University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

BluePIL 2.0: Toward Automated Deployment and Operation

Alain Küng
Zurich, Switzerland
Student ID: 18-717-017

Supervisor: Dr. Bruno Rodrigues, Eder Scheid, Simon Tuck (LiveAlytics), Prof. Dr. Burkhard Stiller
Date of Submission: 27.03.2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Jüngste Studien auf dem Gebiet der Lokalisierung und Identifizierung von Personen in Innenräumen haben gezeigt, dass es verschiedene Möglichkeiten gibt, Personen in Innenräumen zu verfolgen. Solche Lokalisierungstechniken lassen sich in aktive oder passive Innenraum-Lokalisierungssysteme unterteilen, die eine direkte bzw. keine direkte Verbindung mit dem System benötigen. Die Corona-Pandemie hat gezeigt, wie wichtig und nützlich die Anwendung solcher Techniken ist, um mögliche Infektionen von Personen aufzuspüren und die Ausbreitung einzudämmen. In anderen Bereichen ist sie auch für Marketinganalysen von Nutzen. In einer kürzlich an der UZH durchgeführten Masterarbeit wurde in Zusammenarbeit mit Livealytics ein passives Lokalisierungs- und Identifizierungstool zur Lokalisierung von Geräten nur mit Hilfe von erkannten Bluetooth-Signalen untersucht. Die Lösung mit der Bezeichnung *BluePIL* hatte ein funktionsfähiges System mit mässiger Genauigkeit geliefert. Die zeitaufwendige Bereitstellung, Kalibrierung und Analyse forderte eine zweite Version, auf die sich diese Arbeit konzentriert. Unter Verwendung neuer Hardware, zusätzlicher mobilen Ladegeräten und durch Hinzufügen eines Analysetools mit einstellbaren Parametern wird eine neue Version entwickelt, genannt *BluePIL 2.0*. Die durchgeführten Experimente haben gezeigt, dass durch *BluePIL 2.0* das System verbessert wird, indem es die manuellen Schritte zum Einsatz des Systems reduziert und die Mobilität verbessert. Die Zeit bis zum Start des Systems beträgt im Durchschnitt 7,32 Sekunden. Darüber hinaus trägt das Analysetool zu einem besseren Verständnis der durchgeführten Innenraumszenarien bei, indem es eine aktualisierende Darstellung der verfolgten Geräte anzeigt und die Möglichkeit bietet, den Pfadverlustkoeffizienten des Lokalisierungsalgorithmus während der Datenerfassung zu kalibrieren. Die echtzeitnahe Darstellung ist so eingestellt, dass sie sich innerhalb von 0,5 Sekunden aktualisiert. Im Allgemeinen erfüllt *BluePIL 2.0* die Ziele in einer anforderungsbezogenen Bewertung.

Abstract

Recent studies in the field of localization and identification of persons indoors have shown that there are various possibilities to track individuals indoors. Such localization techniques can be divided into either active or passive indoor localization systems, needing a direct or no direct connection with the system respectively. The corona pandemic unveiled the importance and benefit of applying such techniques to trail possible infections of persons to contain the spread. In other fields, it has also been valuable for marketing analysis. In a recent master's thesis at UZH, a passive localization and identification tool for tracking devices using only passive Bluetooth signals was explored in collaboration with Livealytics. The solution called *BluePIL* provided moderate accuracy, which can be considered to work despite the underlying naturally noisy data. The time consuming deployment, calibration and analysis yield a second version of it, called *BluePIL 2.0*, on which this thesis focuses on. Using new hardware, additional power-banks and by adding a new analysis tool with adjustable parameters, a new version is developed. The experiments have shown that *BluePIL 2.0* improves the system by reducing the manual steps to deploy the system and improving mobility. The time to launch the system is 7.32 seconds on average. In addition, the analysis tool contributes to a better understanding of the indoor scenarios by displaying an updating plot of the tracked devices and providing the option to calibrate the path loss coefficient of the localization algorithm during data collection. The near-real-time plot is set to update within 0.5 seconds. In general, *BluePIL 2.0* met the goals in a requirements-based evaluation.

