./client.py ../configurations/". It should be noted the directory structure mentioned in Chapter 4. This command will start a graphic interface from where the user can choose which broker is to be used, the load (as mentioned in the previous chapter) and what protocol to run. For each scenario it will be noted the value of these options.

When the simulation is over, the capturing script is interrupted, extracting the container logs from Docker, and the parser is ran. It should be noted that the Prometheus and cAdvisor containers must be still running when the parser starts, as it needs to fetch information from them. As to the broker itself, it can be terminated as soon as the simulation is over. It is advisable to let the broker run for one minute after the capturing is stopped, that way it can be visualized how the metrics collected by cAdvisor change after the simulation.

Despite the parser only needing the capture files, independent of the simulation platform or how they were captured, such tests won't be contemplated in this thesis. Besides, when doing so the user must consider that the data from Prometheus+cAdvisor won't be available.

5.2 DeviceHive Results

Given that these benchmarking tests are focused on understanding what is going on inside the broker itself, the results will be presented and analysed separately, starting with DeviceHive. The simulations will be also split in protocols and it should be noted that the information available from the simulation tool, as opposed to the information we get from the parser, is relative to the time between publishing a notification and receiving it on the client. Since the captures run from the moment the broker is up, the parser information also contains the authentication and device registering requests.

Before the simulations are described we should first overview the microservices that compose DeviceHive, as stated in its documentation and observed along with the development of this dissertation:

dh_proxy Proxy component that received all external requests and sends them to the appropriate microservice;

dh_auth Microservice responsible for handling validation, generation and refreshing of JWTs;

dh_frontend Frontend microservice which provides a REST and Websocket API;

dh_backend Responsible for storing data in Hazelcast, managing subscriptions and retrieving data by request from other microservices either from Hazelcast either from Postgres;

postgres Database to store persistent data about devices, networks, users and important configuration settings;

mqtt_1(_2) Microservice that provices a MQTT API to DeviceHive, using Mosca MQTT broker library;

kafka Responsible for the communications between microservices and balancing the load between them;

wsproxy Wraper to Kafka which offer a Websocker API for the other microservices to communicate with Kafka;

hazelcast An in-memory data grid that stores notifications;

redis Redis server for persistence functionality when using MQTT.

5.2.1 Websocket Simulations

As a first simulation we'll setup the scenario with one client subscribed to one device, where the device will send 1000 notifications with a 30 byte payload, at a maximum rate of 100 messages per second, using only the Websocket protocol. Right from the simulation tool we know that all publishes were successful as well as all subscriptions, and all the 1000 notifications arrived to the client, with an average time of 0.405 seconds.

After running the parser we obtain more in-depth information, starting with the information obtained by cAdvisor+Prometheus. This information can tell us what to expect from each microservice on a real-life scenario, giving opportunity for better resource allocation. We can observe that the most CPU intensive microservices (Figure 5.1) were "wsproxy" and "kafka", this also verifies with the amount of transmitted data (Figure 5.2) in and out of those containers, most likely because there is continuous communication between them. Such information should be taken into consideration when deploying a broker or designing a solution which makes use of the target broker.

With that in mind we can now take a look at the communication times between microservices, and try to better understand what is going on inside the broker as well as search of ways to improve its performance. Since "dh_proxy" is the entry point of communications we will start there, and we can see that besides receiving the requests from the client, it communicates with "dh_frontend", most likely redirecting the requests. We can also take a look at the Websocket data from packets which are neither requests nor responses, and these are most likely related to delivering notifications to the subscribed clients. In Figure 5.3 we can see that there is a spike of data received from "dh_frontend" and at the same time data being sent back to the client, which suggests a route from where notifications travel when being delivered to the subscribers.

From here we can then check the "dh_frontend" times and see that the biggest time intervals are in requests made to both "dh_auth" and "wsproxy". The fact that the longest interval

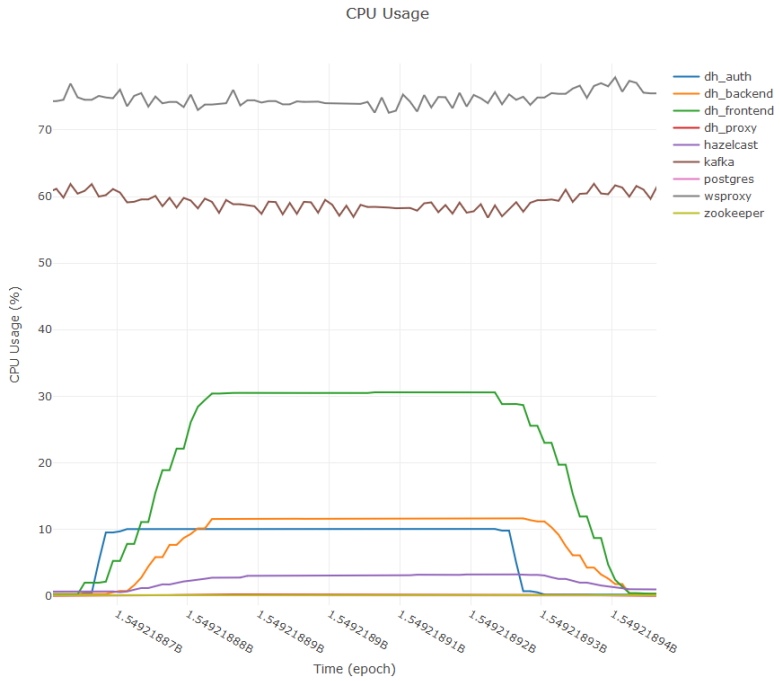

Figure 5.1: Prometheus+cAdvisor data on CPU usage per second.

