Tampere University

Wouter Legiest

DESIGN OF A BACK-END FOR A CAMERA BASED PERSON DETECTION SYSTEM

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
August 2019

ABSTRACT

Wouter Legiest: Design of a back-end for a camera based person detection system
Master of Science Thesis
Tampere University
Master of Electronics and ICT Engineering Technology (KU Leuven)
August 2019

In this thesis, a back-end web server is developed for the CityTrack project. The project uses modern Deep Learning techniques to provide object and people detection on embedded devices. By using multiple of these devices, detection nodes, statistical data can be collected about a certain venue or event. To expand this project, a web application is needed to visualise the data with the possibility to watch in real time. In addition to the web application, a central database should be established to provide long-time structured storage for the detection data.

To make well-considered choices, different technologies are discussed and weighed against each other. For instance, for the communication between the detection nodes and the web application, the HTTP-based REST architectural style and SOAP protocol are compared to the MQTT protocol. Furthermore, the real time capable communication technologies WebSocket, Sever-Sent Event and HTTP Long Polling is reviewed. The system uses the REST architectural style due to practical implementation reasons and WebSocket due to the limitations of the other alternatives. The layered architecture is then discussed to arrive at a proposal for a more modern version of the web architecture. The theoretical background and implementations of all components are then discussed. The advantages and disadvantages of each implementation are reviewed and a thoughtful choice was made.

To make a sustained choice, the performance of different WSGI server implementations are tested. A WSGI server is an interface between a web server and a Python-based framework. The ApacheBench stress testing tool examines different aspects of the performance. The result is that the *uWSGI server* performs the best on both latency and throughput aspect compared to the other candidates tested.

Also, the performance of the various implementations of ASGI server has been tested analogously. An ASGI interface is a superset of WSGI with additional support for asynchronous communication technologies. Implementations of the ASGI interface are tested on the WSGI functionalities. In this way, it is investigated whether the current implementation of ASGI could replace the WSGI server. The results show that the current implementations of various ASGI servers underperform to replace a WSGI server.

Keywords: web application, WSGI server, Deep Learning, web server, containerisation, CityTrack

PREFACE

This thesis is a product of my Erasmus exchange between the KU Leuven and the Tampere University and would not have been possible without the support of various people and institutions. During my time in Tampere, this thesis was established.

First, I would like to thank the KU Leuven, Ghent Technology Campus for allowing me to study at their university and more specifically my local supervisor Assistant Professor Tony Wauters.

Secondly, I would like to express my deep gratitude to my first supervisor Associate Professor Heikki Huttunen for the confidence that I received as a foreign student and for giving me the unique opportunity to join the Machine Learning Group during my Erasmus. This academic period would never have been possible without him. I also would like to thank my second supervisor Associate Professor Kari Systä, for the support of the technical aspect of this thesis.

Finally, I would like to thank the members of the Machine Learning Group for creating an educational and pleasant work environment. I also would like to acknowledge all my family and friends who supported me during this period in Finland and the writing of this thesis.

Tampere, 8th August 2019

Wouter Legiest

CONTENTS

1 Introduction . . . 1
2 Background of the project . . . 3
  2.1 Goal of this thesis . . . 3
  2.2 The requirements of the server . . . 4
  2.3 Practical realisation . . . 5
    2.3.1 Deep learning . . . 5
    2.3.2 Hardware . . . 6
  2.4 Related work: comparing to existing IoT frameworks . . . 6
3 Background of the general web server . . . 8
  3.1 Fundamentals of a web server . . . 8
    3.1.1 Evolution of web pages . . . 8
    3.1.2 Software stack . . . 10
    3.1.3 Deployment . . . 10
  3.2 Communication protocols . . . 12
    3.2.1 Hypertext Transfer Protocol . . . 12
    3.2.2 Message Queuing Telemetry Transport . . . 14
  3.3 Real time websites . . . 14
    3.3.1 HTTP Long Polling . . . 15
    3.3.2 WebSocket . . . 16
    3.3.3 Server-Sent Events . . . 16
4 Principles of designing a server platform . . . 17
  4.1 Layered architectures . . . 17
  4.2 Modern web server model . . . 19
    4.2.1 Web server . . . 19
    4.2.2 Application server . . . 20
    4.2.3 Database . . . 21
    4.2.4 Caching service . . . 22
    4.2.5 Task queue . . . 22
    4.2.6 Third party services . . . 23
    4.2.7 Cloud services . . . 23
    4.2.8 Content Delivery Network . . . 24
  4.3 Virtualisation . . . 24
5 Cloud implementation of the server platform . . . 26
  5.1 Used technologies . . . 26
  5.2 The implemented stack . . . 26
    5.2.1 Selection procedure . . . 27

5.2.2 Overview . . . 27
5.2.3 Web server . . . 28
5.2.4 Web framework . . . 30
5.2.5 Web Server Gateway Interface . . . 31
5.2.6 Asynchronous Server Gateway Interface . . . 32
5.2.7 Distributed workers . . . 32
5.2.8 Time series database . . . 33
5.2.9 Specific library and packages . . . 33
5.2.10 Overview of the web application . . . 33
5.3 Implementation details . . . 34
5.4 Deployment . . . 35
5.4.1 Containerisation . . . 35
5.4.2 Container stack . . . 36
5.4.3 Cloud platform . . . 37
6 Performance evaluation . . . 38
6.1 Performance of different WSGI servers . . . 38
6.1.1 Situation of the research . . . 38
6.1.2 Test setup . . . 39
6.1.3 Results . . . 40
6.2 ASGI as a replacement for WSGI . . . 43
7 Conclusion & Future . . . 48
7.1 Development process and choices . . . 48
7.2 Discussion of the testing results . . . 48
7.3 Future challenges . . . 49
References . . . 51

LIST OF FIGURES

2.1 High-level description of the project 4
2.2 Artificial neural network with three layers 5

3.1 Providing of web pages, adapted from [86] 9
3.2 Visualisation of cloud services, adapted from [109] 12
3.3 Publish-subscribe-based messaging protocol 14
3.4 Example of the HTTP Polling and HTTP Long Polling technique 16

4.1 2-layer architecture for Web application, by Anastopoulos et al. [5, p. 40] . . 18
4.2 n-layer architecture for Web application, by Anastopoulos et al. [5, p. 42] . . 19
4.3 Modern web server architecture, based on [46] 20
4.4 Example of a star schema 22
4.5 Advantage of using CDN [111] 24
4.6 Visualisation using virtual machine and container 25

5.1 Current implementation of the project 34
5.2 Visualisation of the Docker Compose stack 37

6.1 Latency of multi workers WSGI servers 40
6.2 Throughput of multi workers WSGI servers 41
6.3 Error Rate of multi workers WSGI servers 41
6.4 CPU Usage of multi workers WSGI servers 42
6.5 Memory Usage of multi workers WSGI servers 42
6.6 Latency of single workers ASGI and WSGI servers 44
6.7 Latency of single workers ASGI and WSGI servers - magnified 44
6.8 Throughput of single workers ASGI and WSGI servers 45
6.9 Error rate of single workers ASGI and WSGI servers 45
6.10 CPU Usage of single workers ASGI and WSGI servers 46
6.11 Memory usage of single workers ASGI and WSGI servers 46

LIST OF TABLES

3.1 Overview of different software stacks 11
3.2 Different levels of quality of Service 15

4.1 Extra functionalities of a web server [15] 20
4.2 Several third party services 23

5.1 Questions used to choose a package 27
5.2 Overview of considered implementations 28
5.3 Extra implemented Python packages 33
5.4 List of the implemented containers 36

6.1 Summary of all the tested connections 39
6.2 Settings of each WSGI server 40

LIST OF SYMBOLS AND ABBREVIATIONS

ANN	Artifical Neural Network
API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
CDN	Content Delivery Network
CRUD	Create, Read, Update, and Delete
CSS	Cascading Style Sheets
DBMS	Database Management System
DNS	Domain Name System
DRY	Don’t repeat yourself
FIFO	First-in-first-out
HTML	HyperText Markup Language
HTTP(S)	Hypertext Transfer Protocol (Secure)
IaaS	Infrastructure-as-a-Service
IP	Internet Protocol
JSON	JavaScript Object Notation
MPM	Multi-Processing Module
MQTT	Message Queuing Telemetry Transport
MVT	Model-Template-View
ORM	Object-Relational Mapping
OS	Operating System
PaaS	Platform-as-a-Service
RDBMS	Relational Database Management System
REST	Representational State Transfer
RPI	Raspberry Pi
RTT	Round-Trip Time
Saas	Software-as-a-Service
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSE	Server-sent events

SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSDB	Time Series Databases
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
WSGI	Web Server Gateway Interface
WWW	World Wide Web
XML	Extensible Markup Language

1 INTRODUCTION

The Internet is one of the most important inventions of the past 50 years. This interconnected computer network is used to provide applications of the World Wide Web (WWW), electronic mail, telephony, and file sharing. It all began in 1965 at the National Physical Laboratory in England with the NPL Network. This was the first local area computer network that uses packet switching.

At the same time, the development of the similar Advanced Research Projects Agency Network (ARPANET) was done in the United States of America [95]. The reason for the creation of the ARPANET was an economic one, computers were expensive and through this network, devices could be shared [19]. With the usage of packet switching, different networks could be linked together to create a network of networks. During a redefinition of ARPANET, in 1982, the Internet protocol suite (TCP/IP) was standardised. This made it possible to connect other networks anywhere in the world. The ARPANET itself was deactivated in 1990. [95]

With the standardisation of the Internet protocol suite, the widespread global Internet as we know it was born. In essence, the Internet is still nothing more than a network of networks, making it possible for computers to communicate with other computers and providing the infrastructure to design the World Wide Web with the first web server in 1994.

On the other hand, during the same period, research on neural networks won a lot of interest. Back in 1943, Walter Pitts and Warren McCulloch realised the first development of the artificial neuron, a mathematical model of the biological neurons. They tried to mimic the thought process of the brain. This was the first step toward the research of the artificial neural network. [49, 72]

In 2006, the reputation of neural network rose again with Hinton et al. proposing a new more efficient method to training a deep belief network, a specific neural network [55]. Due to the limited computing resource at the beginning of the 70s and after that the creating of too high expectations about the field, the popularity dropped again in the past [76]. These days the computing power of GPUs has increased significantly and by using GPU for the training of a network, the training time of a network reduces noticeably. Furthermore, the term *Deep Learning* was used to indicate that it was possible to train deeper neural networks than had been possible before. [49]

Today, the World Wide Web and Deep Learning are two of the most favoured topics in computer science. The merging of the two fields is inevitable.

The purpose of this study is to design the first iteration of a back-end server. This server will provide a stable web application with a database for the project. The project uses low-cost cameras to track and detect people, the back-end server should then be capable of receiving, storing and processing the data that is coming from the cameras. To design this application, existing research was combined and additional research was conducted on the Web Server Gateway Interface (WSGI), an interface to provide communication between a web server and a Python-based web framework.

In the second chapter, more explanation is given about the CityTrack project. The requirements of the server are declared and the practical realisation of the project is discussed. Chapter 3 provides a brief overview of the history of web servers and describes the available protocols for communication and real time capabilities. A modern version of the web server architecture is proposed in Chapter 4, while Chapter 5 discusses the implemented stack. In Chapter 6 experiments around the performance of various WSGIs are conducted. This thesis ends with Chapter 7 which proposes a conclusion of all the previous parts.

2 BACKGROUND OF THE PROJECT

CityTrack is one of the research projects at the Faculty of Information Technology and Communication Sciences of the Tampere University. In this project, the research of the machine learning (ML) group is used to detect objects and people from a camera view in an indoor environment. On an embedded device with an attached camera a neural network is running. The camera images are entered in the network to detect different objects and persons. The multiple detections are then sent to a collection point. One of the assets of the project is that the camera footage is never leaving the embedded device, only the results of the neural network are sent to a collection point.

The goal of the project is to develop a sensor network with an associated server to collect statistical data about an indoor venue, while the challenge with the hardware is that the computational resources are limited.

One of the advantages of this project is that in the case of people detection, a person does not need to interact with any kind of device. As an illustration, to measure the number of attendees at an event, multiple different pointed lasers could be used to discover people at the entrance. If someone with a smaller figure walks in front of a larger person, it is possible that the small person not will be detected. This project uses different angle cameras directed at the entrance of the event hall. The same person can be recognised by various cameras and the coordinates of the persons could be gathered. To prevent double-counting, re-identification is used to deduce it. This algorithm is capable of connecting two different person detections to each other. This way the same person can be detected in multiple cameras and a much more accurate counting can be done. A disadvantage of using cameras is that the room must be sufficiently illuminated, the contrast between the person and the background should be large enough.

Another use case is the discovery of trajectories on conferences. When a camera detects a person it is possible to track that person inside the camera area. Then using re-identification is possible to build the walked paths. With the complete set of data, popular parts of the conferences can be recognised.

2.1 Goal of this thesis

This thesis is about the research and design of the build-up of the back-end server for the CityTrack project. (Fig 2.1 on the following page) The back-end server should provide a modern web application to visualise the data, whether or not in real time. Also, the server should provide a database for long-time structured storage to calculate easily and fast the statistical data and a possibility to run a resource-intensive ML-related algorithm on the server because the computational resources of the embedded devices are rather limited. The system should only use quick and stable component, this way a reliable back-end server can be built.

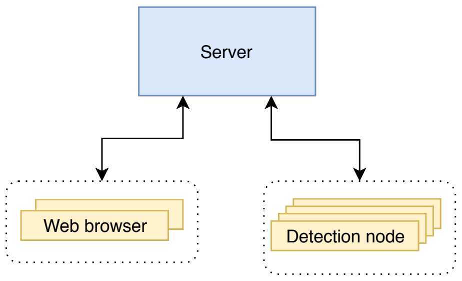


Figure 2.1. *High-level description of the project*

This thesis focuses more on the development and implementation of the back-end side of the web application for providing a foundation to the other parts of the project. With back-end is meant everything with which the visitor does not come into contact. Everything that the visitor can interact with through the browser also called the front-end, is not part of the scope of this thesis.

2.2 The requirements of the server

The purpose of this research project is to develop a useful end product which can be implemented in different areas. This gives the project functional properties that determine the appearance of the project. To achieve this, a few general goals as listed below.

1. Easy to use website, the user-base could include non-experts
2. No camera images leave the embedded device, to avoid privacy issues, only the detection data is sent out of the devices.
3. Scalable: The system should be capable of handling 200 detection nodes
4. The data is secured using contemporary standards

Besides these general project needs, the server-side has are also some more non-functional system specifications. These specifications are used a basis for this thesis.

1. Making use of containerisation, for easy mobility and deployment
2. Making use of Python, the standard programming language inside the ML Group
3. Real time possibilities to view the currently occurring detection
4. Try to use as much open-source software as possible
5. A robust and high-performing system

6. Designed to run for a long time, it is possible that the server will run on a server of the university in the future
7. For this initial setup, make use of the cloud services of the university

2.3 Practical realisation

To realise this project, low-priced detection nodes are used. These nodes include a computing unit and an attached camera. By keeping the cost of a single node low, it is possible to install more nodes inside an indoor venue. Hereby, more data from different points can be gathered, resulting in a more detailed overview of the venue.

The upper limit of this project would be system consisting out of 200 detection nodes. Every node is possible to process ten frames per second and in one single frame the neural network could detect up to 50 people or objects. The top culmination point of one device can be estimated at $10 \cdot 50 = 500$ detections per seconds per device. The total web server should be capable of handling 100 000 detections per second. This requirement will be taken into account.

2.3.1 Deep learning

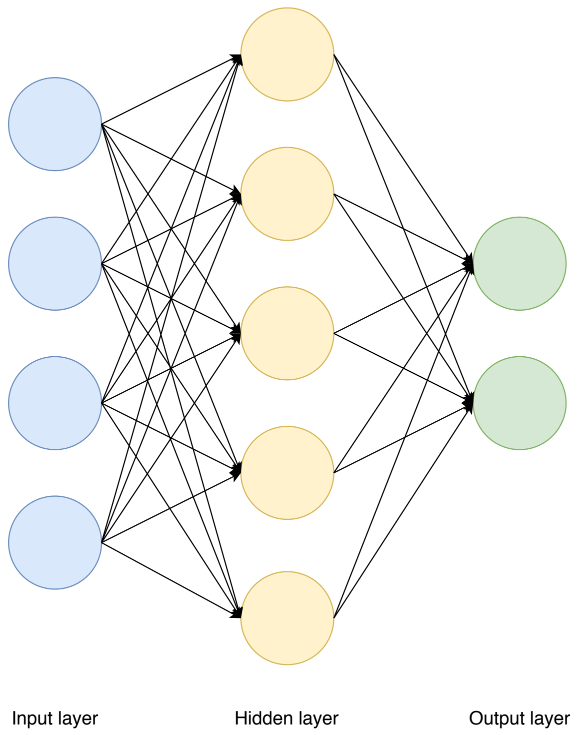


Figure 2.2. *Artificial neural network with three layers*

The detection nodes use deep learning technologies to detect objects and person on the camera images. Deep learning is based on the usage of artificial neural network (ANN).

The structure of an ANN is inspired by the biological neural networks found in animal brains. This is realised by building a collection of artificial neurons. Each connection can transmit a signal from one neuron to another, analogous to the synapses of the biological brain. The artificial neuron can process the incoming signals and send an outgoing single to all the connected neurons (Fig. 2.2 on the previous page). [49]

In common ANN implementations, the output of each neuron is calculated by some non-linear function of the sum of its inputs, generating a real number to transfer to the other neurons. The connections between the neurons are called *edges*. Each edge typically has a weight that adjusts as the learning proceeds. By adjusting the weight, the strength of the signal adapted. A neuron itself can have a threshold, an outgoing signal is only sent if the aggregate signal is higher than the threshold. Multiple neurons all typically aggregated into layers. [49]

The ANN *learns* a specific task through a training process. During this process the network will be shown correct inputs and outputs, hereby learning this mapping by adjusting the weights. Due to the training-based approached of the ANN, the network can output erroneous data. For example, a false positive error could occur. In this situation, the result indicates that a given condition exists, when it does not. Another example is the occurrence of a false negative error. Here, the results indicate that the condition does not exist, while it does. [49]

At first, ANN was developed to solve a problem in the same way that the human brain would. Later, there was more interest to use this technology for specific tasks. This tasks, such as among others, computer vision, speech recognition, machine translation and social network filtering can use a neural network. In this project, the SSD algorithm is used, this is an object detection method that uses deep learning-based approaches [68].

