UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Asset Localization using the Bluetooth Fingerprinting technique

Menke L. Veerman

B.Sc. Thesis
July 2023

Supervisors:
Dr. Duc Viet Le
Son Nguyen

Pervasive Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7522 NB Enschede
The Netherlands

Acknowledgements

I would like to thank my supervisors, Dr. Duc Viet Le and Son Nguyen, for guiding me through this graduation project. The meetings were helpful to ask for some elaboration or to discuss some feedback. Additionally, I would also like to thank them for their fast responses to email. These were really helpful when there was no meeting scheduled and I did not have to wait before continuing.

Abstract

Knowing where something is, and being able to quickly look it up on a phone. While also having the ability to know where something has been can be of great value to evaluate and improve workflows. A typical scenario could be in a warehouse where an employee on the workfloor needs to locate a piece of equipment. Or that management would like to evaluate the forklift's usage.

Both of these can be done with the system that is developed as part of this research project. The system described in this graduation project is divided onto two devices, a Raspberry Pi and a remote server. The Raspberry Pi listens to Bluetooth Low Enegergy (BLE) advertising packets that were emitted by a Thingy 52 from Nordic Semiconductor. The Thingy 52 is a small, mobile device with many internal sensors that has a battery and can be programmed. This device emits a BLE advertising packet every few seconds with data such as temperature, humidity, CO2 Parts per million (PPM) and battery percentage while remaining completely connectionless. This packet is received by a Raspberry Pi using a nRF 52 development board running the nRF BLE Sniffer software from Nordic Semiconductor. The packet is then queued into a Redis Stream and later sent to Apache Kafka on the remote server.

The remote server uses KSQL and Kafka Connect to join and send the packets with location to a Time Series Database (TSDB) named QuestDB. An Express.js webserver interacts with Apache Kafka and QuestDB, serving a Vue frontend that visualizes the Thingies on a live map using MapBox. It also features a page where thingies can be added/deleted, whose changes are immediately put into effect via the usage of Apache Kafka. The last page is a historical page where Thingies can be inspected and viewed over a longer period of time using different visualizations such as a heatmap, lines or only dots using downsampled data from QuestDB.

The system was tested by applying a load at various aspects to find potential issues. This identified a flaw regarding the data retrieval for the historical page on the dashboard.

It also showed that the packets received by a Raspberry Pi can be sent without any data loss to Apache Kafka. Overall, the system is not perfect, but is built to be improved and expanded. Both in terms of features as scale.

Acronyms

ACL	Access control list
AoA	Angle of Arrival
API	Application Programming Interface
BLE	Bluetooth Low Enegergy
CRC	Cyclic Redundancy Check
FOTA	firmware over-the-air
IoT	Internet of Things
LTS	Long term support
MVC	Model View Controller
MVT	Model View Template
NCS	nRF Connect SDK
PDU	Protocol Data Unit
PPM	Parts per million
RF	Radio frequency
RPS	Requests per Second
RSSI	Received Signal Strength Indicator
RTOS	Real-Time Operating System
SDK	System Development Kit

SFLOAT	Short float
SoC	System on a Chip
SOC	State of charge
SPA	Single Page Application
TDoA	Time-Difference of Arrival
TSDB	Time Series Database
TWR	Two Way Ranging
UWB	Ultra Wide Band

Contents

Chapter 1

Introduction

Indoor asset tracking can be a tool of big utility in places such as warehouses, hospitals and other industrial sites. This tool stores the locations of assets in an indoor environment, making it easy to locate assets. This can be useful in a variety of applications. It can be used to find things that are lost quickly, reducing the time lost searching for it. Another application would be to track fire extinguishers to prove conformation to safety guidelines. These are just two examples out of the many other applications possible.

1.1 Problem statement

The system can be divided into two main parts; the localization and the visualization. Although the work in this thesis will mainly focus on the visualization, it is constructed around the localization. Because of this, there is first a technical overview of the localization. After which the problem will be defined.

1.1.1 Technical background

The localization is the part that is responsible for actually getting a position (e.g. coordinates) of a tracked asset. A tracker continuously broadcasts a BLE packet with some data in it, a beacon. This beacon is received by various receivers across a room. A receiver consists of a disc with 5 antennas, they listen for the signal strengths of the transmissions made by the trackers. This perceived signal strength is also known as Received Signal Strength Indicator (RSSI), which varies a bit for all antennas on a receiver. A receiver then transmits the contents of a packet accompanied by the measured RSSI values to a centralized computer. The usage of multiple antennas extend the operating range as well as improve accuracy as there is more data to use. A receiver then transmits these perceived RSSI values along with the beacon's content as the centralized computer isn't guaranteed to be in range of a receiver, this extends the operating range of the system. Once all RSSI values of multiple receivers have arrived at the centralized computer, machine learning is used to compute a location of the origin of the beacon.

1.1.2 Problem definition

Once a position is computed, it is saved and passed on to the visualization part. Which is responsible for everything onwards. The visualization of the data is an important aspect of the system as it has the ability to make the data meaningful. As the number of trackers can quickly grow to large numbers, it is important that the system can effectively and efficiently show the locations of the trackers on a map. Independent of the number of trackers. This dashboard should also be quickly accessible, allowing it to be used from anywhere where something needs to be looked up quickly.

1.2 Research questions

With this the following research question has been defined:

How to design an accessible and efficient system that can visualize the locations of the Bluetooth trackers?

To aid in answering the main research question, several sub research questions have been defined:

1. What is the best way to store the locations from the trackers?
2. How can the dashboard be designed to be accessible and easy to use?
3. How can the reliability of the system be maximized?
4. What strategies can be used to optimize the scalability of the system?

Chapter 2

Background research

2.1 Hardware

A maximum of around 30 Thingy 52's from Nordic Semiconductor were made available by Dr. Duc Viet Le. They will be the trackers that emit the BLE beacons.

2.1.1 Thingy 52

The Thingy52 was released in 2017. It is a small development device based around their nRF52832 Bluetooth 5 System on a Chip (SoC), which is quite similar to the popular ESP32 from Espressif, but without WiFi. The Thingy52 has a built-in battery, a few LED's and many built-in sensors: [1]

1. LIS2DH12 : Low power accelerometer
2. MPU-9250 : 9-axis Magnetometer, accelerometer and gyroscope.
3. HTS221 : Humidity and temperature sensor
4. BH1745NUC : Color sensor
5. CCS811 : Gas sensor
6. MP34DB02 : Digital microphone

All of which are available via the nRF Connect SDK (NCS) System Development Kit (SDK) from Nordic Semiconductor [2]. With this SDK it is also possible to enable firmware over-the-air (FOTA) updates.

2.1.2 Raspberry Pi

Not only were there many Thingy's provided but also a Raspberry Pi 3B+. This is a small 64-bit quad-core ARM-based microprocessor. Released in 2016 and has 1GB of DDR2 ram, WiFi and Bluetooth 4.2 (BLE). The operating systems on this that can be run are Linux such as Raspbian and Ubuntu. [3]

2.1.3 Conclusion

There are many sensors available inside a Thingy 52 that can be used for transmitting of auxilary data such as humidity, temperature and gas levels. A current limitation is that the given Raspberry Pi 3B+ is limited to Bluetooth 4, limiting the amount of data that can be transferred via advertising (see 2.2.3).

2.2 Bluetooth

As of the writing of this report, the localization aspect nor the trackers are ready and thereof need to be programmed to emit BLE beacons as well. As both the trackers and visualization need to interact with BLE beacons, there will be taken a look at their core functionality.

2.2.1 Bluetooth Low Energy vs Bluetooth (classic)

Bluetooth Low Energy (BLE) was included in the Bluetooth 4.0 standard released in 2010. It was developed by Nokia, Nordic Semiconductor and a few others in search for a less power consuming Bluetooth mode. Bluetooth (classic) is nowadays used for applications which require a high bandwidth, such as file- and audio transmissions. However, Internet of Things (IoT) applications have adopted BLE as it consumes less power and has a lower bandwidth, which is often fine. As of this the project will continue with the usage of BLE, with any occurrences of ”Bluetooth”, the Low Energy variant is intended. [4]

2.2.2 What are Bluetooth beacons

Bluetooth beacons are devices that keep transmitting Bluetooth advertising packets but do not accept or make any connections. As they continuously transmit, they can be discovered at any time and do not suffer from connection losses. Which is convenient for trackers that move around a lot where maintaining a connection could be difficult [5].

2.2.3 Bluetooth Low Energy advertising

Devices can use advertising to indicate to other devices that they want to make a connection or remain connectionless and keep transmitting data via beacons. In the Bluetooth spectrum there are 40 channels of which 3 are dedicated to the advertising.

An advertising packet consists of 4 parts (figure 2.1):

1. Preamble
2. Access Address
3. Protocol Data Unit (PDU) (part of the Header in fig 2.1)
4. Cyclic Redundancy Check (CRC)

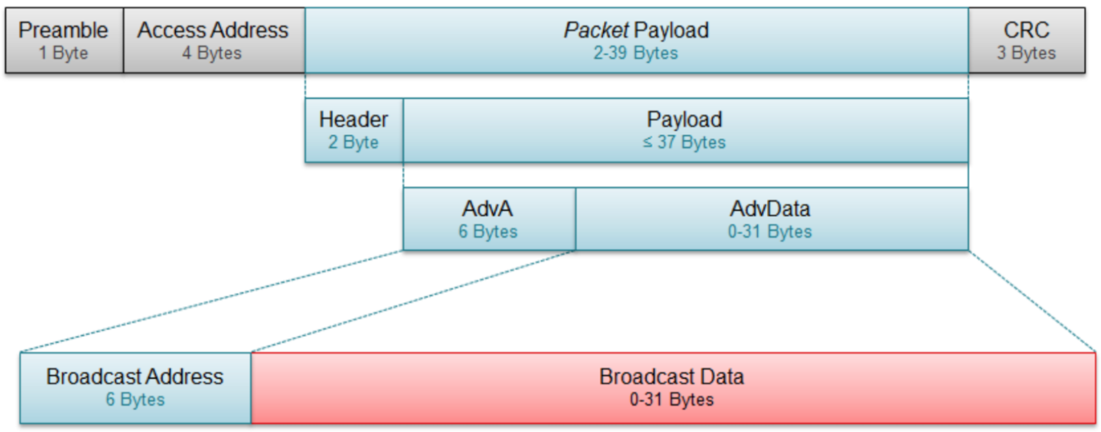
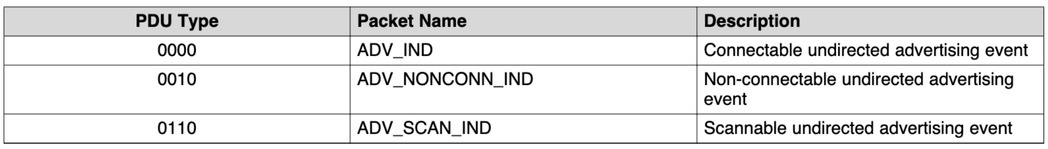


Figure 2.1: BLE advertising packet [6]

The preamble is 1 byte and used to synchronize the transmission and has a fixed value of 0xAA. The Access Address is 4 bytes long and an identifier of what kind of packet it is, advertising packets have the standard value of 0x8E89BED6. The value of the PDU determines the purpose. A broadcasting packet can have 3 values for this; ADV_IND, ADV_NONCONN_IND and ADV_SCAN_IND. [7] [6]



PDU Type	Packet Name	Description
0000	ADV_IND	Connectable undirected advertising event
0010	ADV_NONCONN_IND	Non-connectable undirected advertising event
0110	ADV_SCAN_IND	Scannable undirected advertising event

Figure 2.2: PDU types for broadcasting data [6]

The payload for the PDU is the longest with the ADV_NONCONN_IND, it can then hold a maximum of 31 bytes of data. The last 3 bytes of a packet is the CRC, which is used to check the packet for errors.
However, there is a possibility to send out more data with advertising via the extended advertising feature included in Bluetooth 5. It still keeps the 37:3 data/advertising channel ratio, but makes a distinction between primary and secondary advertising packets. The primary advertising packets are used in the 3 advertising channels whilst the secondary packets flow via the 37 data channels. Extended advertising uses mostly the same packet as Bluetooth 4 does, but with a different PDU (the primary advertising packet). The payload then contains a reference to a packet that will be sent later via the data channels (secondary advertising packet). This allows a total of 255 bytes via the secondary advertising packet. [8] [9]

2.3 Database

A lot of beacons can be generated each day depending on the emission frequency and the number of beacons. For example, if there are 30 Thingy 52's emitting every 5 seconds, after 24 hours are that 518.400 entries to the database. However, all this data is temporal, it needs to be stored by timestamp and later retrieved in large amounts by timestamp. This is where TSDB come into play. A TSDB is built from the ground up to store temporal data and is thereof good at querying over months of data. It can also automatically downsample the data over time [10].

2.4 Dashboard layout

As a lot of data from trackers needs to be visualized, a clear and easy way to view this data by the operators is of great importance to create accurate analyzations of the current operations. An effective website is essential to this. The dashboard should be aesthetic as it defines the credibility and interaction with a website [11]–[14]. As of this, there will be looked at how the aesthetics impact the usability (/performance) of a website. This literature research starts with the layout of a website and continues with the colors of a website.

2.4.1 Layout

The layout of a site is the first thing people see upon entry and has impact on the usability of a site. Once landed, the first thing people do is skimming it. Francisco-Revilla and Crow [15] did a study and showed that this is a process that only takes a few seconds. They also argue that a user starts skimming it from left to right, top to bottom. When skimming, a user does not look at the content of a page, but rather at the structure and layout of it [12], [15]. People are looking for familiar parts such as the search bar, navigation bar and ads. One thing to note is that ads are merely used for reference points, people are aware of their presence, but will mostly ignore their contents [15].

Once the page has been skimmed, users focus their attention on the body of a page. Parush et al. [16] performed an experiment with 71 students who had previous experience with Internet Explorer regarding the structure of the contents by analyzing their performance (search time and eye movement). They tested 4 aspects of the body of a page: the grouping of text, alignment of text, density of the text and the link quantity (figure 2.3). The experiment concluded with 2 major factors that impact on the performance. The biggest impact on performance was the number of links, where more links negatively affected the performance. The secondary factor was the density of the text. The uniform density had a positive impact on the performance. Contrastingly, good alignment did not have a positive effect. Grouping did not have a significant effect. Bonnardel et al. [17] supports the effect of uniform density as they claim that bullet points are easier to scan than full text. Bullet points are a uniform set of stacked text, similar to the uniform density showed by Parush et al. [16].

All in all, Francisco-Revilla and Crow [15], and Michailidou et al. [12] showed that on first entry, users start skimming the site for the layout and recognizable points for a few seconds. Starting at the top left corner and continuing like reading a book. The structure of the body is also important for the usability of the site as too many links negatively impact the performance.

2.4.2 Color

Although the layout being a big impact on the first impression, colors also contribute and cannot be left out. They are used in almost all sites nowadays and can be used to achieve different things. To illustrate, Swasty and Adriyanto [18] classified colors by what behavior they promote (figure 2.4). Mirza [19] evaluated a few sites and falls in line with the table generated by Swasty and Adriyanto [18](figure 2.4).

Bonnardel et al. [17] executed an experiment involving 50 users and 30 expert (web) designers to analyze the impact of website appeal by colors. The experiment was limited to the hue of colors. Participants were shown the same site in 18 different colors in a random order and were asked to rate it. The regular users favored the colors orange and blue while the designers also favored gray. A secondary experiment was conducted from the results of the first experiment. Aspects such as time spent, memorization and number of pages visited were measured across the same group for the orange, gray and blue site. The gray site resulted in the least engagement and memorization. The blue and orange site were appraised more, however, more time was spent on the orange site than the blue one. Implying that the orange site would engage the users more and that the blue site would be more efficient. Mirza [19], and Swasty and Adriyanto [18] come to a similar conclusion regarding blue and orange. For orange, Mirza [19] states that orange can be playful and engaging which falls in line with uniqueness, friendliness and sensation of movement stated by Swasty and Adriyanto [18]. As for blue, Swasty and Adriyanto [18] state safety, calmness, strength and reliability where Mirza [19] argues dependability and strength.

Another aspect is how a color is used on a website; they should be harmonious to really engage users [18]. Cleveland [20] analyzed the color ratios of a set of good/bad categorized websites. He describes that in most cases, the better designed websites, used colors differently than those poorly designed. The better websites leaned more towards 'softer' colors while the lesser sites had colors with stronger hue values. Another difference is present at the usage of secondary or supportive colors. The lesser categorized sites used more distinct colors for this while the better sites used a smaller selection of colors that are mostly close to the primarily used color.

However, they need to be used carefully as colors can have different meanings for different nations or continents, which can be quite challenging [11], [18]. Kondratova and Goldfarb [11] studied the usage of colors on websites of 29 nations and were able to create a global color palette that is safe to use internationally. This was done by analyzing the preferences and common usage of the studied nations. From this, the palette consisting of shades of blue and gray accompanied by yellow was formed to be safe among different nations (figure 2.5).

So, colors can be used to set the user in a certain mood or to portray a certain site more accurate. They can be best used in combination with other supportive colors of a similar hue. However, one must be careful as they can be tricky due to cultural differences.

2.4.3 Discussion and Conclusion

This literature research aimed to identify aesthetic aspects of websites that affect the usability/performance of a website. From research it is found that both layout and color have a significant effect. Upon first entry, users skim the page in a few seconds and look for familiar key elements such as search- and navigation bars. Therefore, a website should facilitate such elements to make this easy for the user. It has also shown that a uniform density (figure 2.3), much like bullet points, enhances performance. Contrastingly, too many links decrease it.

As layout is mostly used for performance, color has also the ability to create additional effects. This is how an orange site is better at engaging people while a blue site is perceived as more efficient. However, color is not perceived the same in every nation. Some colors might have different effects/meanings, but there is a palette that is safe to use amongst most of them (figure 2.5).

This also brings a limitation of the study. The studied papers originate from different nations which might make papers regarding the effects of color not generalizable, and hard to align with each other. Another limitation is that web usage and technology has evolved over the years and that some papers might have lost a bit of their relevance towards modern websites.

Illustrations/pictures and fonts were not included in this research. It is thereof unsure what their effects are on a website. Further research could expand on this paper by including the aforementioned aspect. Another point of improvement would be to include more modern research papers to make it up to current internet standards.

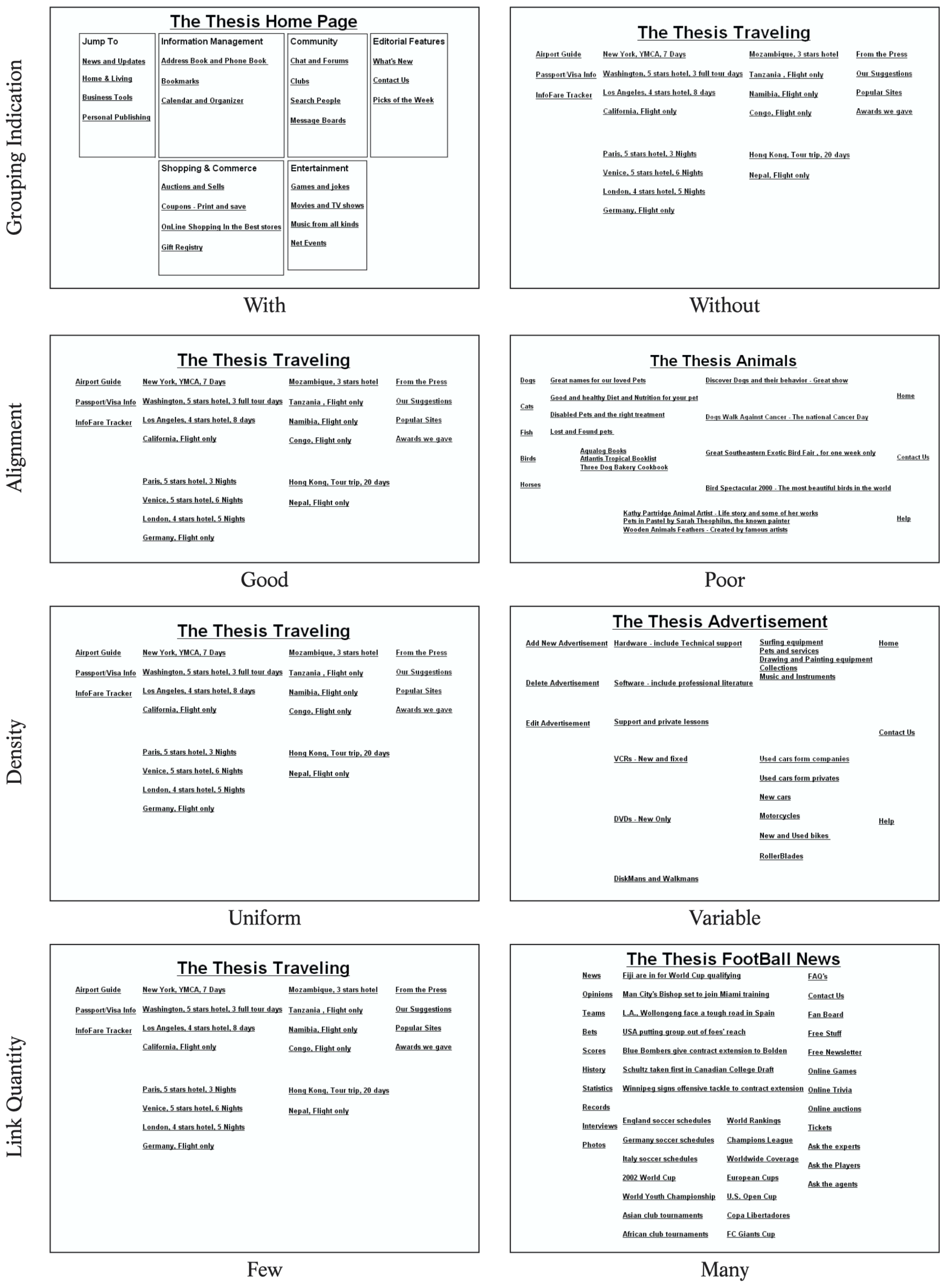
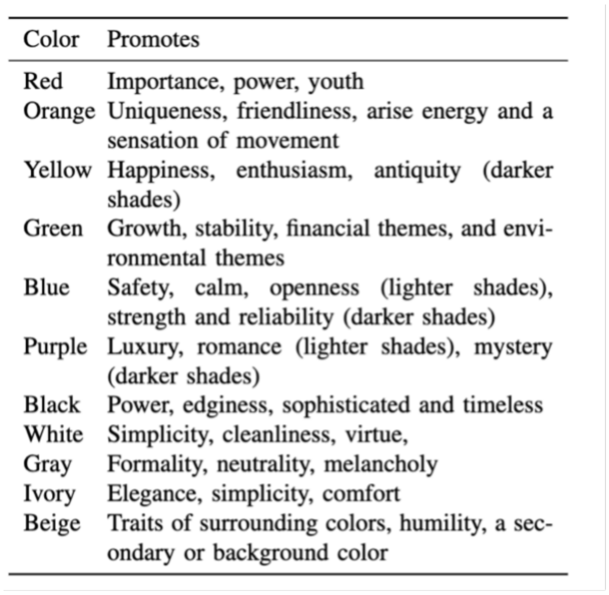
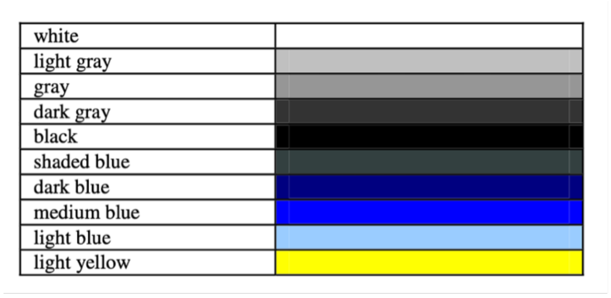


Figure 2.3: 4 tested aspects of the body of a website [16]



Color	Promotes
Red	Importance, power, youth
Orange	Uniqueness, friendliness, arise energy and a sensation of movement
Yellow	Happiness, enthusiasm, antiquity (darker shades)
Green	Growth, stability, financial themes, and environmental themes
Blue	Safety, calm, openness (lighter shades), strength and reliability (darker shades)
Purple	Luxury, romance (lighter shades), mystery (darker shades)
Black	Power, edginess, sophisticated and timeless
White	Simplicity, cleanliness, virtue,
Gray	Formality, neutrality, melancholy
Ivory	Elegance, simplicity, comfort
Beige	Traits of surrounding colors, humility, a secondary or background color

Figure 2.4: Different colors and what they promote [18]



white	⬜
light gray	░
gray	▒
dark gray	▓
black	⬛
shaded blue	▓
dark blue	🟦
medium blue	🟦
light blue	🟦
light yellow	🟨

Figure 2.5: Internationally safe color scheme [11]

2.5 Web frameworks

A website is composed of a frontend, that what can be interacted with and is visible to the user. And a backend which is responsible for things such as authentication, retrieval of data from a database and more. As there are many frameworks out there to choose from, a selection was made from known/familiar, popular or most-loved frameworks. [21] [22]

2.5.1 Backend

Django

Django is written in Python. It is a rapid full-fledged Model View Template (MVT) framework that takes care of things as routing, authentication and the database usage.

Flask

Flask is also written in Python, but is less extensive than Django. It doesn't come with as many built-in features, giving the developer more freedom to choose his own packages.

Laravel

In contrast to Flask and Django, Laravel is written in PHP and using the Model View Controller (MVC) structure. The framework also takes care of things such as routing, authentication and database usage. Unlike Python, PHP does not have support for Bluetooth.

Spring

Spring is written in Java, using the MVC structure. It is quick to set up and modular in its usage.

Express.js

Express.js is a web framework for Node.js which uses JavaScript. It is like Flask, a minimal web framework.

Ruby on Rails

Ruby on Rails is a Ruby full-fledges MVC framework that takes care of routing, authentication and database usage.

ASP.NET

ASP.NET is a web framework from Microsoft written in c# using the MVC structure. Taking care of authentication, routing and database usage.

However, not all frameworks have the same performance. Some are faster than others. The Benchmarker [23] created a benchmarking tool so many frameworks can be tested as equally possible. TechEmpower [24] performed also synthetic benchmarks, but on much more recent hardware. The performed tests are not identical but the trend of scores can still be evaluated. Figures 2.6 and 2.7 have the results for 64, 256 and 512 concurrent connections respectively from the Benchmarker [23], the combined results are available in table 2.1. The performance is measured through Requests per Second (RPS), which is how many requests per second a web server is capable of, and latency, the delay it takes to respond to a request. Looking at the results in figures 2.6 and 2.7, Spring and ASP.net are noticeably faster than the others, topping out at about 78.000/100.000 requests per second. It also has the lowest latency throughout the test. Laravel on the other hand is the slowest of all, managing only about 2.800 requests per second. Ruby on rails is slightly faster with 4.500 requests per second, although its 99 percentile latency is not. They both also have the highest latencies. Django and Flask are also steady across all three tests, managing 8.500 and 15.000 requests per second respectively. Express takes the third place at 23.000 requests per second.

TechEmpower [24] agrees that ASP.NET can have the most requests per second and that Express.js is on the second place. However, other scores such as Django, Spring and Flask were all over the place. Scoring suddenly a lot better or worse. Although it might be a bit like comparing apples to pears due to the many differences in testing and setup, ASP.NET is the fastest framework concluded by both tests.