Acknowledgments

I would like to thank Dr. Bruno Rodrigues, Eder Scheid and Prof. Dr. Burkhard Stiller at the Communication Systems Group of the University of Zurich to give me the opportunity to write this bachelor thesis. Especially, I want to express my gratitude towards Dr. Bruno Rodrigues for his support by discussing complications, answering questions and provide valuable feedback. I also want to thank Nora Fischer and Raffael Mogicato for giving me their insightful advice and support that helped me accomplishing this task.

Contents

Chapter 1

Introduction

1.1 Motivation

Indoor people tracking is a controversial activity with positive and negative aspects: it can be considered harmful concerning the aspect of loss of privacy of tracked people, but it can also be considered positive in use cases that aim at an increased organization and efficiency of public spaces. In this sense, different indoor tracking technologies can offer a higher or lower level of privacy in the tracking activity. As such, several works in literature have explored these different technologies that range from the perception of wireless signals emitted by electronic devices (either IEEE 802.11 [1], or Bluetooth [2]), to the use of cameras [3], as well as the reflection of emitted lights (such as LiDAR) [4], among several others listed in literature surveys [5, 6].

One of the technologies that enable indoor tracking activity is Bluetooth, which is similarly based on the emission and reception of wireless signals as IEEE 802.11 (*i.e.,* WiFi) protocols, but has particular benefits in the sense of passive tracking of signals by not having an address randomization feature. In other words, the identification of people is based on the identification of the mobile devices it carries, and as standard communication protocols, these devices are also identified by addresses. Therefore, the unique identification of these addresses allows the unique identification of its bearer. In the case of the WiFi protocol, such identification is made impossible due to the usage of using address randomization protocols in specific cases. For example, when a mobile device performs a lookup of which access points are available, request packets are issued containing the address of the requesting device. The address scrambling strategy is used to prevent the device from being uniquely identified when requesting the availability of access points. However, such a strategy is explicitly used in 802.11 protocols and not in Bluetooth-based ones.

This thesis extends existing work by proposing improvements to the deployment and operationalization of the tool *BluePIL* [2], which has been proposed as an academic alternative to investigate the possibility of tracking mobile devices based solely on the emission of passive Bluetooth Low-Energy (BLE) signals. *BluePIL* proposes a streaming data architecture and location algorithm based on multilateration positioning data, which uses

information obtained from Ubertooth trackers [7]. Ubertooth allows access to lower layers of Bluetooth protocols at low cost, which are typically hidden in off-the-shelf Bluetooth modules..

1.2 Thesis Goals

Based on the existing *BluePIL* system, the goal is to deploy and expand the *BluePIL* system to further enhance the systems deployment time, analysis and parameter adjustments. The three main goals are:

- **Automation**: Automating the deployment and operation of *BluePIL* is a major goal for this thesis to significantly reduce the deployment time for future data collection.
- **Path-loss coefficient**: The n path-loss coefficient is part of the Log-Distance Path Loss Model [8] used to model radio signal decay over distance. Therefore, it is a vital part for the calculations of the *BluePIL* systems device positioning [2]. It is set before the data collection is running and can not be change during the procedure. For time reduction and better scenario adjusting, it is necessary to be able to modify the n coefficient on the fly during the data collection.
- **Analysis**: For better analysis during data collection, near-real-time plotting of the collected data should enhance the understanding of the scenario, indicating how many devices are sensed and positioned, providing visual representations of the devices and therefore viable information to adjust the n coefficient.

1.3 Methodology

This thesis includes two parts. The first part involves the underlying fundamentals and related works inspiring the *BluePIL* system. In addition, design considerations are made on how to improve the system in order to achieve the goals. In the second part, the considerations will be transitioned into practise by implementing the system with new hardware and updated software. The final part involves evaluating the evolution of the system and whether it is fulfilling the thesis goals.

1.4 Thesis Outline

The following Chapter 2 introduces background knowledge about the underlying *BluePIL* system and its concepts. Chapter 3 describes the requirements to meet the goals and shows the architecture behind the different components that should fulfill those. Chapter 4 includes information about the hardware and software implementation of *BluePIL 2.0*. Chapter 5 explains and discusses the different experiments taken to evaluate *BluePIL 2.0*. The last Chapter 6 sums up the thesis in a final consideration including a summary, conclusions and future work.