2.3.2 Hardware

Every detection node consists of an embedded device, a camera and a neural compute stick. As an embedded device the Raspberry Pi (RPI) 3 Model B+ is chosen [91]. This device has a built-in WiFi chip and a 1.4 GHz 64-bit quad-core processor. The WiFi chip provides an easier installation and due to the form factor of the board, the detection node can be as big as a hand. Attached is the Raspberry Pi Camera Module v2 [17]. The Intel Movidius Neural Compute Stick is used to run the neural network [58]. The stick has optimised low-power hardware to generate better performance for deep learning tasks.

2.4 Related work: comparing to existing IoT frameworks

In this project, a web application is build using a web framework. An alternative would be using an IoT framework. This already provides communication interfaces for the embedded devices, database handles and visualisation and plotting tools. In the coming section, the choice to design our own web site is motivated.

Alternatives for designing a complete own web application would be using one of the IoT frameworks *ThingsBoard* or *FIWARE*. In 2016, López-Riquelme et al. proposed a software architecture based on the FIWARE platform. They have developed a cloud-based Internet of Things (IoT) platform of Precision Agriculture applications. The paper both describes the development of the IoT sensor nodes and the FIWARE server. Their server consists of combining FIWARE components, MySQL Database and Tomcat Server to provide complete web services. [69] On the other hand, De Paolis et al. have been using ThingsBoard to build a real time IoT platform. While ThingsBoard handles the MQTT connection of the sensor nodes, the added Spark Streaming framework provides a cluster computing platform for data analysis. [23]

FIWARE is a collection of components to build an IoT-enabled server. Each provides a distinct service, like a central context broker or a connection tool for communication with a database. [25] Further, ThingsBoard is a complete IoT platform providing an all-in-one environment. [47]

Both platforms offer tools for displaying an interactive map. The project would also be applied to an indoor environment, hereby needing the support of indoor mapping tools. Both platforms do not support this feature. Because of this, full control over the design of the web page is needed, both frameworks do not provide this. Moreover, the collaboration between ThingsBoard with and another framework or platform is only supported by the Professional Edition of ThingsBoard.

In this first iteration of building and designing the server, a regular web framework will be used to construct the server and website. Both frameworks do support a range of communication technologies like Message Queuing Telemetry Transport (MQTT), Hypertext Transfer Protocol (HTTP) and Constrained Application Protocol (CoAP). Integrating one of the IoT frameworks would only be beneficial if a different communication technology than HTTP was used. In this phase, HTTP is used to communicate with embedded devices. This is discussed more in Section 5.1 on page 26. [25, 47]

3 BACKGROUND OF THE GENERAL WEB SERVER

The first version of the World Wide Web (WWW) was designed by Sir Tim Berners-Lee in the early 1990s. Then, he was working at the research facility CERN as a software engineer. Berners-Lee wanted to develop a system to share the big amount of data through hyperlinked plain-text documents. These documents could then contain a hypertext, a text with references to another document that can be accessed directly. [48]

At the end of 1990, Berners-Lee had built and designed the following components to build a first version of the Web:

- HyperText Transfer Protocol (HTTP)
- Hypertext Markup Language (HTML)
- Web browser
- Web server, with accompanying HTTP server software
- the Web pages that described the WWW project itself [12]

Later on in 1994, when the Web began to expand, Berners-Lee also constructs the Uniform Resource Locator (URL), regularly referred to as a web address. A method to specify the location of a resource. [48]

Both HTTP, HTML and URL are standards that are used today in contributing the Web. This chapter discusses the general overview of a web server and associated technologies.

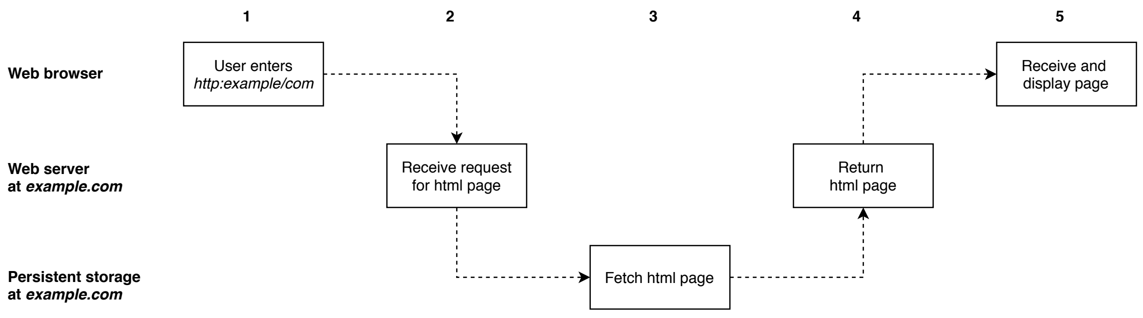
3.1 Fundamentals of a web server

The WWW has since its origin undergone many advances. Also, the web browser and web server have been technologically improved. In the following paragraphs, the progress of the web server is discussed in more detail.

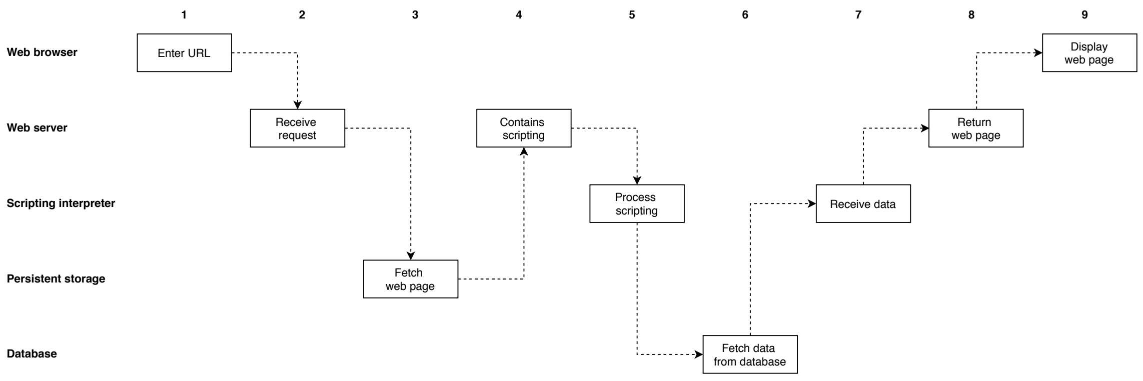
3.1.1 Evolution of web pages

The first web pages were defined in HTML and were built out of plain-text and hypertext to link specific pages to each other. These web pages were hosted by a web server in

CERN. A browser sends a request to a web server to obtain the web pages. The server handles the request by getting the HTML file from the persistent storage and returning the requested page to the browser. It will send to every browser the exact same file, this kind of web pages are static web pages. In Figure 3.1a the basis client-server request-response sequence is illustrated. [86]


1
2
3
4
5
Web browser
Web server at example.com
Persistent storage at example.com
User enters http:example/com
Receive request for html page
Fetch html page
Return html page
Receive and display page

***(a)** Static web pages*



***(b)** Server-side dynamic web pages*

***Figure 3.1.** Providing of web pages, adapted from [86]*

Since the first web pages, a lot of web pages were made more interactive and better looking. This has been achieved by the addition of Cascading Style Sheets (CSS). CSS is a style sheet language for specifying the presentation of an HTML file. Furthermore, JavaScript support was added to the browsers in 1995. JavaScript is a scripting language specially developed to make web pages more dynamic [92, Ch. 4]. In this way, the client-side scripting could be provided.

Meanwhile, also in 1995, the first web framework *ColdFusion*, was born [21]. A web framework is a software framework that provides a standardised way of building web-sites and access to various libraries. Server-side scripting was therefore made possible. Through the framework, it is also possible to link a database, to provide long-term structured storage.

These developments made it possible to display pages with variable content. There are two kinds of dynamic web pages, *server-side dynamic web page* and *client-side dynamic web page*.

A **server-side dynamic web page** is a web page that is constructed by the web framework, whose building the page by server processing server-side scripts.

Figure 3.1b on the preceding page illustrates this process. The server retrieves the web page from his persistent storage and checks what elements should be added. The scripting interpreter collects the desired data from the database and builds the requested web page. The newly formed web page is then sent to the web browser. [86]

A **client-side dynamic web page**, on the other hand, is a web page using HTML scripting to modify the web page. JavaScript or other scripting languages determine then the look of the web page.

The last couple of years, the World Wide Web is changing from static, resource-based web sites, to dynamic web application. An example of this is the single-page application. A website where the user dynamically rewrites the current page rather than fetching a new page [43, p. 497]. Illustrations are Google's *Gmail Web App* and the online LaTeX editor *Overleaf*. [7, 9]

3.1.2 Software stack

The evolution to collaborate web framework, database and web server led to the definition of the software stack. A software stack, or solution stack, is a set of segments to achieve a common goal or a result. For a web application, the solution stack typically consists of an operating system (OS), web server, database, and scripting language. Examples are giving in Table 3.1, all with the common goal to host a website. Subtables 3.1a to 3.1c on the next page are the strictly bounded to one OS and one relational database. While on the other hand, Subtable 3.1d is purely a JavaScript-driven stack for building dynamic websites. [44]

3.1.3 Deployment

Besides the selection of the software stack, choosing the place of deployment is also an important aspect of hosting a website. Here there are two possible options: providing a server yourself or using a remote location, a cloud platform.

Bare metal server

The first option is the usage of a bare metal server. This refers to purchasing the actual hardware and connecting it to a business-class internet service provider. In addition to the maintenance of the physical server, the network infrastructure must also be taken into account. This solution has the highest degree of freedom. [62]

Table 3.1. *Overview of different software stacks*

(a) *The LAMP stack [44, p. 7]*

L	Linux	*operating system*
A	Apache	*web server*
M	MySQL or MariaDB	*database management systems*
P	Perl, PHP, or Python	*scripting languages*

(b) *The WIMP stack [104]*

W	Windows Server	*operating system*
I	Internet Information Services	*web server*
M	MySQL or MariaDB	*database management systems*
P	Perl, PHP, or Python	*scripting languages*

(c) *The LEPP stack [56]*

L	Linux	*operating system*
E	Nginx	*web server*
P	PostgreSQL	*database management systems*
P	Perl, PHP, or Python	*scripting languages*

(d) *The MEAN stack [44, p. 7]*

M	MongoDB	*document-oriented database*
E	Express	*app controller layer*
A	Angular	*front-end framework*
N	Node.js	*web server*

Cloud platform

The cloud platform is divided into different services. To begin with, Infrastructure-as-a-Service (IaaS) the platform provides processing, storage, networks, and other fundamental computing resources. The consumer can run arbitrary software. Illustrated on Figure 3.2 on the following page. IaaS is the most elementary service of the cloud platform. [74]

Secondly, Platform-as-a-Service (PaaS) is an extension of IaaS. It also provides certain libraries, services, and tools to the consumer. PaaS is designed to support the complete web application life cycle. For example, it is possible for a web developer to build a website using a web framework and then place this website on a PaaS, without worrying about the complexity of the practical implementation. [74]

Last, with Software-as-a-Service (SaaS) the consumer does not need to design the software themselves. For instance, the *Microsoft Office 365* applications and communication tool *Slack* are SaaS services.

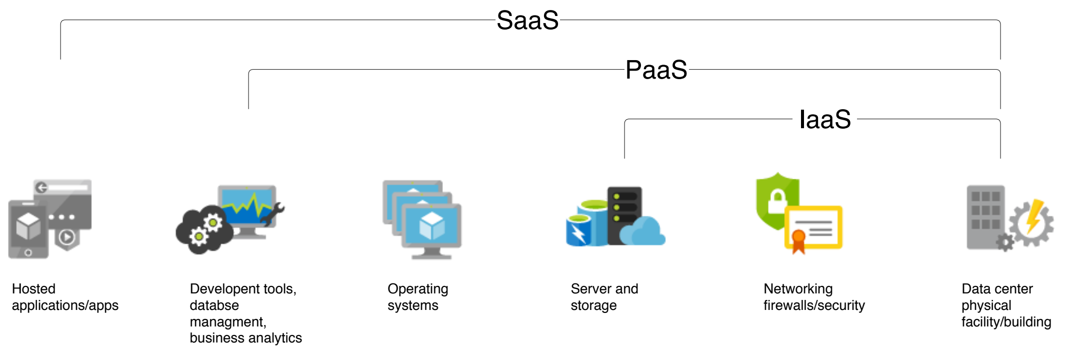

SaaS
PaaS
IaaS
Hosted applications/apps
Developent tools, databse managment, business analytics
Operating systems
Server and storage
Networking firewalls/security
Data center physical facility/building

Figure 3.2. *Visualisation of cloud services, adapted from [109]*

3.2 Communication protocols

The purpose of the very first web server was to host a static website. When the software stack came in, there was a need for an interface to receive and manage data on the web server. To accomplish this, different communication protocols can be used. In this thesis only Message Queuing Telemetry Transport (MQTT) and Hypertext Transfer Protocol (HTTP) will be discussed. Both of the protocols are located in the application layer of the OSI model.

3.2.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol is a protocol that is used, among other things, to deliver web pages to a web browser (client). It is request-response-based, which implies that all the server-sent message originates from a specific client request. The server-sent message then consists of the requested HTTP resources. The addressing of the resources is done by using Uniform Resource Locator. Inside the URL string, the resources are identified and located. A URL is defined by using the Uniform Resource Identifier (URI) *http* scheme. Furthermore, HTTP uses the underlying Transmission Control Protocol (TCP) and Internet Protocol (IP) for message delivery and TCP port 80 to communicate. [41]

To add secure communication between the client and server the Hypertext Transfer Protocol Secure (HTTPS) protocol was designed. HTTPS is an extension of the HTTP protocol. It uses Transport Layer Security (TLS) protocol for encrypting the HTTP messages. However, HTTPS uses the *https* URI scheme and TCP port 443 for communication. [93]

Besides the serving of web pages, HTTP also can be used to compose a web service. A web service is a service with the purpose to facilitate interaction between two machines through the WWW [71]. In contrast, an Application Programming Interface (API) is defined as a common component between different software. Therefore a web service is an API that is restricted to communication between machines. [14, 96]

This thesis will be limited to describing two ways to realise a web service who is using

HTTP as a communication protocol. In the next paragraphs, the Representational State Transfer (REST) architecture style and the Simple Object Access Protocol (SOAP) will be explained.

Representational State Transfer

One way to implement a web service is by using the Representational State Transfer architecture style. This way you become a RESTful web service [94]. REST was originally proposed in the PhD dissertation of Roy Fielding in 2000. In this dissertation following constraints are proposed to qualify as a RESTful system [42] :

1. Client-server architecture
2. Statelessness
3. Cacheability
4. Uniform interface
5. Layered System
6. Code-On-Demand.

In addition, when using an HTTP-based RESTful web service, the HTTP methods and URL can be used. This way create, read, update, and delete (CRUD) functions are provided to the web service [70].

The REST architecture is also not bounded to one specific media type. A media type is used to identify a file format for transmitted data over the Internet [45]. For example, the `application/json` format is used for the JavaScript Object Notation (JSON) or the `text/xml` format for the Extensible Markup Language (XML). [42]

Simple Object Access Protocol

Unlike REST, SOAP is a messaging protocol. The protocol uses a remote procedure call mechanism that occupies XML technologies to define the message format. HTTP can be used for message agreement and transportation. A SOAP message is the basic communication unit between different SOAP nodes. The message consists of a SOAP envelope which includes a SOAP header, SOAP body and a SOAP fault. Only the SOAP body is mandatory, the header and fault field are optional. The most recent media type for a SOAP message is `application/soap+xml`, the original definition of 1999 was using `text/xml-SOAP`. [53]

The big difference between REST and SOAP is that REST is an architectural style and SOAP a protocol. Both support the usage of the encryption protocol Secure Sockets Layer (SSL)[1]. In contrast, SOAP does support the Web Services Security to, hence

[1]The predecessor of TLS

SOAP can be made safer. A RESTful web service, however, produces significantly lower network traffic, lower latency and smaller messages in size than a SOAP-based web service [2, 81].

3.2.2 Message Queuing Telemetry Transport

MQTT is a publish-subscribe-based messaging protocol and uses the underlying TCP/IP transport protocol. In a publish-subscribe pattern, the sender does not send a message directly to specific receivers. An MQTT system communicates with the clients through a server, which can be called a *broker*. This realises one-to-many message distribution. To organise the message flow, MQTT has a topic-based system. A message is published to a specific topic, the broker then sends the message to all the clients who are subscribed to that topic. (fig. 3.3) A topic can be divided into different levels, for example, `example/topic` is a two-level-topic. [8, 100]

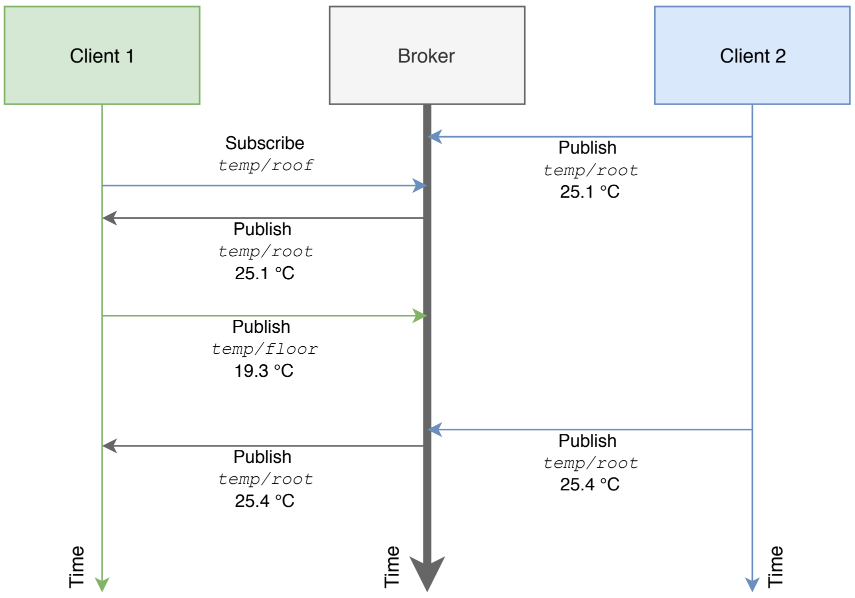


Figure 3.3. *Publish-subscribe-based messaging protocol*

The protocol has different implementations for guaranteeing the delivery of a message. This Quality of Service is divided into three levels, which can be viewed in the Table 3.2 on the following page.