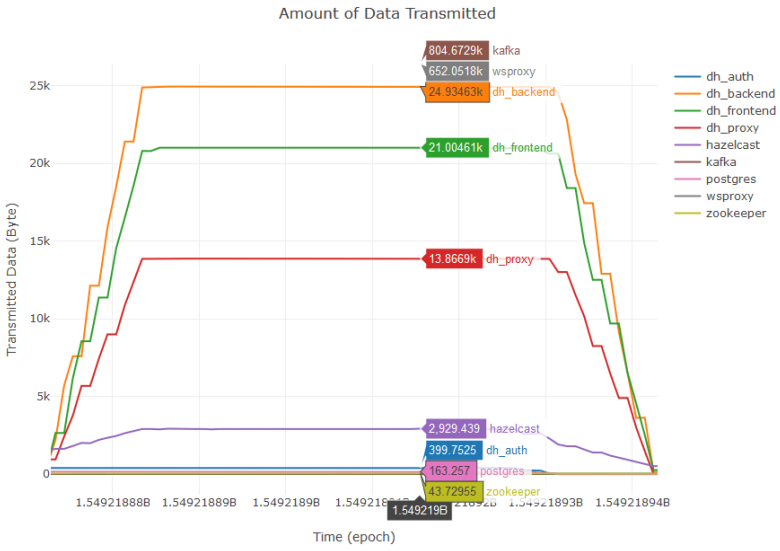

Figure 5.2: Prometheus+cAdvisor data on bytes transmitted per second.

between "dh_frontend" and "wsproxy" is bigger that the longest interval between "dh_frontend" and "dh_proxy", alongside the fact that the intervals overall are fairly short, suggests that the comunications between these components are not what is affecting the interval times between "dh_frontend" and "dh_proxy". On the other hand, we have the communications with "dh_auth" which have fairly large intervals that fit with the delay with "dh_proxy", making "dh_auth" a probable cause for the longer response times. Moreover, we also have available information about Websocket data not associated with requests/responses, and we see spikes of data received from "wsproxy" and sent to "dh_proxy", which leads to the next component from where notifications

might be routed through.

Figure 5.3: Websocket data on "dh_proxy" not associated with a req/res exchange.

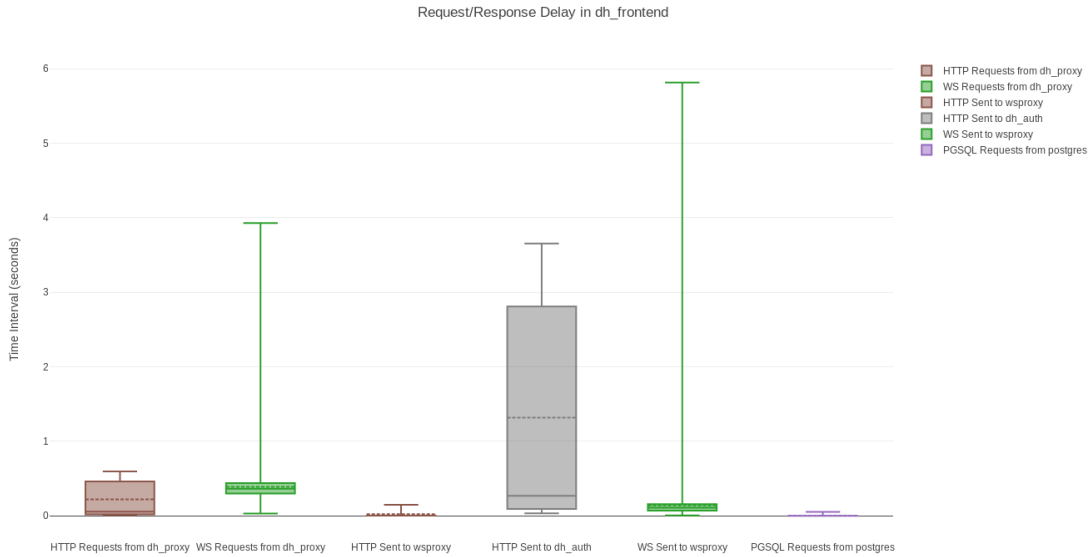

Figure 5.4: Websocket req/res delay on "dh_frontend".

Focusing now on "dh_auth" we can see only two sets of intervals that call our attention. The first one is with requests received from "dh_frontend" and another is with requests sent to "wsproxy" which a quick look at the captures can show us that is only related to setup operations of the broker. The only relevant communications besides those are with "postgres", but the intervals are so brief that we can disregard them. That suggests that "dh_auth" can be a bottleneck in this scenario, but it should be taken into consideration that it is only relative to the authentication

of nodes and generation of JWTs, which doesn't usually take much time in a real-life scenario, where most of the time the broker is dealing with the publishing and treatment of notifications.

Taking a step back to "dh_frontend" and this time following to "wsproxy", we can ignore the HTTP requests, since those intervals are very small and most likely related to setup operations (witching protocols request to establish Websocket connections) as well as the Websocket requests from "dh_auth" which are also for establishing the connections, as it was seen. We can see the communications with "dh_frontend" and "dh_backend" from this graph, although we do know that it communicates with "kafka", it is not displayed here as the parser doesn't handle Kafka communications. But what we can do is take a look at the Websocket data which is not associated with req/res packets (Figure 5.5). Here we can see a similar spike as seen so far, but this time both being sent to "dh_frontend" and "dh_backend".

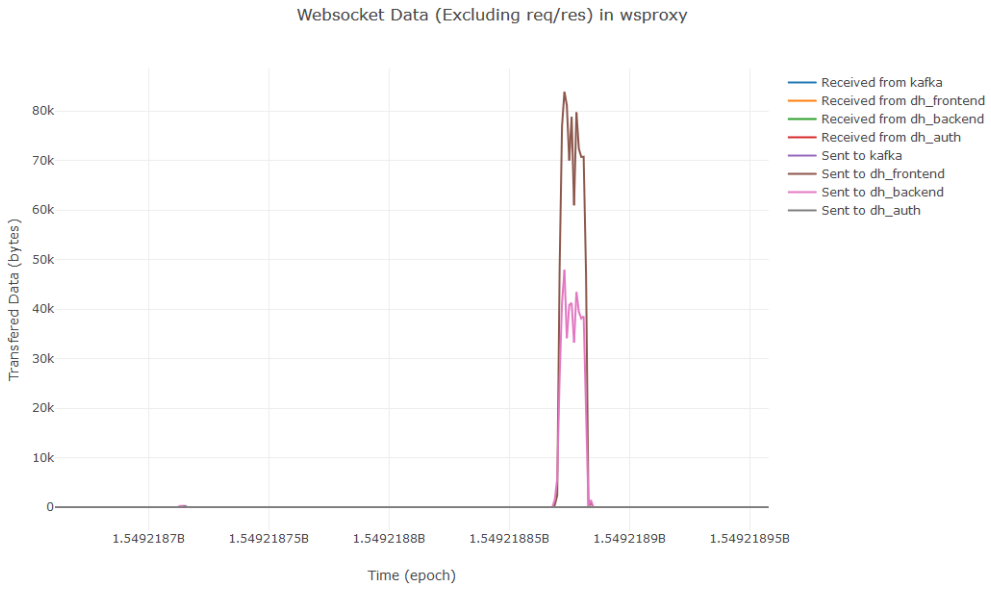

Figure 5.5: Websocket communications in "wsproxy" that are neither requests or responses.