Chapter 2

Fundamentals

2.1 Background

This chapter contains brief explanations of the underlying technologies used in *BluePIL 2.0*. In addition, [2], which introduced the first version of the system, and other related work are described in short. Further reading of these specific papers is recommended to deepen the knowledge if needed.

2.1.1 Bluetooth

BluePIL, the baseline work which this thesis is built upon, relies on Bluetooth signals for device localization and identification [2]. Therefore, it is a vital part to understand the underlying technology, the designed data streaming pipeline, in order to enhance it.

The core concept on which this work is based on is BLE. In principle, Bluetooth was developed with the intention of wirelessly connecting electronic devices over short distances [9] and was composed by the Bluetooth Special Interest Group (SIG) [10]. The organisation is composed of members who work together to develop new Bluetooth specifications for worldwide Bluetooth technology standards. Bluetooth first version was published in Core Specification Version 1.0 [9]. It is often titled as *Classic Bluetooth*, however, the official name is Bluetooth Basic Rate / Enhanced Data Rate (BTBR/EDR). In later stages, BLE was added to the Bluetooth Core specification version 4.0. It is a more energy-efficient implementation in contrast to the previous versions at the expense of less data traffic and complexity for modern Internet-of-Things (IoT) devices.

Both specifications are unique protocol stacks that are not compatible. The latest version of the Bluetooth Core Specification is Version 5.2 from 2019. To put the usage of Bluetooth into perspective, the total annual Bluetooth device shipments is estimated to grow over 6.4 Billion by 2025 [11]. 70% of it is approximated to be peripheral devices, such as various wireless earbuds, sensors, and smart lights. The other 30% contain laptops, tablets and smartphones.

In general, both protocol stacks are operating in 2.4 GHz unlicensed industrial, scientific and medical (ISM) band and are utilizing the Frequency Hopping Spread Spectrum (FHSS) scheme over 79 channels at a hopping rate of 1600 hops/s using a Gaussian Frequency-Shift Keying (GFSK) modulation for binary message encoding [9]. Depending on which class of device and protocol is used, the transmission range is between 0.1 and 100 metres.

2.1.2 Project Ubertooth

Concerning localization and identification of Bluetooth devices, *BluePIL* requires a specific form of frequency capture, *i.e.,* a sniffer. The solution proposed and implemented is a fully open source project called Project Ubertooth, which provides a hardware and a software package for wireless development [12]. Ubertooth is capable of accessing the lower layers of the Bluetooth protocols, including parts of the Bluetooth address. Utilizing the fact that Bluetooth connections use a hopping pattern, Ubertooth collects data using a single transceiver, that mimics this behaviour, such that eavesdroping frequencies is made possible. It offers experimentation with both BLE and BTBR/EDR though the hardware can only receive at a maximum of 1 Mbit/s [13], which makes it not fully compatible with BTEDR that can raise up to 2.1 Mbit/s [14].

It is relatively cheaply available for 125$ [15] (at the time of writing this thesis) considering that alternatives, such as the *Ellisys Bluetooth Explorer 400*, can easily cost up to 20'000$, making such a device unsuitable for research purposes [14]. The processed data can be sent conveniently to the host via USB if you have a USB 2.0 port running [12].

2.1.3 Kalman Filter

Bayes filters are often used in location estimation of 2D or 3D objects, in which sensors are used [16]. It aims to estimate if an object can be at location x if the past sensor measurements have been $z_1, z_2, , z_t$ for all locations x. A commonly use of Bayes filter is the Kalman Filter, which is also used and implemented in *BluePIL* [2]. It helps to smooth the collected data and eliminate outliers.

2.1.4 Multilateration

To geometrically estimate the position of a object in space, the process multilateration is used by *BluePIL* [2]. The distance of a object can be determined by the distance measures between at least three points. The measurements can be mathematically broken down into a determined linear system of equations which provide solutions for any order of multilateration problems [17].

2.1.5 The Log-Distance Path Loss Model

A frequently used model for radio signal decay over distance is the Log-Distance Path Loss Model [8]. With a logarithmic function it can estimate the decay of a signal over distance. The model can be simplified such that it only consist of the parameters received signal strength at 1 m RSS_C, the path-loss coefficient n and the distance d.

$$RSS(d) = RSS_C - 10n\log(d) \tag{2.1}$$

The path-loss coefficient n depends on the environment setting. As example, for grocery stores, it is often set to 1.8.