3.3 Real time websites

With a real time capable technology, a server can send the client new information a soon as it is available. In other words, with this kind of technologies, a server is not depending on a client request for providing new information. When the WWW was designed, the real time capabilities were not considered, although it is possible to achieve with the

Table 3.2. *Different levels of quality of Service*

QoS level	description
QoS 0 - at most once	The sender sends the message only once and the sender and receiver do not acknowledge the delivery. Also called *fire and forget* and preserves the same guarantees as the TCP protocol.
QoS 1 - at least once	Once the client receives the message it sends an acknowledgement to the sender. The sender will resend the message until an acknowledgement has arrived.
QoS 2 - exactly once	This level guarantees that at least two request/response flows happen between the sender and receiver. The receiver sends the first acknowledgement to the sender. The sender replies on this message by sending another package to the receiver. As of last, the receiver sends a second acknowledgement to the sender. This way the sender is assured of the delivery of the message.

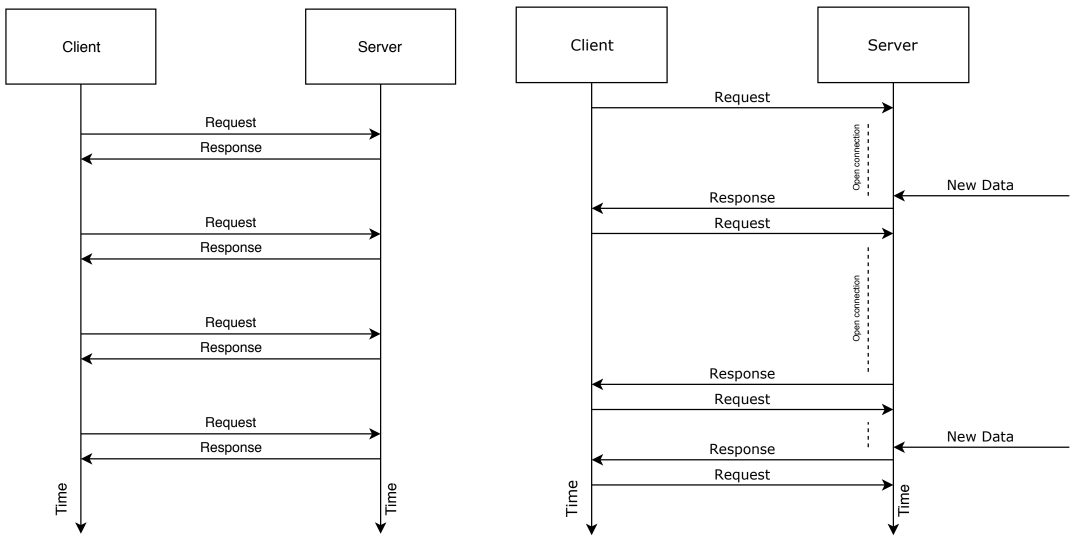
traditional HTTP. The *HTTP Long Polling* technique is an example of this. In the next paragraphs HTTP Long Polling, the WebSocket protocol and the Server-Sent Events (SSE) are discussed.

3.3.1 HTTP Long Polling

A naive solution is running client-side JavaScript code that periodically sends a request to the server for updates. If the period between the HTTP request is small enough, the website can be experienced as real time. An example of the HTTP Polling is shown in Figure 3.4a on the next page. A drawback of this technique is that the server often has no new data and therefore sends empty messages to the client. This creates a lot over unnecessary overhead on the network and on the server. [65, 99]

The problem is resolved in the HTTP Long Polling technique. The server keeps the connection open for a set period of time. If in this period new data arrives the server sends the server it immediately. On the contrary, if the server did not receive new data it will terminate the open connection. The client will then instantly open a new connection. An example is given in Figure 3.4b on the following page. [65, 99]

The Long Polling technique provides a mechanism by which the server can notify the client about new data without requiring any action of the client. The first problem with Long Polling is that it does not support bidirectional communication. If the client already has opened a connection, the only way to communicate with the server is by sending another HTTP request. The second problem of HTTP Long Polling is it can happen that



(a) *HTTP Polling* ***(b)*** *HTTP Long Polling*

Figure 3.4. *Example of the HTTP Polling and HTTP Long Polling technique*

new data is available right after the moment the time span had ended. [65, 99]

3.3.2 WebSocket

Both of the problems of HTTP Long Polling are resolved in the WebSocket protocol. WebSocket provides a bidirectional communication channel over a single TCP connection. The protocol is located in the application layer of the OSI model and depends on TCP protocol. To handle the addressing, WebSocket uses the `ws` or `wss` scheme for the secure version. [40, 51]

Despite the fact that HTTP and WebSocket are various protocols, they are intertwined. WebSocket uses the TCP port 80 and 443 to respectively plain-text and TLS-encrypted communication. To open a WebSocket connection, an HTTP request is sent to the server to *"upgrade"* the connection to the WebSocket protocol. [51]

3.3.3 Server-Sent Events

Server-Sent Events is a technology that makes it possible for a server to send text-based event data to a client. The client initiates the communication by sending a regular HTTP request. The server will send all the data over a long-lived HTTP connection. If the server determines that the connection has been open long enough, it will be terminated. The data is from the `text/event-stream` media type. In other words, SSE creates a unidirectional communication channel that only supports server-to-client messages. [51]

4 PRINCIPLES OF DESIGNING A SERVER PLATFORM

As discussed in the previous chapter, web applications have undergone a major transformation. A new architecture was defined to represent all the necessary components to construct a web application. One of the definitions is given by Bass et al., *an architecture describes a structure*. According to them, the architecture consists of a software system of structures, decomposed into components, and their interfaces and relationships. [10] To summarise, an architecture is a means to reproduce the composition of a web application.

Note that the term *web server* is used ambiguously in practice to describe all the necessary components of a web application or on the other hand the component. In this thesis, the term *web server* is referred to as the component, the piece of software with the purpose to handle the incoming network requests (Sec. 4.2.1). This chapter will handle the former architecture of a web platform. From there, an updated version will be proposed. In the rest of this chapter, the components of this model will be discussed. As a concluding paragraph, two practical implementation methods for isolation are discussed.

4.1 Layered architectures

In 2001, Anastopoulos and Romberg proposed an architecture for web applications based on the layering aspect [5]. Layering means implementing server tiers for structuring the software systems. This way the *"separation of concerns"* can be realised. The first implementation they proposed was the 2-layer architecture (Fig. 4.1 on the next page). It is also called *client/server architecture*. This architecture is appropriate to deliver dynamic pages which can contain data from the database and static HTML pages to the client. During the generation of the dynamic pages, the application logic can use services to build the pages. For instance, data encryption or user identification are valid services. [63]

In contrast, the multi-layered architecture is suitable for more complex web applications and is possible to serve a large number of concurrent clients. Illustrated in Figure 4.2 on page 19, an n-layer architecture is proposed. The more specific three-tier architecture can be found in the figure. It consists of a *presentation layer*, *business layer* and *data layer*. Each of these layers has its own task. Remark, the *reusability* is an important

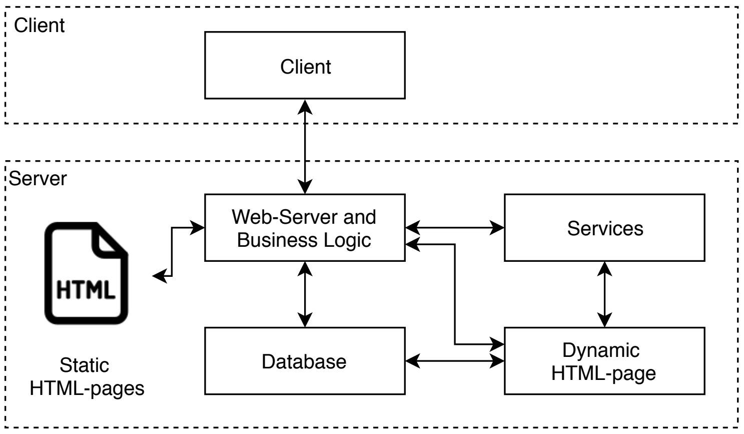


Figure 4.1. *2-layer architecture for Web application, by Anastopoulos et al. [5, p. 40]*

factor in the designing of a web application. The usage of a multilayered architecture can provide this feature.

Presentation layer

The first and topmost layer is responsible for the presentation of the content. The presentation layer, also called view or UI layer, passes the information from the user to the underlying business layer. It is the first point a client will connect to. In the model of Anastopoulos and Romberg the graphical side of the layer is not taken into account. Later this was also added to the presentation layer [90, 98]. Different client-side technologies as the style of the page (CSS) and client-side scripting (JavaScript), but as well the browser, are included in the layer. [90]

Business layer

The second layer, also called application layer, is in charge of the functionality of the web application. In this tier, the core functionalities are placed. Data is fetched from the underlying layer and processed by logic inside this layer. Otherwise, it is also possible that data arrives from the upper layer and is handled by this tier. Furthermore, the layer has also services to expose the functionalities to applicants. [90]

Data layer

The third and last layer is mainly focused on the storage and retrieval of application data. With the help of file server or database server, persistence storage is provided. The layer also provides an API to the upper layer that exposes an endpoint for managing the data. [90]

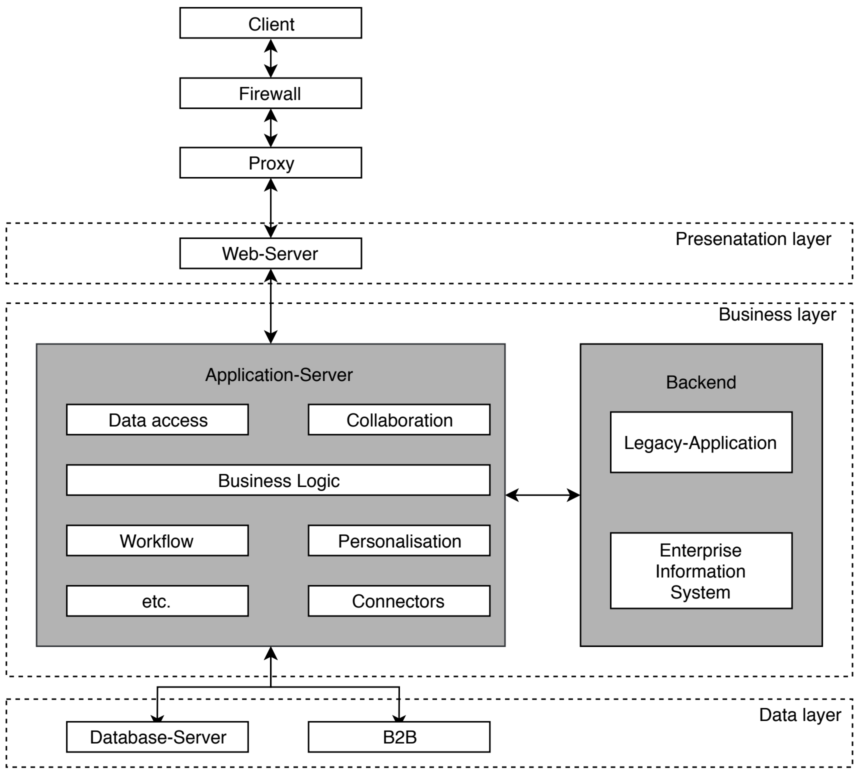


***Figure 4.2.** n-layer architecture for Web application, by Anastopoulos et al. [5, p. 42]*

4.2 Modern web server model

Since the definitions by Anastopoulos and Romberg, a lot of progress and development has been made. For example, cloud-services have become more popular, but also the real time possibilities in web pages. An updated model of a server platform for web applications is given in Figure 4.3 on the next page.

When a person asks a web browser to visit a web page, the browser first contacts a Domain Name System (DNS) to translate the URL into an IP-address. The IP protocol implements an addressing method by giving each device an IP-address. This address is used to locate a host or network interface and location addressing. With this IP-address of the website, the browser contacts the web server to retrieve the web page. In the following paragraphs, the handling of this request will be discussed.

4.2.1 Web server

The web server is a piece of dedicated server software with the purpose to handle the incoming network requests. If the request is invalid or a request for static content, the web server will handle it itself. Otherwise, the component will pass the request to the application server. [15] Besides, a web server can have extra functionalities, a few of the possible ones are listed in table 4.1 on the following page.

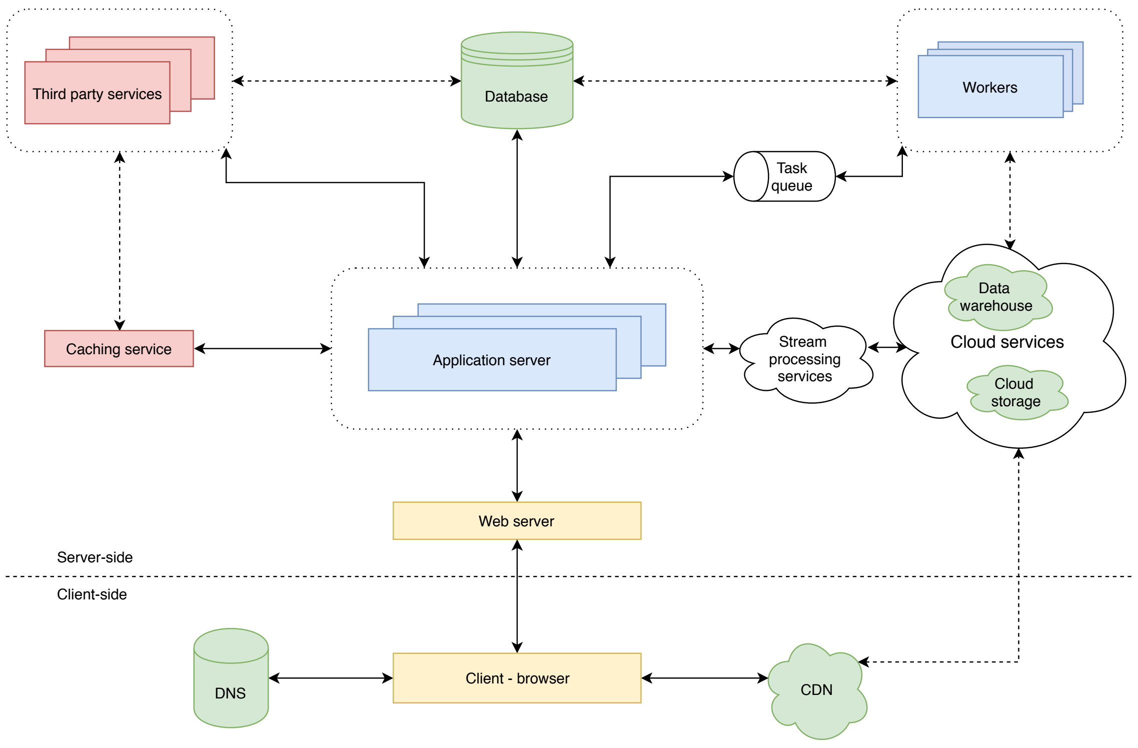


Figure 4.3. *Modern web server architecture, based on [46]*

Table 4.1. *Extra functionalities of a web server [15]*

functionality	description
Load balancing	Distributes the workload over multiple connected application servers
TLS support	Provides communications security by encrypting the outgoing data.
Reverse proxy with caching	Service that requests network resources on behalf of a client from one or more destination server
Solving the C10k problem	Being able to handle more than 10 000 simultaneous connections. Handling concurrent connection requires efficient connection scheduling while handling many requests requires a high throughput to process them. Notice that handling a connection is not the same as handling a request.

4.2.2 Application server

The application server is the core of the web application, it houses the business logic. The server communicates with all the necessary surrounding components to easily build dynamic web page. As an illustration, in Figure 3.1b on page 9, the interpreting of the script happens in this layer. [52]

The functionalities of the application server can be defined by using a web framework. A framework implements solutions for common activities in web development. For instance, it provides libraries for database access, user authentication, session management and templating frameworks. This way only the functionalities that are specific to the web application should be designed. The working mechanism of the application server is related to the programming language of the web framework. The discussion of this falls outside the scope of this thesis. [52]

To increase throughput, reliability and availability and secure the performance, often multiple instances of the application server run simultaneously. The load balancer of the web server then orchestrates the message flow to each application server. [15]

4.2.3 Database

A database is a systematic set of data, designed for flexible storage and management of the data. By the use of a database management system (DBMS), a software package that can construct, manipulate, fetch and manage data in a database. The logical structure of a database is drafted by a database schema. All the related data is saved inside a table. A schema consists of one or more table(s). [11]

To communicate with a DBMS the Structured Query Language (SQL) was created. Data can be requested with a SQL statement, also called a query. SQL support the CRUD mechanism for data and tables. [11]

One of the most used types of DBMS is the relational database management system (RDBMS). This specific type makes it possible to define relationships between tables. One of the downsides of RDBMS is that it cannot efficiently handle time series data. The problem is the big amount of data where also the order of the elements matter. To store time series in an RDBMS, a suitable solution would be to use the star schema. This schema works by storing the core data in a fact table and all the details of the data inside a dimension table. Illustrated in Figure 4.4 on the following page, sales is the fact and employee, time and product are the dimensions. This approach is not optimal for inserting and retrieving data at a high rate. To overcome this problem the data can be converted to a compressed blob form. But in this data form, queries can not be executed and all the benefits of a relational system are lost. [37, p. 28-37]

The RDBMS is not suited well enough to model recursive structures and handling heterogeneous sets. Furthermore, the approach towards time is not very sophisticated. To overcome this problem time series databases (TSDB) can be used. [36] A time series databases must be able to process and store a large number of data points. All these points are time-related to each other, so the timestamp of each point is important. A TSDB gives priority to the timestamp and is optimised to handle very large datasets [64].