Knowing that the publish of notifications returns a response from the broker, we can follow a publish up to this point, where it gets published to "kafka" through "wsproxy", and from there, sent to whomever is subscribed to that particular topic. "dh_backend will most likely receive the notification process it and store it in "hazelcast", while "dh_frontend" received it back in order to redirect it to the subscribed client.

Since the information we obtain from "dh_backend"'s graphs doesn't add anything to what we already know of the notification route we step into the capture files to try to further understand what is happening. In the graph we can see that the spike in communications to "dh_backend" by "wsproxy" starts around epoch time 1549218850, so we'll skip to captures starting from there. The first captures create a device, and this can be followed back to the "dh_frontend" request. Afterwards, we get a subscription to "NOTIFICATION_EVENT" with the filters of the device with ID 1, which is our client subscribing to the device. By this point we notice that "dh_backend" receives notifications with information on what is requested from "wsproxy" and then it sends back

the answer on a different stream. This is likely to do with how the system is setup, with request and response topics being separate and dealt with by "kafka", which may result in misleading outputs by the parser.

Furthermore, taking a look at how "wsproxy" deals with notifications, we can see that it receives them from the frontend, redirects them to the backend, and immediately responds with success or failure to the frontend. Afterwards the backend sends a notification telling that a notification was processed, and from here "wsproxy" sends it back to the frontend in order to arrive to subscribers.

This first experiment helped understand what is going on inside the broker, and even if the captures are not taken into account it is still possible to follow the notifications path inside the broker and see where it may be taking longer to process information or what components are more heavy on resources.

5.2.2 MQTT Simulations

The first simulation using MQTT, with also 1000 notifications with a 30 byte payload, at a maximum rate of 100 messages per second and one client subscribed to one device resulted in all publishes being successful and all messages arriving to the subscriber, with an average time of 0.507 seconds.

As we can see in the graph with the MQTT publishes, there are exchanges with both "mqtt_1" and "mqtt_2", where the proxy sends publishes from the client to "mqtt_1" and receives publishes from "mqtt_2", sending them back to the client (delivering to the subscribed node). Once we jump to "mqtt_1" we can see that it communicates with "dh_frontend" via Websocket and that's were communications using MQTT end and the broker now assumes communications using Websocket, which result in a scenario similar to the described in Sub-section 5.2.1. This suggests that the MQTT API serves as a wrapper to the Websocket API provided by "dh_frontend", instead of communicating with the rest of the broker directly.

Despite that, another thing to note from the graphs, in this case with the information about transferred data in and out of containers as is shown in Figure 5.6, is that the "dh_proxy" does show less amount of bytes transmitted per second, which might be a benefit to consider.

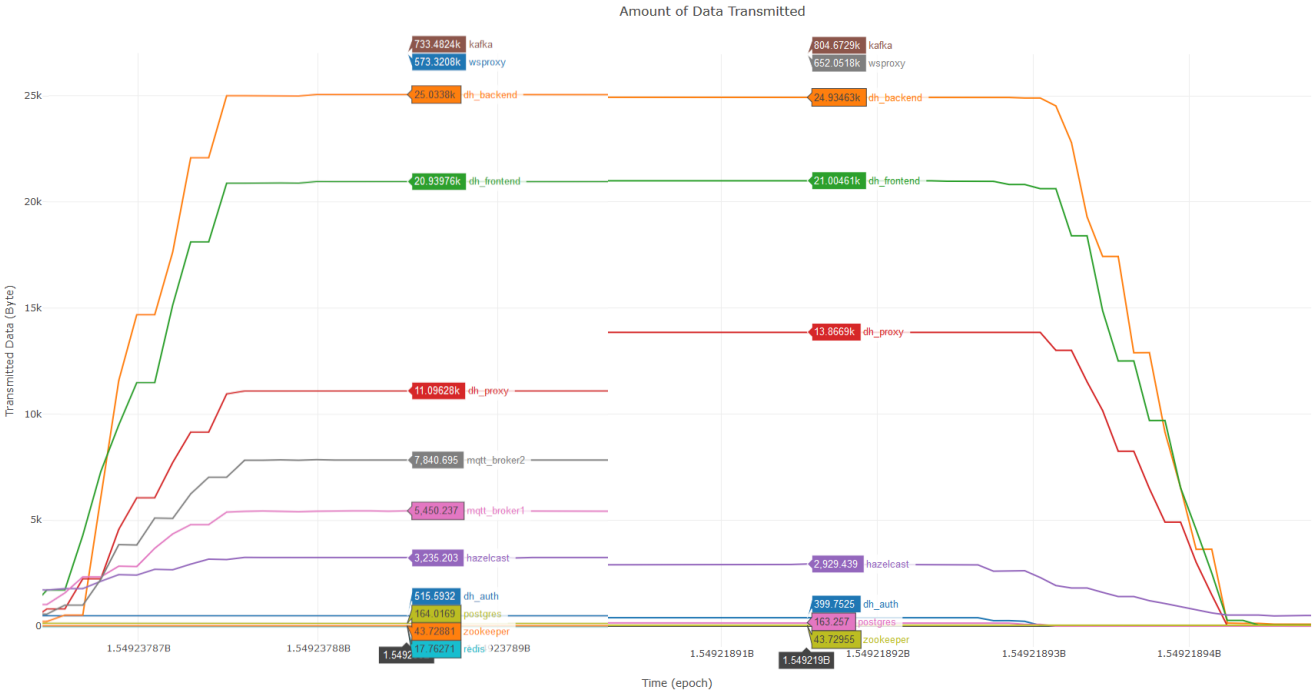

Figure 5.6: Comparison of transmitted bytes between using MQTT or Websocket.

5.2.3 Time Comparison

For comparison, in Figure 5.7 we can see the subscribe time when using Websocket as opposed to using MQTT while also adding the times when traffic is not being captured.