Framework	TechEmpower RPS [24]	The Benchmarker RPS [23]
Django	15.038	8.500
Spring	22.478	78.000
Flask	1.869	15.000
Express.js	40.737	23.000
Ruby on Rails	9.925	4.500
Laravel	8.437	2.800
ASP.NET	394.144	100.000

Table 2.1: Web backend frameworks throughput (RPS) of TechEmpower vs The Benchmarker

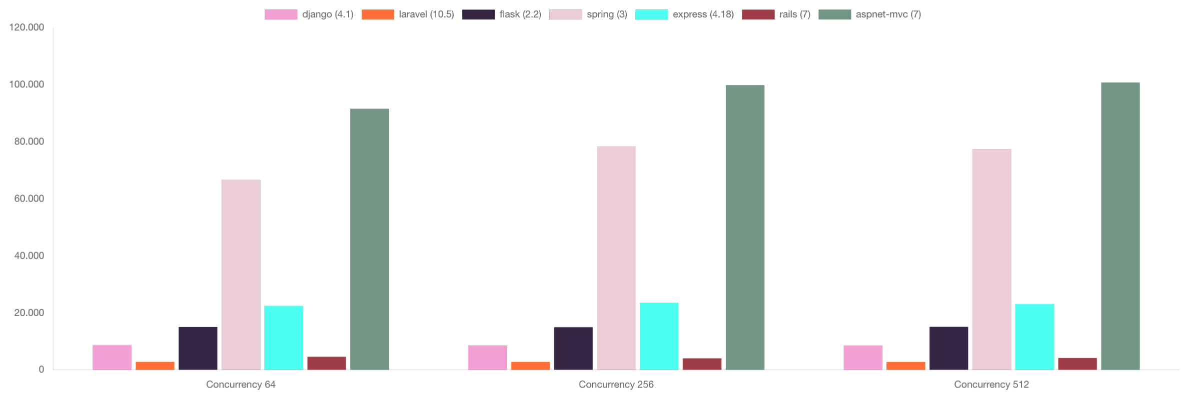


Figure 2.6: Web frameworks requests per second from The Benchmarker [23]

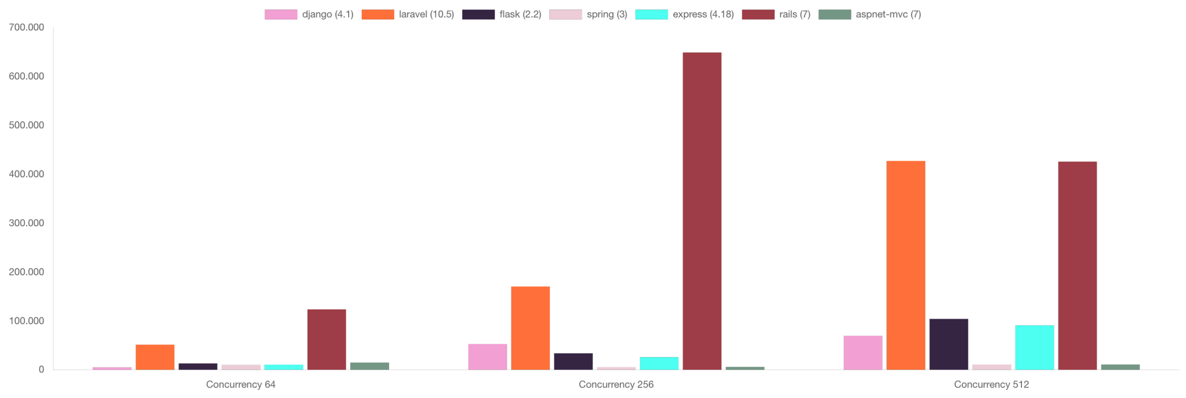


Figure 2.7: Web frameworks 99 percentile delay (ms) [23]

2.5.2 Frontend

The frontend of a website is visible to the user. The development experience can be a lot easier with the inclusion of a frontend framework. There are 3 popular frameworks out there, Vue.js, React.js and Svelte (table 2.2). In essence, they all do something similar, making it easier to create an interactive frontend. However, some frameworks have a better integration with a certain backend than others. In the case that the frameworks will be used as a Single Page Application (SPA) it does not matter anymore as they then communicate with a backend via an Application Programming Interface (API).

Framework	GitHub stars (K)	NPM weekly downloads	Latest update
Vue.js	203+36.7 [25] [26]	3.544.594 [27]	1 day ago
Svelte	66.8 [28]	494.002 [29]	10 days ago
React	206+109 [30] [31]	18.389.003 [32]	2 days ago

Table 2.2: Weekly downloads with GitHub stars of frontend frameworks, 9-april-2023

2.5.3 Styling

CSS is responsible for the styling of a website and can be an extensive job. A framework can ease and speed up this development experience. To do this, frameworks with the most GitHub stars are analyzed (table 2.3).

Framework	GitHub stars (K)	NPM weekly downloads	Latest update
Pure.css	22.9	33.916 [33]	4 years ago
Bootstrap	163	4.568.632 [34]	6 days ago
Bulma	47	206.263 [35]	1 year ago
Foundation	29.4	N/A	9 months ago
TailwindCSS	67	5.529.998 [36]	2 days ago
Materialize	38.8	N/A	4 years ago
Normalize.css	50.1	1.086.135 [37]	4 years ago
Semantic UI	50.5	6.342 [38]	6 months ago

Table 2.3: Weekly downloads with GitHub stars of various CSS frameworks, 9-april-2023 [39]

Normalize.css

Although Normalize.css sees a lot of downloads, it is not really a framework to be used for styling, but to reset the browser's default styling. All browsers ship by default with a bit of their own styling. Normalize.css resets this to a consistent default for all browsers.

Bootstrap

Bootstrap is modular, making it good to create consistent pages. As it is so widely used, it comes with a lot of community support that has been gathered over the years. A downside is that it is not that flexible to use.

Bulma

Bulma is much like Bootstrap, but has more flexibility. It's grid system is also based on flexbox, which is more powerful than Bootstrap's grid system.

TailwindCSS

TailwindCSS is a minimal class-based wrapper around CSS. It gives the developer a lot of freedom in the usage of CSS whilst still being consistent across a page. It can also compile smaller than Bootstrap or Bulma. Making it faster to download for users.

2.5.4 Conclusion

Section 2.5.1 has shown that most backend frameworks can be used to achieve the same, but they are not all equally fast. It showed that ASP.NET was the fastest among 2 benchmarks and was followed by Express.js. Regarding the styling, TailwindCSS is a popular package (table 2.3) that gives a lot of freedom in its usage.

2.6 State of the art

2.6.1 Current applications

When googling for ”indoor asset tracking”, the first page immediately fills up with companies and advertisements offering this service. This section will take a look at a few of them and analyse how they do the tracking and visualization.

Inpixon and INTRANAV

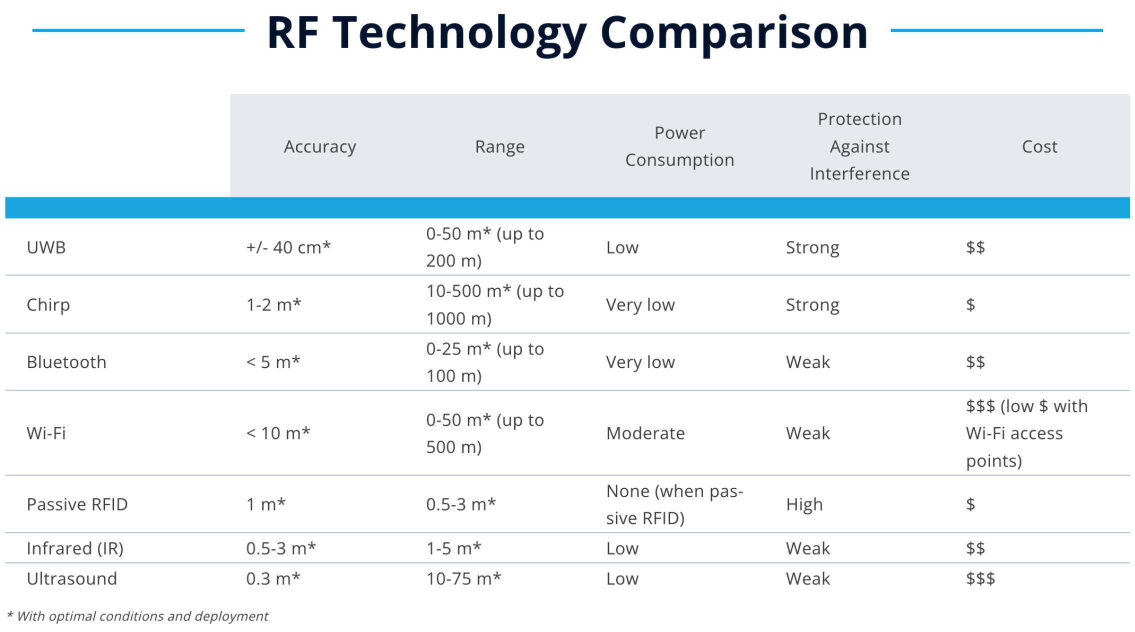
Inpixon offers indoor and outdoor asset tracking. Indoor, they can track assets via their tags or via mobile devices and phones. They use a so-called ”fixed anchor structure”, meaning that they have a set of receivers in fixed locations that scan for devices. These anchors can also actuate the trackers by sending out commands or firmware updates. Inpixon uses various technologies based on the environment of their customers, as each technology has its own dis- and advantages (figure 2.8) [40].

They also describe 5 localization techniques, getting a location from a signal. They mention that RSSI (figure E.4) is the lowest cost option but also tends to be easily affected by interference and thereof is not the most accurate. [40]

Angle of Arrival (AoA) (figure E.6) is recently developed technique that uses the physical angle at which packets are received. The advantage of this that it is more accurate than RSSI and needs less reference points. [40]

Time-Difference of Arrival (TDoA) (figure E.3) is used with Chirp or Ultra Wide Band (UWB) tags. This technique looks at the time differences of the packet arrival between different anchors. A challenge with this technique is that the clocks need to be synchronized in all anchors. [40]

Two Way Ranging (TWR) (figure E.5) is similar as TDoA, but does not use a fixed infrastructure. With TWR, devices communicate to each other and leverage the distances between them. [40]



RF Technology Comparison

	Accuracy	Range	Power Consumption	Protection Against Interference	Cost
UWB	+/- 40 cm*	0-50 m* (up to 200 m)	Low	Strong	$$
Chirp	1-2 m*	10-500 m* (up to 1000 m)	Very low	Strong	$
Bluetooth	< 5 m*	0-25 m* (up to 100 m)	Very low	Weak	$$
Wi-Fi	< 10 m*	0-50 m* (up to 500 m)	Moderate	Weak	$$$ (low $ with Wi-Fi access points)
Passive RFID	1 m*	0.5-3 m*	None (when passive RFID)	High	$
Infrared (IR)	0.5-3 m*	1-5 m*	Low	Weak	$$
Ultrasound	0.3 m*	10-75 m*	Low	Weak	$$$

** With optimal conditions and deployment*

Figure 2.8: Inpixon's comparison of Radio frequency (RF) technologies [40]

Inpixon's visualization is handled by INTRANAV, owned by Inpixon. They have a dashboard where you can see the overview and everything of the system. From live views to heat maps and previous locations (figure 2.9). They also have virtual zones which you can create. Those virtual zones can then be used to trigger actions upon entering or leaving. It also logs sensory information on from a tracker such as temperature and battery percentage. [41]

Even though they do not specify which software they use, there was a small overview in a demo video which showed that everything is containerized through Docker. By analyzing a few container names it is visible that they use Apache Kafka quite a lot as there are containers running named ”intranav-docker_kafka-mqtt-connector_1”, ”intranav-docker_kafkdrop_1” and ”intranav-docker_kafka-1_1”. Apart from that it does not say too much as most other names are generic. Meaning that it is entirely custom or generic (such as ...db_1, db is often referred to with database).

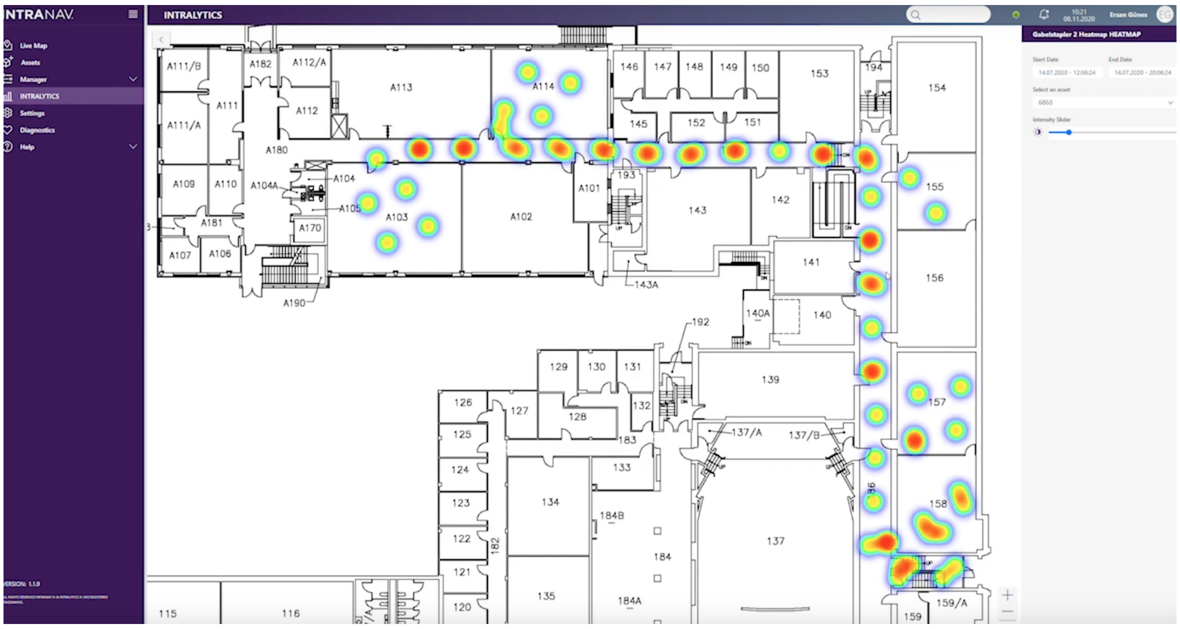


Figure 2.9: INTRANAV’s dashboard with a heatmap [41]

Ubisense

Ubisense aligns with the same technologies as in figure 2.8 from Inpixon. They do seem to favor UWB as it is present in all of their shared technology. This because the UWB signals are shorter and therefore do not overlap with reflections of the signal (figures E.2 and E.1), reducing interference. They mention using AoA and TDoA for localization (figures E.6 and E.3). [42]

Although they do not show pictures of their dashboard. They focus more on inventory management with the help of asset localization. For example, a worker gets a notification that product x needs to go to location y. The worker then searches for an available forklift on his phone and sees the location of the product. Reducing time spent looking for one.

Nextome

Nextome's technology can not be discussed in depth due to a restriction put on the informative paper explaining the operation of the system. Something that can be shown is that they solely use BLE, which is present in every smartphone nowadays. They utilize this as they allow smartphones to be the trackers. With this, users can see where they are on a map and follow live directions to their destination. They also have small BLE wristbands that could be put on patients for example. [43]

Conclusion

It is visible that there is already a diverse application of asset tracking applied in practice. Using various localization techniques and methods, with UWB and BLE being the most popular technologies. Although it was hard to get an insight into technical details as this was often left out. It was possible to get a sneak peek into Inpixon's services used.

2.6.2 Current frameworks

However, it is also important to consider current frameworks that can be used. This section will therefore look into current database and indoor mapping tools. The web frameworks have already been discussed in section 2.5.

Databases

The concept of TSDB was discussed in 2.3. There are 4 main players in this field; InfluxDB, Prometheus, QuestDB and TimescaleDB. InfluxDB and Prometheus have a focus towards monitoring and alerting, while QuestDB and TimescaleDB focus more on data aggregation. InfluxDB is NoSQL, meaning that it has its own querying language. Prometheus also has its own querying language called PromQL. On the other hand, TimescaleDB and QuestDB both use SQL and are built from PostgreSQL. TimescaleDB allows direct querying while the other 3 also have an API available through which data can be requested.

Indoor mapping tools

Mapping tools are tools that can be used to draw maps on a website and interact with them. There are quite a few that allow regular mapping (of streets and such) such as Mapbox, OpenStreetMap, Azure Maps, Google Maps and MapKitJS from Apple. All of them can be integrated into websites through available SDK's, although some have setup or usage fees. For indoor mapping there are different packages available such as MapsPeople, Situm, Visioglobe and Azure Maps.

Azure Maps is from Microsoft and allows creating and render your own indoor maps. They have a web SDK available for easy integration with a website. They do not have any setup fees and have a free tier for up to 5000 maps each month. After this you enter the first tier which is ±4€ for up to 100.000 maps loads [44]. However, the Azure Maps Creator (the part that allows you to upload custom maps) is paid at a price of 0.39€/hour, or ±280€/month [44].

MapsPeople is also available for the web and extends Google Maps or MapBox to indoor environments. They also provide indoor wayfinding. Using this is not as easy as Azure Maps, as this is on request.

Visioglobe also offers a web SDK and can be used to create custom indoor maps. However, they also have a feature called "Asset tracking", this is a mapping solution that facilitates a dashboard for data generated from asset tracking. This solution does not come with localization, but is solely the visualization of it. This is also only available on request.

Situm offers a web SDK as well and is also available on request. They provide indoor mapping, indoor navigation and a tracking visualization as well. This tracking solution shows the live location along with other data, and is integrable with any localization system. It also features a geo management system, which gives tasks to staff as they move around.

All in all, they all feature an ability to create indoor maps and allow them to be used through a web SDK. Although MapsPeople, Situm and Visioglobe are on request, Azure Maps can be used by anyone.

Chapter 3

Methods and techniques

The design process created by Mader & Eggink [45] tailored for the study Creative Technology is applicable to this project. This is a design process that consists of 4 phases; Ideation, Specification, Realisation and Evaluation.

Ideation

It all starts with the ideation phase, this starts with a first preliminary idea, problem or creative inspiration. In this phase background research is performed along with research into existing solutions to build upon and inspire from. This is also where the first requirements are defined and the context of the problem is clearly defined. This results in some preliminary ideas for the defined context of the problem.

Specification

This phase is about exploring the design space and further refinement of the current concept. The prototypes resulting from the Ideation phase are evaluated, merged and improved. The earlier defined requirements can also be more specifically set and better fine-tuned, possibly resulting in new ideas.

Realisation

The Realisation phase is about the implementation of the initial, refined concept according to its specifications.

Evaluation

The final phase is the Evaluation. In this phase, the prototype is (functionally) evaluated to see if it fulfills the requirements which were specified in the Ideation and Specification phase. Lastly, there is a personal reflection on the prototype and the process.

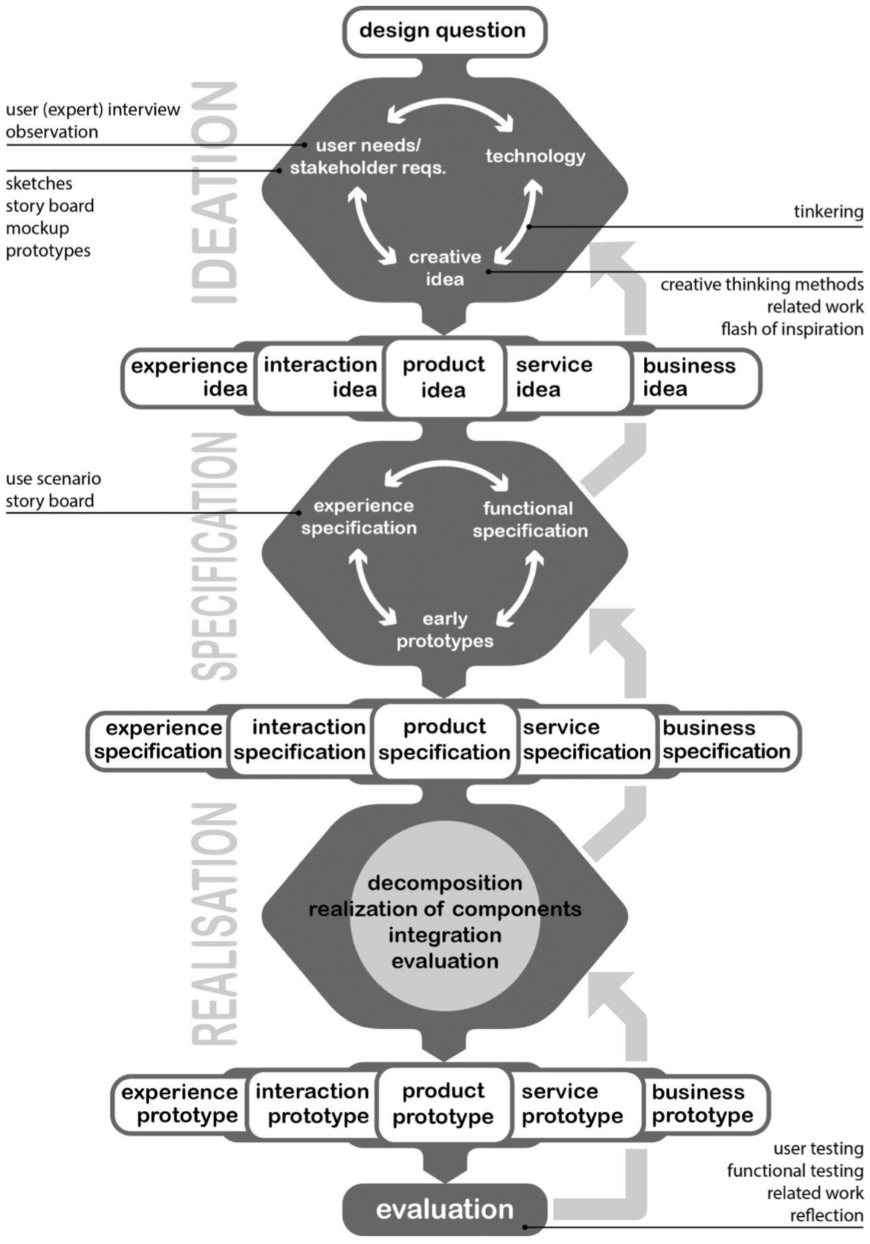


Figure 3.1: Creative Technology design process [45]

Chapter 4

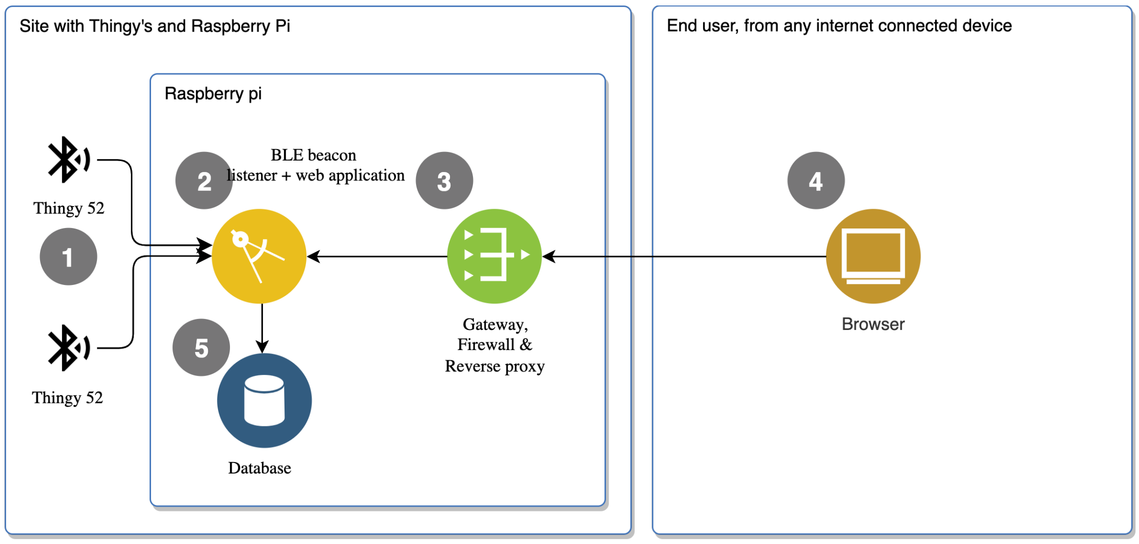
Conceptualization

The background research from chapter 2 has been transformed into an idea which has evolved through multiple iterations to attempt to address visible flaws in such a concept.

4.1 Preliminary concept

First concept

The very first concept was to use Laravel with Vue and TailwindCSS as I am familiar with those (which speed up development a lot), they also have a good integration with each other (figure 4.1). The idea would then be that this is all deployed on a Raspberry Pi and that events would be emitted inside Laravel when Bluetooth beacons arrived. However, this concept was quickly revised as PHP (which Laravel is written in) does not support Bluetooth. Secondly, a web framework was chosen to make it easily accessible, but that is not the easiest to do locally on a Raspberry Pi. This because you would have to go through the trouble of exposing it (by port forwarding it e.g.), and there might also be concerns about data security and reliability.

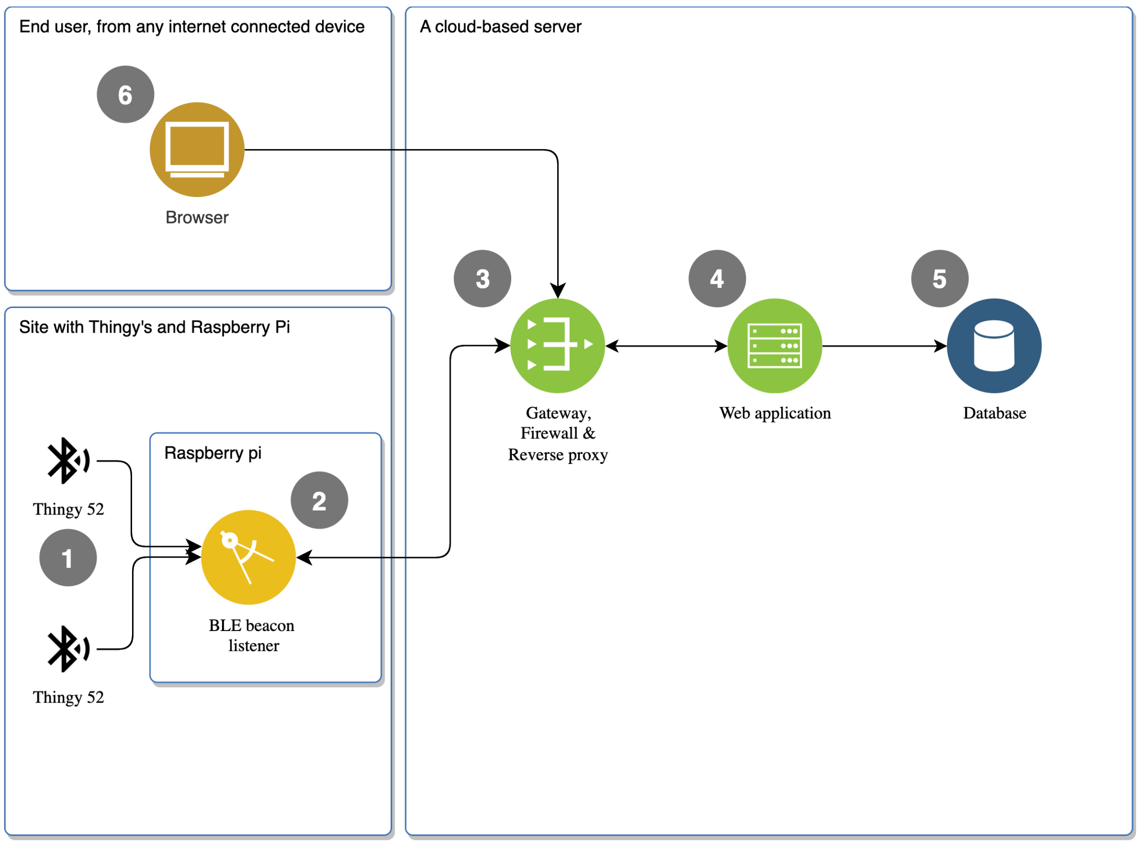

Site with Thingy's and Raspberry Pi
Raspberry pi
Thingy 52
1
Thingy 52
2
BLE beacon listener + web application
3
5
Database
Gateway, Firewall & Reverse proxy
End user, from any internet connected device
4
Browser

Number	Frameworks/software used	Functionality
1	NRF SDK	Thingy 52's that emit Bluetooth beacons
2	Laravel	Backend of the website that also interacts with the beacons
3	N/a	Repsonsible for the (web)routing
4	Vue + TailwindCSS	Frontend of the website that is executed in the browser of the user
5	MySQL	Relational database for the backend of the website

Figure 4.1: Initial concept architecture

Revised concept

This version started its core by reducing the functionality of the Raspberry Pi, which was transferred to be handled in the cloud (figure 4.2). The main advantage of this is that the cloud has a high reliability if it comes to uptime and data retention. It also features a good internet connection and there are no storage limits either. Because of this split, there would need to be a separate application running on the Raspberry Pi. One that listens for the Bluetooth beacons and then transmits them to the website to save them in the database. As seen in chapter 2.5, Laravel is not the fastest framework out there. Node.js was chosen for the backend instead for its speed, familiarity and available packages through NPM. A great addition would be its tight integration with the Bluetooth application that would run on the Raspberry Pi. If both applications are in Node.js (which is capable of Bluetooth), the exact same packages can be used which ensures good compatibility and ease of maintenance. With this, it is also still possible to implement a frontend framework which will then run in SPA mode and use the Node.js application in the cloud as its API. Both applications however, still need to communicate with each other. The application on the Raspberry Pi needs to upload data, and the web application would possibly like to send commands to the Raspberry Pi. As exposing a Raspberry Pi to the internet is not the easiest solution, an API running on the Raspberry Pi is not an option. Instead, the Raspberry Pi could either continuously poll the web application for updates or it could use a full-duplex communication channel, like web sockets. With polling, the server still has to reply to each poll, regardless of there being a command waiting or not. It also might induce a delay between the initiation and execution of a command, based on the polling interval. Web sockets on the other hand are full-duplex, meaning that it can both be used to upload the data and to receive commands instantaneously. This is also another great feature to implement with Node.js as you can use the same library in both applications.