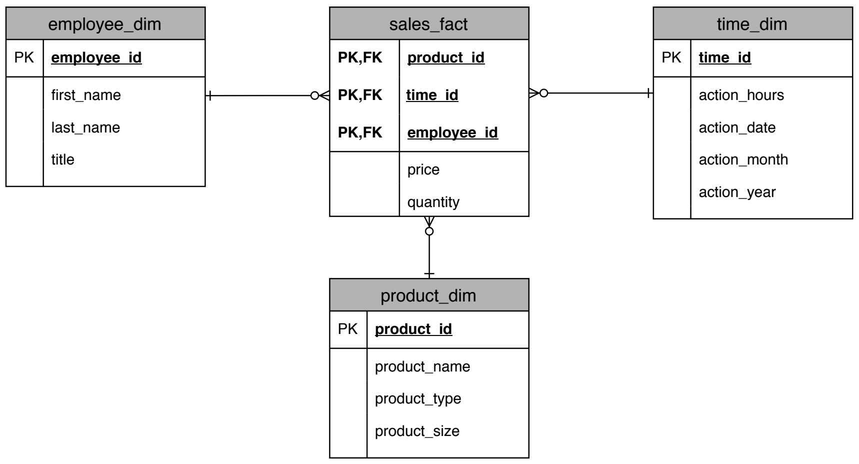


Figure 4.4. *Example of a star schema*

4.2.4 Caching service

This component makes it possible to cache newly arrived information. It provides a simple key/value data store to manage the data. By using this technology, it is possible to insert and retrieve information close to $\mathcal{O}(1)$ time [83]. The result of expensive computations can hereby be kept close. For example, search engines keep the result of common queries like *"cat videos"* rather than recalculate them each time again. [16]

Notice, this component has not the same purpose as a reverse proxy of a web server. The reverse proxy is meant to cache frequently visited web pages, while a caching service is meant to cache the newly generated data of the application server.

4.2.5 Task queue

Besides the traditional requesting and retrieving of web pages, it can be necessary to do work in the background of the web application. This means tasks that are done asynchronously without being part of the HTTP request-response cycle. Long-running jobs would otherwise affect the performance of the cycle. [18]

As an illustration, to spread out the inserts of a large data set into the database instead of inserting everything at once asynchronous tasks can be used. Another use case is the collections of data values on a fixed interval.

To provide asynchronous workers two components are used: a *task queue* to schedule the tasks and an instance that is running the task, a *worker*. The tasks queue stores the list of tasks that needs to run asynchronously. This can be accomplished by implementing a simple first-in-first-out (FIFO) scheduler. A worker polls a task form the queue when he is free to execute it. Workers can run concurrently to maintain the performance of the web application. [18]

4.2.6 Third party services

Through the use of external services, the development of very specific functionalities can be alleviated. Sometimes the cost is too large to build them yourself because some services require very specific knowledge and infrastructure to develop. To illustrate, several third party services are listed in table 4.2.

Table 4.2. *Several third party services*

Service	description
Full-text search service	This service provides a search feature on the website. The *full-text search* technology is made possible by using an inverted index. Hereby, keywords can be found quickly. The service also provides a query interface.
SMS service	Providing an interface for SMS communication.
Payment service	Provides methods for payment by using a credit card or mobile payment.
Web analytics service	Implement a service to measure, collect, analyse and report the user-generated web data. For purposes of understanding and optimising web usage.
Legacy service	An older component from the previous system that should be integrated into the web application.

4.2.7 Cloud services

As discussed in Section 3.1.3 on page 11, a cloud platform provides remote resources for doing computational work and storage. Besides the hardware facilities, cloud computing also offers useful services. In the first place, it is possible to store the user-generated data on *object cloud storage*. This way all the advantages of an IaaS and object storage are ensured.

In object storage, the entire clumps of data are stored into *objects* which contain the data, metadata, and the unique identifier. On the contrary, the traditional *block storage* splits the files into blocks with their address. Because object storage uses the metadata and unique identifier, the availability and durability of the data are increased. This makes it more attractive to use in a distributed setting. [66]

Secondly, *stream processing services* are used to transfer the data from the application server to the cloud services. Streaming data is a continuous stream of data that is usually sent simultaneously. [110] Furthermore, the streaming data services are capable of transforming the data during the transmission and automatically scales the ingest capacity according to the throughput of the data. By using these services real time analysing capabilities can be achieved. [4]

In the third place, the data can be load into a *data warehouse* for analysis. A data warehouse is a central system where data is brought together from one or more distinct sources. It is a place separated from the other databases and stores both current and historical data in one place. The main purposes of this system is to generate data analysis and reports. This aggregate data can be used to make business discussions. [54]

Finally, it is possible to run all the other components of this chapter. There are cloud services for running application servers, databases and caching services. This way all the benefits of an IaaS is offered.

4.2.8 Content Delivery Network

A Content Delivery Network (CDN) is a cloud service that offers a geographically distributed network consisting of alternative server nodes for users to download resources. As an illustration, in Figure 4.5, connecting to a node server closer to the physical location can be more beneficial than connecting to the origin server. Using CDN can create a faster response and reduced latency can be insured. Typical static content, like images and CSS & JS files, is cached on a CDN node. By connecting the cloud storage to a CDN, all the static files of a web site can be served.

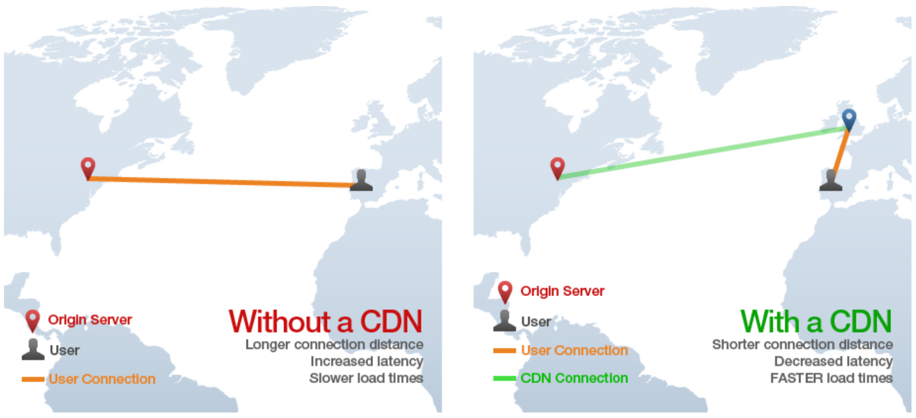


Figure 4.5. *Advantage of using CDN [111]*

4.3 Virtualisation

While building a server platform, it is possible that different components need different versions of the same library or programming language. Usually, only one version can be installed on the OS. To isolate the applications and their dependencies from each other virtualisation can be used. This also ensures that the applications can run on different machines.

A virtual machine (VM) emulates a system that executes applications like a real computer.

A VM runs on top of a *hypervisor*. This is software, firmware or hardware that is used to create and run virtual machines. A hypervisor itself is executed on a host machine, which is either an OS or bare-metal. All VM requires a certain part of the resources of the host machine. The hypervisor distributes the pre-made distribution across all VMs. Figure 4.6a illustrates a hypervisor running on a host OS with three VMs. [79]

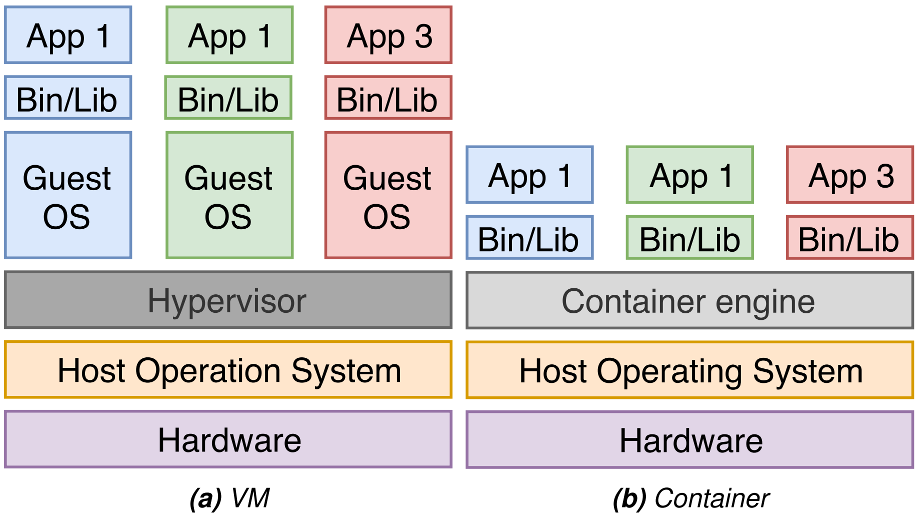


Figure 4.6. *Visualisation using virtual machine and container*

Another option to isolate application is using containerisation, illustrated in Figure 4.6b. In containerisation, the kernel of the host is shared between all the running containers. For this reason, the container is limited to the kernel of the host system. Similar to a hypervisor, the *container engine* is used to manage containers. Because a container does not use a full OS and shared resources with the host system, containers are more lightweight, efficient and faster in startup than VMs. [79]

There are two kinds of containers. An **application container** runs only one application inside the container, which can consist of one or multiple processes. The **system container**, on the other hand, can support the execution of multiple applications within the same container. This kind of containers has an inside init system[1] that makes process management possible. They are primarily designed to run a full OS inside a container. As an illustration, the Linux Containers (LXC) and its extension, the Linux container hypervisor (LXD), both providing limited dedicated resources to host Linux systems containers. [50]

[1]System to manage the start and stop of services, examples: SysV, Upstart, and Systemd

5 CLOUD IMPLEMENTATION OF THE SERVER PLATFORM

The two previous chapters give a high-level description of technologies and protocols used for designing a back-end server platform. This chapter discusses the practical implementation of the web platform of the project. All the architectural decisions, as well as the chosen protocols, components and technologies, are discussed and substantiated. Throughout this chapter, the discussed requirements of Sections 2.1 to 2.2 on pages 3–4 are used as a basis for the design choices.

5.1 Used technologies

The initial state of the project uses an HTTP-based web service for the communication between the web application and the embedded devices. As stated by Yokotani et al., MQTT performs better than HTTP in message delivery. MQTT has a lower payload size and uses less bandwidth than HTTP. [112] Although, the implementation of an HTTP-based system is easier on both the server- and client-side. This was the main reason for choosing an HTTP-based system in the first stage of the project. Accordingly, the **REST architecture style** is chosen for designing the web service. As stated in Section 3.2.1, REST is a better alternative than SOAP. The file format JSON is used to represent the data. Even within the REST Web Service, JSON is better for transmitting over the network than XML [2].

To fulfil the real time functionalities the **WebSocket** protocol is used. SSE is not implemented in the latest versions of Microsoft Internet Explorer and Edge browsers [26]. Using this technology would mean that a portion of the browser cannot display the intended web pages. The WebSocket protocol, on the other hand, has a big coverage on web browsers field [27].

5.2 The implemented stack

Section 4.2 introduced a modern version of a server platform. In this initial stage of the project, some of the components of Figure 4.3 on page 20 are not necessary. As stated in Section 2.1 on page 3, cloud services, caching service and third party services are not needed in the project for now. For each of the used component, serveral viable options are discussed, advantages and disadvantages are reviewed and the final decision is substantiated.

5.2.1 Selection procedure

A wide variety of options is available to implement in the server platform. To determine with package or framework, general guidelines were stated. The guidelines state that the same selection procedure has been followed throughout the construction of the whole server platform. The questions in table 5.1 are a short survey of the novelty, popularity and sustainability of the package. It is recommended to use a stable package with an active community behind it. The questions are ordered on importance, this means that the most influential questions are listed on top of the table. The questions are divided into fields. Some questions are related to the possibility of long-time support and others are related to support.

Production status is the most important field. One of the underlying goals of this project is providing a stable web application. This cannot be achieved with an unstable or beta version. To only list acceptable options, the production phase was the most important factor to find suitable candidates. The second most important field is popularity. A more popular package is likely to have a bigger community. Hereby, the discovery and reportage of faults in the software happen a lot faster. This results in an active community on the web helping to solve the problem of other people.

Table 5.1. *Questions used to choose a package*

Field	Question
Overall quality, future perspective	What is the development status of the package? (alpha, beta, production/stable, inactive)
Future perspective	Has the package's development been regular and is it currently active?
Support, popularity	Does the package have an active community?
Overall quality	Is the package well documented and the documentation gives relevant examples?
Future perspective, support	Does the package have a list of known issues and an issue tracker? $\rightarrow$ If yes, do the issues get solved?

5.2.2 Overview

Due to the wide variety of the implementation of all the necessary components, the most suitable solutions are selected and compared. Table 5.2 on the following page provides an overview of all implementations that meet the requirements of the previous section. In each coming section, the compared solutions are discussed in more detail and finally, the chosen packages are placed in bold.

Table 5.2. *Overview of considered implementations*

Component	Compared solution
Web server	Apache, NGINX
Web framework	Django, Flask
Web Server Gateway Interface	Gunicorn, uWSGI
Asynchronous Server Gateway Interface	Daphne, Uvicorn
Distributed workers	Celery, Dramatiq

5.2.3 Web server

Accordingly to the monthly survey of Netcraft, NGINX [84], Apache [35] and Internet Information Services (IIS) [107] are the most used web servers by the top million busiest sites[1] [61]. IIS is not an option, because it is proprietary software from Microsoft. IIS runs on Windows, who needs a commercial license to operate, while the two alternatives are open-source and free to uses. Also, it is the least popular in the Netcraft survey. Due to the mature state and regular development of Apache and NGINX, only they will be discussed further.

Apache

Apache initiates in 1995 and can be found in the LAMP stack. To process the requests, Apache uses one of the Multi-Processing Modules (MPMs). This is a group of modules with the common goal to server requests. The three most important modules are: [80]

1. **pre-fork model**: This is the original process method. It implements a non-threaded, pre-forking web server. This way at the start-up of Apache, a predefined number of servers will be launched. Each server handles requests. This approach is suitable for sites that work with non-thread-safe libraries and need to avoid the usage of threads.
2. **worker model**: This module implements a hybrid multi-process multi-threaded server. In other words, it creates a master process that is accountable for starting up child processes. Each child process then generates a fixed number of threads on its own, including a listener thread. This listener thread is responsible for detection connections and passing them through to a server thread for processing. By using this method, this MPM is more capable of handling a larger group of requests.

[1]in June 2019

3. **event model**: This model is an adaptation of the worker model. It is possible to pass along processing work to the listener threads. Hereby, redeem other worker threads, to keep them unoccupied for new connections. This model is capable of handling long-running connections more efficiently on a single thread.

Altogether, the event model needs a system that supports both threading and thread-safe polling. The worker model only needs support for the threads and the prefork model does not need support from either.

It is necessary to load one MPM module on the server at any time. Apache has a numerous set of features, like a reverse proxy with caching, load balancing. Apache Modules should be installed to implement this feature. Moreover, the modules can run at either compile-time or at run-time to improve the performance of Apache. [6]

NGINX

NGINX was created in 2002 to solve the C10k problem[2]. It uses an asynchronous, event-driven architecture to handle this prodigious load. Therefore, this architecture makes the resource usage, in terms of RAM, CPU usage, and latency more predictable with fluctuating and high loads. [67]

In terms of the event models, the main difference between the two web servers is that Apache sets up extra worker processes per connection, NGINX does not. NGINX recommends to run one worker process per CPU, hereby maximising the hardware's efficiency. [67] Furthermore, Schroeder and Mikalauskas proves, independently of each other, that NGINX is better in serving static content than Apache. Mikalauskas also concludes that for serving dynamic content, both web servers are equivalent. [75, 97].

Besides this architectural difference, NGINX also has an elaborate arsenal of functions. For example, it features a reverse proxy server for the HTTP(S) and possibilities to support the email protocols IMAP, POP3 and SMTP. Besides, it also features a load balancer and a front end proxy for Apache. Therefore, it is possible to combine the flexibility of Apache with the static content providing of NGINX. [67]

Because of the asynchronous nature of NGINX, the project uses an **NGINX web server**. This is beneficial to handle the incoming message from the detection devices. Besides, the popularity of Apache has been declining for a few years, while that of NGINX has been rising [61]. NGINX also supports a Python-based application server by default, while Apache needs a specific module to provide the same feature.

[2]Capable of serving at least 10 000 simultaneous connections on a single server

5.2.4 Web framework

The web framework is the heart of the web application. Due to one of the requirements, only Python-based frameworks are viewed. For this component, the popularity was the most influential decision factor. Django [29] and Flask [108] are the most popular and stable web frameworks at the moment. Both have two times more GitHub stars and forks than the next competitor Tornado. [103] That is why Django and Flask are the two most suitable candidates.

Django

Django is a full-stack web framework, which generally includes a lot of out-of-the-box functionalities. Libraries, database management and templating engines are examples of services that are provided by a full-stack web framework.

Django is based on the don't repeat yourself (DRY) design philosophy and wants to focus on automation. It uses a model-template-view (MVT) architecture pattern, a variant of the model-view-controller pattern, to build dynamic web pages. Django provides by default an MVT enabled templating engine. A custom object-relational mapping (ORM) is implemented to interact with the database. Django support by default only SQL databases. Furthermore, it has components for routing, forms, authentication. [29]

Flask

Flask nevertheless is a microframework. Generally, a microframework is focused on receiving HTTP requests and to handle them. Functionalities like templating engine and ORM are often not supported.