2.2 Related Work

This section will first describe existing work on both Bluetooth device localization and identification and its applications followed by the work which this thesis is build upon. Due to the SARS-CoV-2 outbreak, a mobile application has been developed for contact tracing. The DP-3T application was deployed in Switzerland as the official contact tracing application for the pandemic [18]. In [19], an airport can be estimated as crowded with pedestrians by using Bluetooth and WiFi in combination. With repeated inquiry scans, they give information on the number of Bluetooth devices available around the sensors. In [20], they use Bluetooth to identify individual devices, given that they can be discovered. Using a Bluetooth beacon system, [21] proposes a system for detecting the presence of individuals in smart homes, enabling energy savings, assistance for elderly or impaired people, and personalizing of the smart home experience. In [22], they develop a passive, energy-efficient indoor tracking and pattern recognition system based on a managed BLE network. More specifically, the system relies on passively monitoring the position of a network of BLE tags. In order to reliably locate moving objects indoors, multiple protocols were implemented, including broadcasting the unique identifiers and current timestamp of moving objects.

BluePIL [2], the main work this thesis is build upon emerged out of a InnoSuisse-founded cooperation between Communication Systems Group UZH (CSG) and Livealytics which offer Live marketing analytics, IoT devices, and other services [23, 24]. The goal was to find a completely passive localization and identification of Bluetooth devices, as most approaches until then required cooperation between the system and the target devices. Livealytics approach uses passively measure Wi-Fi signals to collect marketing statistics. The problem, however, was that Wi-Fi Media Access Control (MAC) randomisation complicates the device identification step [2]. Since randomization does not exist in Bluetooth, the cooperation spawned the idea of *BluePIL*. *BluePIL*: Fully Passive Identification and Localization of Bluetooth Devices in Near-Real-Time has been implemented by Cyrill Halter as a proof-of-concept in his master-thesis [25] in 2020. Four Ubertooth sensor nodes placed in each corner of a room collect data of the incoming Bluetooth signals. During the collection, the data is continuously sent to a sink where the installed *BluePIL* system

then calculates the location of a device using a quadlateration algorithm merged with a Log-Distance Path Loss Model. Due to the signals having a unique identifier, different devices can be sensed and localized within a room. After the system has run for several minutes or hours, the collected data can then be analyzed in a second step, such that the devices can be plotted into a graph to visualise their positions. For better accuracy of the noisy observation, the Kalman Filter is applied. The experiments provided an accuracy of around 1 m in a 12 m^2 area and 1.4 m in a 25 m^2 area.

Chapter 3

System Design

3.1 Requirements

The following main requirements for the second version can be obtained from the thesis goals:

R1 Automating the deployment and operation of *BluePIL* to reduce the number of manual steps required in its bootstrap.

R2 The system must be able to plot the data near-real-time.

R3 Parameters must be adjustable for better visualisation of the plot.

- The n path-loss coefficient should be adjustable during data collection.

R4 Prototype must enable mobility of experiments allowing to be easily deployable in different scenarios.

3.2 Assumptions

As this thesis is based on the first *BluePIL* version, assumptions are similar but not quite the same [25]. It is necessary to simplify the use within the requirements, therefore the following assumptions are made or adopted:

- The number of sensors available for the system remains at four.
- The localization of the devices are within the predefined area of the sensors.
- The localization problem remains in two dimensions, *i.e.* in a planar space.