End user, from any internet connected device
6
Browser
A cloud-based server
3
4
5
Gateway, Firewall & Reverse proxy
Web application
Database
Site with Thingy's and Raspberry Pi
Raspberry pi
2
Thingy 52
1
BLE beacon listener
Thingy 52

Number	Frameworks/software used	Functionality
1	NRF SDK	Thingy 52's that emit Bluetooth beacons
2	Node.js	Application that listens for the emitted beacons and uploads them via a websocket
3	N/a	Repsonsible for the (web)routing
4	Node.js + express.js	Backend of the website
5	MySQL	Relational database for the backend of the website
6	Vue + TailwindCSS	Frontend of the website that is executed in the browser of a user

Figure 4.2: Revised concept architecture

Proposed concept

Although the revised concept addresses a few issues, it still has a few that can be addressed. In the current situation, the data is sent through a web socket from the Raspberry Pi to the web application which will then store it in a relational database (MySQL, PostgreSQL etc.). There are two main issues with this design. The first one is that all data now flows through the web application, which will do nothing with it except for storing it. Inducing a constant load on it while it also needs to serve its API for the website. Another thing is that the generated data is timestamped and can grow vastly depending on the beacon intervals and number of Thingy 52's. If there are 30 Thingy 52's emitting at an interval of 5 seconds. There would be 24 * 60 * 12 * 30 = 518.400 new entries to the database per day.

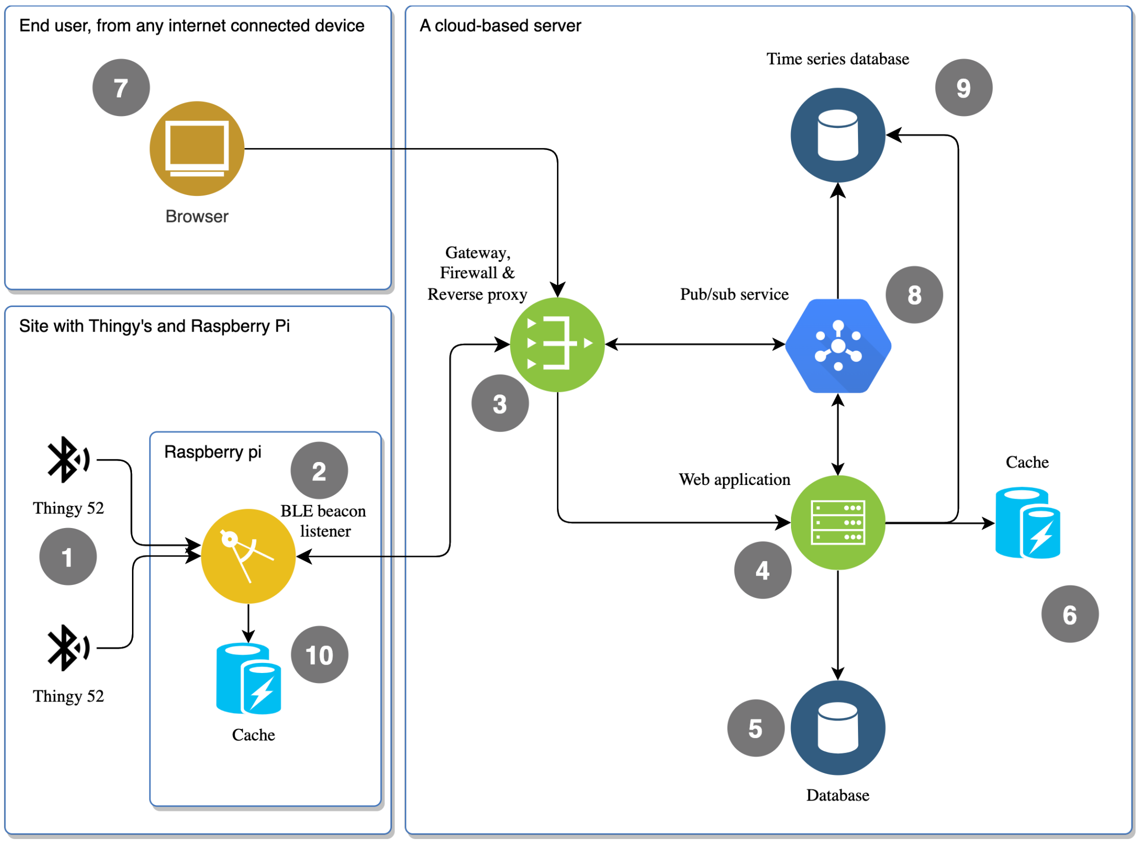
This is where the main change takes place, instead of the data flowing through the web application it will send it directly to the database (figure 4.3). Entirely omitting the web application which initially was a proxy, it only needs to read data.

The database would also be a TSDB, this type of database is purposely built to store time-series data. Which these beacons are.

The communication scheme also received a revision where the websockets have been replaced with Apache Kafka. Apache Kafka works with the publish and subscribe model. Meaning that you can send data to it, so-called producers, and subscribe to ingested data, the consumers. Apache Kafka is an application that can have a high throughput, is easily scalable and used by 80% of the Fortune 100 [46]. Another great feature is that it ”buffers” data, which is great in our case as the TSDB can ingest it at its own pace, reducing packet losses.

Once a user visits a web application, it is likely that it needs data from the TSDB. So the TSDB is queried and responds with a dataset. However, it could be that the user reloads the page or that another user needs to have the same dataset. This would result in the continuous querying of the TSDB, giving nearly identical responses each time. This is where a cache could serve well, once a query has been made, the response is cached for a few minutes. If another similar request is made, it can be served from a quick cache. Resulting in faster response times and a lower load on the TSDB. A downside is that the data will not be updated until the cache expires.

A similar cache has been added to the Bluetooth application on the Raspberry Pi as well to deal with internet interruptions. In the event that the internet connection is lost, beacons are cached instead and later uploaded when the internet comes back online, reducing the possibility of data loss.



Number	Frameworks/software used	Functionality
1	NRF SDK	Thingy 52's that emit Bluetooth beacons
2	Node.js	Application that listens for the emitted beacons and uploads them
3	N/a	Repsonsible for the (web)routing
4	Node.js + express.js	Backend of the website
5	MySQL	Relational database for the backend of the website
6	Redis	Cache to cache queries towards the TSDB
7	Vue + TailwindCSS	Frontend of the website that is executed in the browser of a user
8	Apache Kafka	Pub/sub service that ingests all data from the Raspberry pi's
9	InfluxDB	Time series database (TSDB) that stores all the data from the Thingy's
10	Redis	Upload cache to prevent data loss on a network loss

Figure 4.3: Proposed concept architecture

4.2 Dashboard concept

Initial layout concept

Figures 4.4, 4.5 and 4.6 are the initial designs for the dashboard. This dashboard allows the user to whitelist and configure Thingy's. The main screen (figure 4.4) contains a map with the live location where Thingy's can be clicked to inspect them by name and see some generic information about them. On the left-hand site are the battery percentages, temperature stream and the packet transmission rate of the Thingy's.

The layout page (figure 4.5) allows Thingy's to be named, configured and added. It also allows to identify thingy's by clicking them. Then a command will be sent to a Thingy to light it up.

The last page (figure 4.4) has the option to show a heatmap or to show the previous trails. There is also a slider to select a time range. In the big box on the left specific Thingy's can be selected to show a smaller selection of Thingy's.

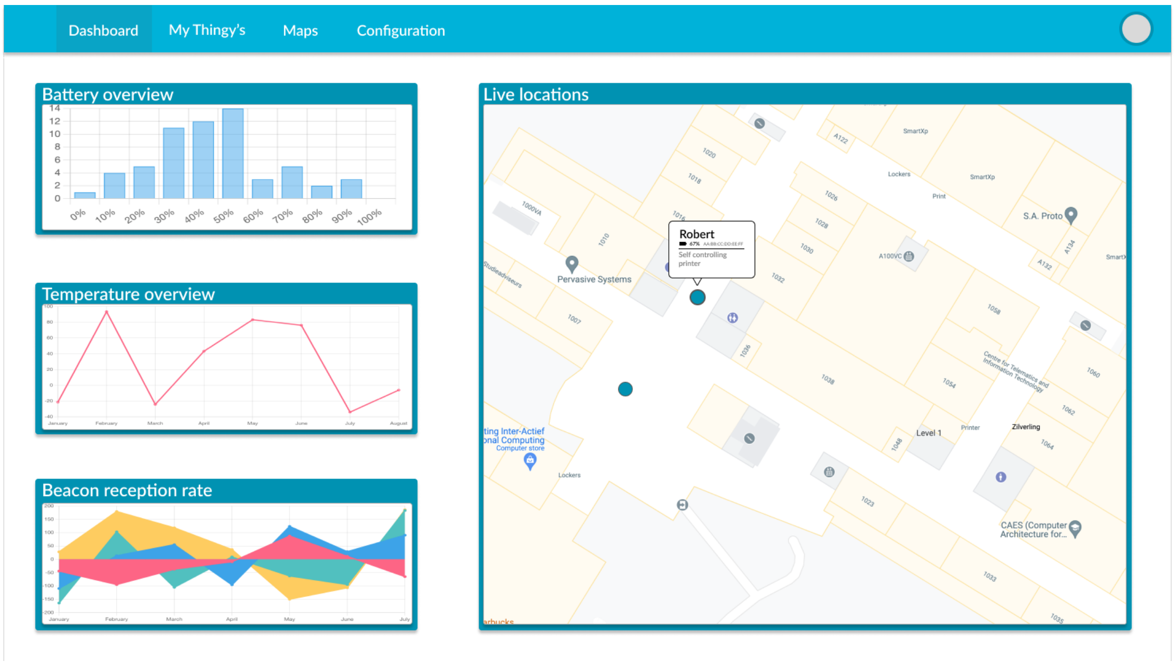


Figure 4.4: Dashboard homepage concept where the live location is available.

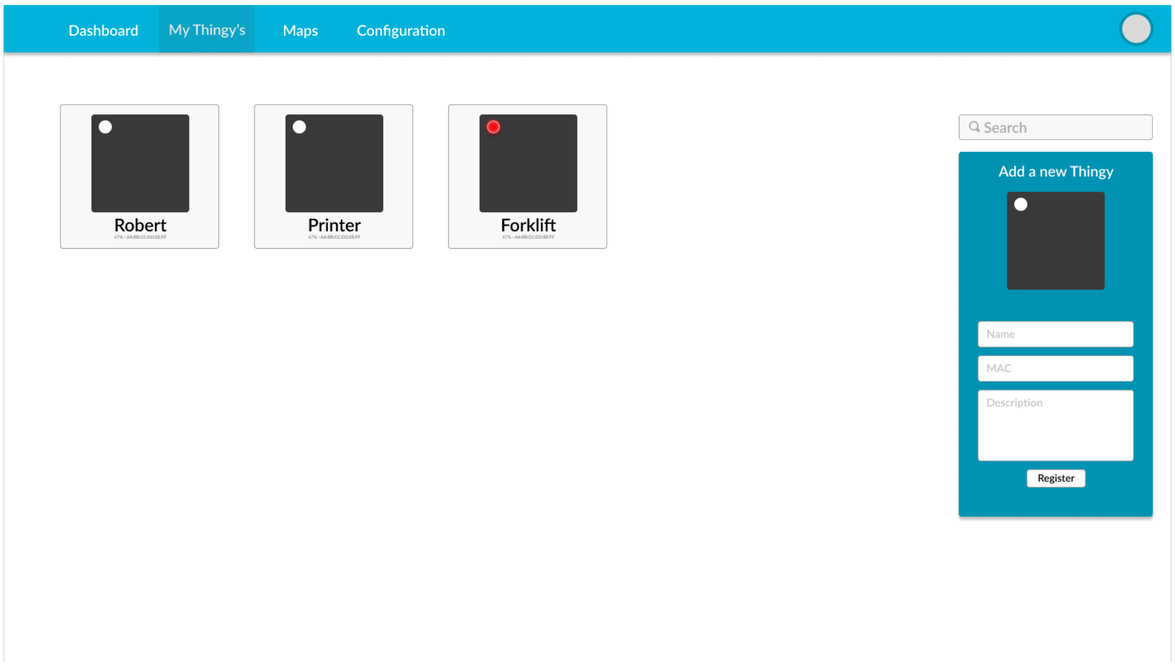


Figure 4.5: Dashboard concept of the whitelisted Thingy 52's, here Thingy's can be added and configured.

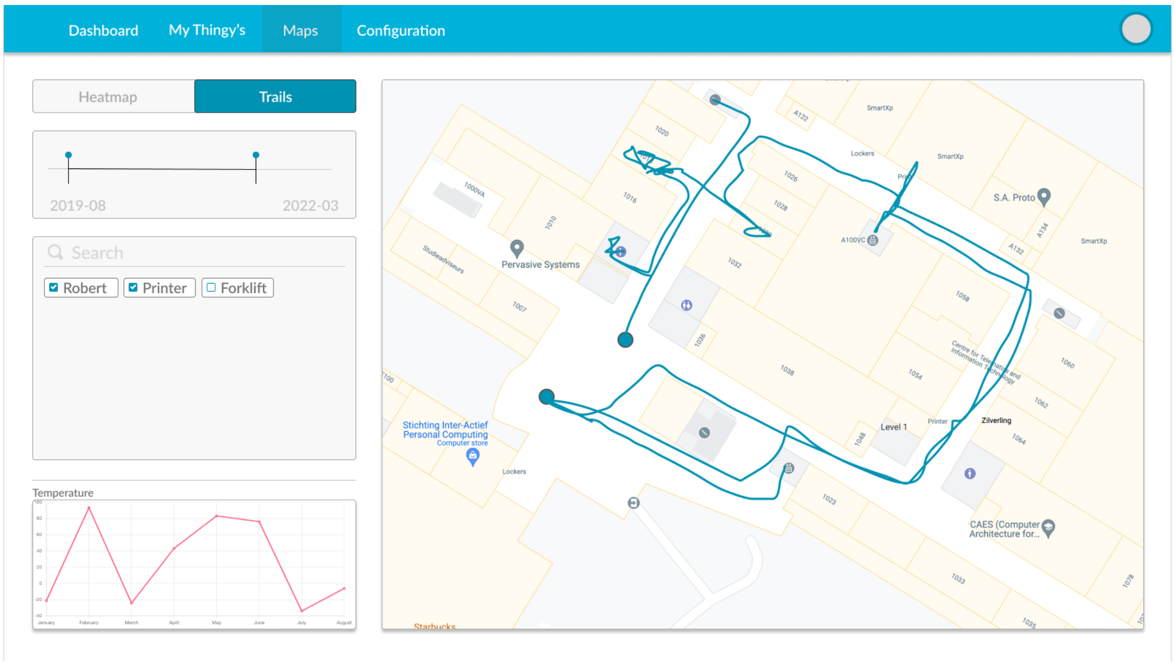


Figure 4.6: Dashboard concept to show previous locations of trackers via trails or a heatmap. It also allows filtering by tracker and time range.

Revised layout concept

Although the initial concept shows a lot of information, it might be a bit too much for a single page. As shown in figure 4.4, there are 3 graphs available with a live map. By removing those graphs, there is more space available on the dashboard for the map. Allowing a larger map to be displayed for live locations. These graphs have therefore been moved to a new, dedicated page where the Thingies can be inspected individually. As of this the navigation bar layout has also been restyled to allow the map also to grow a bit vertically. As a result of this foundational change in the layout, the overview page (figure 4.5) and historical data page (figure 4.6) have also been restyled to adapt to the new layout and style.

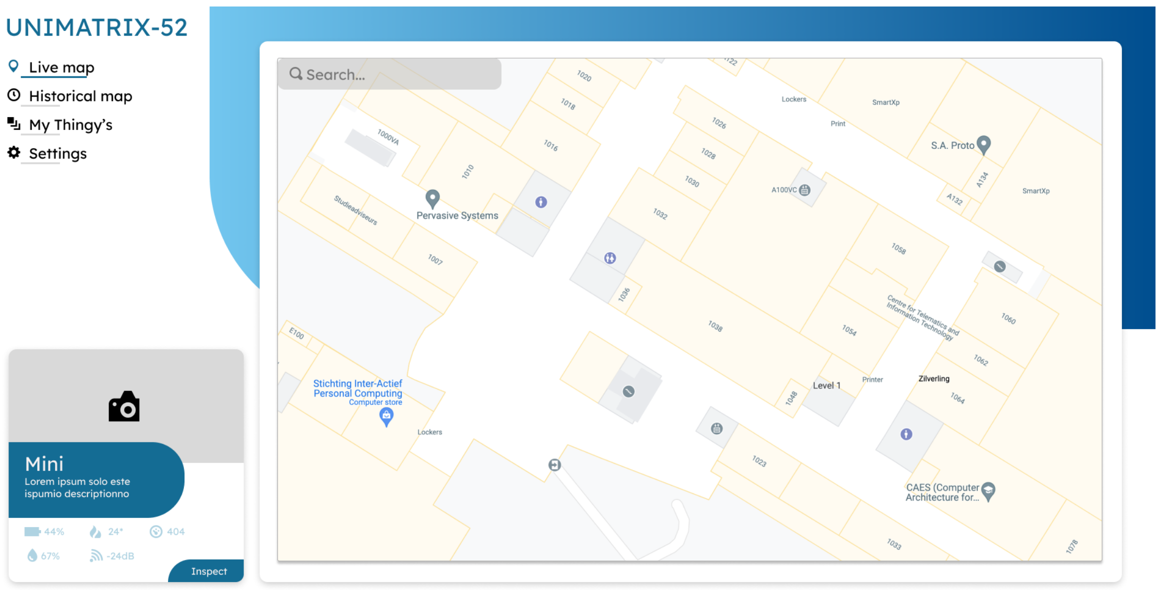

UNIMATRIX-52
Live map
Historical map
My Thingy's
Settings
Search...
Pervasive Systems
Stichting Inter-Actief Personal Computing
S.A. Proto
CAES (Computer Architecture for...
Mini
Lorem ipsum solo este ispumio descriptionno
44%
24°
404
67%
-24dB
Inspect

Figure 4.7: Revised dashboard homepage concept where the live location is available.

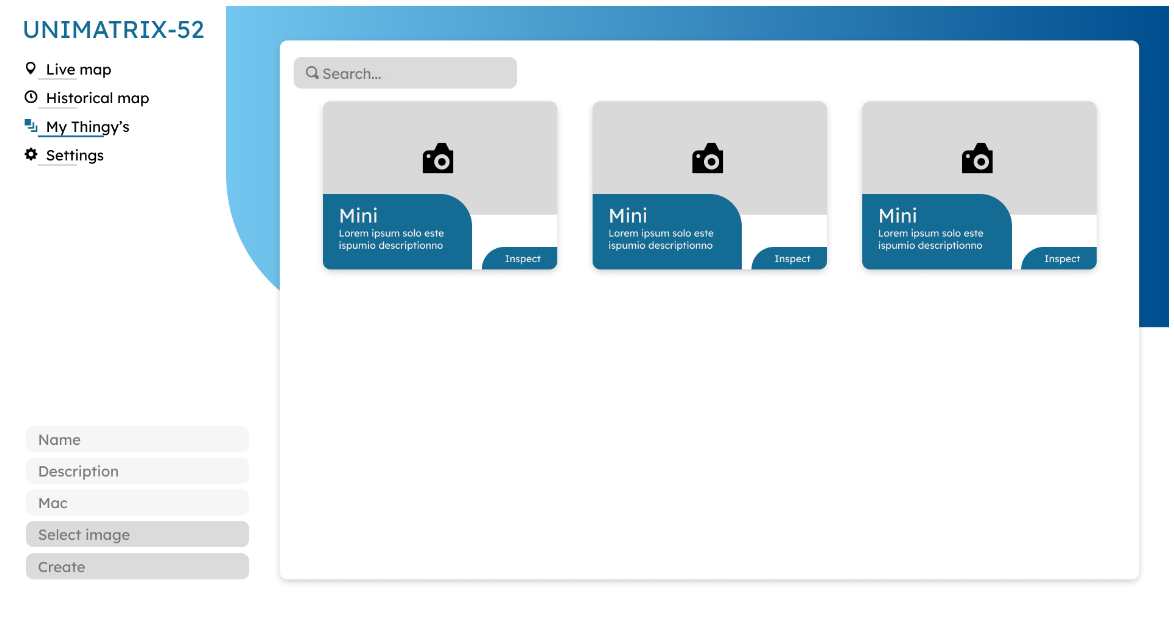
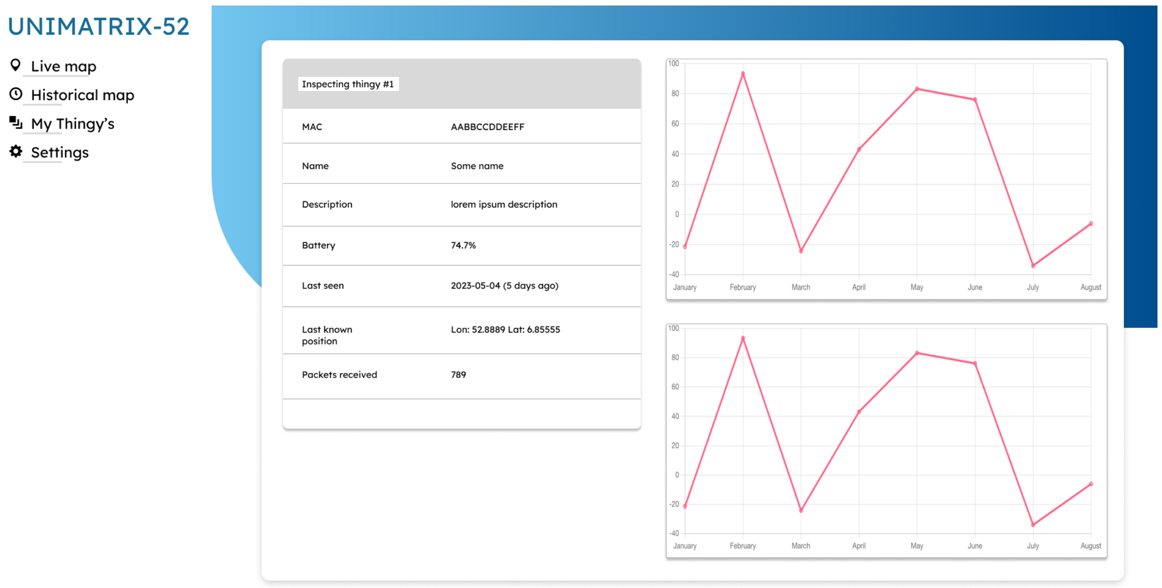


Figure 4.8: Revised thingy overview page where they can be added and searched for.



Inspecting thingy #1

MAC	AABBCCDDEEFF
Name	Some name
Description	lorem ipsum description
Battery	74.7%
Last seen	2023-05-04 (5 days ago)
Last known position	Lon: 52.8889 Lat: 6.85555
Packets received	789

Figure 4.9: Revised Thingy inspection page where a Thingy can be analyzed individually.

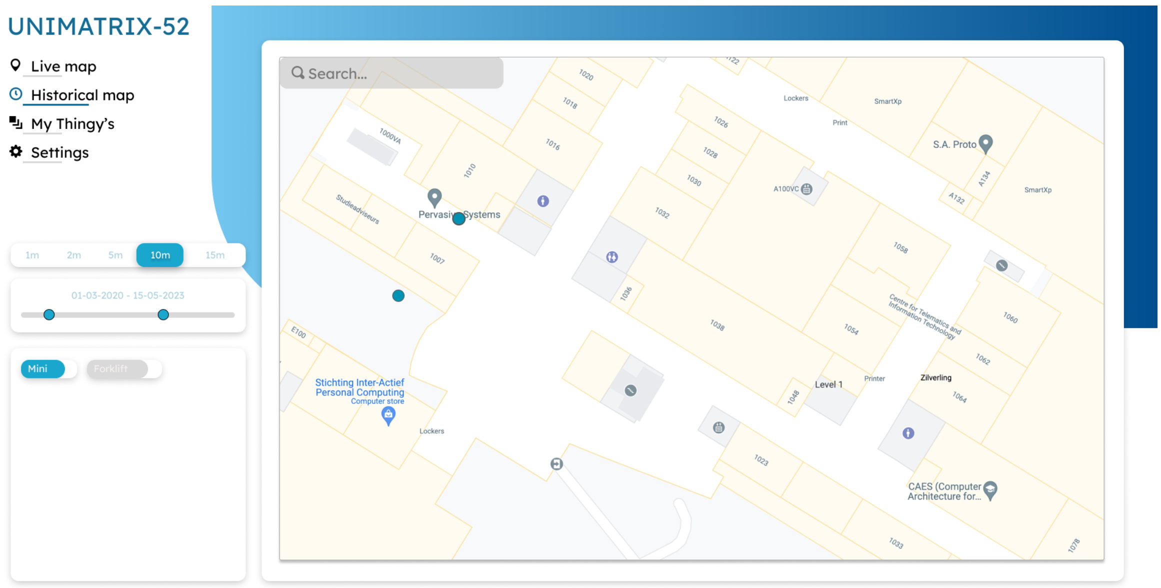


Figure 4.10: Revised historical page where all thingies can be analyzed on the map.

4.3 Stakeholders

There have been identified 3 stakeholders for this project. The identified stakeholder have been placed in an analysis matrix, which is shown in figure 4.11.

Developers

At some point this project might be continued by other developers or needs to be integrated with the localization. Although those developers do not have a direct impact on the end product, they should be minded during the development.

Pervasive Systems Group

The Pervasive Systems Group of the University of Twente and has set certain requirements to the product. In the end, they will deploy the project and are therefore a major stakeholder in this project.

End users

These users need to directly interact with the dashboard and are thereof substantial stake-holders. However, they only interact with the dashboard and are not interested in how everything is managed.

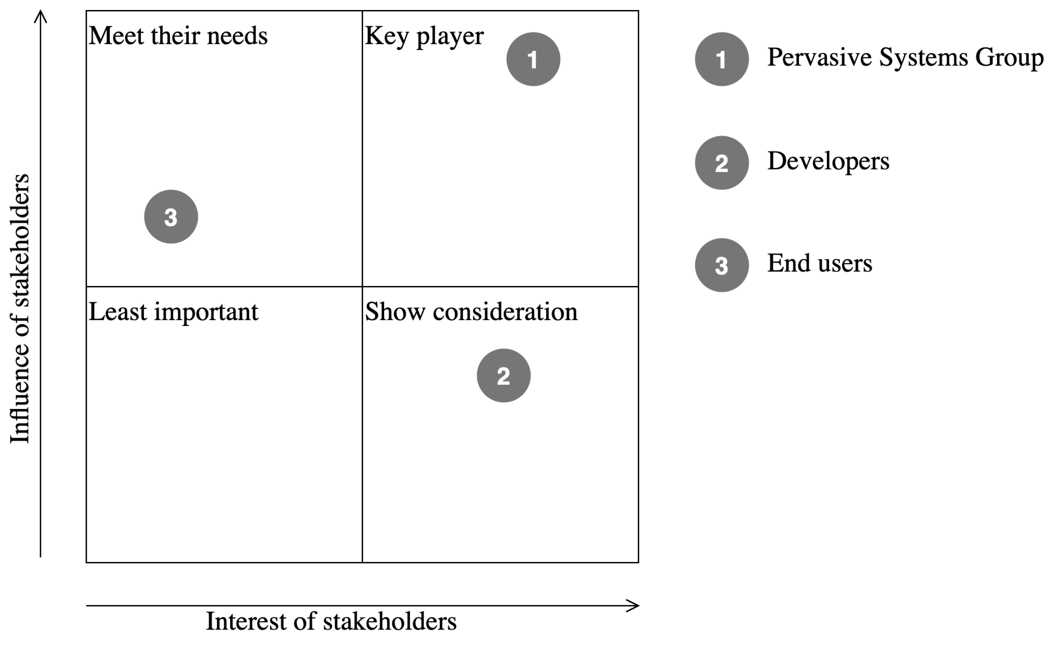


Figure 4.11: Stakeholders matrix

Chapter 5

Specification

5.1 Requirements

First, a list of requirements is set using the MoSCoW [47] method. This is a list of requirements that the product must have (Mo), those that the product cannot function without. Should have's (S), which should be in there, but the product could function without. Could have's (Co), which would be nice to have if time allows it. And won't have's (W) which the project will not have.