By default, Flask does not support an ORM but does support the templating engine Jinja. The philosophy of this framework is that does not force you to use a certain architecture or model. There are extensions to implement a variety of functionalities, like ORM and providing a REST web service. [108]

For the project, **Django** is used as the web application framework. The project needs an informative home web page, a database to provide long-time storage and specific environment to monitor all the embedded devices. By default, Django already provides many of the needed functionalities.

5.2.5 Web Server Gateway Interface

The Web Server Gateway Interface (WSGI) provides an interface between a web server and a Python-based web application or framework. The first definition of the interface was proposed in the Python Enhancement Proposal (PEP), `PEP 333 - WSGI` in 2003 and solved the problem of the negative interoperability [38]. WSGI wants to provide an API that has analogue flexibility as the *servlet* API of Java. In other words, WSGI makes it possible to freely choose a web server and a web application or framework. [38] In 2010 an updated version was published under `PEP 3333`, adding the support and improving the usability of Python 3. [39]

The most popular WSGI servers, at the moment of writing, are Gunicorn [20], uWSGI [101], Waitress [106] and mod_wsgi [77]. [30] All the other WSGI implementations were not considered because of the selection criteria of Table 5.1. For this component, it is important to implement a stable production level product. Only Gunicorn and uWSGI are qualified enough for the project. Because Waitress does not support the start-up of multiple workers and mod_wsgi is only compatible with Apache [77, 106].

Gunicorn

Gunicorn “Green Unicorn” is the Python alternative of the Unicorn project for Ruby, which is an HTTP server to serve Ruby web applications. Gunicorn is built on the pre-fork worker model. At the core, there is a central master process that manages a collection of worker processes. The master is only a conductor and all the requests are handled by the workers. By default, Gunicorn uses synchronous workers that serve a single request at the time. [20]

uWSGI

uWSGI, on the other hand, is developed to be more than just a WSGI server. There are plug-ins available for logging, monitoring, load balancers, proxies.uWSGI also supports different interface, for instance, WSGI, PSGI[3] and Rack[4]. It uses the same pre-fork worker model for handling request as Gunicorn. During the start-up, it is possible to activate the multithread mode in uWSGI, this enables support of running threads inside a process. [101]

Different implementations of WSGI are tested in Section 6.1 on page 38. The experiments show that **uWSGI** is the better option. The choice is more substantiated in Section 7.2 on page 48.

[3]The Perl alternative of WSGI
[4]The Ruby alternative of WSGI

5.2.6 Asynchronous Server Gateway Interface

Due to the requirement concerning the real time possibilities of the web application, a real time protocol should be implemented. The biggest limitation of WSGI is that it is bounded to a single, synchronous callable that process a request and answers it. Long-lived connections, like long-polling HTTP and WebSocket, are not supported. Asynchronous Server Gateway Interface (ASGI) corrects this issue, it is structured as a single, asynchronous callable. Providing hereby support for multiple incoming and outgoing events for each application and possibilities for doing background coroutines. ASGI is designed as a superset of WSGI, which makes it possible to run WSGI applications inside an ASGI server. ASGI has, at the moment of writing, only two stable servers: Daphne [24] and Uvicorn [105]. They are both HTTP/WebSocket server with support for HTTP/1.1. [57, 59]

The project uses the **Daphne server**. Because Daphne is maintained as part of the Django Channels packages. The project uses this package to provide a WebSocket connection.

5.2.7 Distributed workers

When observing a venue, multiple detection nodes will be used to get a general idea of the venue. The same person may be detected by different nodes. To discover that same person a re-identification algorithm can be used. Using distributed workers makes it possible to run multiple instances of this algorithm at the same time. Each instance can then process the data asynchronously from the application server.

The most popular stable options are Celery [60] and Dramatiq [78], with Celery being eight times more popular[5]. [31]

Celery is based on distributed message passing to provide an asynchronous task queue. It mainly focuses on real time processing, but also offers task scheduling. Celery requires a *message broker* to receive and send messages. It supports, among other things, the collaboration with RabbitMQ [88] and Redis [83]. Furthermore, it is possible to bound a caching service to Celery to store the results. [60]

Dramatiq is developed to be an alternative to the approach of Celery. For instance, Dramatiq sends an acknowledgement when the task is done, while Celery, sends it when the task is pulled out of the queue by default. Celery does not support task prioritisation, only by organising multiple sets of workers priority can be included. Dramatiq offers this by default. Dramatiq also provides a lock and rate limit, while Celery does not. [78]

Both of the packages are suited to work together with the Django framework. Because the one-sided tasks and the simple use case, **Celery** is chosen to integrate with the

[5]Based on the number of GitHub stars

project. An advantage is also the popularity of the package. **Redis** is used as the supporting message broker because it has a high performance and larger community than RabbitMQ [89].

5.2.8 Time series database

Every detection that happens on the embedded devices has a timestamp and is a new entry. As estimated in Section 2.3, the amount of detection would be 0.1 M/s. Hereby, the detection stream can be classified as a time series.

By default Django only supports four SQL databases, namely MySQL, PostgreSQL, SQLite and Oracle Database. None of those four is optimised for time series. TimescaleDB [102], on the contrary, is an expansion of the PostgreSQL [87] database. Hereby adapting PostgreSQL from a relational DBMS to a TSDB. **TimescaleDB** provides the same entire SQL interface as PostgreSQL, making it suitable to work together with the Django framework. [102]

5.2.9 Specific library and packages

External packages were required in the application server to fulfil all the required functionalities. Django does not provide by default tools for all the features. An overview of all the mentionable packages is given in Table 5.3.

Table 5.3. *Extra implemented Python packages*

Package	Usage
Django REST framework [33]	Used to build a REST web service. Hereby, an entry point for the embedded devices is realised.
Django Channels [28]	This package adds the support of SSE and WebSocket to the synchronous web framework. Through this package a WebSocket, connection can be implemented to display real time information.
Django reCAPTCHA [32]	Enables the addition of a reCAPTCHA to the forms. This way it is possible to add reCAPTCHA to the login page and register page.

5.2.10 Overview of the web application

The complete overview of the back end platform is displayed in Figure 5.1 on the following page. It can be concluded that the *LEPP stack* is used in this initial stage of the project.

The application server is utilising a Python-based framework and the database is an extension of the PostgreSQL server. Further in the coming section, the practical aspects of deployment are discussed.

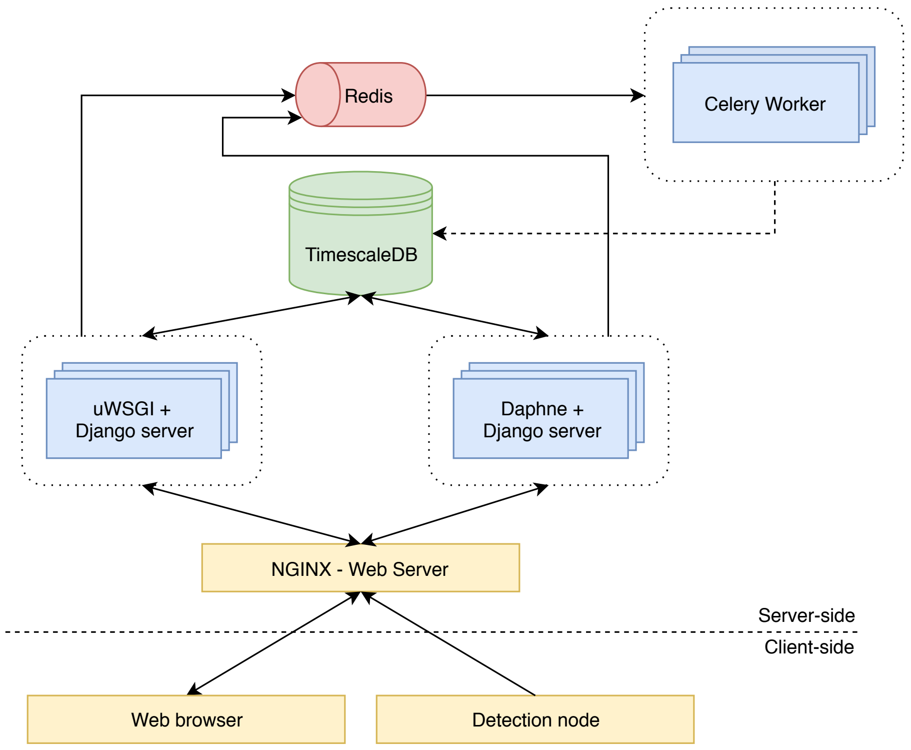


Figure 5.1. *Current implementation of the project*

5.3 Implementation details

Inside a Django server, separate applications are used to increase the re-usability. The back-end of the project uses three Django applications, namely `web`, `rest` and `stat`. The `web` application is for providing a home page with all the information about the CityTrack project. The second `rest` application is responsible for the RESTful web service. The last `stat` application provides the statistical data and the possibility to view the incoming data in real time. This implementation makes it possible to set up a separate Django server that is only concerned with providing one application. For example, a Django server with dedicated computational resources can be arranged to provide the RESTful web service.

The `web` application is a common Django application for providing a dynamic web site. The `rest` application is also a basis Django REST Framework (DRF) application for providing a RESTful web service. For arranging the real time functionalities the following implementation is used.

Real time application

To provide the application with real time functionalities, both WSGI and ASGI server need to work together. The WSGI server is responsible for delivering the web page to the browser, while the ASGI server is accountable for the start up of the WebSocket connection.

One of the functionalities is providing a real time view of the most recent detections. The data from the detection nodes is sent to DRF. Due to performance reasons and easy implementation, the DRF is placed on a WSGI server. The newly arrived data needs to be sent immediately over the WebSocket connection, who placed at the ASGI server. To transport the data from one server to another, the newly arrived data at the RESTful web service is saved twice once in the database and once in a Redis queue. A Celery worker will then be responsible for handling the remaining part.

Once a WebSocket connection is opened, the ASGI server adds the metadata of the connection to a group. This group will contain the data of all the currently active connection. After this, the ASGI server will start a container to run a dedicated Celery worker. The purpose of the worker is to check the Redis queues if they are empty. If not, the worker will pop the first element from the queue and sends it over the WebSocket connection to the browser. Once a second WebSocket connection is opened, the ASGI server adds the credentials to the same group as the first connection. This way the Celery worker will send the data to all the active connection. If a browser closes the web page, the associated connection will be removed from the group and the worker will not send any message to that connection. Once all the last connection is closed, the Celery container will also be terminated. On the browser, a JavaScript file is used to settle the incoming messages.

5.4 Deployment

To provide this project to the WWW, the web platform should be continuously available. As discussed before, this can be achieved by hosting the application on a cloud service. In this section, the practical aspects of containerisation and cloud deployment are explained.

5.4.1 Containerisation

Due to the requirement to use containerisation in this project, application containers are used to build the whole web server platform. All the previously discussed components can fit in their own application container. Because these containers are lighter and faster, a component can easily be replaced or duplicated to maintain system performance. The Docker platform [34] is used to build and manage the container stack. It is the widest used platform in 2019, with major support for a variety of cloud platforms [13]. Docker utilises the Linux kernel to provide application containers. The platform *Docker Hub* provides multiple official images for all the chosen components of the web application. The container version is defined through `REPOSITORY:TAG`. The repository container name defined in the `REPOSITORY` part, the specific version is defined in the `TAG` part. For example, the web server runs on version 1.15 of NGINX and uses a Linux distribution Alpine. To always obtain the most recent version of the software, the tag `latest` can be used. [34]

Table 5.4 gives an overview of all the used containers and container versions. All the containers are Alpine-based [3]. Using this Linux distribution results is a lighter container than Debian-based containers. The application server and distributed worker are Python programs, therefore the base of is a Python container. The TimescaleDB container is only provided with an Alpine base. Finally, a conscious choice was made to choose a specific version of the software containers. This to guarantee that the web application will have the same composition in the future.

Table 5.4. *List of the implemented containers*

Component	Technology	Container version
Web server	NGINX	`nginx:1.15-alpine`
Application server	Django	`python:3.6.9-alpine`
Message broker	Redis	`redis:5.0.5-alpine`
Database	TimescaleDB	`timescale/timescaledb:latest-pg11`
Distributed worker	Celery	`python:3.6.9-alpine`

5.4.2 Container stack

To run these multi-container applications, the Docker Compose tool is used. Because of this tool, all the containers and their connection to each other can be defined in one single file. This has also simplified the start of the web application.

In Figure 5.2 on the following page, visualisation of the container stack is given. As stated in Section TODO, is the Celery container only started if a WebSocket connection is opened. The circles on the figure represented the TCP/IP ports that are accessible for clients. The NGINX container has an open port 80 and 443, for HTTP and HTTPS communication, respectively. Due to development reasons, the 5432 port of the database is also opened.

The data that is stored inside the database and Redis container is brought outside by linking a folder on the host system to a folder inside the container. This way, the data is not lost when the containers are stopped. The same binding is set for the static content of the Django servers and the configuration files of the NGINX server. In the figure, the file path of the container is placed on the dotted line and the file path of the host system is placed inside the folder icon.

The diamond-shaped quadrilaterals represent the networks. There are three networks in this system, one network for all the database traffic, one for the Redis traffic and one for Django servers to connect to the NGINX server. These networks provide an interconnecting for the containers.

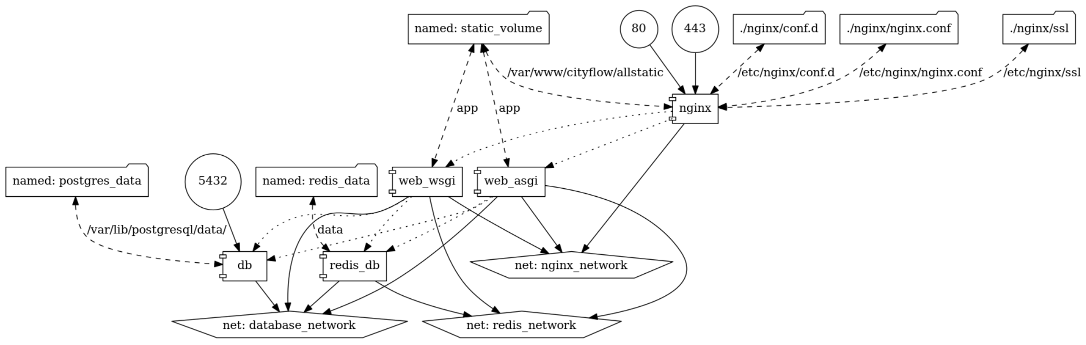


Figure 5.2. *Visualisation of the Docker Compose stack*

5.4.3 Cloud platform

The web application is deployed on the cPouta IaaS cloud computing service, offered by CSC [22]. CSC is an IT Center for Science providing ICT services to Finnish higher education institutions. cPouta is an IaaS service that provides a virtual machine. Inside one VM the whole container application is deployed. In this way, the settings, including firewall and storage capacity, are provided by the provider.

6 PERFORMANCE EVALUATION

To design a high-performance server platform, the choice of each component is very influential. The WSGI server has been investigated in this thesis. To make a substantiated choice, several aspects of the performance of each WSGI are tested. Secondly, it is tested if the current implementations of the ASGI server could be a worthy replacement of the WSGI server. Finally, the boundaries of the system are exposed.

6.1 Performance of different WSGI servers

In this first test, the performance of different WSGI servers will be investigated. The goal is to determine which implementation achieves the best total results. Therefore, five different aspects are examined of each server.

6.1.1 Situation of the research

As declared by Meier et al., *“Performance testing is a type of testing intended to determine the responsiveness, throughput, reliability, and/or scalability of a system under a given workload”* [73]. There are two kinds of performance tests, namely load testing and stress testing. Both tests are related to each other. Stress testing is an extreme case of load testing.

Load testing A performance test designed to determine the performance qualities of a server under normal workloads.

Stress testing A performance test designed to determine the performance qualities of a server under unnatural high workloads. This can include using all the available computational resources.

The two most important characteristics of the performance of a web server are throughput and latency. Throughput is the number of the requests handled in a certain window of time, generally expressed as request per second. While latency focuses on the time that a request is served. In 1993, Nielsen stated three response time limits.

The first limit is the **0.1 second**, this gives the user the feeling that everything is reacting instantaneously. Special feedback is not necessary to display. The **1.0 second** is the second limit. The user will notice the delay, but the flow of thought of the user will not be interrupted. No additional feedback is needed here either. Altogether, if the response time is between 0.1 and 1.0 seconds, special feedback is not necessary. The last is the **10 seconds** limit. This border is about keeping the attention of the user focused on the dialogue. Extra feedback, about when the computer expects to be done, is necessary. The user possibly wants to perform other tasks during the waiting.

Altogether, next to the latency and throughput, the error rate is calculated from results. Together with the CPU and RAM usage, a complete overview of the performance of a WSGI can be composed.

6.1.2 Test setup

As stated in Section 5.2.5, the most popular WSGI at this moment are Gunicorn, uWSGI, and mod_wsgi [30].

To test the different servers, the ApacheBench tool (ab) - version 2.4 [1] is used. For 30 s the tool requests packets, using the HTTP/1.0 GET method, from the server with a given amount of connections. All the 28 tested, arbitrarily chosen, connections can be found in Table 6.1. Through this range, an overview of the performance under different concurrency can be formed. With this tool, data is gathered about the Round-Trip Time (RTT). This is the time between the departure and the arrival at the client of the same package. Furthermore, the Request per second and Error Rate is also collected with the ab tool.

From the output of the testing tool, the total connections time of one packet can be collected. This total connection time consists of the connection and processing time. For this experiment, the 90% border is used, 90% of all request will be handled under this time. The remaining 10% consists of packets with total connection times greater than the 90% border. This border gives a good indication of the average RTT of the WSGI server.

Note that the request per second is not the same as the number of users the system can handle. A regular user requests more than one packets if he wants to view a web page.