3.3 BluePIL 2.0

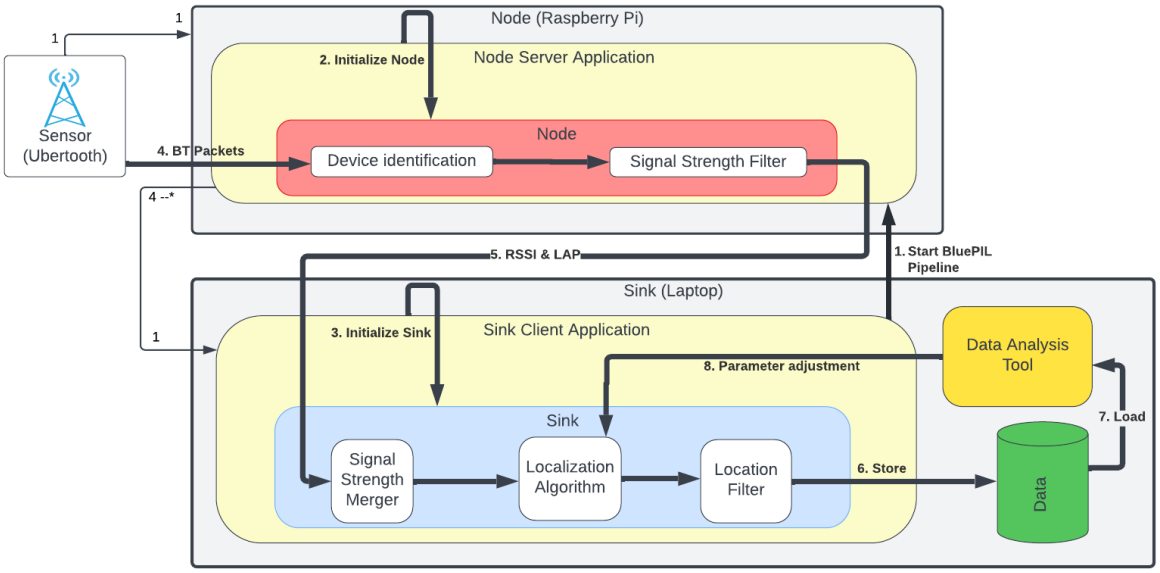

Figure 3.1: *BluePIL 2.0* high-level system architecture

This section describes the general architecture of *BluePIL 2.0*. The main system architecture of the first *BluePIL* version [2] is carried over into the second. Figure 3.1 shows the high-level system architecture of *BluePIL 2.0*.

As the previous *BluePIL* system is a streaming multi-component architecture, implementing individual components can be done with sufficient flexibility, so the architecture itself will not be compromised. Different configuration options are available for structuring the deployed system for physical or virtual processing entities. The previous version was based on a distributed node-sink setup, in which data is sent from many nodes to a single physical sink via network.

In detail, it works as a streaming data pipeline starting in step 4 of the Figure 3.1. The sensors collect the passively captured Bluetooth packets and port them into the nodes. The node then identifies and filters the signal strengths of the devices. In step 5, the received signal strengths and the device lower address part (LAP) are forwarded to the sink component. There, the different signals received from each sensor of the same device are first merged in the Signal Strength Merger, then used to calculate the device location in Localization Algorithm and filtered in Location Filter before they get stored into a file in step 6.

In addition to the streaming pipeline, we designed a server-client architecture for *BluePIL 2.0* to be able to operate both the sink and the node from the client side. Step 1 in Figure 3.1 sends the starting command from the Sink Client Application, which both trigger step 2 and step 3 to initialize and start the pipeline. This is further explained in Section 3.4. Additionally, a data analysis tool was added to the *BluePIL 2.0* system. The tool is designed to fetch data directly from the storage in step 7 and plotting the data

in near-real-time described in Section 3.5. To calibrate the localization algorithm, the n coefficient value is adjustable during data collection, represented in step 8. This addition is embedded into the analysis tool, further explained in Section 3.5.1.

3.4 Automation

For the next section, it is assumed that the sink and the nodes already have a *BluePIL* instance installed and the devices are connected to the same network. Also, a remote Secure Shell (SSH) connection configuration has been established between the sink and the nodes. Furthermore, the location configuration file *bp.json* is adapted to the scenario [25].

To reduce the amount of steps required in its bootstrap, the sequence of actions that start the whole process of data collecting has to be analyzed. The sequence diagram in Figure 3.2 summarises the process for starting the *BluePIL* system, stopping before data collection begins. The *manual* tag is referencing the actions taken by human to deploy the application, further referenced as operator, to initialize the next step.