Must have:

- A dashboard that visualizes locations on a map.
- The dashboard must be accessible from anywhere.
- The system must be reliable.
- The dashboard must be responsive.
- The dashboard must have authentication.

Should have:

- The system should be scalable.
- The system should show a live location of an asset.
- The system should collect auxiliary data such as temperature.

Could have:

- The system could control the Thingy 52's from the dashboard (e.g. turn on a LED).
- The Thingy 52's could be able to update through the dashboard (FOTA).
- The system could send out notifications on specific events, such as a low battery.

Won't have:

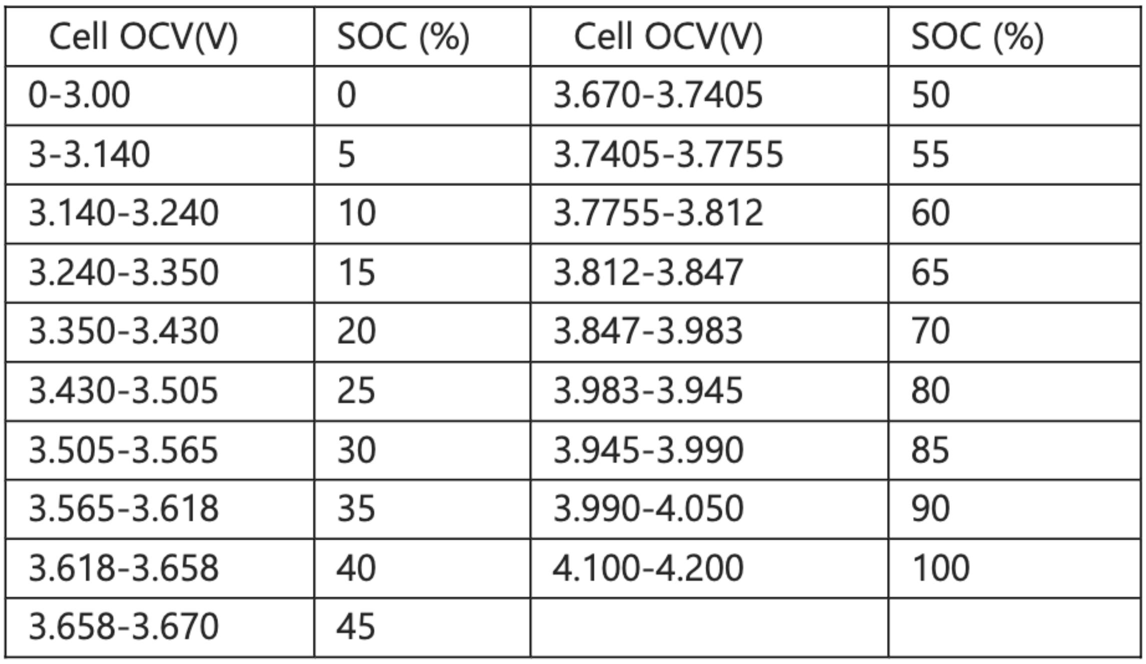
- The system won't have a functional localization system.

5.2 BLE packet transmission and reception

5.2.1 Sensor readout

The Thingy52 has 2 sensors (see 2.1.1 which need to be read out, the humidity- and temperature sensor and the gas sensor. Additionally, the battery percentage also needs to be known. However, this is not as straightforward as reading out the sensors as a battery voltage is read, which needs to be converted to a percentage.

The battery percentage, also known as State of charge (SOC), can be calculated from the percentage. To do this, the lower and upper voltages were measured by fully draining the Thingy 52 until it turned off and leaving it in the charger for over a day. This resulted in a 0% voltage of 3.220V and a 100% voltage of 4.210V.



Cell OCV(V)	SOC (%)	Cell OCV(V)	SOC (%)
0-3.00	0	3.670-3.7405	50
3-3.140	5	3.7405-3.7755	55
3.140-3.240	10	3.7755-3.812	60
3.240-3.350	15	3.812-3.847	65
3.350-3.430	20	3.847-3.983	70
3.430-3.505	25	3.983-3.945	80
3.505-3.565	30	3.945-3.990	85
3.565-3.618	35	3.990-4.050	90
3.618-3.658	40	4.100-4.200	100
3.658-3.670	45		

Figure 5.1: The battery's SOC and matching battery voltage [48].

Looking at the table graphed in figure 5.1, it seems that there is a mistake at 70% where the voltage ranges from 3.847-3.983 V. Which is a significantly higher voltage range than the surrounding entries in the table, secondly the 80% voltage starts higher than it ends at 3.983-3.945 V. Due to these two incongruities, the assumption has been made that 3.983 is a mistake and has been corrected to 3.883 V. Looking at the plotted table in figure 5.2, the read voltage cannot linearly represent a percentage. The trendline (in yellow) has the following formula:

$$\begin{aligned} Percentage = -178,332 * Voltage^4 + 2504,83 * Voltage^3 \\ + -13077,1 * Voltage^2 + 30155,2 * Voltage + -25957,3 \end{aligned} \tag{5.1}$$

This formula needs to be adjusted to the 0 and 100% values measured above. This results in a range of 9,3624% (see calculation B.1) until 100.9455036% (see calculation B.2). Which can be mapped via function 5.2.

$$Mapping = (1,0919045 * input) - 10,2228467 \tag{5.2}$$

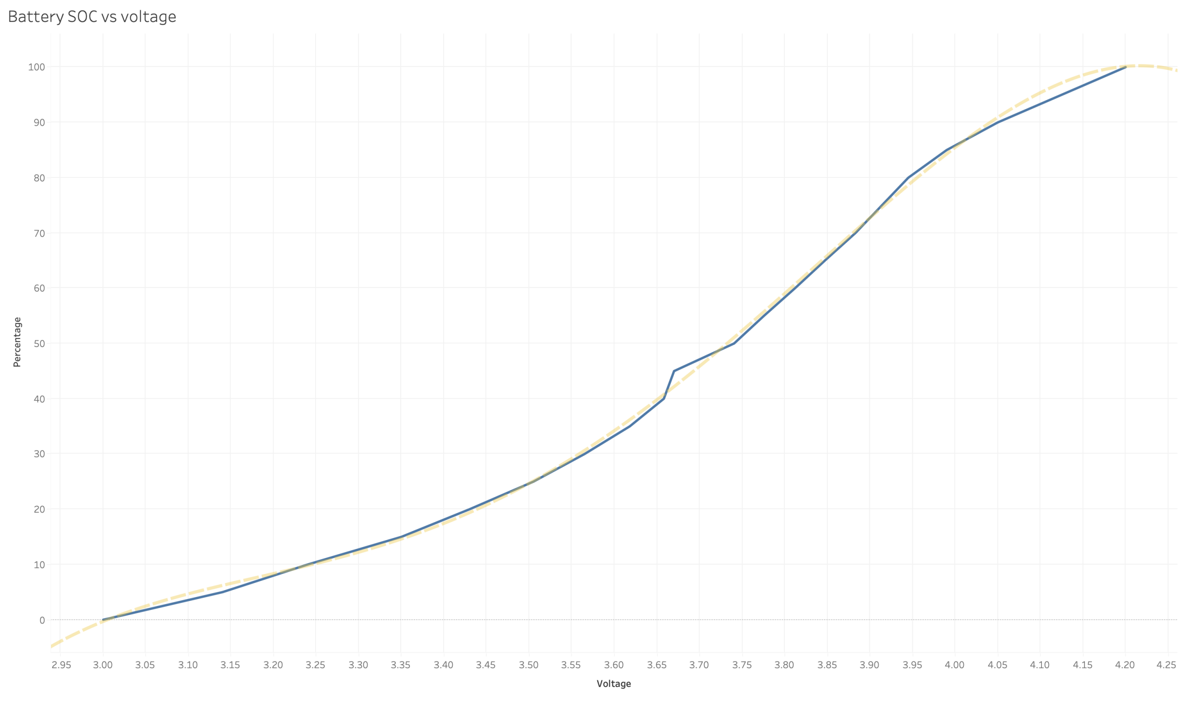


Figure 5.2: The battery's SOC and matching battery voltage graph from figure 5.1 (blue) and the trend line (yellow).

5.2.2 Packet creation

A BLE 4.0 packet has a limited space available to insert data. To better understand this, a list is created with the data which needs to be sent in the packet:

- An identifier (MAC address)
- A packet identifier
- Firmware version
- Humidity
- Temperature
- Battery percentage
- CO2 PPM (from the gas sensor)

MAC address

The MAC address is a great identifier as each Thingy 52 has a sticker with the MAC address, making it easy to identify them physically. It also has a designated field in advertisement packets, the Broadcast Address (see figure 2.1).

A packet identifier

This byte-sized integer increments upon each transmission of a packet. This byte makes it easy to identify if there are any duplicates during reading and whether any packets have been missed. It also eases data processing in Apache Kafka (see section 5.3). This identifier is also referred to as the rollover byte (as it will roll over from 255 to 0).

Firmware version

The packet includes a firmware version to identify how to parse the packet, as a packet arrives in the form of an array of bytes. This byte helps to identify how to parse these raw bytes to their original values. It could also be used for automatic updating, such that an update could be instantiated wirelessly if a packet is received with a specific firmware version.

Humidity, temperature and battery percentage

The humidity, temperature and battery percentage are all represented as a floating value of 4 bytes. Which would be great to store large numbers or have a great accuracy, however this is not needed for these values. The temperature will most likely range between 10-30 degrees Celsius, but for now -40 until 85 degrees is assumed as those are the minimum and maximum rated operating temperatures for the nRF52832 SoC (which is what the Thingy 52 uses) [49]. The battery and humidity are both values between 0-100%.

This is where the Short float (SFLOAT) comes in, it is an encoding standard as per IEEE 11073-20601-2008 [50] to encode floating values to a 16-bit value. The conversion from a 32-bit IEEE 754 float to SFLOAT is fully supported in the nRF Connect SDK 2.2.0. This 16-bit value uses a 4-bit exponent and a 12-bit mantissa. Its encoding, however, is not the same as that of the IEEE 754 standard. The exponent is a signed 2's complement value which represents the power. The mantissa is a 12-bit signed 2's complement representation of the original value. An example will illustrate this better. Let's assume the value of 39,1 which needs to be parsed into a SFLOAT. This value can be represented as $391 * 10^{-1}$. Which is exactly what will be encoded into a SFLOAT, where 391 will be the exponent and -1 the mantissa. As the exponent is 4-bit signed 2's complement, any value ranging from -8 until +7 is possible. The mantissa is 12-bits 2's complement value, which ranges from -2048 to 2047. So the SFLOAT can be used to represent the temperature, humidity, battery percentage with a maximum of 1 decimal.

CO2 PPM

The CO2 PPM represents how many CO2 particles per million particles in the air are present. This is on average 400 outdoor, depending on the environment it could go as high as 40000 [51]. This value contains no decimals and would be too small for a SFLOAT. It could be represented as an unsigned 16-bit integer which ranges from 0-65535, as the CO2 PPM cannot be negative and will not go past 65535.

The packet

Once all data has been encoded, a packet with a payload of 10 bytes is generated (see table 5.1).

Type of data	Bytes
Rollover	1
Firmware version	1
Humidity	2
Temperature	2
CO2 PPM	2
Battery percentage	2

Table 5.1: The packet construction with byte count

5.2.3 Packet readout

Two methods have been attempted to read out a BLE advertising packet on a Raspberry Pi running Raspbian OS.

Node.js/ Python packages

The first attempt used different packages such as 'node-ble', 'bluez', 'bleak' and 'noble' on Node.js. 'PyBluez' was tested using Python. All attempts had one thing in common, they would see the packets occasionally. So although it was possible to read out the data from the packets, none of them was able to capture every packet. The documentation of BlueZ (the official Linux Bluetooth stack), showed that it could be interfaced over the D-Bus of Linux (a communication bus to communicate between different programs) and that it had an option to disable duplicate advertisement detection [52]. However, this attempt was unsuccessful as well. After further investigation, it seemed that the Linux kernel itself filters duplicate advertisement packets [53].

nRF BLE Sniffer

For testing and development purposes, Nordic Semiconductor created a tool named the nRF BLE Sniffer. With this tool BLE advertising packets could be read out using Wireshark. It captures packets via all 3 advertising channels and has Bluetooth 5 support as well. This tool consists of special firmware which needs to be uploaded to a nRF 52 Development Kit which can then be connected via USB to any operating system. To read out this nRF 52 Development Kit they use a Python script and Python API which connects the data to Wireshark. However, this Python API is generic and ready to be used with any other Python code. Using this tool makes the reading of the BLE packets operating system independent and adds support for reading Bluetooth 5 packets.

5.3 The server

The software such as the TSDB, website backend (dashboard) and Apache Kafka will be hosted on a central server that is remotely accessible. The system's overview is given in figure 5.3. All the arrows represent the flow of data. The green arrows are HTTP requests.

5.3.1 Apache Kafka in short

Apache Kafka is the main connector between the data read on a Raspberry Pi and the TSDB. In Apache Kafka so-called topics can be created. A topic is a stream of events that have happened, similar to logs. These topics can be partitioned, which is splitting them across brokers. A cluster can contain various brokers, this to increase the throughput, reliability and scalability of a sytem. A topic is spread across multiple brokers via the partitions, brokers also keep backups of each other in the case one broker fails. A message (an entry) to a topic contains a key and a value, the key is used to partition a message to a broker. Using this it is ensured that all messages with the same key end up in the same partition of a topic. This is great to maintain the order of a given message type.

5.3.2 The architecture

In figure 5.3 the most important parts of the system architecture are shown. The section below explains all elements visible in the figure.

1. nRF BLE Sniffer

This is the nRF 52 Development Kit that runs the nRF BLE Sniffer firmware from Nordic Semiconductor and is connected via USB to the Raspberry Pi. This part listens for the BLE advertising packets and sends them to no. 2.

2. Python BLE sniffing application

This is a Python application that implements the nRF BLE Sniffer API. It reads the received BLE packets from no. 1, does some basic validation whether the packet could have been sent from a Thingy 52. It looks at the packet length and if the Manufacturer ID has been set to a specific value. If a packet passes the validation, it is pushed into the Redis Stream at no. 3.

3. Redis

This is a Redis instance which is deployed on the Raspberry Pi. It serves as a queue and buffer for the received packets by no. 4.

4. Location consumer

This Node.js application reads entries from the Redis Stream (no. 3) and sends them to the 'raw_locations' topic in the Apache Kafka cluster (no. 17) if a packet's MAC address is present in the whitelist. It only sends the timestamp, rollover byte, MAC address and a longitude and latitude. This application initially fetches the whitelist from the backend (no. 11) and receives live changes of the whitelist via the 'whitelist' topic in the Apache Kafka cluster (no. 18).

5. Packet consumer

This application is for the most part similar in functionality to no. 4, except that it sends all other data that was present in the packet and not a location. It also sends it to a different topic, 'raw_beacons' (no. 16). This application, as the Location consumer (no. 4), is horizontally scalable. Meaning that multiple instances can be run parallel to each other while they do not upload duplicates. This could have the advantage that, in the event that one replica unexpectedly fails, that it can take over the packets of the crashed instance which it was processing.

6. The manager

This is a Node.js application and its function is to manage the Redis Stream (no. 3). It removes entries if they have been acknowledged by both consumers (no. 4 & 5) to free up space. A secondary function of this is to report the metrics of the Raspberry Pi and the packet processing. It looks at the number of incoming packets and if any of the 2 consumers (no. 4 & 5) are lagging behind, or not processing at all. Because of these reported metrics, it can be known if any of the consumers have failed (large lag), if the Python application has failed (no. 2) by looking at the incoming packets or if the entire Raspberry Pi or manager has failed by a lack of metrics at all.

7. Browser

This is any browser on a desktop, laptop or mobile phone that can access the webserver (no. 11) via the reverse proxy (no. 8).

8. Reverse proxy

The reverse proxy is responsible for creating HTTPS certificates and forwarding/loadbalancing an URL to a service. In this case it will forward all traffic towards the webserver (no. 11).

9. Redis

This Redis instance is used as a cache for QuestDB (no. 12) and the webserver (no. 11). The webserver uses this cache to temporarily store downsampled historical data from the QuestDB server (no. 12). By doing so, it reduces the load on the QuestDB server, simultaneously improving the response time to the browser.

10. MySQL database

This MySQL database will only be used by the webserver (no. 11) and store the username with hashed password for users. For the thingies it will store metadata such as name, MAC address and a description.

11. Webserver

The backend will be written in Node.js v18 (latest Long term support (LTS) at the time of writing). This webserver is the only application on the server that is exposed to the public internet and accessible via a reverse proxy (no. 9). This backend will serve the website and handle interactions with the website such as data requests and authentication. The backend can also send out whitelist commands

12. QuestDB

QuestDB is the TSDB which will store all received packets including a location. It receives the data from the Apache Kafka QuestDB sink connector (no. 13). The webserver (no. 11) accesses this TSDB via the PostgreSQL protocol.

13. QuestDB sink connector

A connector in Apache Kafka is a program that can be installed in an Apache Kafka cluster and can be considered similar to a plugin, it facilitates a pre-defined set of tasks. In this case it is a sink connector (it extracts data), it extracts data from the 'merged_location_beacon' topic (no. 14) and inserts that into the QuestDB TSDB (no. 12).

14. 'merged_location_beacon' topic

This Apache Kafka topic is the output topic from the ksqlDB JOIN query that is executed in no. 15. Messages on this topic are consumed by the QuestDB sink connector (no. 13).

15. ksqlDB location beacon join

The Apache Kafka cluster also runs a ksqlDB server. KSQL can be used to interact with topics to filter, join or detect anomalies. It is used similar to SQL, actions are defined via SQL-like queries. This is therefore used to merge the 'raw_locations' (no. 17) and 'raw_beacons' (no. 16) topic and output that to the 'merged_location_beacon' topic (no. 14). It does this by creating a key based on the MAC address and the rollover bytes. It then merges the 2 raw topics into one if the index matches and the timestamp of the messages is within 1 minute of each other. It also has a grace-period set to 5 minutes. Meaning that a packet with a matching MAC and rollover byte index can arrive within 5 minutes of each other in either of the raw packet topics (no. 16 & 17).

16. 'raw_beacons' topic

The beacons from the Packet consumer (no. 5) are pushed to this topic. It also has a schema defined via the schema-registry. A schema in the schema-registry server defines the format and types of data that needs to be present in a message. This schema-registry is used by all consumers and producers that interact with the Apache Kafka cluster. By doing so, it allows for consistent data entries (as invalid entries will not be pushed) and versioning of messages.

17. ’raw_locations’ topic

This topic is identical in functionality as the ’raw_beacons’ topic (no. 16) except that it does not store the packet data but the location.

18. ’whitelist’ topic

The whitelist topic contains of MAC addresses and a boolean stating it is whitelisted or not. The Packet- and location consumer (no. 4 & 5) consume from this topic. Meaning that they will receive instantaneous updates when the webserver (no. 11) makes any changes to the whitelist.

19. ksqlDB whitelist

Another function of ksqlDB is to create a so-called KTable. This is a table with all the latest values from messages pushed to the ’whitelist’ topic (no. 18). So a new whitelist event can be sent to the whitelist topic and this automatically updates current whitelist statuses to the table shown in no. 20.

20. Whitelist status KTable

This table contains all the latest whitelist commands that were pushed onto the ’whitelist’ topic (no. 18). It is queried by the webserver (no. 11)

21. ’pi_metric’ topic

This topic contains the metrics that are generated from the Manager (no. 6). KsqlDB can be used to detect anomalies on this topic (and therefore issues with the Raspberry Pi).

22. Thingy 52

These are the BLE beacons that send out beacons with information of their sensors. The rollover byte is increased with 1 on every transmission.

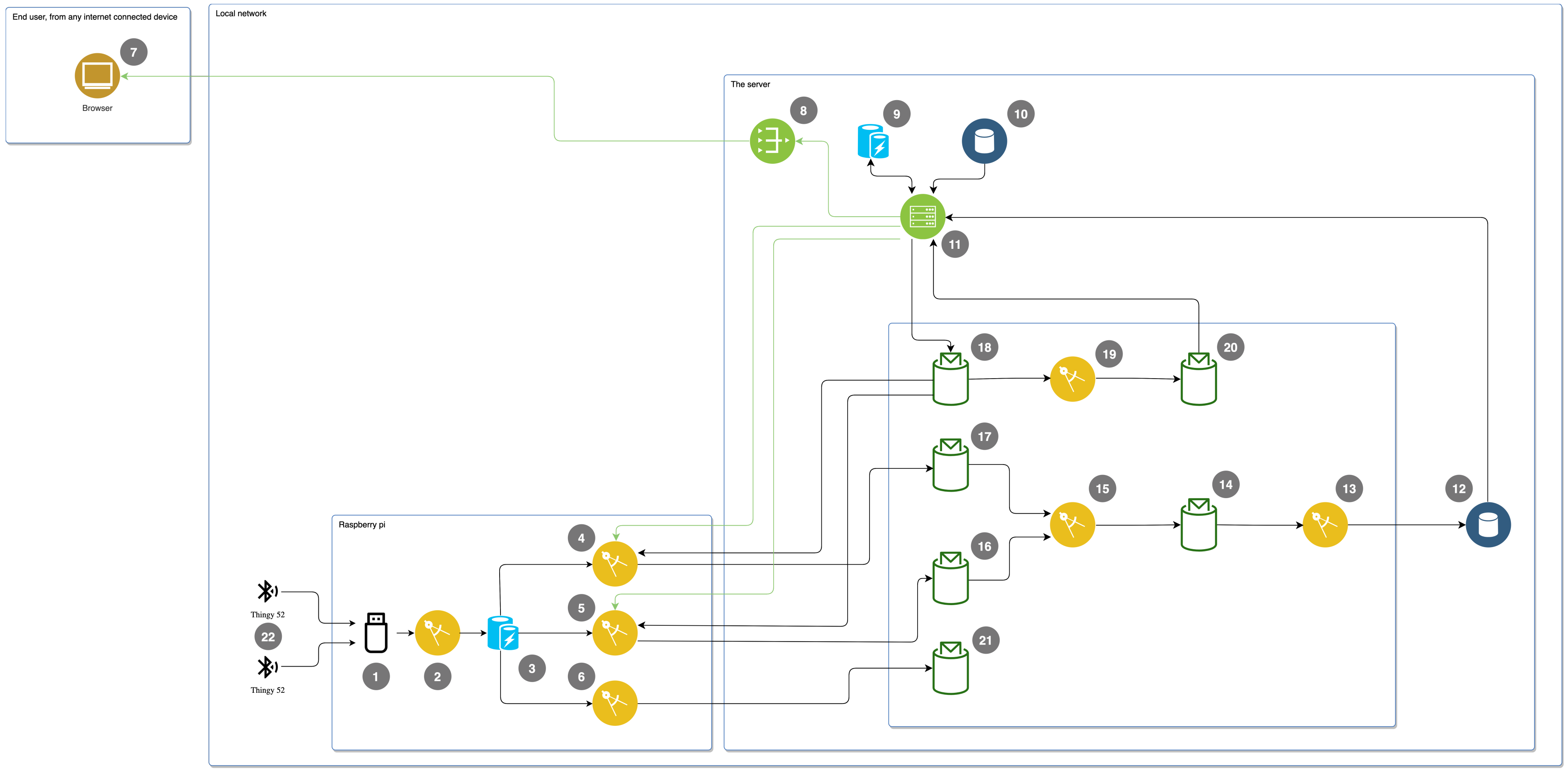


Figure 5.3: The system's architecture overview

5.3.3 Splitting the location and packet into two topics

The received BLE packets are read from the Redis Stream (no. 3) by 2 applications (no. 4 & 5). These are then pushed onto separate Kafka topics (no. 16 & 17) to only be joined back together by ksqlDB (no. 15). The main reason for this is to ease the integration with the localization part of the system. The localization is going to be a Machine Learning model which is going to be deployed on the Raspberry Pi. However, its parameters and requirements are currently not yet that well-defined. It could be that it needs to be written in a different language than Node.Js. Meaning that an entire application would need to be transferred to extend it with the Machine Learning model, meaning that a lot of work has to be re-done. By splitting this, it saves a lot of work in such an event. All a future application would have to do is to read from Redis and send messages to Apache Kafka without having to worry about the packet's contents. Another argument is that it is more expandable. A Raspberry Pi has limited computational power and depending on how the model is trained, it could be that the Machine Learning model is too complex to handle many beacons per second. In this event it could also be computed elsewhere. The packet's information could be consumed from the 'raw_beacons' topic in the Apache Kafka cluster (no. 16) and a location could be pushed into the 'raw_locations' topic (no. 17). Adding the option that the Machine Learning model has the ability to run elsewhere without having to modify the current packet reading and processing that is done on the Raspberry Pi.

5.3.4 Rasperry Pi Redis

It might seem a bit cumbersome to first store a BLE packet to a Redis Stream (no. 3) and then read it again to send it to the Apache Kafka cluster. The main reason for this is reliability. The reading application is kept as light as possible to reduce the possibility of any crashes or unexpected failures. It also does not have to wait for packets to have been acknowledged by a broker in Apache Kafka and worry about data-loss. The application can listen for the BLE packets without interruptions and send them to the locally-running Redis (no. 3). Because of this it can also keep collecting when there is a network loss as the packets can be sent later by the Packet- and Location consumer (no. 4 & 5) when the connection is back up again. It also acts as a buffer, the nRF BLE Sniffer (no. 1) is read periodically which could come with bursts of data rather than a continuous stream of packets.

The reading from a Redis Stream (no. 3) happens in 3 stages:

- Unread

- Pending
- Acknowledged

The packets are inserted in the ’Unread’ state. They are ready to be processed and waiting for a Packet or Location consumer (no. 4 & 5). Once a packet is retrieved from the Redis Stream it enters the ’Pending’ state until the Packet or Location consumer actively acknowledges it that this packet has been processed. Only then it will be updated to the ’Acknowledged’ state. A packet is only acknowledged by a Packet or Location consumer if it has been pushed successfully to the corresponding topic in Apache Kafka (meaning that it was acknowledged by at least 1 broker in the cluster). In the event that the connection times out, application crashes or any other event that does not trigger an acknowledgement. The packets will remain in the ’Pending’ state. As the application reads from a Redis Stream it is horizontally scalable, multiple instances of a Packet or Location consumer can run simultaneously to improve reliability and/or throughput. In the event that a Packet or Location consumer fails, it could leave certain messages at the ’Pending’ state. Due to this possibility, the Packet or Location consumer periodically checks if a message has been in a ’Pending’ state for too long (e.g. longer than 10 seconds). If it finds any, it reassigns those packets to itself.

However, acknowledged packets are not automatically deleted. Without the Manager (no. 6), the Redis Stream would keep already pushed packets, which take up storage and are of no use on the Raspberry Pi. This is why the Manager periodically deletes all acknowledged entries until the first ’Pending’ or ’Unread’ entry.

5.3.5 Dashboard & webserver

The dashboard is a Vue.js frontend that communicates to with a backend, the webserver (no. 11). The webserver only allows authenticated users to interact with the website. The authentication is handled by ’passport.js’, which has over 2 million weekly downloads [54].