Table 6.1. *Summary of all the tested connections*

Connections: 1 to 5000									
1	2	4	5	10	50	100	150	200	250
300	350	300	450	500	600	600	800	900	1000
1500	2000	2500	3000	3500	4000	4500	5000		

The WSGI is set to use the project web application and not a dummy application. The WSGI and Django server runs inside a container, with a limit of two CPUs and 1500 MB of RAM. Using the docker command `docker stats`, data is collected about the CPU and memory usages. The testing tool requests the homepage[1] of the web application and connects directly to the WSGI server, there is no web server placed in front of the web framework. To test the full capacity of all WSGI servers, they are launched with one worker for each CPU core and two threads inside. Except for the Gunicorn server, there it is recommended to use the following rule of thumb: `(2 x $num_cores) + 1` number of workers to handle all the requests [20]. An overview of the configuration of the workers and threads can be found in table 6.2.

All the tests are conducted using a Dell Optiplex 9020 with an Intel Core i7-4790 CPU 3.60 GHz and 24 GB of DDR3 RAM, running Linux Mint 19.1. Each server is tested four times with all the different connections. From the four corresponding data points, the average was taken to obtain the result. The results can be viewed in Figures 6.6 to 6.11 on pages 44–46.

Table 6.2. *Settings of each WSGI server*

WSGI server	Version	Workers	Threads
Gunicorn	19.9.0	5	1
uWSGI	2.0.18	2	2
mod_wsgi	4.6.7	2	2

6.1.3 Results

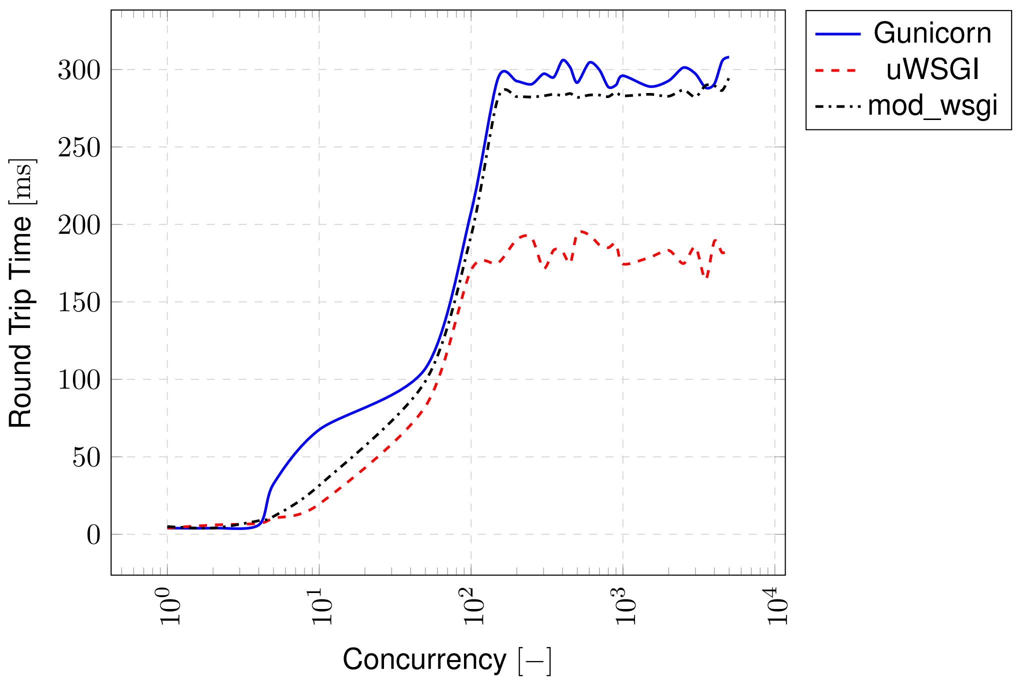


Figure 6.1. *Latency of multi workers WSGI servers*

[1]which is a static web page

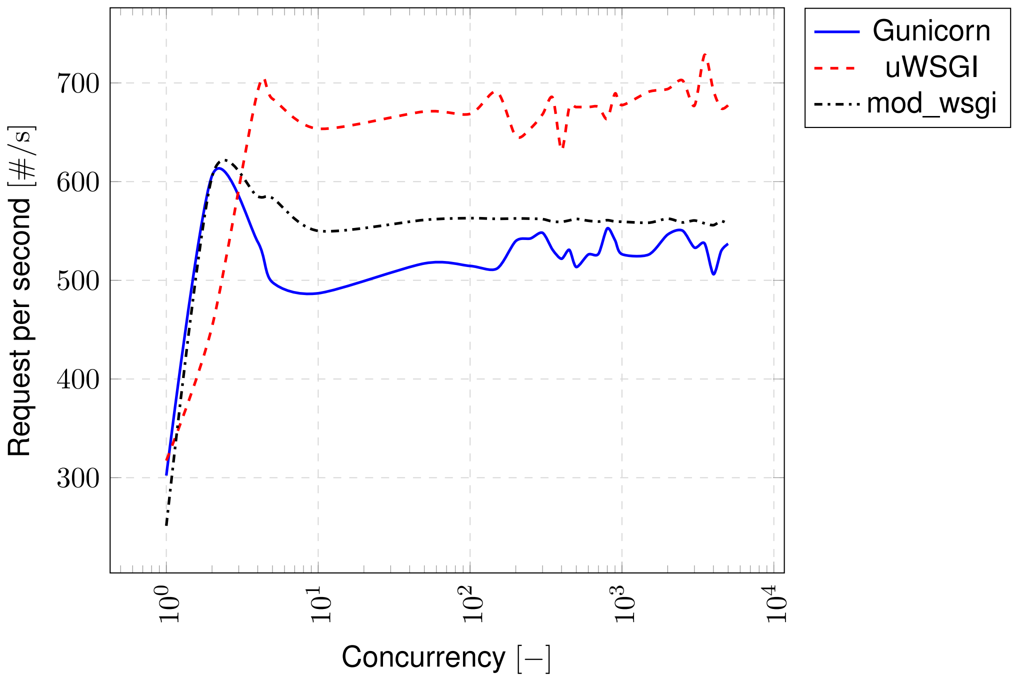


Figure 6.2. *Throughput of multi workers WSGI servers*

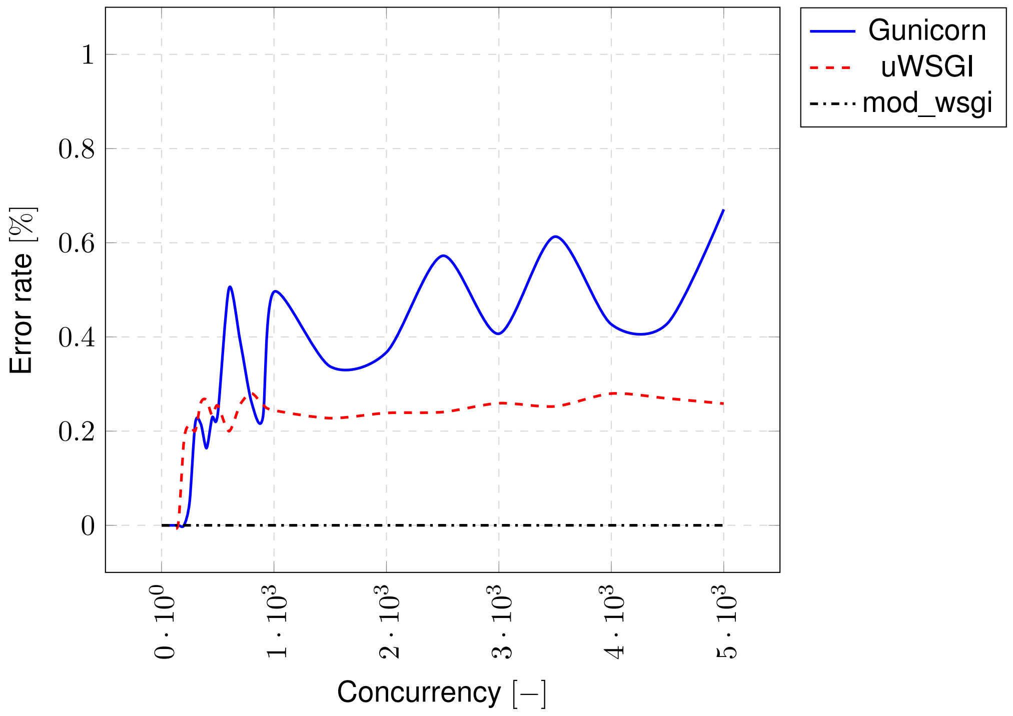


Figure 6.3. *Error Rate of multi workers WSGI servers*

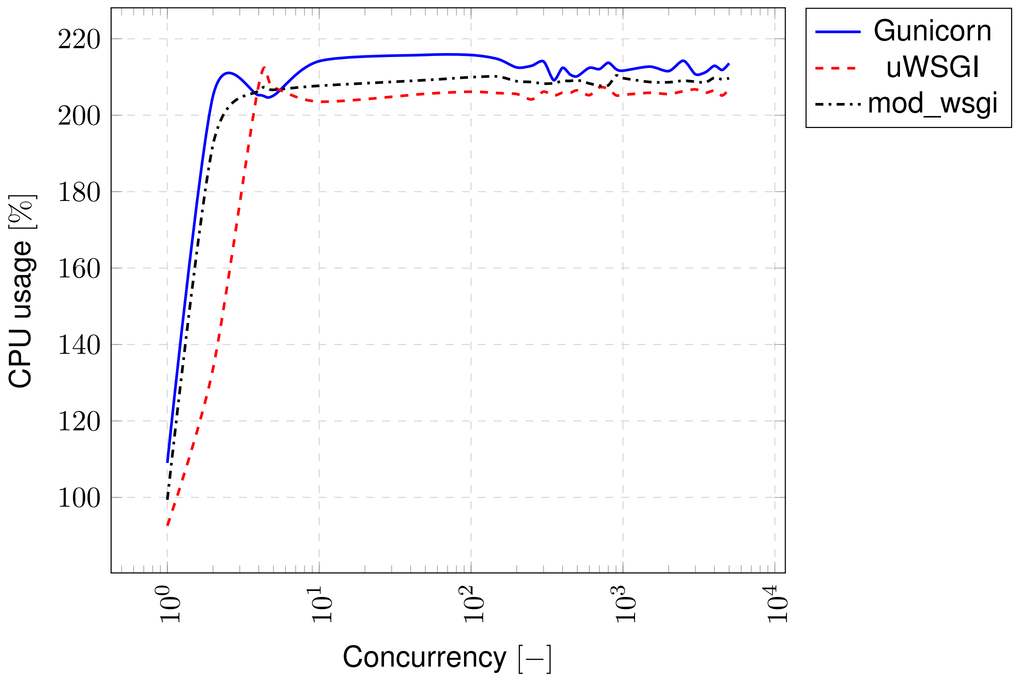


Figure 6.4. *CPU Usage of multi workers WSGI servers*

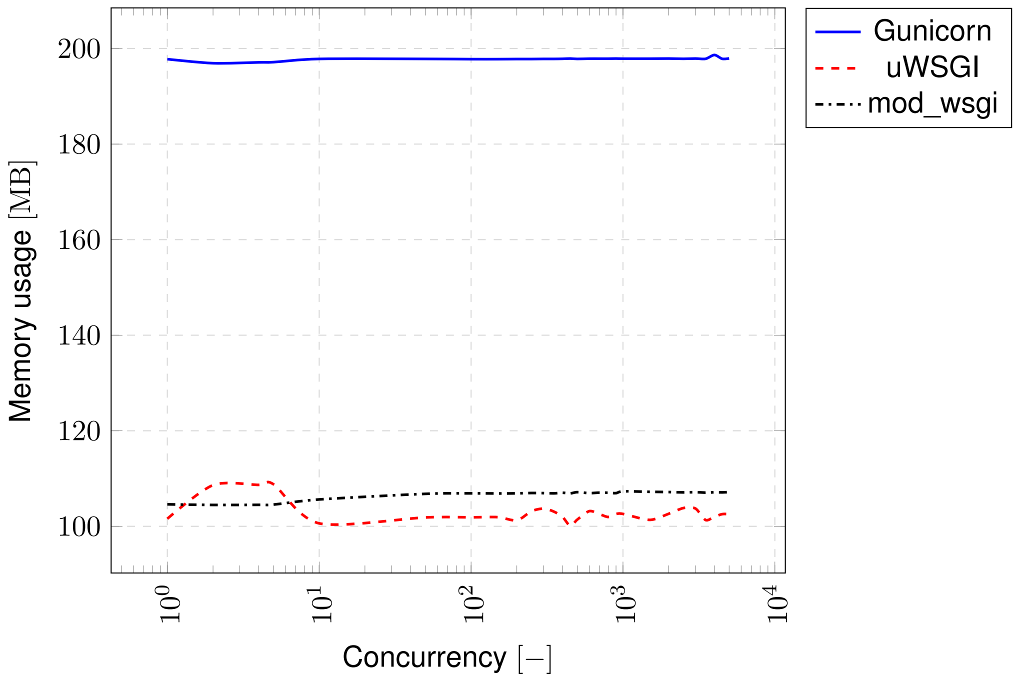


Figure 6.5. *Memory Usage of multi workers WSGI servers*

In Figure 6.1 the latency of every WSGI server is plotted in function of all the tested connections. According to Nielsen, the smaller the latency, the better. [85] The graphs show that all servers evolve to a stable value once there are more than 100 connections. With Gunicorn and uWSGI located in the same region and uWSGI significantly smaller.

The throughput can be seen in Figure 6.2. On the plot of Gunicorn and mod_wsgi there is a distinguishable peak with 2 connections. Thereafter, both plots stabilise. Both are launched with two CPUs. The best circumstance for these both servers is clear when there are only 2 connections and therefore the hardware can be used optimally. On the uWSGI plot, on the other hand, an analogous peak at four connections can be detected. The same high values return at higher concurrency. The best circumstance for the uWSGI server in the low concurrency domain is with four connections. In this case, the hardware is used optimally. The CPU plot of uWSGI, in Figure 6.4, supports this.

Remarkably, mod_wsgi manages to handle all requests given a certain load, resulting in an overall 0% error rate. Gunicorn and uWSGI do not always succeed in responding to all requests. Gunicorn has a slight fluctuating error rate, while uWSGI ultimately has a stable error rate. The average of both error rate keeps under 1%, meaning that once a request fails, the browser is capable to handle this.

In Figure 6.4 and 6.5 shows respectively the CPU and memory usage. The evolution of all the three the graph is similar. With one connection all the servers use only 100%, which is equivalent to the usage of one CPU. Once there is more than one connection, the usage is more than 200%. As for memory usage, this is not affected by the number of connections.

6.2 ASGI as a replacement for WSGI

As discussed in Section 5.2.6 on page 32, ASGI is an enlargement of WSGI. Meaning that every WSGI application also can run on an ASGI server. This test is to research if a current implementation of the ASGI server is suitable to replace the WSGI counterparts. In this test, the three recently reviewed WSGI servers are tested against the only two stable ASGI servers, Daphne and Uvicorn.

The test is conducted in the same circumstance as the first tests. All the connection of Table 6.1 on page 39 are used, the test is done using the ab tool with a period of 30 s and the ASGI server is running inside a container using two CPUs and 1500 MB. The only difference is that all servers are start-up with only one worker. Daphne does not provide the possibility to start-up multiple workers. To get a comparable result therefore, all servers are tested with one worker.

The overview of the results is given in Figures 6.1 to 6.5 on pages 40–42.