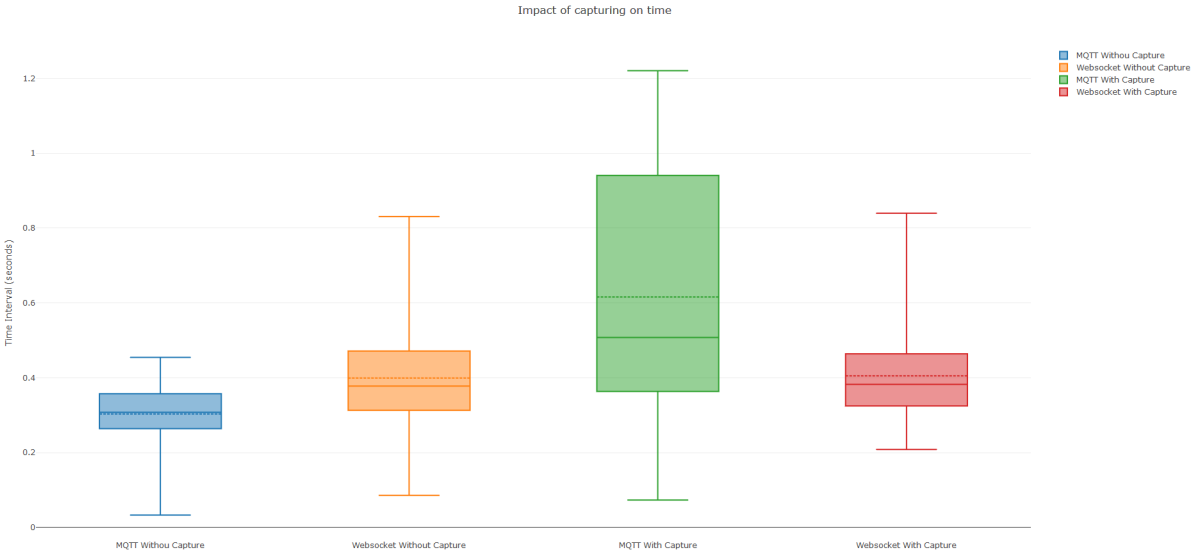

Figure 5.7: Comparison of the subscribe times in DeviceHive.

The biggest impact noted is between using MQTT with and without the capturing. This comes probably from the fact that with MQTT we have three more containers which result in three more captures running, resulting in a bigger difference.

5.2.4 Multiple Devices

Another experiment ran was the simulation of multiple devices, with the same rate of notifications arriving to the broker. In Figure 5.8 we can see how the interval between sending a notification and its arrival on the subscriber is affected by having multiple devices. Here we have one client subscribed to all the running devices, and the notification rate is set in order to maintain the 100 messages per second arriving at the broker.

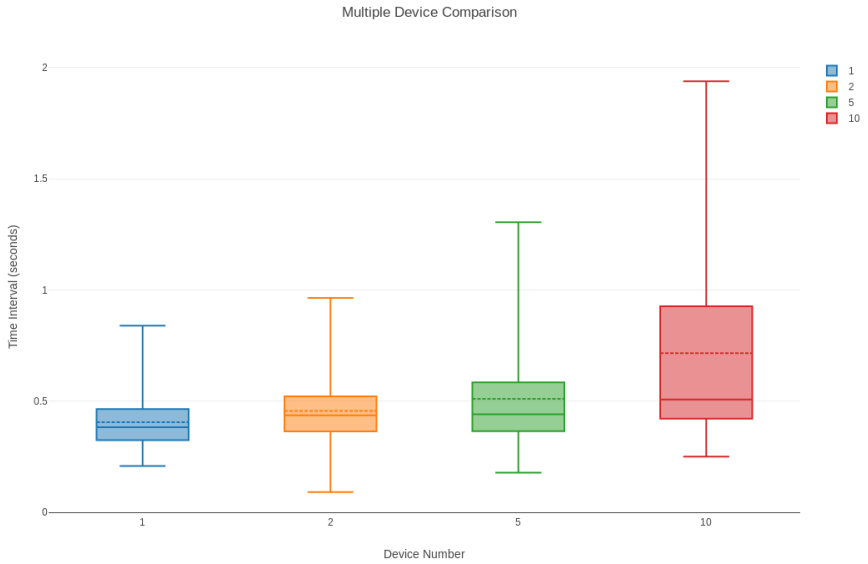

Figure 5.8: Comparison of the subscribe times in DeviceHive with multiple devices.

5.3 Meshblu Results

Moving now to Meshblu, the same tests as the ones described for DeviceHive were conducted. Before the simulations are conducted a UUID and Token were generated and saved in the configuration file relative to Meshblu, to be used as the client to subscribe. Before describing the simulations, here lies an overview of Meshblu's microservices:

dispatcher Meshblu's backend which processes requests and handles operations with the database.

http Protocol adapter that offers a REST API.

websocket Protocol adapter that offers a Websocket API.

mqtt Protocol adapter that offers a MQTT API.

mongo Persistent database were all the information is stored.

redis In-memory data structure that is used for communication between microservices.

5.3.1 Websocket Simulations

As it was said, the simulations ran are identical to the ones ran with DeviceHive, starting with 1000 messages with payload of 30 bytes, and a maximum rate of 100 messages per second. Again, we're using a client subscribed to a device, but it should be noted, as it was mentioned previously, that Meshblu's Websocket API sends double notifications to subscribed devices.With that said, the client registered 1384 notifications arriving, but despite that, 283 notifications weren't delivered, and the ones that arrived did so with an average time of 17.82 seconds.

The simulation platform only registers a failed publish if it is caused by the broker, meaning, every publish is considered successful unless an error response is sent by the broker. To counter that we can take a look at the parser's output and see what was going on in Meshblu. To do this we'll start by taking a look at the text files produced by the parser, in this case the "websocket.txt" and more specifically at the device's TCP stream, where the Websocket messages that don't need a response are. We do this because Meshblu doesn't answer to successful publish requests, and when the parser runs and encounters messages from the client that do not require a response (specified in the configuration file) it adds them to a dictionary with the message type as the key ("message" in this particular case).

From the "websocket" text file we can see that one notification did not arrive, which still doesn't explain why the other 282 messages didn't reach the subscriber. The next step now is to see if the notifications were sent by the "websocket" adapter, and to do this we check, in the TCP stream of the client, the array "WS from Server", which is the array that holds messages sent by the server which are not responding to any requests. The parser assumes such packets are notification publishes, either traveling through the broker or directly reaching the client (which is the case here). Here we have, just like the client registered, 1384 packets, which makes the protocol adapter most likely not at fault as well as eliminates the possibility of it being a problem with the simulation tool.