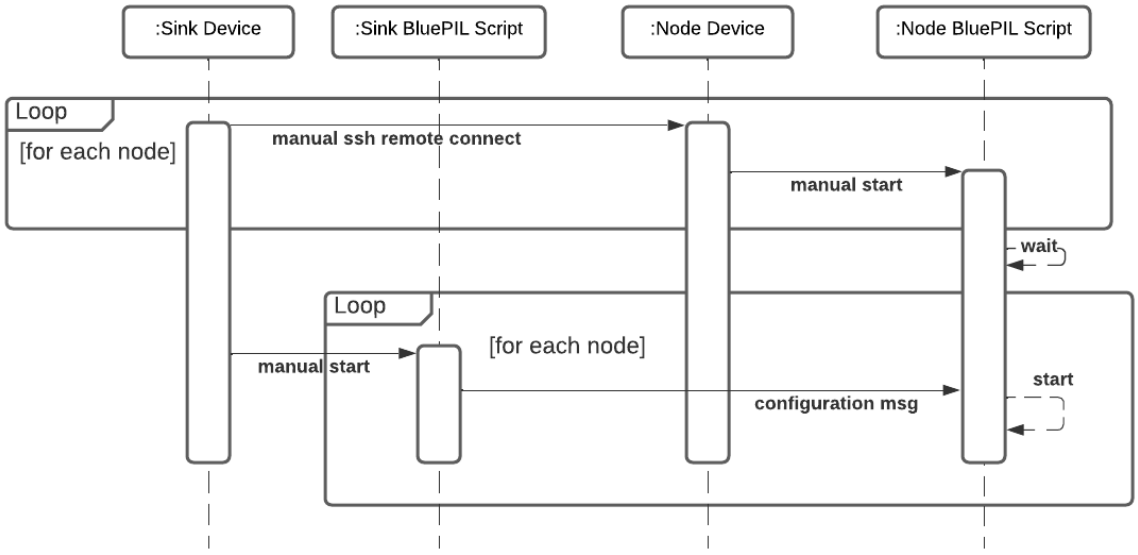

Figure 3.2: Sequence diagram presenting the starting process for the sink and the nodes in the first *BluePIL* version

First, the sink (*e.g.,* a laptop) must be manually connected to each node by the operator via a SSH connection, as the nodes are small computers without a screen. These low-power units were chosen for economic and physical reasons, as they improve margins by being cheap, consume less power and are small in size. Secondly, the operator has to start the node script on each node manually within the SSH connection. Via the command line window the *BluePIL* python scripts get executed for each node, starting their pipeline. The nodes are then designed to be waiting for the sink script to send a

configuration message, which includes both the starting signal and connection parameters. The advantage of this master-slave architecture is that many nodes can be deployed simultaneously, while the sink retains control over the devices it connects to. Lastly, the actor has to start the bootstrap script on the sink device. The sink sends the configuration message to the nodes. Then the *BluePIL* system is ready for operation and already starts collecting data from received Bluetooth signals automatically. The streaming pipeline does not need any further instructions, except the stopping message. Considering that *BluePIL* was implemented for four nodes, following nine manual steps are necessary to get the system up and running.

1. Four times connecting to the nodes via a SSH connection.
2. Four times starting the node script on every node.
3. Start the sink script on the sink device.

To meet the first requirement, these nine manual steps must be significantly reduced, as each step and each additional node requires more time to deploy the system. It would also be practical to be able to restart the application without having to perform all the steps again. There is a start-up function in the last step, to keep control of the system, while the connection via SSH and the start of the individual scripts on the nodes have to be automated. These also take the most time for deployment. The next section introduces the new architecture design to address this issues.

3.4.1 Architecture

The location configuration of the nodes was done using a *json* file in which the location is manually adjusted. This required the operator to open the source code. We have therefore proposed a graphical user interface (GUI) that reads the *json* file and, for practical reasons, the variables can be adjusted within the GUI before the *BluePIL* system starts. After this, the system is ready for the *BluePIL* system start. Since the *BluePIL* system is encapsulated into the node and sink component, a server-client architecture can be integrated. In this case, nodes have an additional layer that acts as a listening mini-server, while the sink receives a client layer. The listening mini-server on the nodes were designed to be a socket implementation. The device is bound to a dedicated port, waiting for a client, *i.e.,* the sink, to make a connection request to the socket. On the client side, if the connection was accepted by the node, a socket is successfully created and the client can communicate through it. After the sink has prepared every socket connection to each deployed node, the operator can then request the mini-servers to start their node script, while the sink proceeds to the start the sink script, starting the *BluePIL* pipeline. Figure 3.3 shows the new sequence diagram of the second version. In addition to the previous components, two other entities, namely *Sink Client* and *Node Server*, are added to the sequence. The *Sink Client* is started by the operator in a first step. Then, the sink connects to each of the listening mini-servers in a loop. After each server has responded, the operator can manually start the application. Each mini-servers launch their *BluePIL* script on command, followed by the sink starting their *BluePIL* script automatically. In