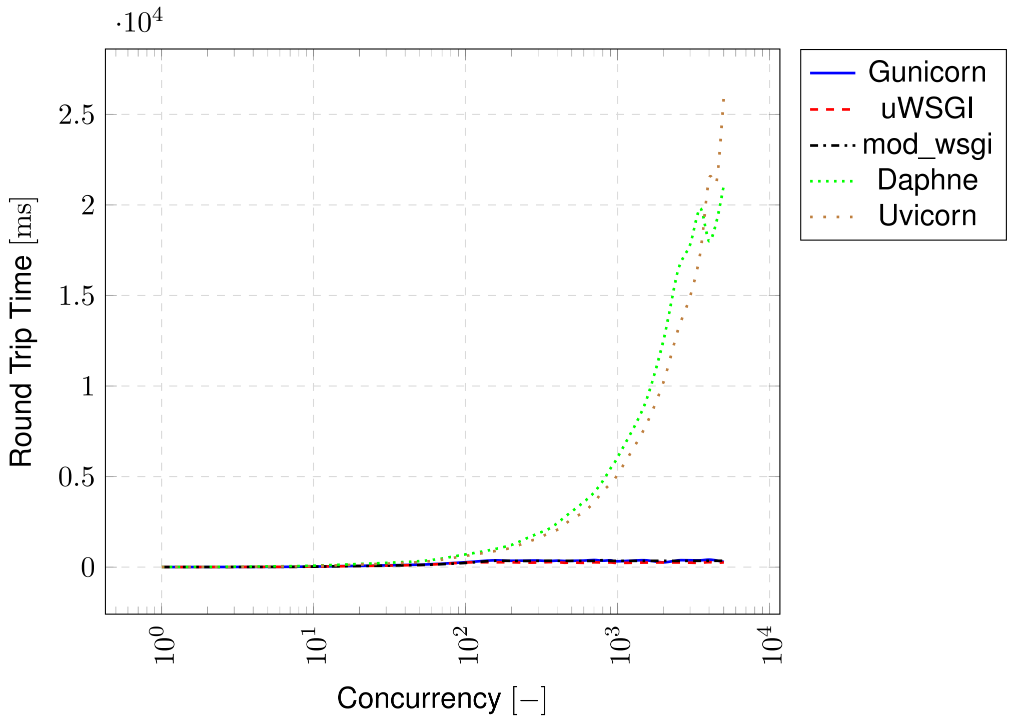


Figure 6.6. *Latency of single workers ASGI and WSGI servers*

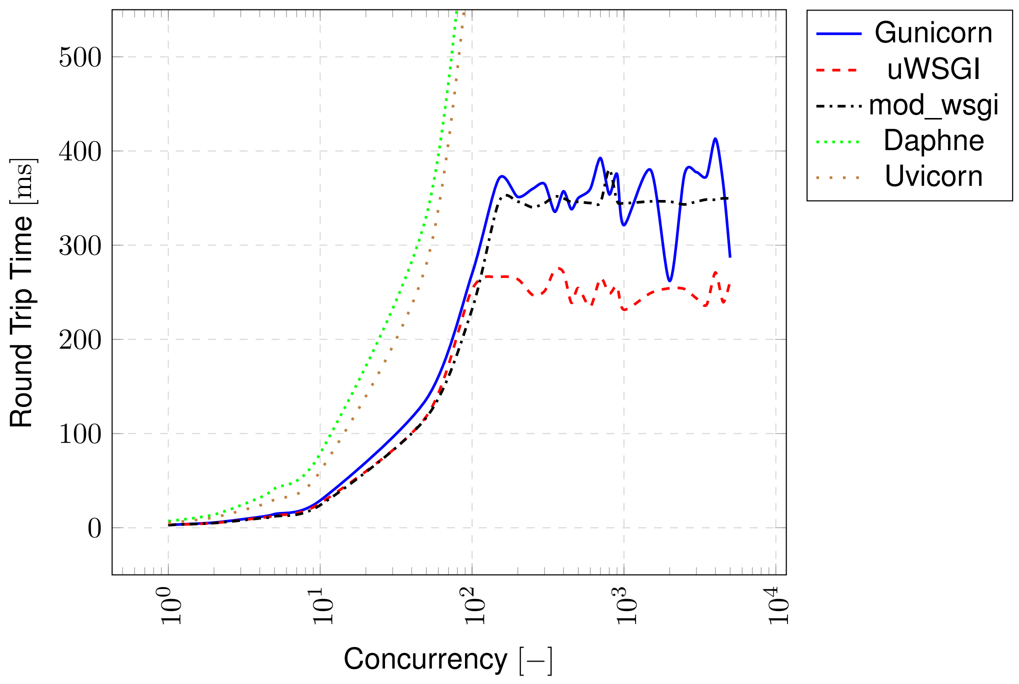


Figure 6.7. *Latency of single workers ASGI and WSGI servers - magnified*

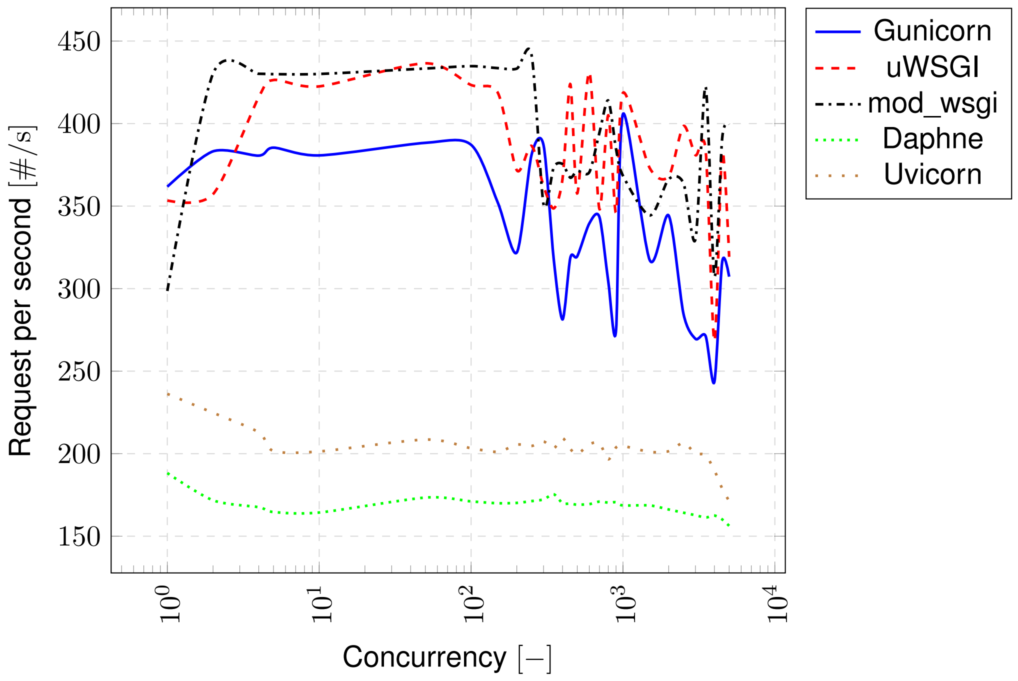


Figure 6.8. *Throughput of single workers ASGI and WSGI servers*

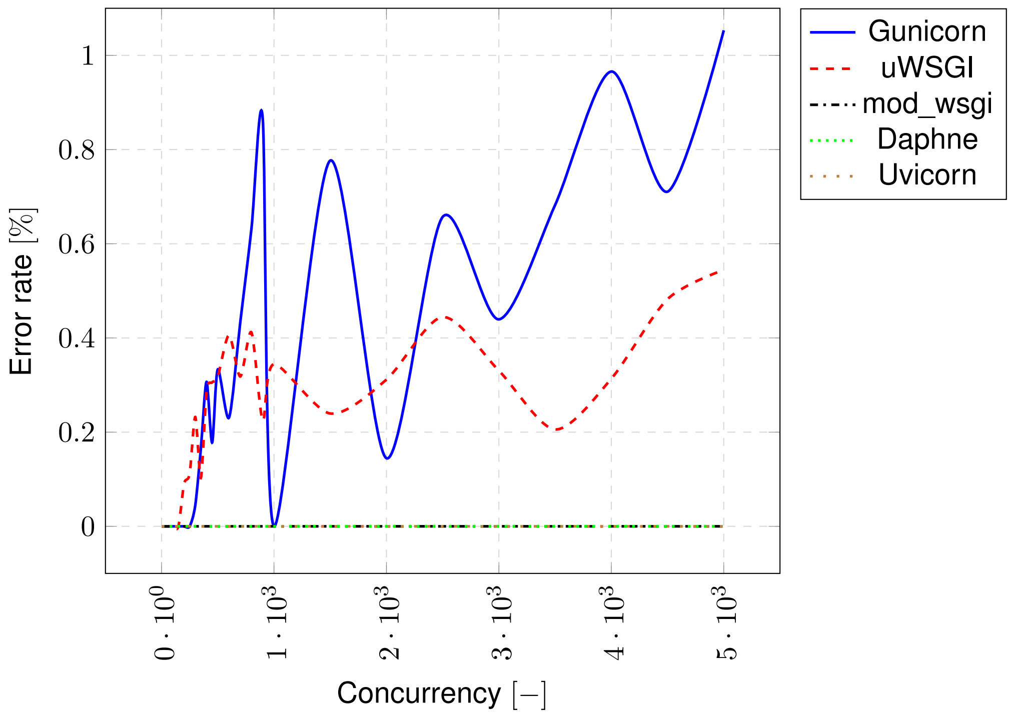


Figure 6.9. *Error rate of single workers ASGI and WSGI servers*

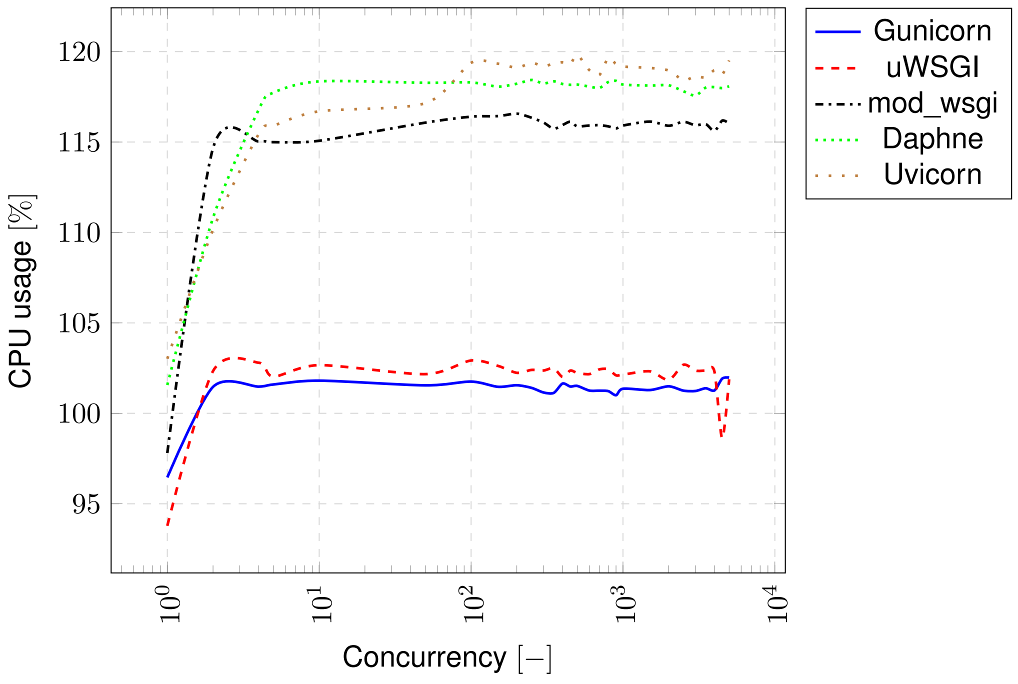


Figure 6.10. *CPU Usage of single workers ASGI and WSGI servers*

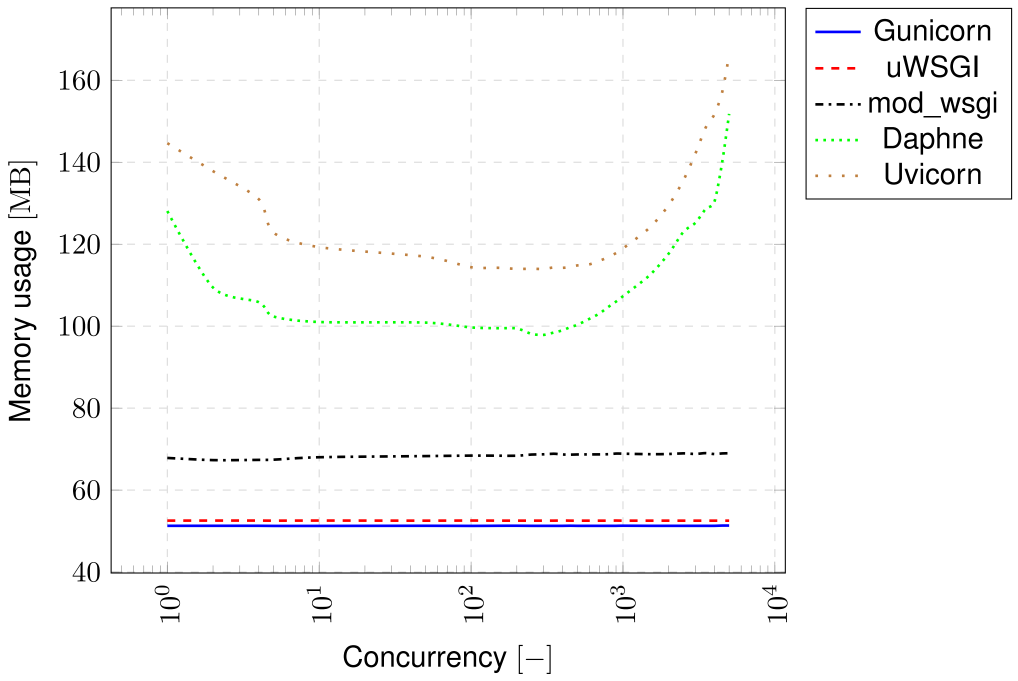


Figure 6.11. *Memory usage of single workers ASGI and WSGI servers*

The same behaviour of the WSGI server can be found in Figure 6.7. However, the Gunicorn and uWSGI are much more unstable with large concurrency. Both ASGI servers rise very instantaneously with additional concurrency, as seen in Figure 6.6.

The same unstable behaviour of the WSGI server with large concurrency can be found in Figure 6.8. Both the ASGI servers clearly underperform in comparison with WSGI versions.

The asynchronous servers answers on every request that it receives. Resulting in a perfect error rate, see Figure 6.9. Again, Gunicorn and uWSGI display unstable behaviour.

In comparison with the first tests, the consumption of the WSGI servers of CPU and memory has changed. For example, in Figure 6.10 on the previous page, Gunicorn and uWSGI consume less CPU and memory than the other three servers. The usage of the CPU of Daphne, Uvicorn and mod_wsgi is in the same order of magnitude. Remarkable is that no server uses the full 200% CPU. All the systems have designed a worker to maximise the usage of one CPU core.

The memory usage (Fig. 6.11) of the WSGI servers are still constant, while the memory usage of the ASGI only stabilises between the 10 and 500 connections. The memory usage of Gunicorn in these tests are the lowest of all the competitors, while in the first test, the usage was noticeable a lot higher than the other two servers.

7 CONCLUSION & FUTURE

This chapter recapitulates the building process of the thesis, the discussions of the testings and to finalise, the prospects of the web platform.

7.1 Development process and choices

The choices to use certain technologies and packages in Chapter 5 are based on their properties and functionalities. The usability can only be assessed after the implementation.

All selected packages and frameworks were very easy to use. There was documentation and online support for each component. Only the implementation of the Django Channels package was difficult. Because of a major update[1], the official documentation is limited and the online support found was mostly for the older versions. On the deployment side, Docker is a useful container platform with a vast online community. It is an easy to use tool with many functionalities. Altogether, all the chosen components were easy to implement and available with sufficient documentation, except for the Django Channels packet.

7.2 Discussion of the testing results

In the first test, the performance of WSGI servers was tested with multiple workers. Finally, uWSGI is used in the web platform. A web browser will open multiple connections to parallel the retrieval, it will happen very little that only one connection is opened. The behaviour at high concurrency is, therefore, more important. uWSGI has the highest average throughput in the high concurrency domain. Together with the overall lowest latency, uWSGI is the best choice. The error rate is less good than mod_wsgi, but still acceptable under 0.4%. In terms of consumption, uWSGI scores the same as the other two in CPU usage. The Apache implementation uses about as much as uWSGI, but much less than Gunicorn. In general, it can be said that uWSGI performs best of all and is therefore used in the web platform.

The second test investigates whether the ASGI implementation can perform equality well, or better, than the WSGI counterparts. This test was focused on if an ASGI can replace an

[1]version 2.0, in February 2018

WSGI. A good ASGI server needs to perform WSGI tasks correctly. Both implementations of the ASGI and WSGI server were tested with the same tool to handle a request for a web page. However, the results are disappointing. Both the ASGI servers have overall higher RTTs and an overall lower throughput than any WSGI server. Although the error rate is very good, both asynchronous servers answer on every incoming request. Unfortunately, the resource usage is then worse again. Both servers consume more CPU and memory than the WSGI counterparts.

From this can be concluded that the current version of Daphne and Uvicorn are not suited to taking over tasks from any WSGI server. This is also the reason that both a WSGI and an ASGI server is implemented in the system, to use the asynchronous capabilities of the ASGI server and the better performance of the WSGI server.

Remarkable from this test is also the unstable behaviour of the WSGI servers at high concurrency. Both the throughput and latency are more fluctuating than in the first test. The error rate of Gunicorn and uWSGI are also more erratic than in the first test. As a result, it can be concluded that WSGI is more reliable with multiple workers to process high concurrency.

7.3 Future challenges

This first thesis especially focuses on the design of the back-end server system of the web application. The following listing shows where upcoming opportunities lie.

Front-end framework

During this stage, little attention was paid to the design of the web pages during the construction. The addition of a JavaScript framework would be an added value. Together with a CSS framework, professional-looking and easy to use web pages can be built.

Different communication technology

As stated by Yokotani et al., MQTT is a lighter protocol than REST [112]. One possibility lies in changing the communication technology to MQTT or the newer Lightweight Machine to Machine (LWM2M) protocol. Thus, smaller and lighter message packets can be created that are easier to process by the back-end server [82].

Cloud services

Making use of cloud storage, all the data can be saved in an extra facility. This way, the data can be extra protected and if linked to a CDN, all the static files can be offered to the website visitors faster.

Infrastructure automation

To provide maintenance options of the detection nodes, an infrastructure automation tool can be used. This way, only one time the update statement can be defined and then be implemented on all nodes. An automated start-up of new nodes can also be achieved with these tools.

Expansion of the functionalities

Another opportunity lies in the implementation of an administrator portal. Through this portal, the status of all the detection nodes could be displayed. Web pages of the initialisation of a new venue and the assigning of which venue which user can track, can be placed here.

REFERENCES

[1] *ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4*. The Apache Software Foundation. Apr. 2019. URL: https://httpd.apache.org/docs/2.4/programs/ab.html (visited on 07/30/2019).

[2] T. Aihkisalo and T. Paaso. Latencies of service invocation and processing of the REST and SOAP web service interfaces. *Proceedings - 2012 IEEE 8th World Congress on Services, SERVICES 2012* (2012), 100–107. DOI: 10.1109/SERVICES.2012.55.

[3] *Alpine Linux*. Alpine Linux Development Team. July 2019. URL: https://alpinelinux.org/ (visited on 07/29/2019).

[4] *Amazon Kinesis Data Firehose*. Amazon Web Services. 2019. URL: https://aws.amazon.com/kinesis/data-firehose/ (visited on 07/03/2019).

[5] M. Anastopoulos and T. Romberg. Referenzarchitekturen für Web-Applikationen. (Dec. 2001). in German.

[6] *Apache httpd Modules*. The Apache Software Foundation. 2016. URL: http://httpd.apache.org/modules/ (visited on 07/09/2019).

[7] A. Babu. *Evolution from Web Sites to Web Apps — PWA*. Medium. Mar. 2018. URL: https://medium.com/beginners-guide-to-mobile-web-development/evolution-from-web-sites-to-web-apps-pwa-6aa25aeecd2b (visited on 06/16/2019).