Live map

Once a user is logged in they will see a live map where they can find Thingies and watch their live movement. The mapping tool that the dashboard will use is MapBox as it has a free tier and extensive documentation and support for features such as markers, popups and heatmaps [55]. The movement is updated through polling (periodically requesting data from the same endpoint). If a Thingy 52 has not published a packet within the last 10 seconds it turns gray, otherwise it is blue to indicate that it has emitted a packet recently.

Thingy overview & inspection

On the overview page thingies can be searched and added. Before a Thingy is added, there is some validation to not allow duplicate names or MAC addresses. Once it is added to the MySQL database (no. 10) an 'add to whitelist' entry is pushed to the 'whitelist' topic in Apache Kafka (no. 18). This then triggers instant updates of the whitelist at the Packet and Location consumers (no. 4 & 5). The same happens upon deletion of a Thingy.

In the inspection page of a Thingy, the last received packet is displayed along with graphs showing the change of the sensor readings over time.

Historical map

The historical map has various options to visualize the locations of the thingies. Thingies can be downsampled by every minute, every 5, 10, 15, or 30 minutes. All data within a sample is taken into account by taking the average value of it. This reduces the amount of data that needs to be sent and visualized in the frontend. There are 3 visualization options, a heatmap, individual data points (which can be hovered to inspect them into further detail) and a line that shows the movement of a Thingy. Each visualization can be toggled on or off individually. There is also a time slider present to select the desired time range of which the data needs to be shown. Each Thingy 52 can be selected individually.

All three visualizations can be customized by changing the sensitivity of the heatmap and changing the radius/thickness of the lines or data points.

To reduce the load on the webserver (no. 11) and QuestDB TSDB (no. 12), a Redis cache is present (no. 9). This cache is used to temporarily store previously requested data samples to quickly serve them to the client. This also improves the latency as a request does not have to be generated from the TSDB, but instead can be directly served to the client. Using this strategy, it could be that the data that is visible is not 100% up to date as a cached response is used. However, its functionality is not to provide a live map.

Chapter 6

Implementation

Multiple programs need to be coded and deployed in order to collect and visualize data. This section describes how this is implemented.

6.1 Code

The entire project consists of code and thereof needs to have some attention.

6.1.1 GitHub

GitHub is used for versioning and keeping track of changes to the codebase. This allows collaboration by other users and insights in what exactly is changed. It also serves as a backup as the code is now also stored at GitHub's servers and not only locally at the developer.

Repository

The repository (available at https://github.com/Menkveld-24/GP-Bluetooth-asset-localization—2023) consists of multiple folders which contain all the software and configuration files to deploy this project. The file tree is in figure A.1 of the appendix, each directory also contains a README markdown file that has instructions on how to develop/deploy that part of the system.

6.1.2 Documentation & styling

As the code for this project is over 9000 lines (excluding comments), it is good practice to add comments to code and add function descriptions in order to make the code easier to understand by yourself and other programmers. As of this, all functions written in a Node.Js application will have descriptions according to the TypeDoc specification [56]. The comments and function descriptions in Python are written according to the PEP8 specification [57].

Node.Js

In Node.Js there are two options available to program in, TypeScript and CommonJS (regular JavaScript). TypeScript is the same as CommonJS except that it is a lot stricter with typing. Using CommonJS it would be an allowed action to assign an integer to a variable that was initially a string. However, it is unlikely that a string should be assigned to a variable that was initially an integer. With TypeScript the type of variable can be declared so the IDE and compiler throw errors on such actions. Additionally to this strict typing, ESLint and Prettier have been configured for the website (both frontend and backend) as well as for the consumer applications that run on the Raspberry Pi. ESLint and Prettier force consistent styling across the codebase, a program does not compile if there is a styling error such as a missing ';' or a newline or space too much or too little.

Python

The only application that is written in Python runs on the Raspberry Pi and reads the packets from the nRF BLE Sniffer. This code is written according to the PEP8 standard which also shows in the IDE where there are styling errors.

6.2 Deployment

Quite a few additional programs need to be run on top of the self-written applications. To deploy these Docker is used, Docker is a containerization platform. A container can be thought of similarly like a virtual machine, it is an operating system that is separated from the host. Containers however, do not boot their own operating system, but rather use the host's operating system and are therefore not 100% separated like virtual machines. Making them more performant than a traditional virtual machine. Containerizing applications with docker has the advantage that there is no need to install dependencies and setup the environment on every deploy. Making it easy to deploy an application on a different machine or operating system, or to scale up an application.

An environment is defined once using a ’Dockerfile’, this file describes what to install, which versions and what commands to execute. A Dockerfile generates an image, this image can be pushed to an image registry (Docker Hub) or kept locally. A ’docker-compose’ file specifies which images and version of that image needs to run along with which ports, volumes (like a USB stick to persist storage from a container) and environment variables. These images are all available for all used Kafka services; ksqlDB, Kafka Brokers, Confluent Control Center, Zookeeper, Kafka Connect and the Schema Registry. Not only are they available for Apache Kafka, but also for the other applications/services: Reverse proxy (nginx), QuestDB, MySQL, Redis.

The Dockerfiles for the self-written applications, docker-compose files and other configuration files are present in the corresponding GitHub repositories.

All detailed instructions for deploying and developing with this graduation project are available in the GitHub https://github.com/Menkveld-24/GP-Bluetooth-asset-localization—2023

6.3 Security

Ideally all applications should have authentication and encryption to identify each other and prevent network sniffing. Applications that run on the Raspberry Pi which need to access the brokers should ideally verify the brokers, and the brokers should verify the applications. Connections made to the Apache Kafka cluster should also use Access control list (ACL)s, restricting access to only topics required by a given application. However, setting this up takes time to do it properly and verify it while it does not impact the functionality of the system. As of this, the decision is made to restrict access to the entire sytem and expose as little as possible to the public internet. The server will be in its own private network along with the Raspberry Pi’s that should use the system. The only remote access will be to the webserver through the reverse proxy. As the dashboard will be remotely accessible (from internet connected device), this has authentication in the form of login-protected routes. A disadvantage of this approach is that a Raspberry Pi cannot connect from another network, however a workaround for this is to use a VPN.

Chapter 7

Evaluation

The system has been evaluated on various aspects to validate or to attempt to find constraints in the current implementation. This has mainly been done by applying a load at various aspects of the system which were then analyzed. All raw measurements from the performed tests along with the refined datasets are present in the GitHub repository.

7.1 Dashboard

The webserver (backend of the dashboard) has been load tested using k6 [58]. With k6 a JavaScript script can be created which defines what endpoints should be tested and how intensively.

7.1.1 Procedure

First, a list of endpoints had been defined of actions which a user could do:

- Visit the dashboard and retrieve live locations
- Go to the thingy overview page and inspect thingies
- Go to the historical page and inspect historical data

From this set of actions the following request scheme could be constructed:

- Visit the home page
  - Request the index.html
  - Request the index.css

    - Request the index.js
    - Request the last live locations
- Visit the overview page
    - Request all available Thingies
    - Inspect 3 random Thingies (3 requests)
- Visit the historical page
    - Request a random sample

This request schema was then configured to be used with k6 and run with various 'virtual users'. These users all approach the site following the defined scheme above, this simulates intensive usage of the website which can be used to define bottlenecks in the system. The test has been run in 3 modes; accessing the site via the proxy, accessing the site directly and accessing the site directly without cache. Each performed test has been run twice of which the average data has been used. All test modes were run with 5, 10, 25, 50, 100, 250, 500, 1000, 2500, 5000 and 10000 virtual users. Each run had a duration of 60 seconds. All tests used HTTP instead of HTTPS as a direct connection to the webserver only accepts HTTP connections.

Initially testing begun with 50 whitelisted Thingies and a dataset of 500 000 locations/packets present in QuestDB. However, upon start a flaw was quickly identified where the dataset was too large, causing the webserver to crash. As of this the dataset was significantly reduced (see section 7.1.2). Also was found that the network connection was limiting the tests due to the loading of the index.js and index.css files. These have been left out for all tests (see section 7.1.2).

7.1.2 Results

Network limits

The network connection from the k6 testmachine (see all specifications and raw results in appendix D) to the server was a Gigabit Ethernet connection. This network connection has a theoretical limit of 125MB/s and maxed out when running k6. To verify that the network connection was throttling the test, the test was also run on the server, yielding faster results in terms of MB/s and RPS (see upper half of 7.1). Looking into the network traffic it was found that the index.js file was uncompressed and was 2MB large. After further analysis it revealed that unused parts of dependencies were also compiled into it, removing this shrunk the file with ±25%. However, 1.5 MB is still quite large. This file was then also gzipped and compressed down with ±65% to 0.5MB. Running tests with the gzipped files showed that the network connection was no longer bottlenecking the tests, but the testmachine itself. When running more than 100 virtual users, CPU utilization approached 100% causing the tests to experience throttling limits which in term could affect the results [59]. Further tests were executed without loading an index.js and index.css as a result.

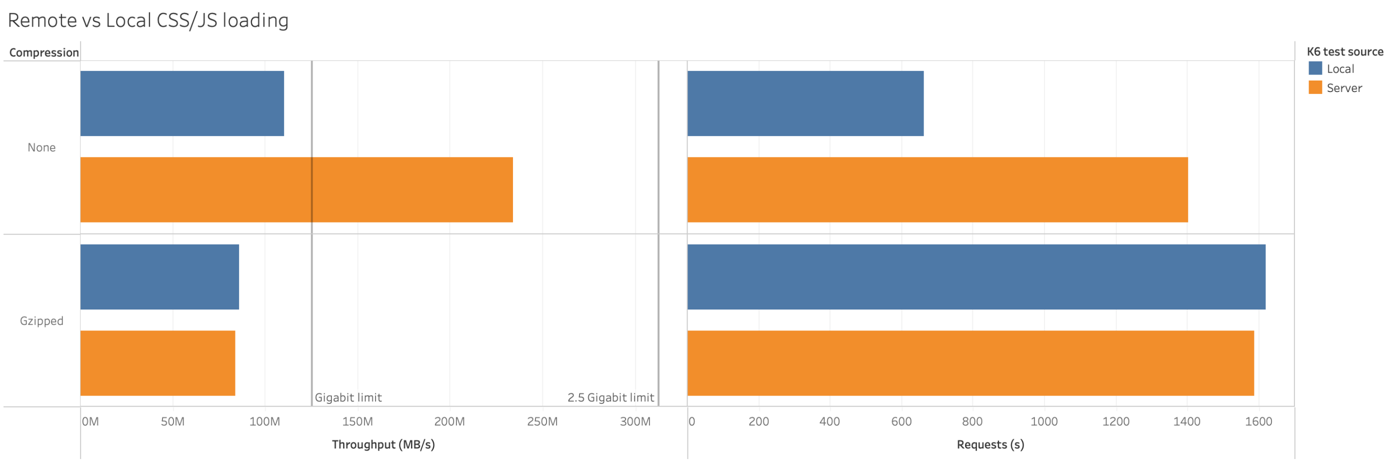


Figure 7.1: The impact of css/js compression and network throttling is visible (100 virtual users, http connection without proxy)

Historical data limitation

When running the first tests, with a dataset of 500 000 records spread out over 2 years, the tests quit almost instantly. Looking at the resource utilization of the system it was noticed that the RAM usage shot up and dropped down. Implying that something inside the webserver starts to consume a lot of ram and then crashes the webserver. This was supported by the logs generated from the webserver. All individual requests were traced in the browser, requests from the historical page took exceptionally long (±10 seconds) and their responses were almost 100MB large.

The frontend requests data for all thingies from a given sampling interval (1m, 5m, 10m, 15m or 30m). The webserver loads the raw data from the database into a variable and does some modification to it before it stores it in the cache and returns a response. However, if you speed this process up it could be that there are multiple virtual users simultaneously requesting a dataset causing multiple identical datasets to be loaded into the RAM. Quickly causing the webserver to run out of memory.

Potential solution

A test was also run with 10 virtual users (due to a RAM limitation) on the server, fetching the data from a static JSON file instead of loading it through the cache/webserver. Revealing that the webserver can serve data at a rate of ±1.4 GB/s, saturating a 10 Gigabit connection.

As of this result, a potential solution would be to serve the historical data through stored JSON files instead of via the cache/webserver. This reduces the CPU power and RAM the webserver requires to operate while still being able to serve the data for the historical page. However, this solution should be accompanied with sending less data as well. This could be done through more aggressive downsampling, having the user to select a time-range before they fetch data and lastly by sending the data per Thingy instead of them all together.

Caching

The webserver implements caching in multiple locations with QuestDB; at requesting the latest locations (live page) of the Thingies, inspecting Thingies and the data for the historical map. Looking at figure 7.2, the tests with caching enabled scored nearly double among all tests. Having a cache enabled improves throughput while also reducing the load on QuestDB.

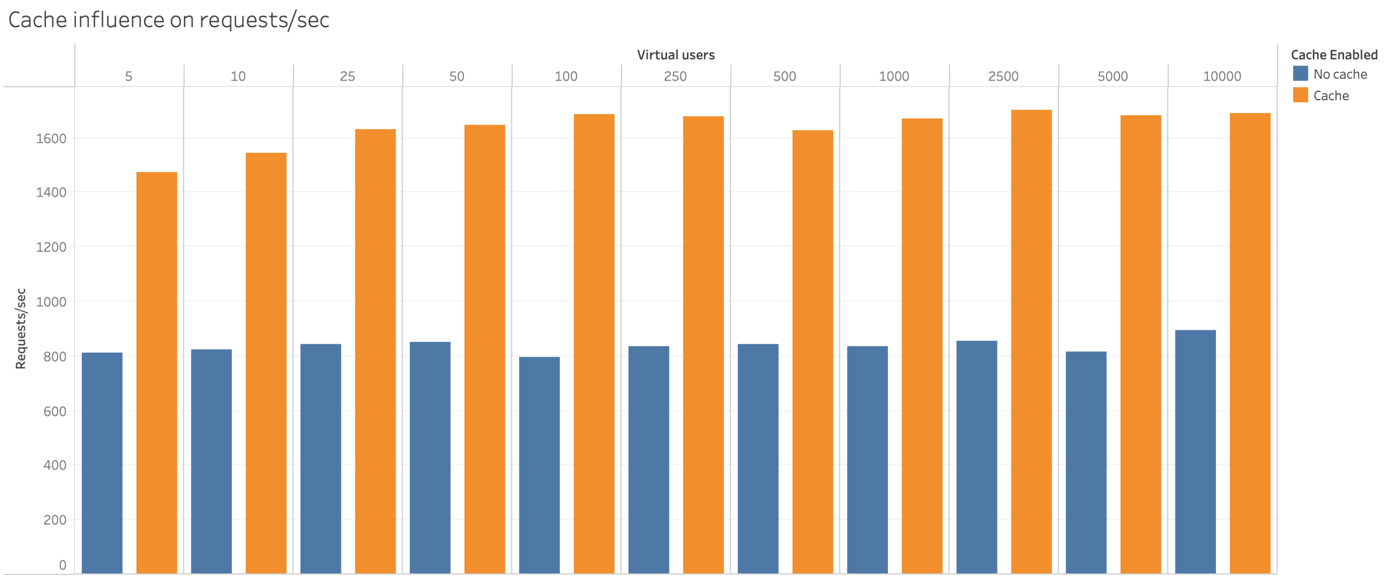


Figure 7.2: The impact of using a cache vs continuous request to QuestDB

Proxy

The proxy serves to direct traffic encrypted traffic from the internet to the webserver. Figure 7.3 shows that all requests sent directly to the webserver are accepted while those via the proxy experience a failure rate of over 50% for all tests. However, looking at figure 7.4 shows that all tests, except the first one with 5 virtual users, sends out more HTTP requests. Especially the tests concerning 2500 or more virtual users for requests via the proxy.

Additionally, looking at the HTTP request timings in figure 7.5 it is visible that the minimum response times for the direct requests shoot up from that point on, while the requests via the proxy have minimum values of 0. On the other hand, the maximum request durations shoot up for requests via the proxy while they do so much less for direct requests.

Figure 7.6 shows the same data as in figure 7.5, but is visualized differently. Looking at the requests with 5000 virtual users for the requests via the proxy, it can be seen that the median (243ms) sits below the mean (907ms). Indicating a left-skewed distribution. The 95th percentile response time is 3300ms, meaning that there are few outliers that reach up to 36794ms. These distributions apply for the tests run directly to the proxy using 2500, 5000 and 10000 virtual users.

The requests that were made directly to the webserver all look similar except for the scaling between them. Deeper analyzation of 5000 virtual users it reveals that the median (2236ms) is also lower than the mean (2972ms). Indicating a left-skewed distribution, similar to the requests via the proxy. However, the 95th percentile is also nearly double at 5690ms.

Despite that there were made significantly more requests during testing via the proxy than directly. It seems that due to the presence of the high failure rate combined with the left-skewness, presence of a minimum request duration of 0ms. Combined with considerably lower means, medians and 90/95th percentiles than the requests made directly, that most failures were due to a rejection by the proxy.

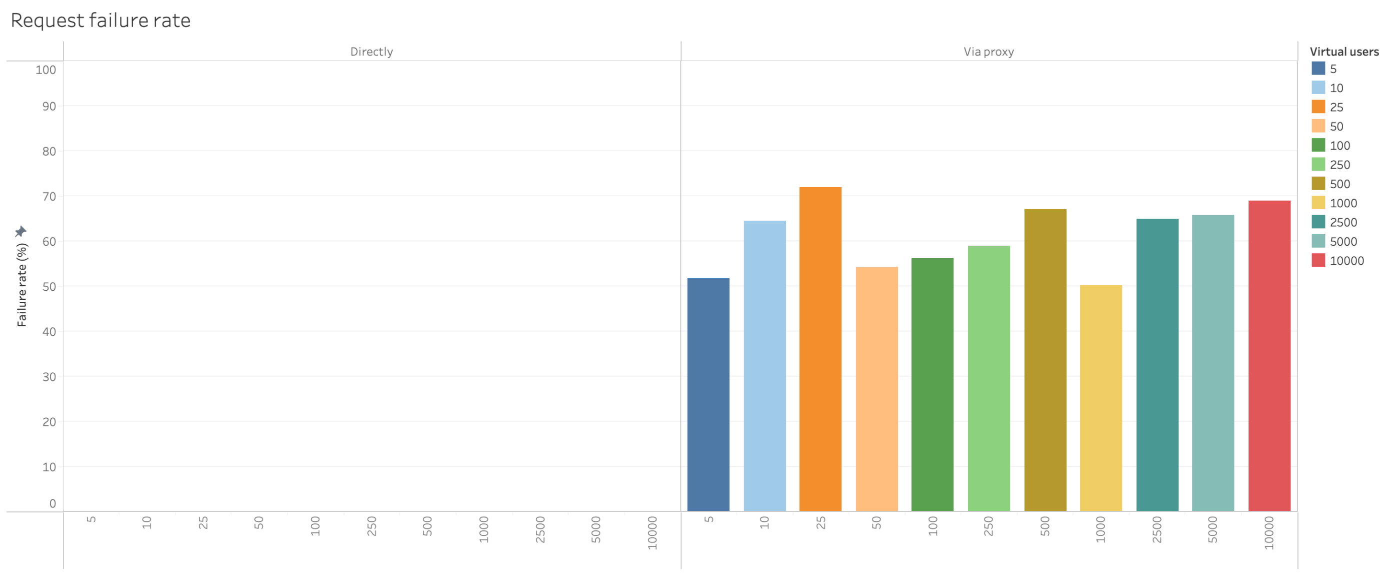


Figure 7.3: The failure rate of requests via the proxy or directly to the webserver

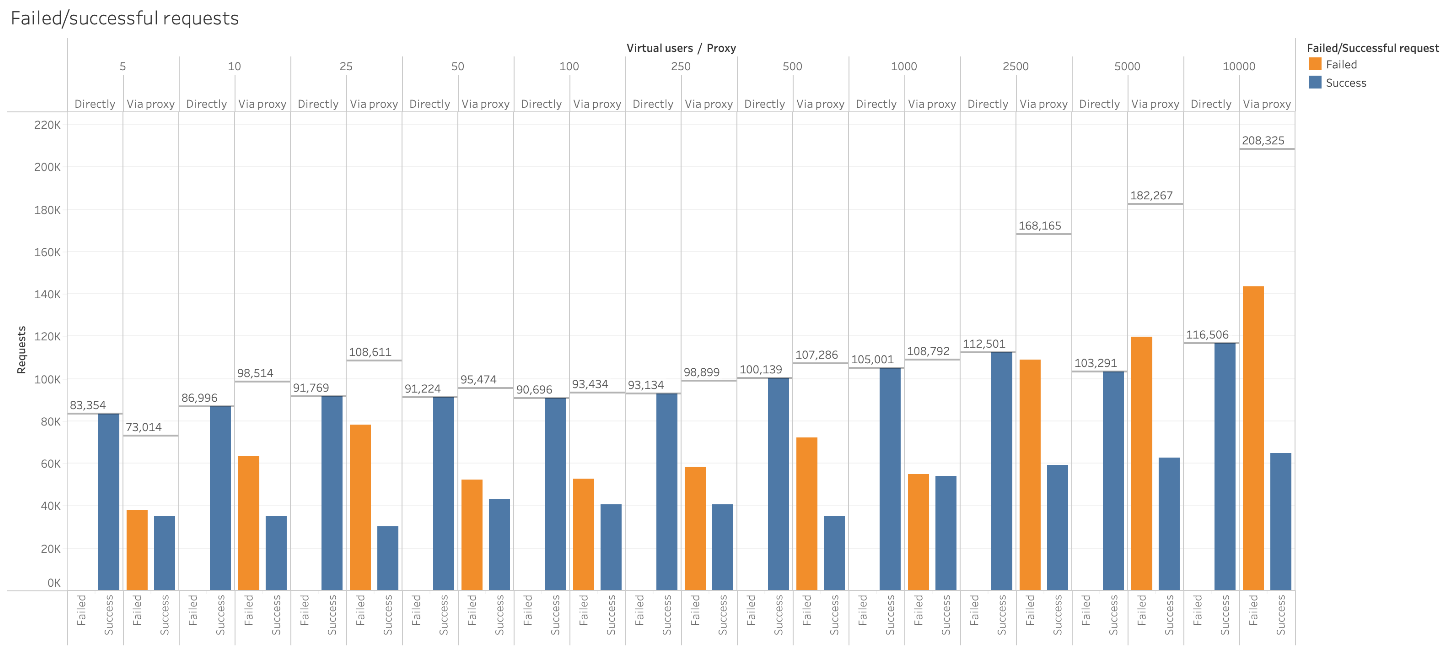


Figure 7.4: Proxy vs direct connection to the webserver failed/successful requests per test

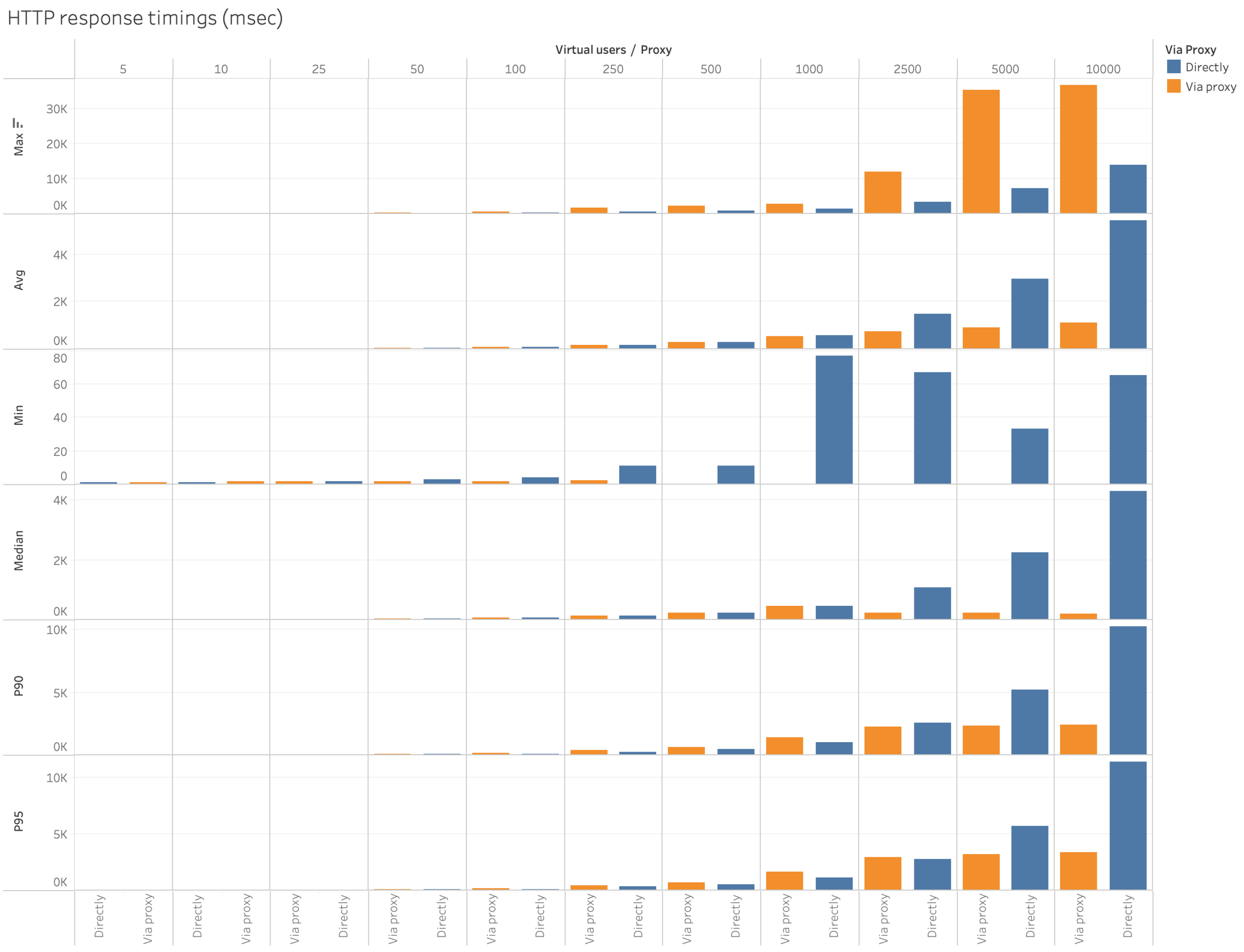


Figure 7.5: Proxy vs direct connection to the webserver HTTP request timings per test.

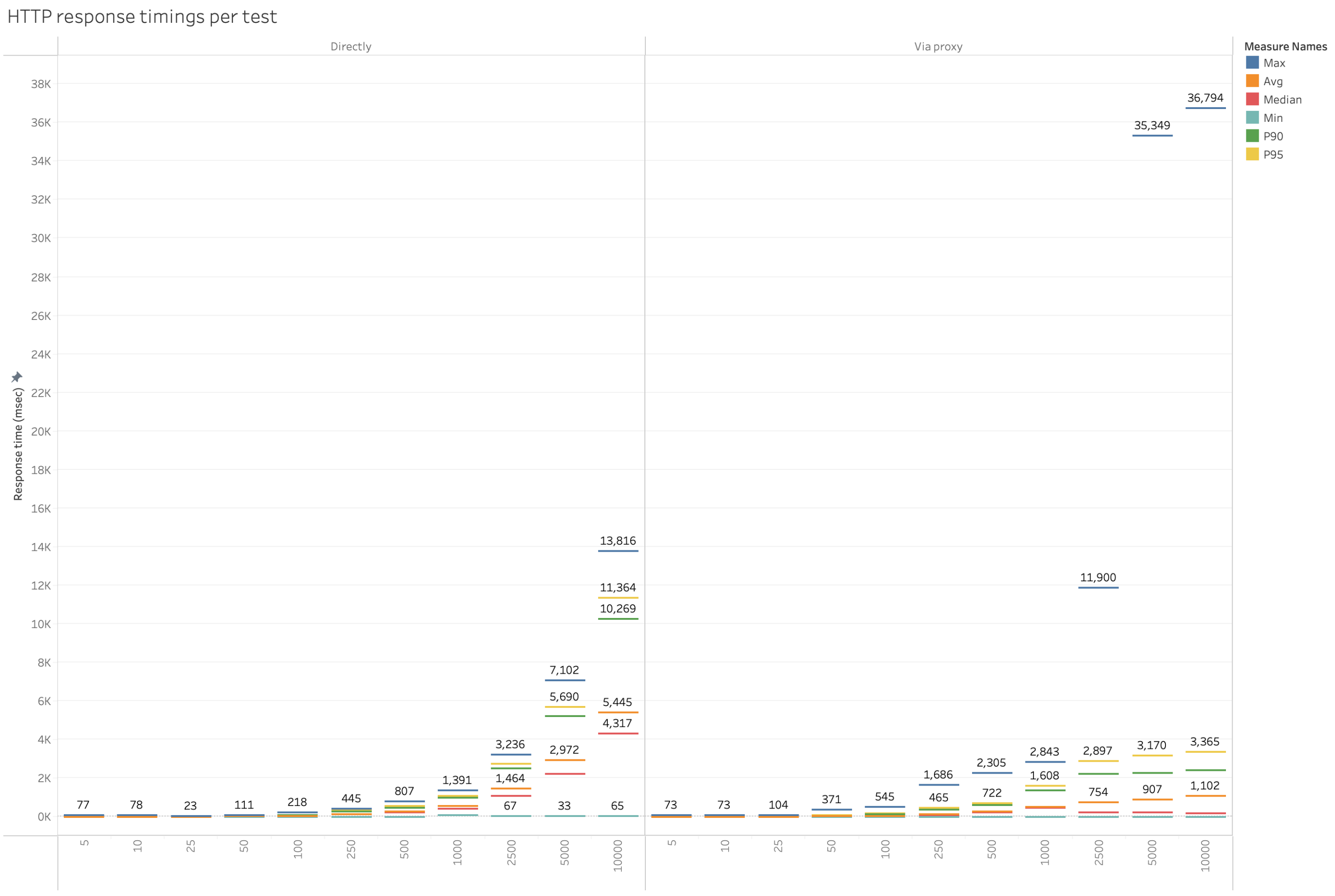


Figure 7.6: Proxy vs direct connection to the webserver HTTP request timings per test.