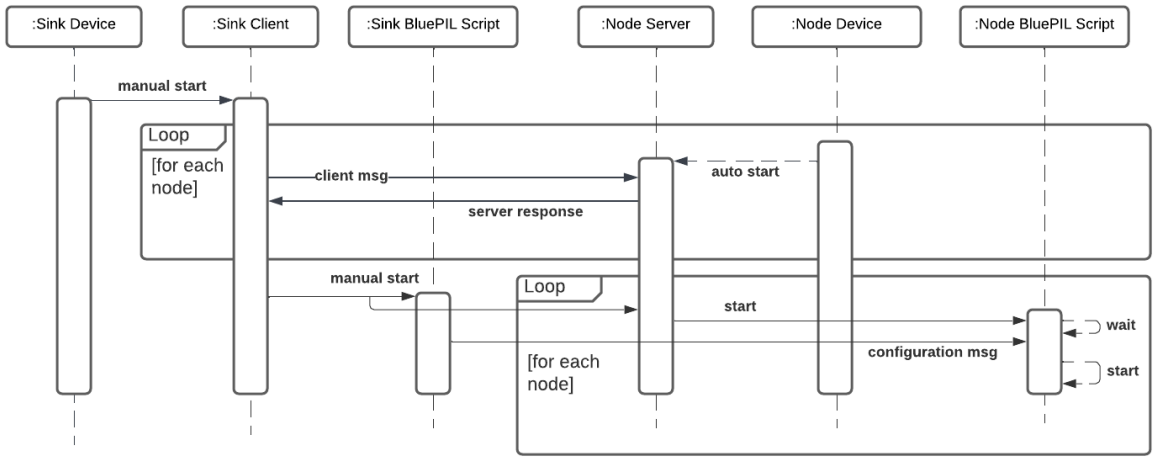

Figure 3.3: Sequence diagram presenting the starting process for the sink and the nodes in version 2.0

general, the underlying *BluePIL* system is untouched. The *Node Servers* would be designed to start automatically upon the *Node Device* boot. Overall, the required manual steps that the operator has to take in the second version of the *BluePIL* system are two, again assuming the configuration has been done beforehand.

1. Initialize the sink to connect to each node.
2. Send the starting message to nodes.

3.5 Analysis

The stored data from the first *BluePIL* version was designed to be analysed by another entity, either by uploading the data to a cloud environment or by writing a script that retrospectively displays the data collection [2]. This setup never allowed the observation of the current situation during data collection, yet a near-real-time plot would have the benefit of observing the data during acquisition. Furthermore, it would allow to calibrate the localization algorithm for different scenarios where the nodes are deployed, as the parameter is environment dependent.

First, we have a look at how the data is stored. The sink of the *BluePIL* system stores the calculated locations and identifications of devices during the collection on the sink device. In Table 3.1, there is a sample of a data collection which constantly updates during the procedure with x, y being the coordinates of the device in planar space with a timestamp when the data was received and the devices *LAP*. The data format is a comma-separated values (CSV) file and is saved in near-real-time, taking into account the sending, calculating and storing of the data, each time all nodes acquire frequency of the same device.

x	y	timestamp	LAP
0.6035623435939044	0.36680444883406643	2022-01-06 23:42:32.303156	8ddf78
0.6035623435939044	0.36680444883406643	2022-01-06 23:42:32.548230	8ddf78
0.26798773861436137	0.7739191039012463	2022-01-06 23:42:33.832163	8ddf78
0.3883890511365365	0.6116109488635245	2022-01-06 23:43:29.927560	8ddf78
0.38840517546606995	0.6115948244538033	2022-01-06 23:43:06.138758	8ddf78
0.3883794731077689	0.6116205267829772	2022-01-06 23:43:08.235642	8ddf78
0.3883833007018804	0.6116166991356671	2022-01-06 23:43:12.681228	8ddf78
0.38733908283768886	0.3844429934106364	2022-01-06 23:44:00.542858	8ddf78
0.3884104662037166	0.3884104662037166	2022-01-06 23:44:12.696629	8ddf78
0.3891682550061562	0.3891682522434169	2022-01-06 23:44:15.579850	8ddf78
0.38840122647199393	0.3884012264719939	2022-01-06 23:45:31.702248	8ddf78