Next we check with whom the "websocket" microservice communicates with, checking first the communications by request and response, in case the adapter redirects messages as in the case of DeviceHive's "dh_proxy". Since that's not true we then move to redis graphs (Figure 5.9), and here we see a spike in sent data, followed by a decreasing amount of received data, and when we move on to the text file, the parser extracts information from the REDIS packets and we can see which notifications were sent to "redis" and sent from "redis". From this analysis we can see that 1000 notifications we sent by the protocol adapter with a "jobType" named "SendMessage", but 1384 notifications were received back from "redis", without a "jobType" name (which are the messages that "redis" sends to the protocol, which sends to the nodes connected). Following this leads to the "dispatcher" component, we can also see communications between "redis" and the "dispatcher" with different "jobTypes", but the interesting thing about this is that the dispatcher sends all the notifications received by "redis", but instead of the "jobType" being "SendMessage" we have "DeliverBroadcastSent", "DeliverBroadcastMessage" and "DeliverSentMessage". When "redis" sends back the notifications with the separate "jobTypes" we see that some notifications

are now missing. Afterwards the dispatcher sends the ones without "jobType" that are redirected to the client, loosing some here too.

This suggests that the problem is actually in "redis" along with the dispatcher and that's where notifications are being lost.

The effects of the double Websocket messages can also be explained here, as the "dispatcher" sends back not only messages to the nodes subscribed to the device (with a "jobType" set as "DeliverSentMessage", to "redis"), but also a broadcast message (with a "jobType" set as "DeliverBroadcastSent"), since the target devices list on the message is "["*"]". This overlook by the author can be easily solved by simply changing the message's target devices.

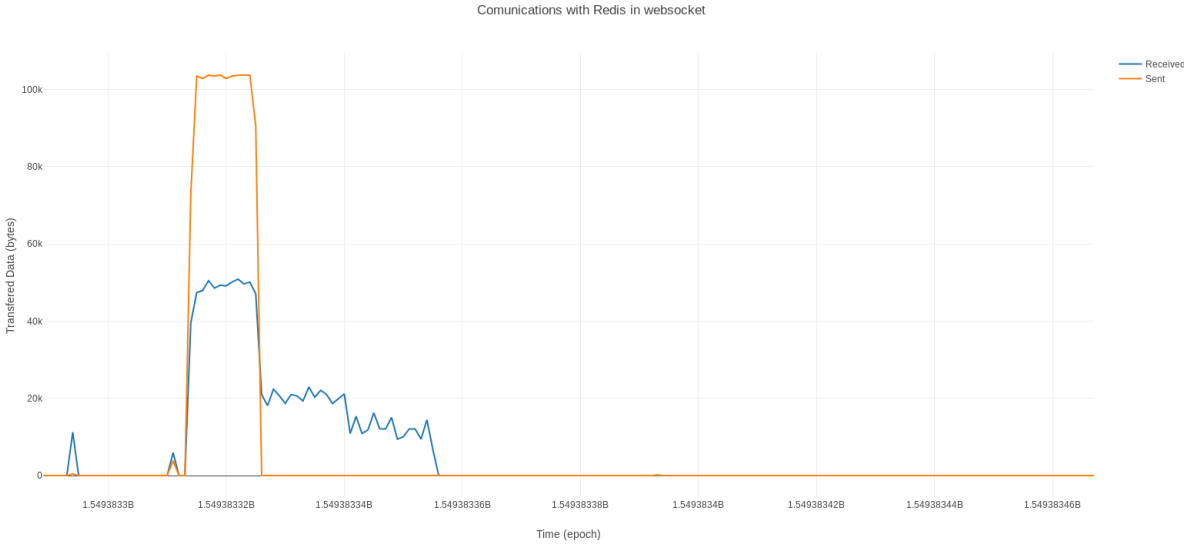

Figure 5.9: Redis data per second captured on the microservice "websocket".

5.3.2 MQTT Simulations

With the same initial scenario as the one used with Websocket, while using MQTT, the broker sent 1000 successful publishes (at least not resulting in an error by the broker) but only received 473 notifications, with an average time of 16.57 seconds. We can refer to Figure 5.10 to see this difference, but noting that, using MQTT, the broker lost more 244 notifications than when using Websocket.

Taking a look at the protocol adapter text file we can see the same thing that happened with Websocket, where the protocol adapter sent all the notifications to "redis" (with no loss), but only received 473 to send to the client. If we now take a look at the "dispatcher" text file we see that it received 999 notifications from "redis", which indicates that one was lost, but when looking at the notifications sent back to "redis" by the "dispatcher" we see that all the 1000 notifications were sent, which suggests a failure of the parsing tool. But since no "REDIS" packet with the notification in question was found on the captures themselves, this issue was considered relative to the capturing and not the parsing.

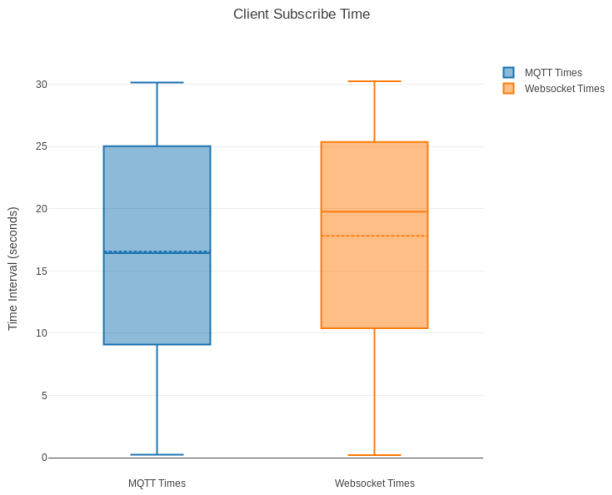

Figure 5.10: Comparison of the subscribe times of MQTT or Websocket in Meshblu.

The "dispatcher" then proceeds to do the same as in Websocket, where it assigns "jobTypes" and sends the notifications back to "redis" which answers with 4 notifications missing. The "dispatcher" then sends back 510 notifications to "redis" to be sent to the client. This shows a lot of missing notifications, which indicates that the problem lies, once again, on the "redis" and "dispatcher" microservice, which is where the notifications are being lost. Since "redis" is only responsible for the microservices communications, we can conclude that using this strategy for the internal communication is not effective and cannot scale well.

5.3.3 Time Comparison

Just as with Devicehive, in Figure 5.11 we present a graph comparing times between using MQTT and Websocket, with and without capturing traffic. Since with the previous experiment, using a maximum of 100 messages per second, Meshblu failed to send back a significant amount of notifications, for this comparison we used a maximum of 10 messages per second. In this experiment, all notifications arrived to the subscriber.