7.2 Raspberry Pi

The software running on the Raspberry Pi such as the consumer, location consumer, BLE Sniffer and manager has been tested for its throughput and reliability. The behavior of the consumers can be influenced in three ways;

- By running multiple consumers in parallel (consumer count)
- By changing the read interval (iteration speed)
- by changing the number of packets read per iteration (read size)

7.2.1 Procedure

The first test that was run is to evaluate the maximum throughput of both the consumer and location consumer (the applications that consume packets from the Redis Stream and upload them to Apache Kafka, see 5.3.4 for more details). This is done by running the tests in multiple stages, each having a different read size. The first set of tests was run using 1 replica of each consumer while the second run was performed with 2 replicas. During each read from Redis, the code would read a maximum of 10, 100, 1000, 2500, 5000 or 7500 packets at a time and upload them to Apache Kafka. In total 100.000 packets are inserted in each run by a simulator where the results of the manager were used to collect metrics, both not running on the Raspberry Pi.

The second test was to validate the integrity of the system using 2 replicas of the consumer (the location consumer shares the same source code responsible for reading and uploading, hence the location consumer was not tested separately). This was done by reducing the iteration speed to 1Hz and a read size of 10. The minimum time for a packet to be in the 'pending' state was configured to 5s before it could be (auto)claimed. One instance would have a bit of code injected where it would have a random chance to abruptly crash between reading from Redis, and pushing the packets to Apache Kafka. After which the other consumer should pick up those 'pending' packets and push them instead (see 5.3.4 for more details).

7.2.2 Results

The detailed environment specifications along with raw test results for both can be found in appendix D. All runs were executed three times of which the average values were taken.

Throughput

Looking at graph 7.7, the gray line is the number of packets pushed to the Redis Stream. There is a significant difference when the read size is set as low as 10 per iteration. However, this difference becomes minor for anything above 1000 packets per iteration. It also shows that there is no significant difference between the upload speed of a consumer or location consumer.

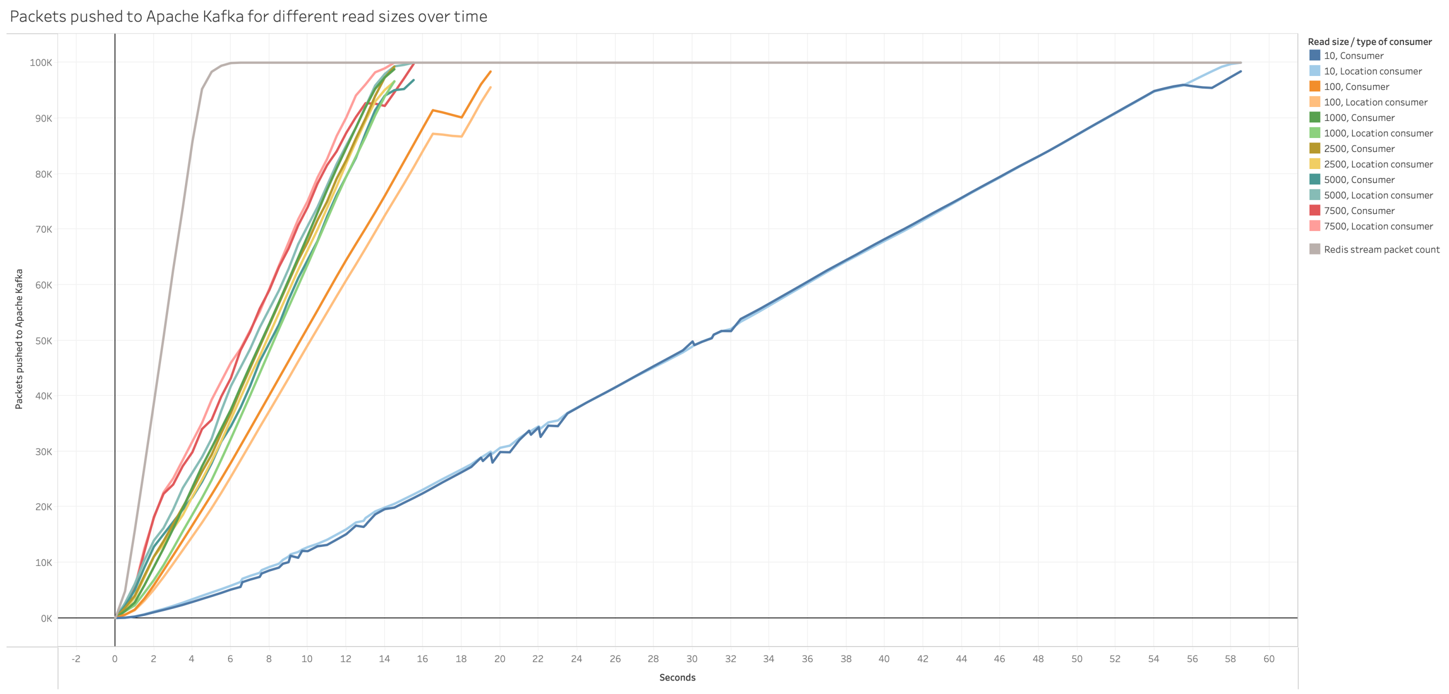


Figure 7.7: The packets that were pushed over time by the (location) consumer (1 replica)

When comparing the consumers using one or two replicas in figure 7.8, the lines describe the number of packets that are in the Redis Stream, but not yet read by any consumer. It shows that there is less lag using 2 replicas. However, the lines are a lot less fluent. This could be because the consumers were all running at max speed while there was also a Redis instance running, which would be 5 applications that now have to share 4 CPU cores. This claim would be supported by figure 7.9. In this figure, the packets pushed to Redis from the simulator running on a remote device experience a reduced and less stable throughput as the lines' slope decreases and becomes less fluent. It would not be a limitation because of Redis as this behavior is not present during low read sizes, which access the Redis at a much higher rate than larger read sizes.

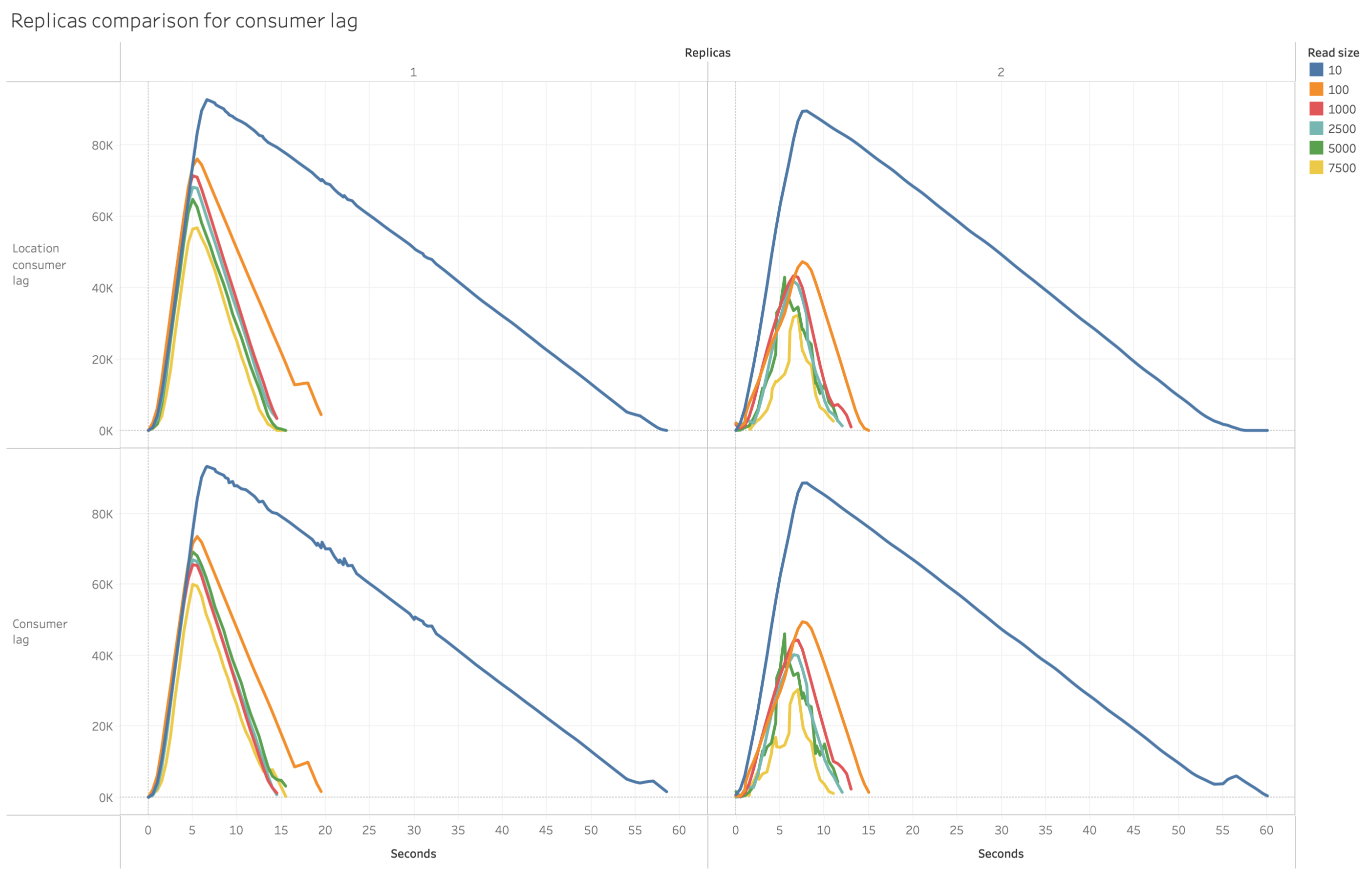


Figure 7.8: The number of packets the consumers lag behind for 1 or 2 replicas

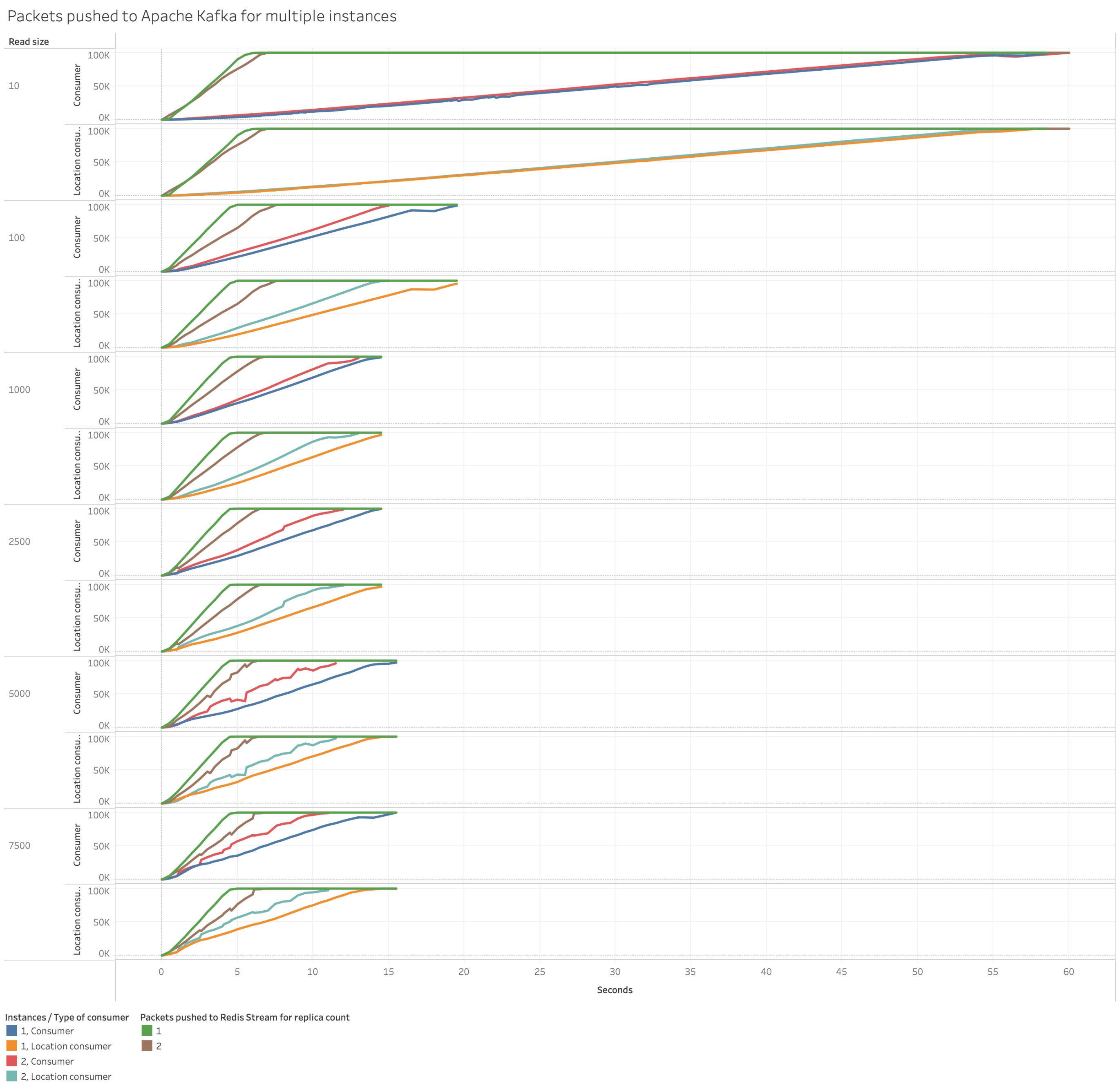


Figure 7.9: The packets pushed to Apache kafka or Redis per read size and type of consumer

Autoclaiming

The second test result is visible in figure 7.10. This test started with 200 packets present in the Redis Stream and the consumers were reduces to a read size of 10 with an iteration speed of 1Hz. The second consumer was intentionally crashed at ±11 seconds from the start. At this timestamp, the packets were read by the second consumer, but not further processed. Leaving them at pending until the first consumer claims these unacknowledged packets and uploads them to Apache Kafka at ±16 seconds.

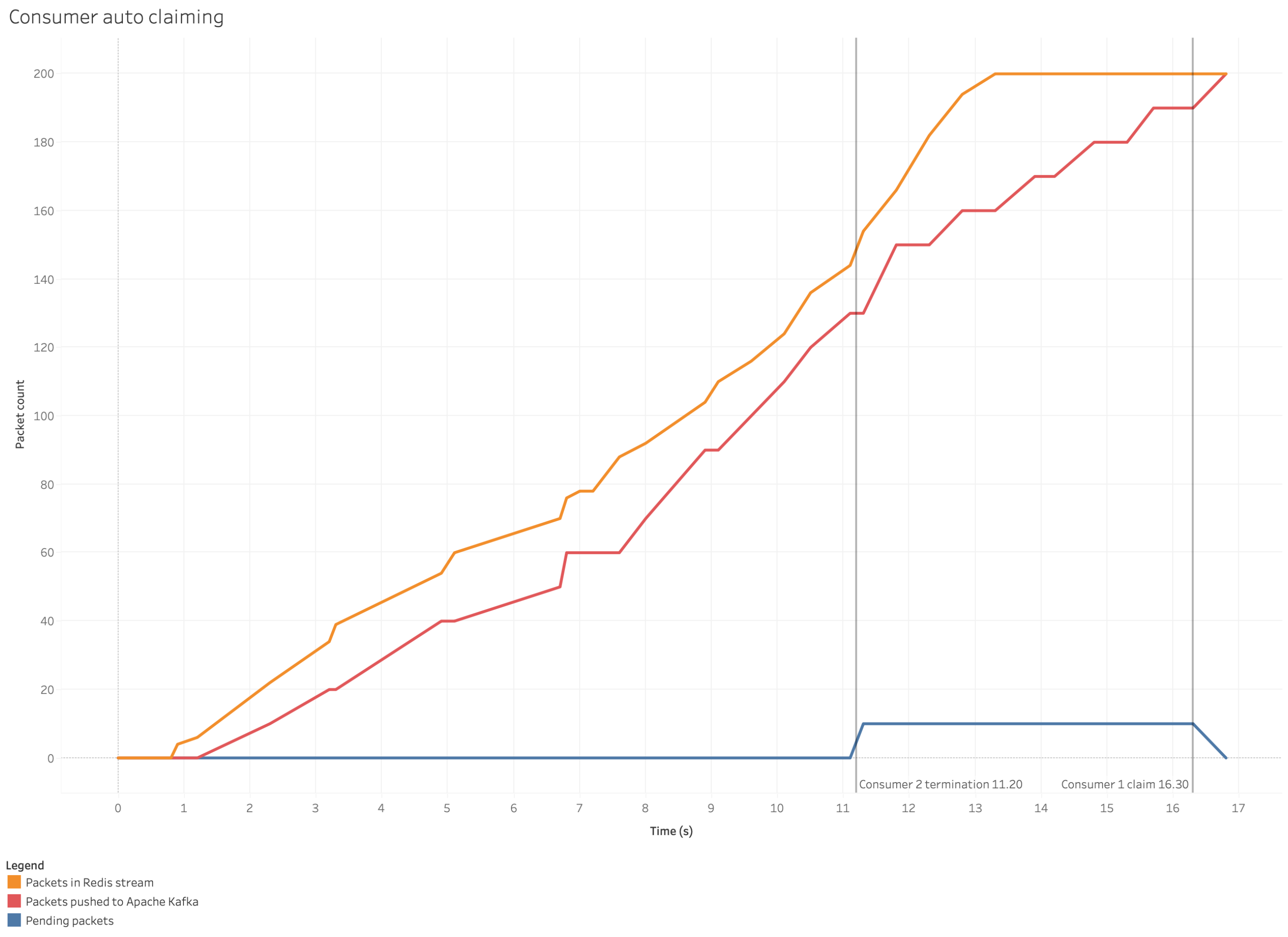


Figure 7.10: The validation of the auto claiming when one consumer fails

7.3 Conclusions

The requirements from section 5.1 are visible below and mostly met.

Must have:

- A dashboard that visualizes locations on a map.
- The dashboard must be accessible from anywhere.
- The system must be reliable.
- The dashboard must be responsive.

- The dashboard must have authentication.

Should have:

- The system should be scalable.
- The system should show a live location of an asset.
- The system should collect auxiliary data such as temperature.

Could have:

- The system could control the Thingy 52's from the dashboard (e.g. turn on a LED).
- The Thingy 52's could be able to update through the dashboard (FOTA).
- The system could send out notifications on specific events, such as a low battery.

Won't have:

- The system won't have a functional localization system.

From the 'must have', only the reliability and responsitivity requirements are partially met. The reliability is a hard factor to fully meet and has many points of failure. Although the reliability of sending packets once they have been received until they are present in Apache Kafka has been validated in section 7.2.2. There are still many more aspects that come into play after the Raspberry Pi such as the observed failure rate that requests have through the reverse proxy (section 7.1.2), SSD storage, brownouts and network stability. Making this a hard box to fully tick. The responsitivity still has some flaws at the historical page which need to be worked out before this can be checked, a potential solution was suggested in 7.1.2.

The 'should have's' shows auxilary data and a live location, ticking 2/3 boxes. The scalability however, was only tested of the consumer and location consumer running on the Raspberry Pi. Although running multiple Raspberry Pi's was not explicitly tested, it should work as Apache Kafka itself can take connections from many devices. One foreseeable issue would be that in the case of multiple Raspberry Pi's, it could be that one packet is received by both, pushing the data twice to Apache Kafka. This can be resolved by filtering duplicates with the help of KSQL. Regarding Apache Kafka and QuestDB, both are enterprise grade software which have full support to distribute data and load.

Although none of the 'Could have's' or 'won't have's' were met, low battery notifications could be implemented by adding anomaly detection inside Apache Kafka on the incoming packets stream.

Chapter 8

Discussion and conclusion

8.1 Discussion

All tests in chapter 7 were run independently of each other. So when the Raspberry Pi was evaluated, there was no other load on the system generated by incoming requests from the dashboard or other Raspberry Pi's. Although this allows for a good evaluation of individual components, it could be different from what might happen in a real-world scenario where the system has other loads applied too. Another limitation is that the server running all the software was continuously at a rather high memory pressure of ±80% or above. This could have had an effect on the results. It should also be noted that the requests in section 7.1.2 were all run on an i7 processor dating from 2014. As this is not the most recent processor, results could vary for that tests when run with a more modern and powerful processor. The tests were also not run with more than 10000 virtual users as this would not run, despite k6 claiming that a single instance should be capable of running tests with 30000-40000 virtual users [59]. Currently, all tests were run without HTTPS and without encryption between components of the system. Which could have a negative impact on the measured results in chapter 7 once this will be configured.

The results from the test in section 7.2.2 regarding the throughput of the consumers on the Raspberry Pi were all run without the BLE Sniffer and manager. Although it now shows a clear difference between the number of replicas, in a real-world scenario the manager and BLE Sniffer would always be running and consuming resources. Resources that were now used by the consumers and Redis. The results also showed that there was no significant difference between the throughput when using read sizes larger than 1000 packets per iteration. This, combined with the results in figure 7.9, could open up a good middle ground where the consumers dynamically scale their iteration speed and read size depending on the number of packets pending and the rate they come in. The main advantage of this would be that the Raspberry Pi remains responsive for other processes when they (temporarily) need more CPU power, while still being able to process many packets when required. Another advantage of the reduced CPU load is a reduction in power consumption and thereof generated heat.

Results of the proxy in section 7.1.2 showed that the proxy is limiting the throughput of what the webserver can handle. Although a specific issue was not identified, it could be as simple as a lack of configuration, or even that this proxy is not the best one out there to handle this job.

Working with Apache Kafka proved to be quite a challenge due to its size and complexity. It had many new concepts such as consumers, producers, topics, partitions which first needed to be studied. However, there is more to Apache Kafka as there is also KSQL and Connectors, parts that were responsible for data processing inside Apache Kafka.

Now is this only a part of the entire system, a complete website containing a frontend and a backend needed to be integrated with other components of the system and MapBox. Although this was partially familiar, it does not take away the fact that it was quite some work to set up.

There was the challenge concerning the Thingy 52's and BLE packets. The nRF Connect SDK using Zephyr Real-Time Operating System (RTOS) is quite different from an ESP32 or Arduino. This, combined with the age of the Thingy 52's and number of sensors that needed to be read out, posed its challenges. However, sending the packets is just the first half. Figuring out a way to reliably receive these BLE advertising packets turned out to be quite the challenge.

Lastly, the software also needed to be deployed on a server in order to learn, develop and run tests. This also took its necessary time before all 18 Docker containers on the server were properly configured along with build and initialization scripts.

Quickly into the evaluation it became clear that the data retrieval for the dashboard was flawed (section 7.1.2). Upon request all data would be sent to the user which became a problem for the user and server as it was just too much data. The results were that too much ram was used on the server causing the backend to crash when a bit of load was applied and that the response times were long for the users (±10 seconds).

Overall the evaluation showed that this system has potential (to expand), but also needs some (re)work on aspects such as data retrieval for the dashboard and the reverse proxy. These aspects will be discussed in chapter 9.

8.2 Conclusion

This graduation project aimed to design a system that visualizes locations of Bluetooth trackers, from the source to the user. To do this, the following main research question was identified:

How to design an accessible and efficient system that can visualize the locations of the Bluetooth trackers?

4 sub research questions were identified to help answer this main research question.

8.2.1 What is the best way to store the locations from the trackers?

Section 2.3 partially answers this question. It states that a TSDB is designed for temporal data, which the data from the trackers is. This has the advantage that time-based querying and downsampling is part of its nature, making queries fast whilst receiving 'lightweight' responses that still take all data into account. Another aspect of storing the locations is the format of it. Although not discussed specifically, locations are stored in longitude and latitude. This makes it easy to implement them in the frontend as no translation layer is required. It also makes it easy for other parts of the system to read and work with this data as the location it receives is really the location without having to apply some offset.

8.2.2 How can the dashboard be designed to be accessible and easy to use?

The dashboard has been made accessible by creating it as a website. This website is exposed via a reverse proxy which handles HTTPS encryption. Making the site accessible from anywhere by any internet connected device without having to install something first.

8.2.3 How can the reliability of the system be maximized?

The reliability of the system is dependent on many factors, both internally as externally. The code that runs on the Raspberry Pi such as the consumer and location consumer were designed to run in parallel. In the event that a replica experiences a critical error or other failure, they have proven in section 7.2.2 that no loss of data has occurred. Reliability is also the main reason that the code was split up and that all received packets first go through a Redis Stream, this acts as a buffer in case Apache Kafka is unreachable, no received packets will get lost. Reliability can also be improved by quickly detecting when something goes wrong. This is where the manager on the Raspberry Pi comes into play. It continuously reports metrics to the Apache Kafka cluster about the throughput of the Redis Stream. By looking at these results (or whether the metrics even arrive at Apache Kafka), anomaly detection could be used to quickly detect failure of any part of the Raspberry Pi. To tackle environmental issues such as network losses, brownouts and hardware failure, it was chosen to deploy and store most parts on a server remotely in the cloud. Running the webserver in the cloud also has the advantage that it is easy to connect to the dashboard as it will have a fast internet connection and static ip address.

8.2.4 What strategies can be used to optimize the scalability of the system?

Not only does running the software remotely on a server improve reliability, but it improves the scalability as well. When using a remote server, it is easy to expand in terms of RAM, storage or computational power. However, expanding resources should be accompanied by scaling the software too by using enterprise grade software such as Apache Kafka and QuestDB, both of which are designed to scale. Also, the usage of a cache for retrieving data from QuestDB had a positive effect on throughput, reducing the need to scale up. However, the system should not only scale in the cloud but also on the ground, by deploying more Raspberry Pi's for example. Although the current implementation does not actively support it (there is no packet deduplication if a packet is picked up by 2 or more Raspberry Pi's e.g.), it should work fine as Apache Kafka allows many producers and consumers to connect to topics.

The answers to the sub research questions together provide a guideline to the main research question. Due to the extensiveness and wide variety of the entire system, it is hard to give a concise answer to the main research question. To conclude, a distributed system deployed over multiple devices is designed around Apache Kafka that maximizes reliability, scalability and accessibility.

Chapter 9

Future work

The first and foremost recommendation is to revise the historical data retrieval from the dashboard. The current implementation retrieves all data at once which is fundamentally flawed as shown in section 7.1.2. A potential solution consists of various changes to reduce the amount of data sent in the first place. This could be done by more aggresive downsampling and sending the data upon request per Thingy instead of sending all data at once. However, sending less data through the current implementation still loads the same data many times through the RAM, not fully fixing the crashing that was observed. Therefore the system could be expanded by a new, additional dedicated application that only prepares data upon request. In the case that some historical data is requested, the request for data is put into a queue for the other dedicated application while the webserver returns a url at which the data will be available as a static json file, the frontend will keep polling this url. When the dedicated application has finished processing the file, it will store it for a limited duration. Doing so would keep the webserver free to handle other tasks instead.