[8] A. Banks, E. Briggs, K. Borgendale and R. Gupta. *MQTT Version 5.0*. Tech. rep. March. OASIS, 2019.

[9] L. Baresi, F. Garzotto and P. Paolini. From Web Sites to Web Applications: New Issues for Conceptual Modeling. (2001). Ed. by S. W. Liddle, H. C. Mayr and B. Thalheim, 89–100.

[10] L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2003. ISBN: 9780321154958.

[11] A. Beaulieu. *Learning SQL*. O'Reilly Media, Inc., 2005. ISBN: 0596007272.

[12] T. Berners-Lee. *The World Wide Web project*. CERN. Dec. 1990. URL: http://info.cern.ch/hypertext/WWW/TheProject.html (visited on 06/15/2019).

[13] *Best Container Management Software*. G2 Crowd, Inc. 2019. URL: https://www.g2.com/categories/container-management?segment=all (visited on 07/14/2019).

[14] M. A. Boillo. Application programming interface (API) for sensory events. 12. 2007.

[15] T. Butler. *NGINX Cookbook*. Packt Publishing, 2017. ISBN: 9781786466174.

[16] *Caching Overview*. Amazon Web Services. 2019. URL: https://aws.amazon.com/caching/ (visited on 06/30/2019).

[17] *Camera Module V2*. The Raspberry Pi Foundation. Apr. 2016. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ (visited on 07/29/2019).

[18] *Celery - Distributed Task Queue — Celery 4.3.0 documentation*. Celery. Mar. 2019. URL: http://docs.celeryproject.org/en/latest/ (visited on 06/30/2019).

[19] P. E. Ceruzzi. *A History of Modern Computing*. 2nd ed. Cambridge, MA, USA: MIT Press, 2003. ISBN: 0262532034.

[20] B. Chesneau. *Settings — Gunicorn 19.9.0 documentation*. July 2018. URL: http://docs.gunicorn.org/en/latest/settings.html (visited on 07/16/2019).

[21] *ColdFusion Versions CFML Documentation*. CFdocs. URL: https://cfdocs.org/coldfusion-versions (visited on 06/18/2019).

[22] *cPouta Community Cloud*. CSC - IT CENTER FOR SCIENCE LTD. 2019. URL: https://research.csc.fi/cpouta (visited on 07/29/2019).

[23] L. T. De Paolis, V. De Luca and R. Paiano. Sensor data collection and analytics with thingsboard and spark streaming. (June 2018), 1–6. DOI: 10.1109/EESMS.2018.8405822.

[24] *Deploying — Channels 2.1.7 documentation*. Django Software Foundation. Apr. 2019. URL: https://channels.readthedocs.io/en/latest/deploying.html (visited on 07/29/2019).

[25] *Developers Catalogue - FIWARE*. FIWARE Foundation, e.V. 2019. URL: https://www.fiware.org/developers/catalogue/ (visited on 07/18/2019).

[26] A. Deveria. *Can I use eventsource*. Caniuse. June 2019. URL: https://caniuse.com/#feat=eventsource (visited on 06/10/2019).

[27] A. Deveria. *Can I use Web Sockets*. Caniuse. June 2019. URL: https://caniuse.com/#search=WebSocket (visited on 06/10/2019).

[28] *Django Channels — Channels 2.1.7 documentation*. Django Software Foundation. Apr. 2019. URL: https://channels.readthedocs.io/en/latest/ (visited on 07/29/2019).

[29] *Django documentation*. Django Software Foundation. Apr. 2019. URL: https://docs.djangoproject.com/en/2.2/ (visited on 07/11/2019).

[30] *Django Packages : Webserver*. Django Software Foundation. Apr. 2019. URL: https://djangopackages.org/grids/g/webserver/ (visited on 07/10/2019).

[31] *Django Packages : Workers, Queues, and Tasks*. Django Software Foundation. Apr. 2019. URL: https://djangopackages.org/grids/g/workers-queues-tasks/ (visited on 07/12/2019).

[32] *Django reCAPTCHA*. Praekelt Consulting. Apr. 2019. URL: https://github.com/praekelt/django-recaptcha (visited on 07/29/2019).

[33] *Django REST framework*. Encode OSS Ltd. July 2019. URL: https://www.django-rest-framework.org/ (visited on 07/29/2019).

[34] *Docker Documentation*. Docker Inc. 2019. URL: https://docs.docker.com/ (visited on 07/14/2019).

[35] *Documentation: Apache HTTP Server*. The Apache Software Foundation. Apr. 2019. URL: https://httpd.apache.org/docs/ (visited on 07/28/2019).

[36] W. Dreyer, A. K. Dittrich and D. Schmidt. Research perspectives for time series management systems. *ACM SIGMOD Record* 23.1 (1994), 10–15. ISSN: 01635808. DOI: 10.1145/181550.181553.

[37] T. Dunning and E. Friedman. *Time series databases*. Ed. by M. Loukides. First Edition. Sebastopol, CA: O'Reilly Media, Inc, Dec. 2015, 73. ISBN: 9781491917022.

[38] P. J. Eby. *PEP 333 – Python Web Server Gateway Interface v1.0*. Python Software Foundation. Dec. 2003. URL: https://www.python.org/dev/peps/pep-0333/ (visited on 07/10/2019).

[39] P. J. Eby. *PEP 3333 – Python Web Server Gateway Interface v1.0.1*. Python Software Foundation. Sept. 2010. URL: https://www.python.org/dev/peps/pep-3333/ (visited on 07/10/2019).

[40] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, Dec. 2011. URL: http://www.rfc-editor.org/rfc/rfc6455.txt.

[41] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, May 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[42] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California, Irvine, 2000, 162.

[43] D. Flanagan. *JavaScript: The Definitive Guide Activate Your Web Pages*. 5th. O'Reilly Media, Inc., 2011. ISBN: 0596805527.

[44] M. Frampton. *Complete Guide to Open Source Big Data Stack*. Apress, 2018. ISBN: 9781484221495. URL: https://books.google.fi/books?id=Y8FHDwAAQBAJ.

[45] N. Freed and N. S. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045. RFC Editor, Nov. 1996. URL: http://www.rfc-editor.org/rfc/rfc2045.txt.

[46] J. Fulton. *Web Architecture 101*. Medium. Nov. 2017. URL: https://engineering.videoblocks.com/web-architecture-101-a3224e126947 (visited on 07/04/2019).

[47] *Getting Started - ThingsBoard*. The ThingsBoard Authors. 2019. URL: https://thingsboard.io/docs/getting-started-guides/helloworld/ (visited on 07/18/2019).

[48] J. Gillies and R. Cailliau. *How the Web was Born: The Story of the World Wide Web*. Oxford paperback reference. Oxford University Press, 2000. ISBN: 9780192862075.

[49] I. Goodfellow, Y. Bengio and A. Courville. *Deep Learning*. MIT Press, 2016.

[50] A. Goodman. *Application vs System Container*. Excella. Oct. 2017. URL: https://www.excella.com/insights/application-vs-system-containers (visited on 07/13/2019).

[51] I. Grigorik. *High-Performance Browser Networking*. Vol. 1. O'Reilly Media, Inc., 2013, 404. ISBN: 978-1-4493-4476-4. DOI: 10.1017/CBO9781107415324.004. eprint: arXiv:1011.1669v3.

[52] C. de la Guardia. *Python Web Frameworks*. O'Reilly Media, Inc., Feb. 2016. ISBN: 9781491938102.

[53] M. Hadley, H. F. Nielsen, A. Karmarkar, M. Gudgin, N. Mendelsohn, J.-J. Moreau and Y. Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation. W3C, Apr. 2007.

[54] J. Han, M. Kamber and J. Pei. *Data Mining: Concepts and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 9780123814791.

[55] G. E. Hinton, S. Osindero and Y.-W. Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.* 18.7 (July 2006), 1527–1554. ISSN: 08997667. DOI: 10.1162/neco.2006.18.7.1527.

[56] S. Hussain. *How To Set Up a Two Node LEPP Stack on CentOS 7*. DigitalOcean, LLC. Mar. 2015. URL: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-two-node-lepp-stack-on-centos-7 (visited on 06/17/2019).

[57] *Implementations — ASGI 2.0 documentation*. Django Software Foundation. Mar. 2019. URL: https://asgi.readthedocs.io/en/latest/implementations.html (visited on 07/11/2019).

[58] *Intel Movidius Neural Compute Stick*. Intel Corp. Nov. 2017. URL: https://software.intel.com/en-us/movidius-ncs (visited on 07/29/2019).

[59] *Introduction — ASGI 2.0 documentation*. Django Software Foundation. Mar. 2019. URL: https://asgi.readthedocs.io/en/latest/introduction.html (visited on 07/11/2019).

[60] *Introduction to Celery — Celery 4.3.0 documentation*. Celery Project. July 2019. URL: http://docs.celeryproject.org/en/latest/getting-started/introduction.html (visited on 07/12/2019).

[61] *June 2019 Web Server Survey*. Netcraft Ltd. June 2019. URL: https://news.netcraft.com/archives/2019/06/17/june-2019-web-server-survey.html (visited on 07/09/2019).

[62] R. M. Junior. *Bare metal vs. virtual servers: Which choice is right for you?* IBM - Cloud computing news. July 2014. URL: https://www.ibm.com/blogs/cloud-computing/2014/07/25/bare-metal-vs-virtual-servers-choice-right/ (visited on 06/17/2019).

[63] G. Kappel, B. Pröll, S. Reich and W. Retschitzegger. *Web Engineering: The Discipline of Systematic Development of Web Applications*. Wiley, June 2006. ISBN: 9780470015544.

[64] A. Kulkarni. *What the heck is time-series data (and why do I need a time-series database)?* Timescale, Inc. Nov. 2018. URL: https://blog.timescale.com/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/ (visited on 05/23/2019).

[65] J. Lengstorf and P. Leggetter. *Realtime Web Apps: With HTML5 WebSocket, PHP, and jQuery*. Apress Media, LLC, 2013, 312. ISBN: 978-1-4302-4620-6. DOI: 10.1007/978-1-4302-4621-3.

[66] Y. P. de León and T. Piscopo. *Druva Cloud Platform*. Druva. Aug. 2014. URL: `https://www.druva.com/blog/object-storage-versus-block-storage-understanding-technology-differences/` (visited on 07/02/2019).

[67] A. Leslie. *NGINX vs. Apache (Pro/Con Review, Uses, & Hosting for Each)*. HostingAdvice.com. Jan. 2018. URL: `https://www.hostingadvice.com/how-to/nginx-vs-apache/` (visited on 07/09/2019).

[68] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg. SSD: Single Shot MultiBox Detector. To appear. 2016. URL: `http://arxiv.org/abs/1512.02325`.

[69] J. López-Riquelme, N. Pavón-Pulido, H. Navarro-Hellín, F. Soto-Valles and R. Torres-Sánchez. A software architecture based on FIWARE cloud for Precision Agriculture. *Agricultural Water Management* 183.C (2016), 123–135. DOI: `10.1016/j.agwat.2016.10.0`.

[70] J. Martin. *Managing the Data Base Environment*. Pearson Education, Limited, 1983. ISBN: 9780135505823.

[71] F. McCabe, D. Booth, H. Haas, M. Champion, C. Ferris, D. Orchard and E. Newcomer. *Web Services Architecture*. W3C Note. W3C, Feb. 2004. URL: `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`.

[72] W. Mcculloch and W. Pitts. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* 5 (1943), 127–147.

[73] J. Meier, C. Farre, P. Bansode, S. Barber and D. Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Redmond, WA, USA: Microsoft Press, 2007. ISBN: 9780735625709.

[74] P. M. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011.

[75] A. Mikalauskas. *What was wrong with Apache*. speedemy. Apr. 2015. URL: `http://www.speedemy.com/apache-vs-nginx-2015/` (visited on 07/09/2019).

[76] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[77] *mod_wsgi — mod_wsgi 4.6.7 documentation*. Graham Dumpleton. 2019. URL: `https://modwsgi.readthedocs.io/en/develop/` (visited on 07/10/2019).

[78] *Motivation — Dramatiq 1.6.0 documentation*. Cleartype SRL. May 2019. URL: `https://dramatiq.io/motivation.html` (visited on 07/12/2019).

[79] A. Mouat. *Using Docker*. O'Reilly Media, Inc., 2015. ISBN: 9781491915769.

[80] *Multi-Processing Modules (MPMs) - Apache HTTP Server Version 2.4*. The Apache Software Foundation. 2019. URL: `https://httpd.apache.org/docs/current/mpm.html` (visited on 07/09/2019).

[81] S. Mumbaikar and P. Padiya. Web Services Based On SOAP and REST Principles. *International Journal of Scientific and Research Publications* 3.5 (2013), 3–6. URL: `www.ijsrp.org`.

[82] N. Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *2017 IEEE International Systems Engineering Symposium*

(ISSE). IEEE, Oct. 2017, 12–18. ISBN: 9781538634042. DOI: 10.1109/SysEng.2017.8088251.

[83] J. Nelson. *Mastering Redis: take your knowledge of Redis to the next level to build enthralling applications with ease*. Birmingham: Packt Publ., 2016. ISBN: 9781783988181.

[84] *NGINX Docs*. NGINX Inc. Apr. 2019. URL: https://docs.nginx.com/nginx/ (visited on 07/28/2019).

[85] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 0125184050.

[86] R. Nixon. *Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5*. 4th. O'Reilly Media, Inc., 2014. ISBN: 1491918667.

[87] *PostgreSQL*. The PostgreSQL Global Development Group. June 2019. URL: https://www.postgresql.org/ (visited on 07/29/2019).

[88] *RabbitMQ - Homepage*. Pivotal Software, Inc. July 2019. URL: https://www.rabbitmq.com/ (visited on 07/29/2019).

[89] *RabbitMQ vs Redis : What are the differences?* StackShare, Inc. 2019. URL: https://stackshare.io/stackups/rabbitmq-vs-redis (visited on 07/12/2019).

[90] P. Raj, A. Raman and H. Subramanian. *Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture*. Packt Publishing, 2017. ISBN: 9781787287495.

[91] *Raspberry Pi 3 Model B+*. The Raspberry Pi Foundation. Mar. 2018. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ (visited on 07/29/2019).

[92] A. Rauschmayer. *Speaking JavaScript*. 1st. O'Reilly Media, Inc., 2014. ISBN: 1449365035.

[93] E. Rescorla. *HTTP Over TLS*. RFC 2818. RFC Editor, May 2000. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

[94] L. and Richardson and S. Ruby. *RESTful Web Services*. Ed. by M. Loukides. First Edit. Vol. 1. 1. Sebastopol, California: O'Reilly Media., 2007, 103. ISBN: 9780596529260.

[95] L. Roberts. The Evolution of Packet Switching. *Proceedings of the IEEE* 66 (Dec. 1978), 1307–1313. DOI: 10.1109/PROC.1978.11141.

[96] R. Romano and M. Kalin. *Java Web Services: Up and Running*. Second ed. O'Reilly Media, Inc., 2009, 316. ISBN: 9780596521134.

[97] K. Schroeder. *Performance of Apache 2.4 with the event MPM compared to Nginx*. ESchrade. 2014. URL: https://www.eschrade.com/page/performance-of-apache-2-4-with-the-event-mpm-compared-to-nginx/ (visited on 07/09/2019).

[98] R. Singh Chowhan. Evolution and Paradigm Shift in Distributed System Architecture. *IntechOpen* (2018). DOI: 10.5772/intechopen.80644.

[99] Ø. R. Tangen. Real-Time Web with WebSocket. Master's Thesis. University of Oslo, May 2015. URL: https://www.duo.uio.no/handle/10852/44808.

[100] T. H. Team. *MQTT Essentials*. HiveMQ. Jan. 2015. URL: https://www.hivemq.com/tags/mqtt-essentials/ (visited on 06/05/2019).

[101] *The uWSGI project — uWSGI 2.0 documentation*. unbit. Feb. 2019. URL: https://uwsgi-docs.readthedocs.io/en/latest/ (visited on 07/11/2019).

[102] *TimescaleDB Docs | Overview*. Timescale, Inc. May 2019. URL: https://docs.timescale.com/v1.3/introduction (visited on 07/12/2019).

[103] D. Tomaszuk. *Python's Frameworks Comparison: Django, Pyramid, Flask, Sanic, Tornado, BottlePy and More*. Netguru S.A. Nov. 2018. URL: https://www.netguru.com/blog/python-frameworks-comparison (visited on 07/11/2019).

[104] *Tutorial: Installing a WIMP Server on an Amazon EC2 Instance Running Windows Server*. Amazon Web Services. 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/install-WIMP.html (visited on 06/17/2019).

[105] *Uvicorn*. Encode OSS Ltd. June 2019. URL: https://www.uvicorn.org/ (visited on 07/29/2019).

[106] *Waitress — waitress 1.3.0 documentation*. Pylons Project. Apr. 2019. URL: https://docs.pylonsproject.org/projects/waitress/en/latest/# (visited on 07/10/2019).

[107] *Web Server (IIS) Overview | Microsoft Docs*. Microsoft Corporation. Oct. 2018. URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831725(v%5C%3Dws.11) (visited on 07/28/2019).

[108] *Welcome to Flask — Flask Documentation (1.1.x)*. the Pallets Projects. Sept. 2019. URL: https://flask.palletsprojects.com/en/1.1.x/ (visited on 07/11/2019).

[109] *What is PaaS? Platform as a Service*. Microsoft. 2019. URL: https://azure.microsoft.com/en-ca/overview/what-is-paas/ (visited on 06/17/2019).

[110] *What is Streaming Data?* Amazon Web Services. 2019. URL: https://aws.amazon.com/streaming-data/ (visited on 07/03/2019).

[111] *Why use a Content Delivery Network (CDN)?* GTmetrix. Feb. 2017. URL: https://gtmetrix.com/why-use-a-cdn.html (visited on 07/01/2019).

[112] T. Yokotani and Y. Sasaki. Comparison with HTTP and MQTT on required network resources for IoT. *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*. Sept. 2016, 1–6. DOI: 10.1109/ICCEREC.2016.7814989.