5.3.4 Multiple Devices

Taking a look now to the behaviour of the broker when facing multiple devices, we start by taking a look at how it handles the same number of 100 messages per second and 1000 notifications. It should be noted that with 2 devices, it lost 483 notifications, with 5 devices, lost 524, and with 10 devices it lost 473. In Figure 5.12 we can see the resulting times.

Since it was already seen that Meshblu can't handle this rate of messages, it was also conducted experiments using a rate of 10 messages per second. In this case all the notifications arrived to the subscriber, and the times can be seen in Figure 5.13. Here we can see a similar behaviour to that of DeviceHive, with an increase of time between sending and receiving notifications according to the number of devices publishing them. It should be noted once again that the rate of messages

stated is the rate at which the messages arrive to the broker, not the rate at which each individual device is sending messages.

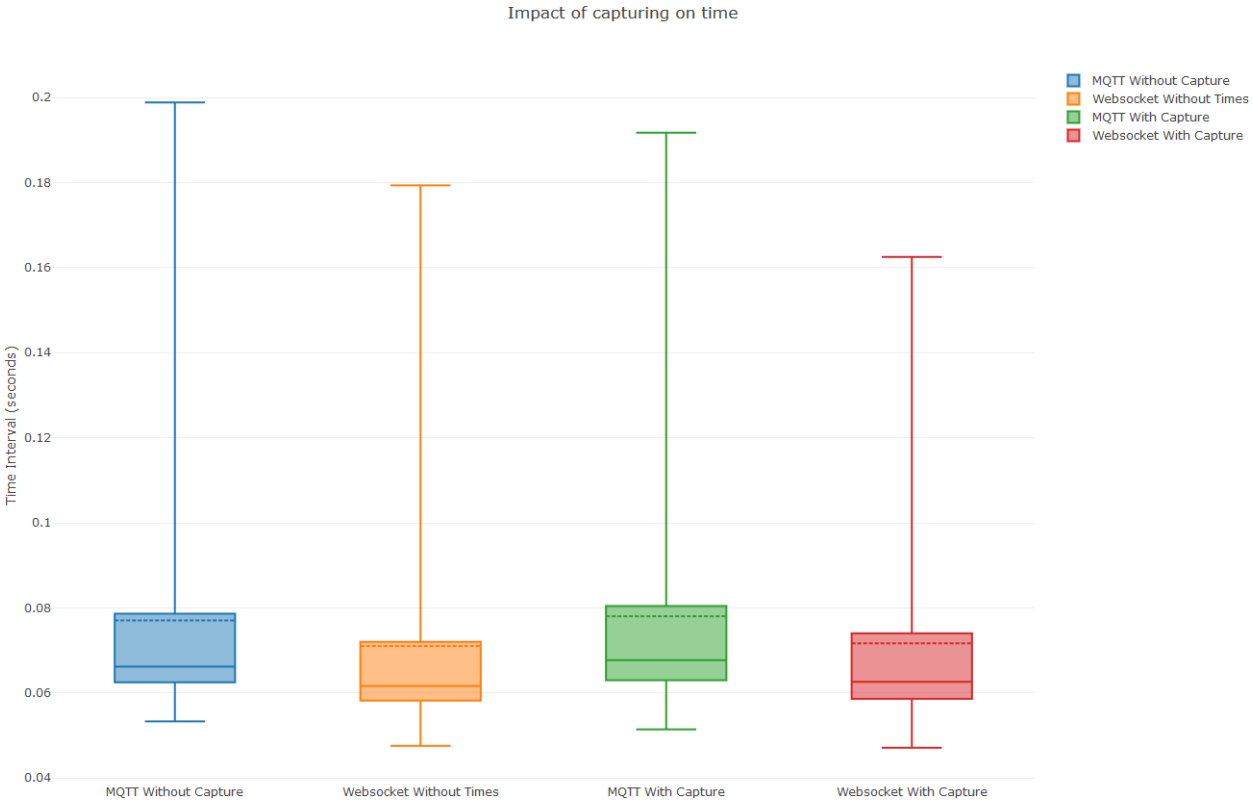

Figure 5.11: Comparison of the subscribe times in Meshblu with and without capturing traffic.

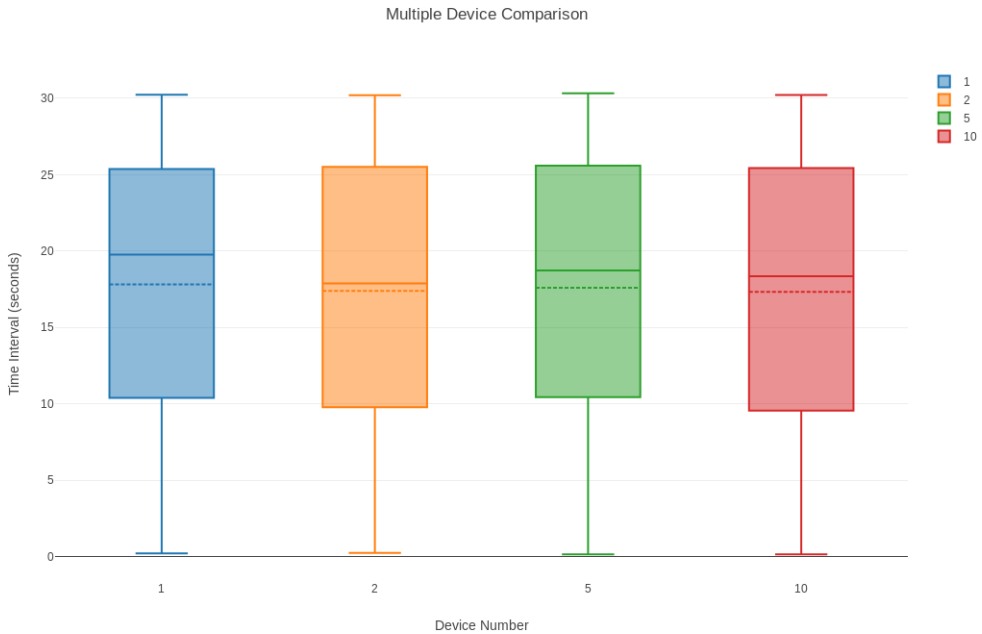

Figure 5.12: Comparison of times while using multiple devices in Meshblu, with 100 messages per second.