A second improvement would be to look into, and set up security such as authentication, encryption and ACL's between all components in the system. Currently, authentication is only required to access the dashboard, which is also the only component that is exposed to the public internet. Apache Kafka does have support for a variety of authentication mechanisms such as LDAP, SASL/SCRAM and SSL. This would greatly improve the system's security and integrity.

Another improvement would be to implement anomaly detection using KSQL in Apache Kafka. This could have many system-wide improvements, such as low-battery notifications for the Thingies and fault detection for the Raspberry Pi's. It could also be used in specific scenarios where the environment's temperature should be monitored, or where all tracked devices could function as additional security by flagging changes in position when the warehouse is closed.

Lastly, the dashboard's designs were only based on a literature research. These designs along with the user experience could be evaluated (and improved) by doing tests with users. This has the potential to address design flaws and improve the usability and effectiveness of the system.

Bibliography

[1] “Nordic Thingy:52 User Guide.” [Online]. Available: https://infocenter.nordicsemi.com/pdf/Thingy_UG_v1.1.pdf

[2] “Welcome to the nRF Connect SDK! — nRF Connect SDK 2.3.99 documentation.” [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/index.html

[3] “Buy a Raspberry Pi 3 Model B+ – Raspberry Pi.” [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/

[4] “The Difference Between Classic Bluetooth and Bluetooth Low Energy.” [Online]. Available: https://blog.nordicsemi.com/getconnected/the-difference-between-classic-bluetooth-and-bluetooth-low-energy

[5] “Intro to Bluetooth Beacons — Bluetooth® Technology Website.” [Online]. Available: https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-beacons/

[6] J. Lindh, “Bluetooth ® low energy Beacons,” 2015. [Online]. Available: https://www.ti.com/lit/an/swra475a/swra475a.pdf

[7] “Bluetooth Low Energy -It Starts with Advertising — Bluetooth® Technology Website.” [Online]. Available: https://www.bluetooth.com/blog/bluetooth-low-energy-it-starts-with-advertising/

[8] “Bluetooth 5 Advertisements: Everything you need to know — Novel Bits.” [Online]. Available: https://novelbits.io/bluetooth-5-advertisements/

[9] “Bluetooth 5 Advertising Extensions.” [Online]. Available: https://blog.nordicsemi.com/getconnected/bluetooth-5-advertising-extensions

[10] "The best way to store, collect and analyze time series data — InfluxData." [Online]. Available: https://www.influxdata.com/the-best-way-to-store-collect-analyze-time-series-data/

[11] I. Kondratova and I. Goldfarb, "Color Your Website: Use of Colors on the Web," in *Usability and Internationalization. Global and Local User Interfaces*, N. Aykin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 123–132. [Online]. Available: https://www.researchgate.net/publication/221098779_Color_Your_Website_Use_of_Colors_on_the_Web

[12] E. Michailidou, S. Harper, and S. Bechhofer, "Visual complexity and aesthetic perception of web pages," in *Proceedings of the 26th annual ACM international conference on Design of communication*. New York, NY, USA: ACM, 9 2008, pp. 215–224. [Online]. Available: https://dl.acm.org/doi/10.1145/1456536.1456581

[13] L. Thorlacius, "The Role of Aesthetics in Web Design," *Nordicom Review*, vol. 28, no. 1, pp. 63–76, 5 2007. [Online]. Available: https://www.researchgate.net/publication/238106570_The_Role_of_Aesthetics_in_Web_Design

[14] O. Wu, Y. Chen, B. Li, and W. Hu, "Evaluating the visual quality of web pages using a computational aesthetic approach," in *Proceedings of the fourth ACM international conference on Web search and data mining*. New York, NY, USA: ACM, 2 2011, pp. 337–346. [Online]. Available: https://dl.acm.org/doi/10.1145/1935826.1935883

[15] L. Francisco-Revilla and J. Crow, "Interpreting the layout of web pages," in *Proceedings of the 20th ACM conference on Hypertext and hypermedia*. New York, NY, USA: ACM, 6 2009, pp. 157–166. [Online]. Available: https://dl.acm.org/doi/10.1145/1557914.1557943

[16] A. Parush, Y. Shwarts, A. Shtub, and M. J. Chandra, "The Impact of Visual Layout Factors on Performance in Web Pages: A Cross-Language Study," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 47, no. 1, pp. 141–157, 3 2005.

[17] N. Bonnardel, A. Piolat, and L. Le Bigot, "The impact of colour on Website appeal and users' cognitive processes," *Displays*, vol. 32, no. 2, pp. 69–80, 4 2011. [Online]. Available: https://www.researchgate.net/publication/256970347_The_impact_of_colour_on_Website_appeal_and_users'_cognitive_processes

[18] W. Swasty and A. R. Adriyanto, “Does Color Matter on Web User Interface Design,” *CommIT (Communication and Information Technology) Journal*, vol. 11, no. 1, p. 17, 8 2017. [Online]. Available: https://journal.binus.ac.id/index.php/commit/article/view/2088

[19] A. Mirza, “Emotional Impact of Colors Using Web-Design,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 10, no. 1, pp. 944–947, 1 2023. [Online]. Available: https://www.researchgate.net/publication/367656544_Emotional_Impact_of_Colors_Using_Web-Design

[20] P. Cleveland, “Colour and Dynamic Symmetry,” *PCA/ACA 2011 Conference*, 2011. [Online]. Available: https://research-repository.griffith.edu.au/bitstream/10072/46215/1/70834_1.pdfhttps://research-repository.griffith.edu.au/handle/10072/46215

[21] “The 11 Best Backend Frameworks – 2023 — CodingNomads.” [Online]. Available: https://codingnomads.co/blog/best-backend-frameworks/

[22] “10 Most Popular Web Frameworks to Use in 2023 — Monocubed.” [Online]. Available: https://www.monocubed.com/blog/most-popular-web-frameworks/

[23] “Web Frameworks Benchmark.” [Online]. Available: https://web-frameworks-benchmark.netlify.app/compare?f=django,laravel,flask,spring,express

[24] “Round 21 results - TechEmpower Framework Benchmarks.” [Online]. Available: https://www.techempower.com/benchmarks/#section=data-r21&test=fortune

[25] “vuejs/core: Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.” [Online]. Available: https://github.com/vuejs/core

[26] “vuejs/vue: This is the repo for Vue 2. For Vue 3, go to https://github.com/vuejs/core.” [Online]. Available: https://github.com/vuejs/vue

[27] “vue - npm.” [Online]. Available: https://www.npmjs.com/package/vue

[28] “sveltejs/svelte: Cybernetically enhanced web apps.” [Online]. Available: https://github.com/sveltejs/svelte

[29] “svelte - npm.” [Online]. Available: https://www.npmjs.com/package/svelte

[30] “facebook/react: The library for web and native user interfaces.” [Online]. Available: https://github.com/facebook/react

[31] “facebook/react-native: A framework for building native applications using React.” [Online]. Available: https://github.com/facebook/react-native

[32] “react - npm.” [Online]. Available: https://www.npmjs.com/package/react

[33] “purecss - npm.” [Online]. Available: https://www.npmjs.com/package/purecss

[34] “bootstrap - npm.” [Online]. Available: https://www.npmjs.com/package/bootstrap

[35] “bulma - npm.” [Online]. Available: https://www.npmjs.com/package/bulma

[36] “tailwindcss - npm.” [Online]. Available: https://www.npmjs.com/package/tailwindcss

[37] “normalize.css - npm.” [Online]. Available: https://www.npmjs.com/package/normalize.css

[38] “semantic-ui - npm.” [Online]. Available: https://www.npmjs.com/package/semantic-ui

[39] “troxler/awesome-css-frameworks: List of awesome CSS frameworks in 2023.” [Online]. Available: https://github.com/troxler/awesome-css-frameworks

[40] “Real-Time Location Systems & Hardware - Inpixon RTLS.” [Online]. Available: https://www.inpixon.com/technology/rtls

[41] “INTRANAV.IO The Enterprise IoT RTLS Platform for the Digital-Twin - INTRANAV.” [Online]. Available: https://intranav.com/en/iot-rtls-suite/intranav-io-the-enterprise-iot-rtls-platform-for-the-digital-twin/

[42] “How Real-Time Location Systems Work - Ubisense.” [Online]. Available: https://ubisense.com/location-technology/

[43] “How it works - Nextome.” [Online]. Available: https://nextome.com/how-it-works

[44] “Pricing - Azure Maps — Microsoft Azure.” [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/azure-maps/

[45] A. Mader and W. Eggink, “A DESIGN PROCESS FOR CREATIVE TECHNOLOGY,” 2014. [Online]. Available: https://www.researchgate.net/publication/265755092_A_DESIGN_PROCESS_FOR_CREATIVE_TECHNOLOGY

[46] “Apache Kafka.” [Online]. Available: https://kafka.apache.org/

[47] "Making Your UX Life Easier with the MoSCoW — IxDF." [Online]. Available: https://www.interaction-design.org/literature/article/making-your-ux-life-easier-with-the-moscow

[48] "(PDF) Optimizing State of Charge (SOC), temperature, and State of Health (SOH) of lithium-ion batteries." [Online]. Available: https://www.researchgate.net/publication/340777901_Optimizing_State_of_Charge_SOC_temperature_and_State_of_Health_SOH_of_lithium-ion_batteries

[49] "nRF52832 Product Specification," 2017.

[50] "11073-20601-2008 - IEEE Health informatics - Personal health device communication - Part 20601 : Application profile - Optimized exchange protocol." 2008.

[51] "Carbon Dioxide — Wisconsin Department of Health Services." [Online]. Available: https://www.dhs.wisconsin.gov/chemical/carbondioxide.htm

[52] "adapter-api.txt ≪ doc - bluez.git - Bluetooth protocol stack for Linux." [Online]. Available: https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc/adapter-api.txt

[53] "bluez: Linux kernel always de-duplicates advertisements · Issue #235 · hbldh/bleak." [Online]. Available: https://github.com/hbldh/bleak/issues/235

[54] "passport - npm." [Online]. Available: https://www.npmjs.com/package/passport

[55] "API Reference — Mapbox GL JS — Mapbox." [Online]. Available: https://docs.mapbox.com/mapbox-gl-js/api/

[56] "TypeDoc." [Online]. Available: https://typedoc.org/

[57] "PEP 8 – Style Guide for Python Code — peps.python.org." [Online]. Available: https://peps.python.org/pep-0008/

[58] "Load testing for engineering teams — Grafana k6." [Online]. Available: https://k6.io/

[59] "Running large tests." [Online]. Available: https://k6.io/docs/testing-guides/running-large-tests/

Appendix A

GitHub repository layout

```
/
├─ server-deployment/
│  ├─ dashboard/
│  ├─ kafka-initializatino/
│  └─ README.md
├─ dashboard/
│  ├─ frontend/
│  ├─ backend/
│  └─ README.md
├─ pi/
│  ├─ python-sniffer/
│  ├─ consumers/
│  ├─ nRF BLE Sniffer/
│  └─ README.md
├─ tesing
│  ├─ k6/
│  ├─ pi/
│  └─ README.md
├─ thingy52-src
│  └─ README.md
├─ README.md
└─ LICENSE
```

Figure A.1: The file tree of the GitHub repository

Note that all code is too large and complex to be added to the appendix. Therefore it is available at https://github.com/Menkveld-24/GP-Bluetooth-asset-localization—2023.

Appendix B

Battery voltage mappings

The formulas below describe the transformations of the voltage mapping to a battery percentage as described in section 5.2

$$\begin{aligned}&Voltage = 3.220\\&Percentage = -178,332 * 3,220^4 + 2504,83 * 3,220^3\\&+ -13077,1 * 3.220^2 + 30155,2 * 3,220 + -25957,3 = 9,3624\end{aligned} \tag{B.1}$$

$$\begin{aligned}&Voltage = 4.210\\&Percentage = -178,332 * 4,210^4 + 2504,83 * 4,210^3\\&+ -13077,1 * 4,210^2 + 30155,2 * 4,210 + -25957,3 = 100,9455036\end{aligned} \tag{B.2}$$

$$\begin{aligned}&Range = 100,9455036 - 9,3624 = 91,583101\\&Desired\,range = 100\\&Slope = 100/91,583101 = 1,0919045\\&Y\ offset = 0 - (1,0919045 * 9,3624) = -10,2228467\\&Mapping = (1,0919045 * input) - 10,2228467\end{aligned} \tag{B.3}$$

Appendix C

Dashboard layout

This part of the appendix contains screenshots of the final version of the dashboard including some sample visualizations.

Unimatrix 52
A Bluetooth asset localization system
Dashboard
@2023 By Menke Veerman as a Graduation Project at the University of Twente
Github

Figure C.1: The login screen of the dashboard

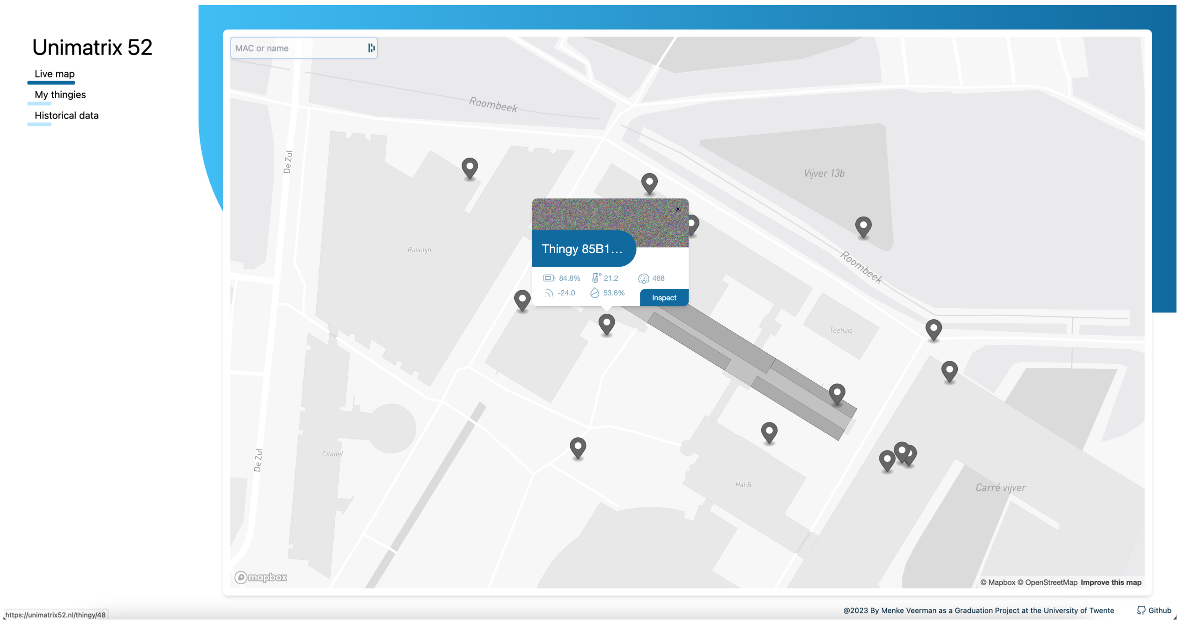


Figure C.2: The live map of the dashboard, here Thingies will more around live. They can be clicked to inspect them. If they have been active within the last 10 seconds, the marker will color blue.

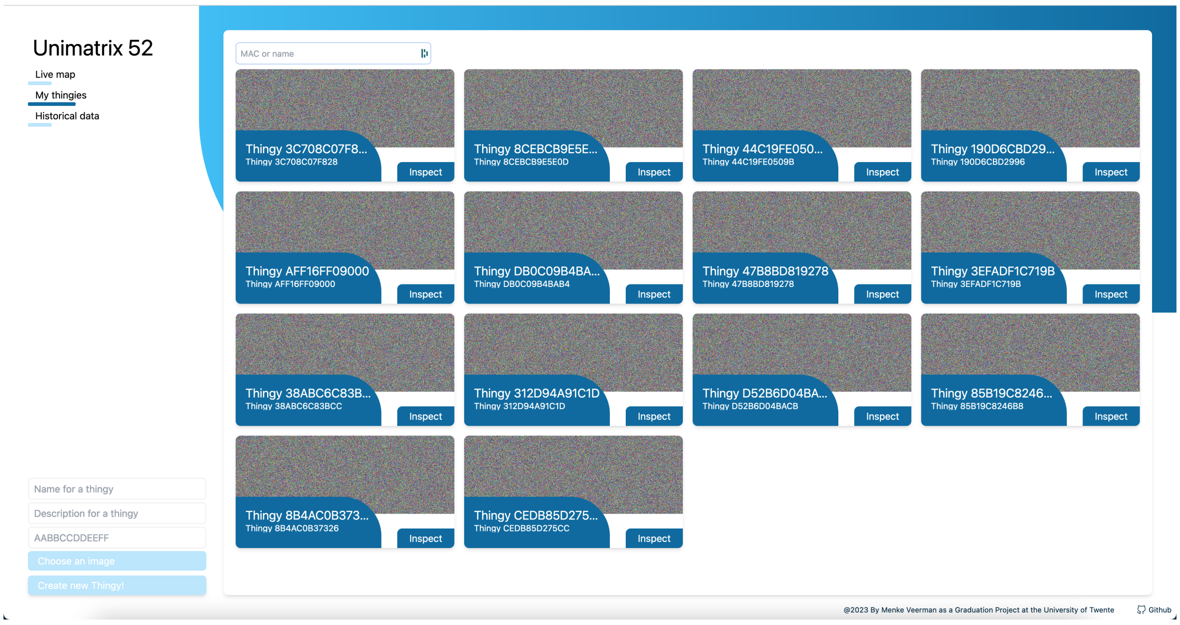
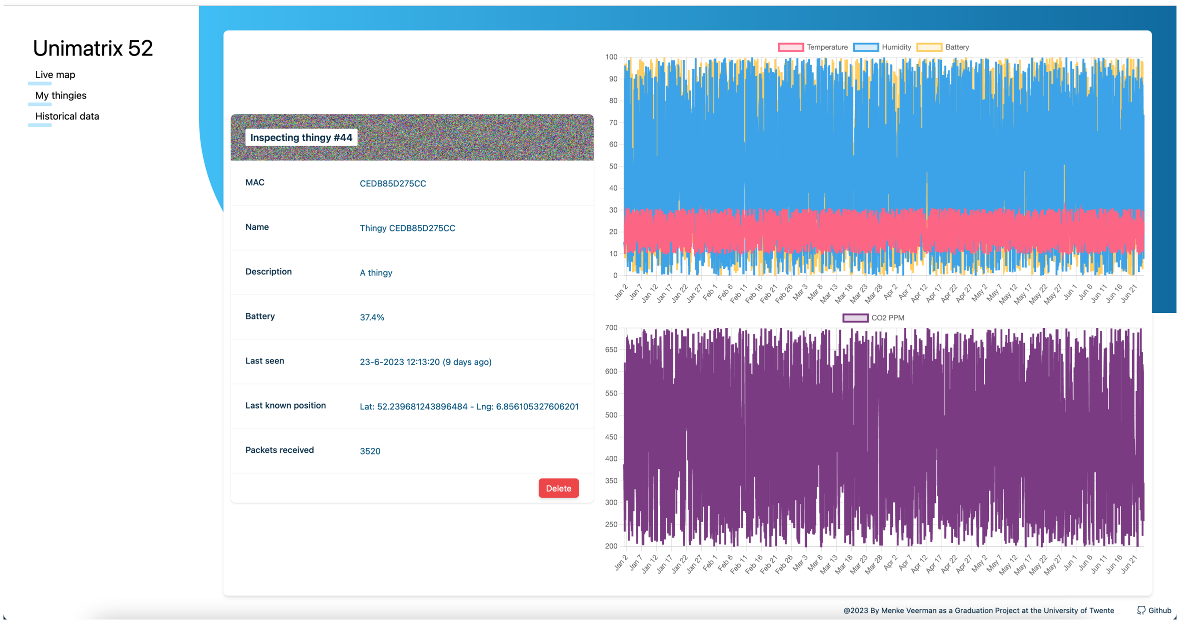


Figure C.3: The overview page containing all Thingies. Here thingies can be further inspected or added.



Unimatrix 52

Live map

My thingies

Historical data

Inspecting thingy #44

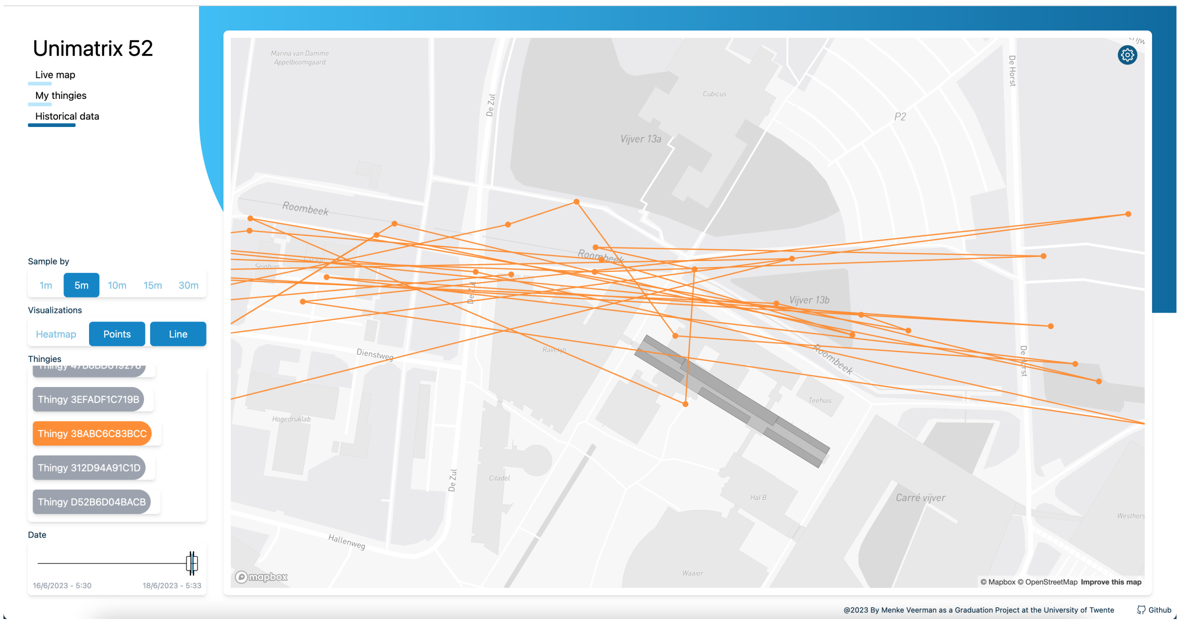
MAC	CEDB85D275CC
Name	Thingy CEDB85D275CC
Description	A thingy
Battery	37.4%
Last seen	23-6-2023 12:13:20 (9 days ago)
Last known position	Lat: 52.239681243896484 - Lng: 6.856105327606201
Packets received	3520

Delete

@2023 By Menke Veerman as a Graduation Project at the University of Twente

Github

Figure C.4: The inspection page of a Thingy. Here the individual data of a thingy can be seen including the last known information.



Unimatrix 52

Live map

My thingies

Historical data

Sample by

1m [✓ 5m] 10m 15m 30m

Visualizations

Heatmap [✓ Points] [✓ Line]

Thingies

Thingy 3EFADF1C719B

Thingy 38ABC6C83BCC

Thingy 312D94A91C1D

Thingy D52B6D04BACB

Date

16/6/2023 - 5:30 18/6/2023 - 5:33

@2023 By Menke Veerman as a Graduation Project at the University of Twente

Github

Figure C.5: The historical page of a Thingy using downsampled data for 10 minutes. The bottom left-hand corner has a slider where the time range can be defined. Hovering over a point shows the information of that (downsampled) packet.

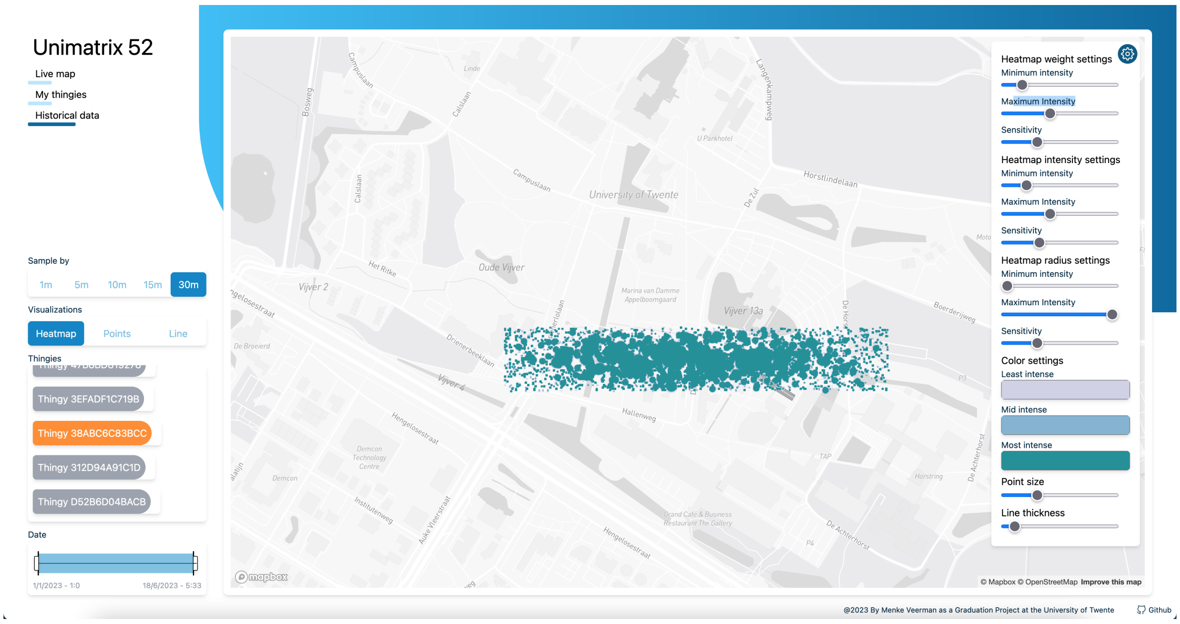


Figure C.6: The heatmap of downsampled data of 30 minutes. The top right-hand corner shows a configuration screen where visualizations can be altered.

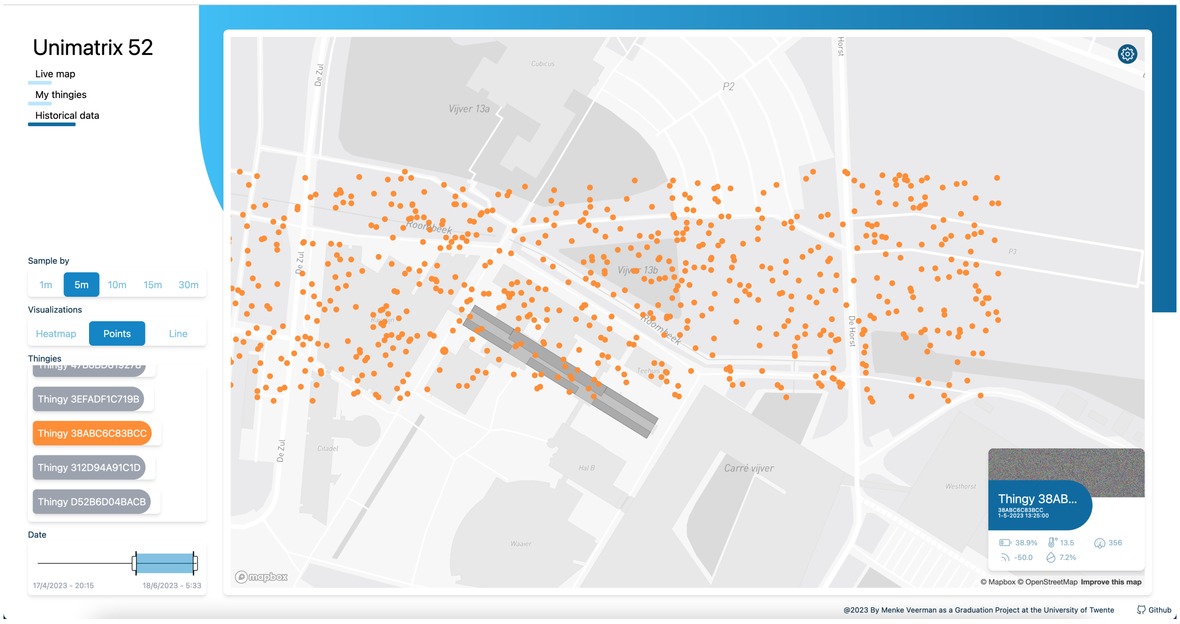


Figure C.7: The individual downsampled packets. They can be hovered for further inspection.

Appendix D

Tests

D.1 Specifications

Most k6 tests were run from the 'testmachine' to the server. See the versions of the specific software run in

D.1.1 Testmachine

- CPU: Intel i7-4980HQ
- RAM: 16GB DDR3-1600
- Network: Gigabit Ethernet
- OS: MacOS Big Sur 11.6.6
- K6 version: v0.45.0

D.1.2 Server

- CPU: Intel 12100F
- RAM: 16GB DDR4-3200
- Network: Gigabit Ethernet
- Storage: 1TB SN750 WD Black
- Docker version: 23.0.4
- OS: Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-73-generic x86_64)

D.1.3 Raspberry Pi

- Model: Raspberry Pi 4gb
- RAM: 4GB
- Network: Gigabit Ethernet
- Docker version: 23.0.5
- OS: Raspbian 5.15.84-v8+

D.2 Test configurations

D.2.1 K6

The script used for K6 along with the results can be found in the GitHub repository at testing/scripts/[virtual_users]_[test_name].js.

Directly (7.1.2)

- Source: Testmachine
- Destination: Server
- Network: Local, Gigabit ethernet
- URL: http://unimatrix52.nl:3000
- Virtual users: 5, 10, 25, 50, 100, 250, 500, 1000, 2500, 5000, 10000
- Runs per 'Virtual users': 2

Via proxy (7.1.2)

- Source: Testmachine
- Destination: Server
- Network: Local, Gigabit ethernet
- URL: http://unimatrix52.nl
- Virtual users: 5, 10, 25, 50, 100, 250, 500, 1000, 2500, 5000, 10000

- Runs per ’Virtual users’: 2

Cache test (7.1.2)

- Source: Testmachine
- Destination: Server
- Network: Local, Gigabit ethernet
- URL: http://unimatrix52.nl:3000
- Virtual users: 5, 10, 25, 50, 100, 250, 500, 1000, 2500, 5000, 10000
- Runs per ’Virtual users’: 2
- Options: Cache: disabled

D.2.2 Raspberry Pi

Single consumer (7.2.2)

- Source: Testmachine
- Manager runs on: Testmachine
- Packets pushed to Redis: 100000
- (Location) consumer on: Raspberry Pi
- Iteration speed: 0 (as fast as possible)
- Consumer count/replicas: 1
- Read sizes: 10, 100, 1000, 2500, 5000, 7500
- Runs per ’Read sizes’: 3

Two consumers (7.2.2)

- Source: Testmachine
- Manager runs on: Testmachine
- Packets pushed to Redis: 100000

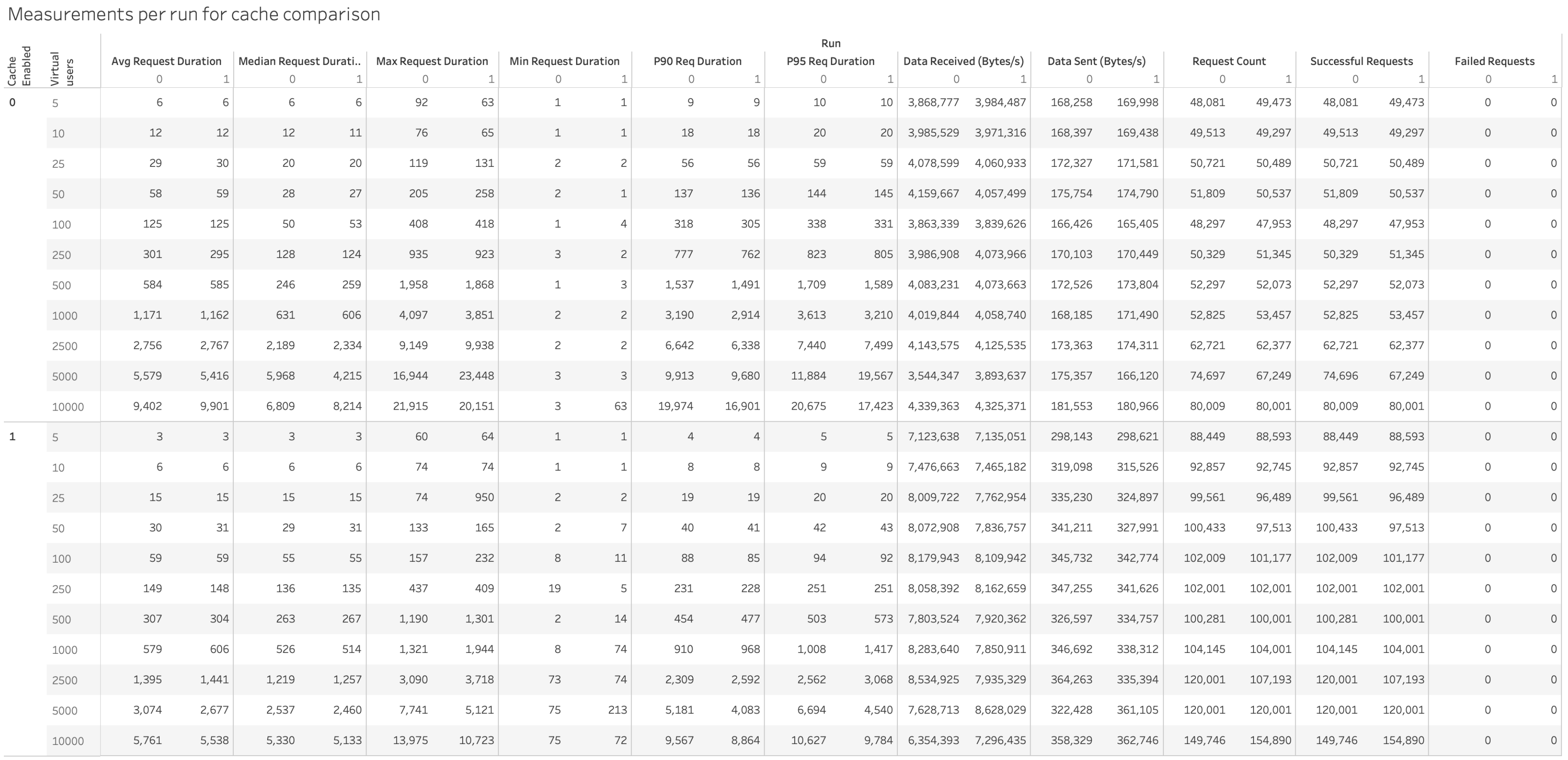
- (Location) consumer on: Raspberry Pi
- Iteration speed: 0 (as fast as possible)
- Consumer count/replicas: 2
- Read sizes: 10, 100, 1000, 2500, 5000, 7500
- Runs per ’Read sizes’: 3

Consumer validation (7.2.2)

- Source: Testmachine
- Manager runs on: Testmachine
- Packets pushed to Redis: 200
- (Location) consumer on: Raspberry Pi
- Iteration speed: 1000ms / 1Hz
- Consumer count/replicas: 2
- Read size: 10

D.3 Raw results

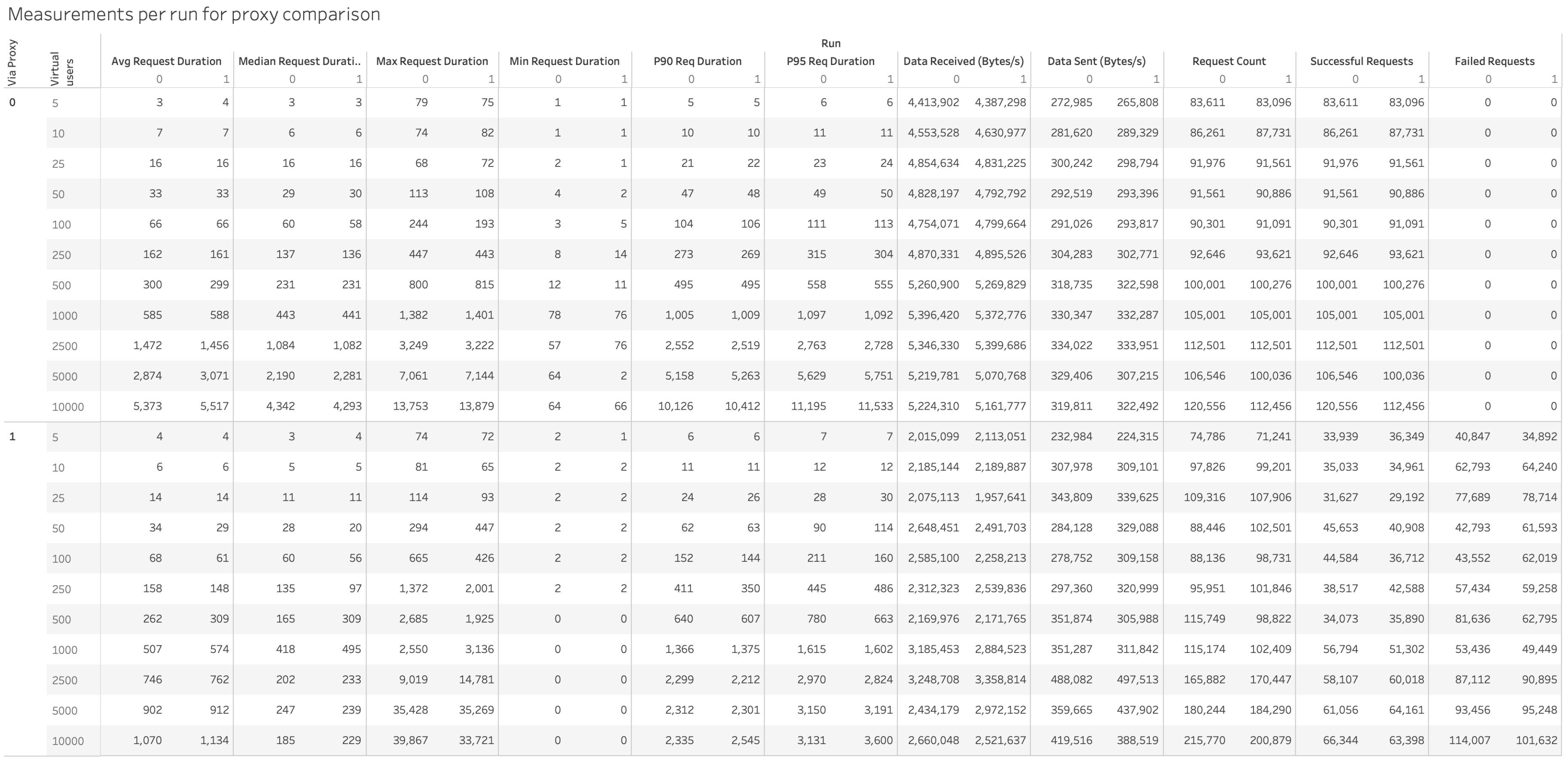
All other raw results are available in the testing/results folder on GitHub.



Measurements per run for cache comparison

													Run										
Cache Enabled	Virtual users	Avg Request Duration		Median Request Durati..		Max Request Duration		Min Request Duration		P90 Req Duration		P95 Req Duration		Data Received (Bytes/s)		Data Sent (Bytes/s)		Request Count		Successful Requests		Failed Requests	
		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	5	6	6	6	6	92	63	1	1	9	9	10	10	3,868,777	3,984,487	168,258	169,998	48,081	49,473	48,081	49,473	0	0
	10	12	12	12	11	76	65	1	1	18	18	20	20	3,985,529	3,971,316	168,397	169,438	49,513	49,297	49,513	49,297	0	0
	25	29	30	20	20	119	131	2	2	56	56	59	59	4,078,599	4,060,933	172,327	171,581	50,721	50,489	50,721	50,489	0	0
	50	58	59	28	27	205	258	2	1	137	136	144	145	4,159,667	4,057,499	175,754	174,790	51,809	50,537	51,809	50,537	0	0
	100	125	125	50	53	408	418	1	4	318	305	338	331	3,863,339	3,839,626	166,426	165,405	48,297	47,953	48,297	47,953	0	0
	250	301	295	128	124	935	923	3	2	777	762	823	805	3,986,908	4,073,966	170,103	170,449	50,329	51,345	50,329	51,345	0	0
	500	584	585	246	259	1,958	1,868	1	3	1,537	1,491	1,709	1,589	4,083,231	4,073,663	172,526	173,804	52,297	52,073	52,297	52,073	0	0
	1000	1,171	1,162	631	606	4,097	3,851	2	2	3,190	2,914	3,613	3,210	4,019,844	4,058,740	168,185	171,490	52,825	53,457	52,825	53,457	0	0
	2500	2,756	2,767	2,189	2,334	9,149	9,938	2	2	6,642	6,338	7,440	7,499	4,143,575	4,125,535	173,363	174,311	62,721	62,377	62,721	62,377	0	0
	5000	5,579	5,416	5,968	4,215	16,944	23,448	3	3	9,913	9,680	11,884	19,567	3,544,347	3,893,637	175,357	166,120	74,697	67,249	74,696	67,249	0	0
	10000	9,402	9,901	6,809	8,214	21,915	20,151	3	63	19,974	16,901	20,675	17,423	4,339,363	4,325,371	181,553	180,966	80,009	80,001	80,009	80,001	0	0
1	5	3	3	3	3	60	64	1	1	4	4	5	5	7,123,638	7,135,051	298,143	298,621	88,449	88,593	88,449	88,593	0	0
	10	6	6	6	6	74	74	1	1	8	8	9	9	7,476,663	7,465,182	319,098	315,526	92,857	92,745	92,857	92,745	0	0
	25	15	15	15	15	74	950	2	2	19	19	20	20	8,009,722	7,762,954	335,230	324,897	99,561	96,489	99,561	96,489	0	0
	50	30	31	29	31	133	165	2	7	40	41	42	43	8,072,908	7,836,757	341,211	327,991	100,433	97,513	100,433	97,513	0	0
	100	59	59	55	55	157	232	8	11	88	85	94	92	8,179,943	8,109,942	345,732	342,774	102,009	101,177	102,009	101,177	0	0
	250	149	148	136	135	437	409	19	5	231	228	251	251	8,058,392	8,162,659	347,255	341,626	102,001	102,001	102,001	102,001	0	0
	500	307	304	263	267	1,190	1,301	2	14	454	477	503	573	7,803,524	7,920,362	326,597	334,757	100,281	100,001	100,281	100,001	0	0
	1000	579	606	526	514	1,321	1,944	8	74	910	968	1,008	1,417	8,283,640	7,850,911	346,692	338,312	104,145	104,001	104,145	104,001	0	0
	2500	1,395	1,441	1,219	1,257	3,090	3,718	73	74	2,309	2,592	2,562	3,068	8,534,925	7,935,329	364,263	335,394	120,001	107,193	120,001	107,193	0	0
	5000	3,074	2,677	2,537	2,460	7,741	5,121	75	213	5,181	4,083	6,694	4,540	7,628,713	8,628,029	322,428	361,105	120,001	120,001	120,001	120,001	0	0
	10000	5,761	5,538	5,330	5,133	13,975	10,723	75	72	9,567	8,864	10,627	9,784	6,354,393	7,296,435	358,329	362,746	149,746	154,890	149,746	154,890	0	0

Figure D.1: The raw measurements for the cache comparison 7.1.2.



Measurements per run for proxy comparison

		Run																					
Via Proxy	Virtual users	Avg Request Duration		Median Request Durati..		Max Request Duration		Min Request Duration		P90 Req Duration		P95 Req Duration		Data Received (Bytes/s)		Data Sent (Bytes/s)		Request Count		Successful Requests		Failed Requests	
		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	5	3	4	3	3	79	75	1	1	5	5	6	6	4,413,902	4,387,298	272,985	265,808	83,611	83,096	83,611	83,096	0	0
	10	7	7	6	6	74	82	1	1	10	10	11	11	4,553,528	4,630,977	281,620	289,329	86,261	87,731	86,261	87,731	0	0
	25	16	16	16	16	68	72	2	1	21	22	23	24	4,854,634	4,831,225	300,242	298,794	91,976	91,561	91,976	91,561	0	0
	50	33	33	29	30	113	108	4	2	47	48	49	50	4,828,197	4,792,792	292,519	293,396	91,561	90,886	91,561	90,886	0	0
	100	66	66	60	58	244	193	3	5	104	106	111	113	4,754,071	4,799,664	291,026	293,817	90,301	91,091	90,301	91,091	0	0
	250	162	161	137	136	447	443	8	14	273	269	315	304	4,870,331	4,895,526	304,283	302,771	92,646	93,621	92,646	93,621	0	0
	500	300	299	231	231	800	815	12	11	495	495	558	555	5,260,900	5,269,829	318,735	322,598	100,001	100,276	100,001	100,276	0	0
	1000	585	588	443	441	1,382	1,401	78	76	1,005	1,009	1,097	1,092	5,396,420	5,372,776	330,347	332,287	105,001	105,001	105,001	105,001	0	0
	2500	1,472	1,456	1,084	1,082	3,249	3,222	57	76	2,552	2,519	2,763	2,728	5,346,330	5,399,686	334,022	333,951	112,501	112,501	112,501	112,501	0	0
	5000	2,874	3,071	2,190	2,281	7,061	7,144	64	2	5,158	5,263	5,629	5,751	5,219,781	5,070,768	329,406	307,215	106,546	100,036	106,546	100,036	0	0
	10000	5,373	5,517	4,342	4,293	13,753	13,879	64	66	10,126	10,412	11,195	11,533	5,224,310	5,161,777	319,811	322,492	120,556	112,456	120,556	112,456	0	0
1	5	4	4	3	4	74	72	2	1	6	6	7	7	2,015,099	2,113,051	232,984	224,315	74,786	71,241	33,939	36,349	40,847	34,892
	10	6	6	5	5	81	65	2	2	11	11	12	12	2,185,144	2,189,887	307,978	309,101	97,826	99,201	35,033	34,961	62,793	64,240
	25	14	14	11	11	114	93	2	2	24	26	28	30	2,075,113	1,957,641	343,809	339,625	109,316	107,906	31,627	29,192	77,689	78,714
	50	34	29	28	20	294	447	2	2	62	63	90	114	2,648,451	2,491,703	284,128	329,088	88,446	102,501	45,653	40,908	42,793	61,593
	100	68	61	60	56	665	426	2	2	152	144	211	160	2,585,100	2,258,213	278,752	309,158	88,136	98,731	44,584	36,712	43,552	62,019
	250	158	148	135	97	1,372	2,001	2	2	411	350	445	486	2,312,323	2,539,836	297,360	320,999	95,951	101,846	38,517	42,588	57,434	59,258
	500	262	309	165	309	2,685	1,925	0	0	640	607	780	663	2,169,976	2,171,765	351,874	305,988	115,749	98,822	34,073	35,890	81,636	62,795
	1000	507	574	418	495	2,550	3,136	0	0	1,366	1,375	1,615	1,602	3,185,453	2,884,523	351,287	311,842	115,174	102,409	56,794	51,302	53,436	49,449
	2500	746	762	202	233	9,019	14,781	0	0	2,299	2,212	2,970	2,824	3,248,708	3,358,814	488,082	497,513	165,882	170,447	58,107	60,018	87,112	90,895
	5000	902	912	247	239	35,428	35,269	0	0	2,312	2,301	3,150	3,191	2,434,179	2,972,152	359,665	437,902	180,244	184,290	61,056	64,161	93,456	95,248
	10000	1,070	1,134	185	229	39,867	33,721	0	0	2,335	2,545	3,131	3,600	2,660,048	2,521,637	419,516	388,519	215,770	200,879	66,344	63,398	114,007	101,632

Figure D.2: The raw measurements for the proxy comparison 7.1.2.

Appendix E

State of the art images

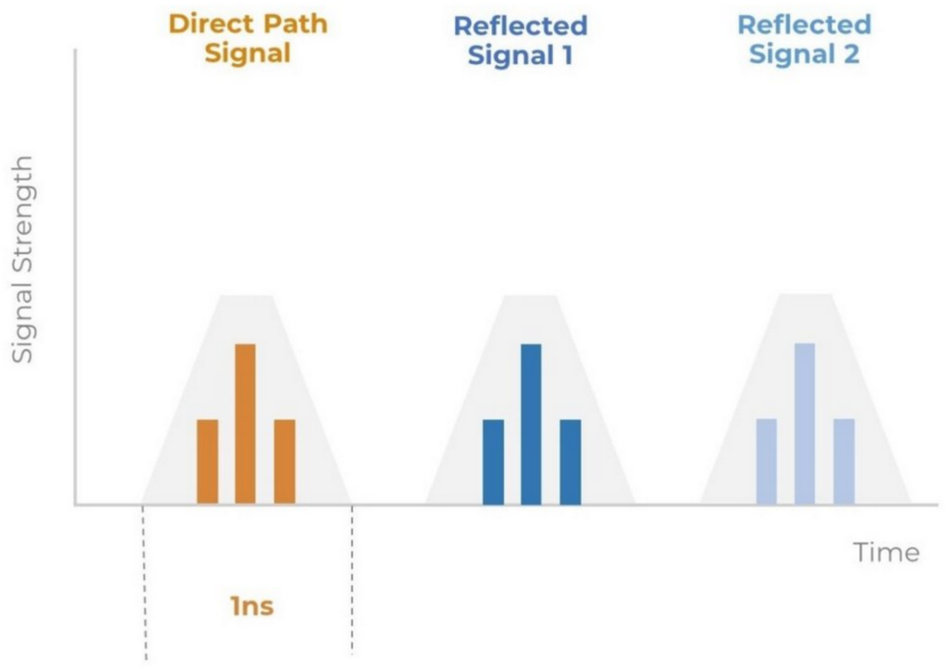


Figure E.1: UWB arrival of a signal [42]

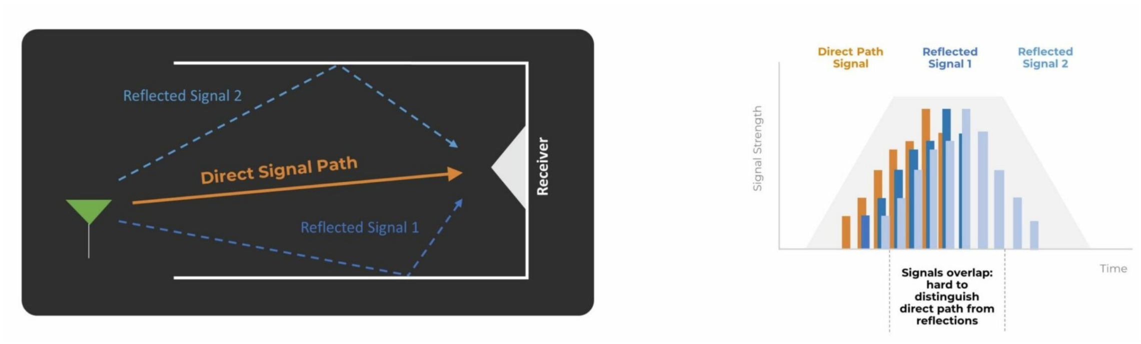


Figure E.2: Reflections of a signal [42]

E.1 Localization techniques

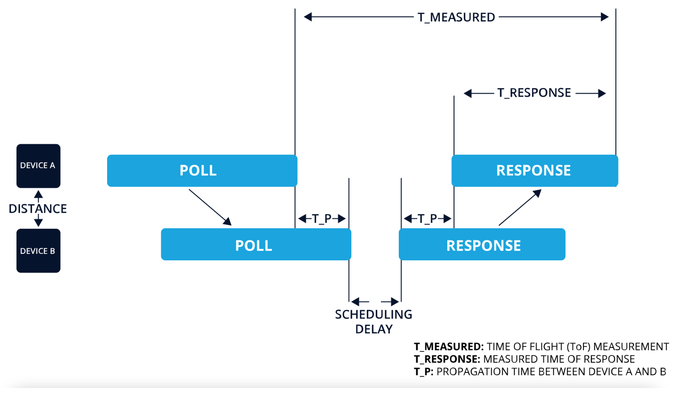

T_MEASURED
T_RESPONSE
DEVICE A
DISTANCE
DEVICE B
POLL
POLL
RESPONSE
RESPONSE
T_P
T_P
SCHEDULING DELAY
T_MEASURED: TIME OF FLIGHT (ToF) MEASUREMENT
T_RESPONSE: MEASURED TIME OF RESPONSE
T_P: PROPAGATION TIME BETWEEN DEVICE A AND B

Figure E.3: TDoA localization [40]

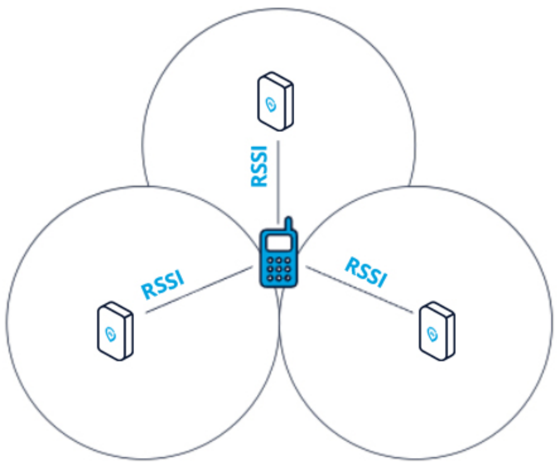


Figure E.4: RSSI localization [40]

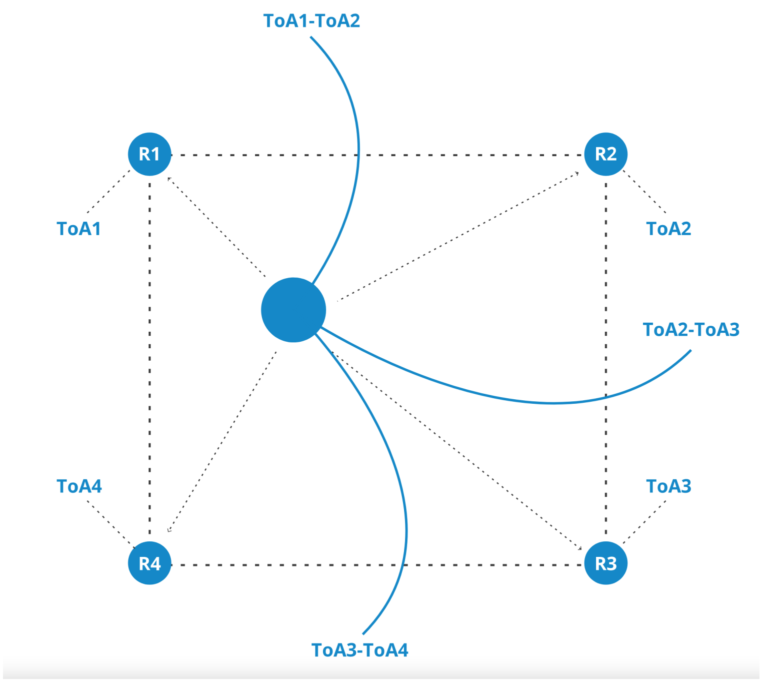


Figure E.5: TWR localization [40]

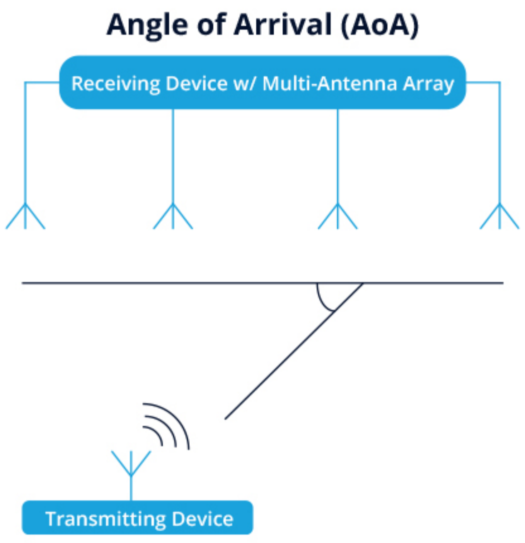


Figure E.6: AoA localization [40]