Comparing both graphs, we can see that past its break point there is no utility in analysing these times for the broker and the information is not really reliable. In this case, an analysis of the internal communications of the broker (as the ones made above) is more useful into understanding what is failing within the broker itself.

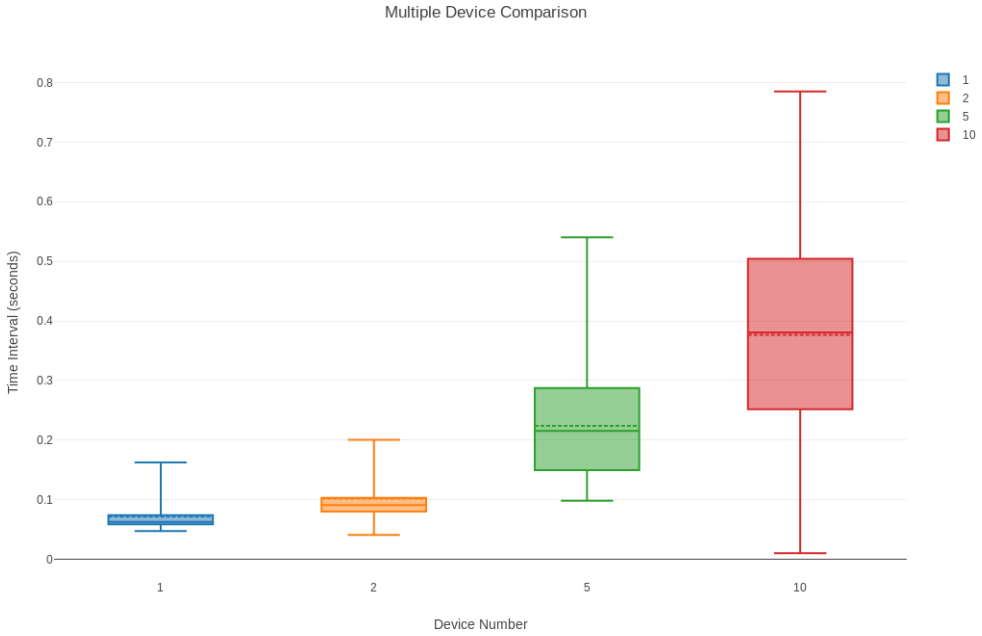

Figure 5.13: Comparison of times while using multiple devices in Meshblu, with 10 messages per second.

Chapter 6

Conclusion

6.1 Goals and Hypothesis

Having performed the experiences and analysis of the results obtained with the platform in Chapter 5, we should now recall the main goals and hypothesis to test with this dissertation:

$\mathscr{G}_1$. **Identify a set of objective measurement criteria to evaluate the message brokers microservices' performance in a Smart City scenario.**

$\mathscr{G}_2$. **Develop a platform able to simulate smart city scenarios.**

$\mathscr{G}_3$. **Develop a platform that automates the collection of information relevant to the performance analysis, and that parses it in order to facilitate said analysis.**

Looking back on these goals, we can evaluate how the work done during this dissertation and the resulting tools developed, accomplished them. In regards to Goal 1, during the development of this dissertation, some metrics and criteria were successfully identified that aid in the performance analysis of a broker and its microservices. Given the nature of the platform, the end result culminated in more of a analysis support tool, nonetheless, some metrics produced allow a comparison between brokers and an evaluation of their microservices performance, and from the evaluations done on the experiments above, it's consider that the current work proposed and made use of a useful set of metrics. About the developed platforms for capturing and parsing information from a test scenario of a broker we can say that they are capable of collecting data without a drastic impact on the broker's performance, and from the analysis made above, it is considered to produce results that aid in a performance analysis of a broker and its internal implementation, as well as to understand the internal operation of the broker. Despite the attempts to make the protocol parsers modular and simple to integrate new ones with the platform, there is still some adjustments to be

made to the platform when a protocol is implemented and it is easily seen that there is plenty of room for improvement. In regards to the compatibility with different brokers, it is considered that to add a new broker is rather simple, since it only requires a configuration file with specifications about messages and variables to use.

Despite the fact that the experiments conducted were limited in regards to the number of brokers taken into consideration, it is seen from the simulations above that DeviceHive is far superior to Meshblu, and should be taken into consideration as an alternative for Citibrain.

Moving now to the hypothesis considered at the beginning of this work:

$\mathscr{H}_1$. It is possible to automate the process of collection and parsing of information in order to obtain the defined benchmarking metrics;

$\mathscr{H}_2$. A simulation platform can be developed in order to simulate smart city scenarios;

$\mathscr{H}_3$. The addition of compatibility with new message brokers or communication protocols can be made simple.

With the work done during this dissertation it is possible to draw some conclusions about these hypothesis, by summarizing the developed tools. By this point it is available a set of tools that collect and parse results which aid in the analysis of a broker and its behaviour given a test scenario. Despite being still crude performance-wise, the platform can still produce a good amount of results in a short period of time, relative the amount of data to parse.

Moreover, the tools developed allow for a somewhat easy integration of new protocols and brokers, which is still a characteristic with the possibility of much more improvement to facilitate said integration further. Moreover, the existing protocol parsers can be easily modified to extract more or less information from the data provided.

In regards to the simulation tool, it offers a good amount of customization, allowing to have more than one client or device active, a client subscribed to multiple devices and customization of payload size and the minimum interval between messages. Besides that, it is not hard to add compatibility with more protocols, and the compatibility with the broker, as it was mentioned, is made using a simple configuration file. It is also capable of simulating multiple protocols at the same time, though it should be noted that the clients and devices are nested according to protocol, for example, a client using Websocket won't get notifications from a device using MQTT. This decision was made at the time to keep things more manageable.

The main issue with the developed simulation platform, is that it simulates clients and devices using threads, with a thread being created for each client and device (and also for message processing, in order to avoid loosing incoming messages). Because of this, simulating a Smart City scenario, with thousands of devices, is not viable as there is always a physical limit associated with the machine being used.

6.2 Conclusion and Future Work

With this study, we developed three separate components that facilitate the analysis of a message broker's internal functioning and performance. We learned from this development that a generalization of such platform is not simple, and for each broker there are many different particularities that interfere with the development of such platform. That said, it is believed that a base platform was designed which facilitates that analysis and the integration of more brokers. From the testing it was also seen how the information produced by the developed platform is useful and how it can be used to narrow a problem with a broker to specific microservices. Nevertheless it should be taken into consideration that some adjustments need to be made in order for the platform to produce more content specific results, such as the case of Meshblu, where an analysis of the content of Redis messages received a more thorough parsing.

Moreover it was also presented a simulation platform which successfully simulates publishers and subscribers, giving the user a fair amount of customization. And despite working separately from the parser it should be taken into consideration that it offers content specific details which enable a more thorough analysis of the content by said parser.

Despite that, it is not hard to conclude that a lot of work can still be done in regards to performance analysis of message brokers and test scenario simulation. Starting with the simulation of Smart City scenarios, which given the scale should be simulated also making use of distributed computing techniques.

Besides that, it should also be considered one of the main limitations of the tool as a whole, which is that is depends on being deployed using containers. As future work it should be considered ways to capture and parse information about a broker distributed across machines and deployed using other technologies. Nonetheless, the parser and the simulator are independent from this issue, and is only relative to the collection of information.

There is also a lot of improvement to be made in regards to the parsing of the information, mainly because there is still not a big variety of protocols implemented, and some metrics that may be relevant could have been overlooked by the author. Moreover, there is some information that can be represented in a more readable way, which for now is only presented in the text files and can be time consuming to go through (for example, the case with the Redis information).

Moreover, it should be also considered the poor performance of the developed parsing tool. Besides not being polished and optimized, it is also developed using Python, which by itself doesn't offer good performance. This can be a drawback when dealing with large amounts of data, and the fact that the tools themselves are not polished just accentuates this, where the parsing of 300MiB capture can take up to 60 minutes. Not only that, but the way Python deals with multithreaded applications is also not optimal, hence deepening the need for an implementation on a more efficient language.

References

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[B+12] M Bauer et al. The internet-of-things architecture. *European Commission with the Seventh Framework Programme*, 2012.

[Bat13] Michael Batty. Big data, smart cities and city planning. *Dialogues in Human Geography*, 3(3):274–279, 2013.

[BH08] Alessandro Bassi and Geir Horn. Internet of things in 2020: A roadmap for the future. *European Commission: Information Society and Media*, 22:97–114, 2008.

[BSMD11] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. Role of middleware for internet of things: A study. *International Journal of Computer Science and Engineering Survey*, 2(3):94–105, 2011.

[CNW+12a] Hafedh Chourabi, Taewoo Nam, Shawn Walker, J Ramon Gil-Garcia, Sehl Mellouli, Karine Nahon, Theresa A Pardo, and Hans Jochen Scholl. Understanding smart cities: An integrative framework. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 2289–2297. IEEE, 2012.

[CNW+12b] Hafedh Chourabi, Taewoo Nam, Shawn Walker, J. Ramon Gil-Garcia, Sehl Mellouli, Karine Nahon, Theresa A. Pardo, and Hans Jochen Scholl. Understanding smart cities: An integrative framework. *Proceedings of the Annual Hawaii International Conference on System Sciences*, pages 2289–2297, 2012.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[GGP05] J Ramón Gil-García and Theresa A Pardo. E-government success factors: Mapping practical tools to theoretical foundations. *Government information quarterly*, 22(2):187–216, 2005.

[GIMA10] Daniel Giusto, Antonio Iera, Giacomo Morabito, and Luigi Atzori. *The internet of things: 20th Tyrrhenian workshop on digital communications*. Springer Science & Business Media, 2010.

[GPM16] Amitranjan Gantait, Joy Patra, and Ayan Mukherjee. Design and build secure IoT solutions , Part 1 : Securing IoT devices and gateways. *IBM developerWorks*, pages 1–20, 2016.

REFERENCES

[Har05] Jean Hartley. Innovation in governance and public services: Past and present. *Public money and management*, 25(1):27–34, 2005.

[HBB+00] Robert E Hall, B Bowerman, J Braverman, J Taylor, H Todosow, and U Von Wimmersperg. The vision of a smart city. Technical report, Brookhaven National Lab., Upton, NY (US), 2000.

[HEH+10] Colin Harrison, Barbara Eckman, Rick Hamilton, Perry Hartswick, Jayant Kalagnanam, Jurij Paraszczak, and Peter Williams. Foundations for smarter cities. *IBM Journal of Research and Development*, 54(4):1–16, 2010.

[HKM+17] Daniel Happ, Niels Karowski, Thomas Menzel, Vlado Handziski, and Adam Wolisz. Meeting iot platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72(1-2):41–52, 2017.

[Kal14] Vivek Kale. *Guide to cloud computing for business and technology managers: from distributed computing to cloudware applications*. Chapman and Hall/CRC, 2014.

[NMMA16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[NP11] Taewoo Nam and Theresa A Pardo. Smart city as urban innovation: Focusing on management, policy, and context. In *Proceedings of the 5th international conference on theory and practice of electronic governance*, pages 185–194. ACM, 2011.

[PCAM18] Carlos Pereira, João Cardoso, Ana Aguiar, and Ricardo Morla. Benchmarking pub-/sub iot middleware platforms for smart services. *Journal of Reliable Intelligent Environments*, 4(1):25–37, 2018.

[RAPR16] M Mazhar Rathore, Awais Ahmad, Anand Paul, and Seungmin Rho. Urban planning and building smart cities based on the internet of things using big data analytics. *Computer Networks*, 101:63–80, 2016.

[RFK+07] Giffinger Rudolf, Christian Fertner, Hans Kramar, Robert Kalasek, Natasa Pichler-Milanovic, and Evert Meijers. Smart cities-ranking of european medium-sized cities. *Rapport technique, Vienna Centre of Regional Science*, 2007.

[Sil16] Ivo Daniel Silva. IoT Standards for Smart Cities. Master's thesis, Faculty of Engineering of the University of Porto, 2016.

[SLM17] Long Sun, Yan Li, and Raheel Ahmed Memon. An open iot framework based on microservices architecture. *China Communications*, 14(2):154–162, 2017.

[TSH09] Ioan Toma, Elena Simperl, and Graham Hench. A joint roadmap for semantic technologies and the internet of things. In *Proceedings of the Third STI Roadmapping Workshop, Crete, Greece*, volume 1, pages 140–53, 2009.

[WHZZ15] Feng Wang, Liang Hu, Jin Zhou, and Kuo Zhao. A data processing middleware based on soa for the internet of things. *Journal of Sensors*, 2015, 2015.

[ZMA18] Luís Zilhão, Ricardo Morla, and Ana Aguiar. A modular tool for benchmarking iot publish-subscribe middleware. 2018.