Table 3.1: Sample of a data collection by *BluePIL* stored in positions.csv

Taking advantage of the fact that the data can be retrieved out of the CSV file during the process without interrupting the *BluePIL* system, we propose a GUI that polls the positions of the devices from the CSV file in near-real-time and displays them graphically for visual analysis. The GUI is structured having a constantly updating plot while also providing parameter fields to adjust the plot and its data. The application is a stand-alone process to be started by the operator of the *BluePIL* system. Therefore, it can be started or closed without affecting the *BluePIL* system and causing more workload to fall on the process.

The plot for *BluePIL 2.0* was designed to represent the two dimensional planar space of a chosen scenario. An example scenario, which [25] chose for his experiment was a room with a 4.2 m x 2.9 m area. In each corner of the room, one of the nodes of the *BluePIL* system was installed. Since *BluePIL* calculates the location of the devices inside the room as 2D coordinates, the plot of the *BluePIL 2.0* data analysis tool therefore also represents the space in 2D. The *Uberteeth* and the data points were designed to have visual representation for better analysis.

Data polling is key for near-real-time plotting, as such the polling rate of the CSV file has to be set to a valid interval. A polling rate that is too high affects the performance of the GUI because it recalculates all the stored data for plotting, but is closer to near real-time. A polling rate that is too low delays the real-time display so that it is no longer considered near real-time. Since one of the purposes of the system is to analyse indoor behaviour from a marketing perspective, e.g. how long the average person stands in front of which shelf, the interval should be adopted to human walking indoors. For this reason, we have set an estimated rate of 0.5 seconds, which is manageable for an ordinary device used as a sink. Furthermore, considering the data collection can be run during longer periods, the CSV file can overload the application. This can be tackled by reducing the deployment time or splitting the data in different CSV file according to time slots.

Considering the fact that data is at a pre-defined upstreaming time interval and the devices can be constantly in motion, we consider a field to set a number for the amount of the last retrieved data points of a particular device we want to plot. Drastically speaking, more

than 100 data points of the last device location may not be suitable for a near-real-time representation for the current location.

Following the retrospective data analysis procedure of [25], a Kalman Filter was applied to the data set in order to eliminate outliers in both static and dynamic device behaviour. Therefore, it was also added to the GUI in *BluePIL 2.0* as an option for further experimenting. It is applied to the whole set of data points. Furthermore, to calibrate the localization algorithm for positioning the devices, we propose a *True Point* that represents the actual position of the device that simplifies the process of aligning the algorithm with the device position, which is further discussed in Section 3.5.1. In addition, the prediction mean between the selected last points provides better accuracy in an inherently noisy data set. In order to detect devices in the environment, it is advantageous to have an additional overview of the sensed but not yet positioned devices, similar to the third experiment in [25]. The last sections described are designed to not interact with the *BluePIL* system itself, except the calibration of the localization algorithm. Therefore, n path-loss coefficient, which is the key parameter to adjust, yields its own section.

3.5.1 Path-Loss Coefficient

The path-loss coefficient n is a variable in the localization algorithm of *BluePIL* [2]. For the first version of *BluePIL*, it is set to a fixed value within the *BpSink* class in the _init_quadlateration function when the *BpSinkStream* class is initialized. From there, it gets passed into the *position* class, directly into the *BpQuadlateration* class as a class variable where the algorithm computes and returns the (x y) position of the device. The n variable corresponding to the path-loss coefficient is, therefore, not mutable once chosen in the *BpSink* though the coefficient is known to be dependent on the environment. [26] conducted experiments in a office environment in which n varied between 1.35 and 1.98. To fully meet the third requirement, the n variable needs to be adjustable during the data collection. The listing 3.1 show a summarized code snippet of the n coefficient that gets passed along. The highlighted parts reference the n coefficient.
In the *BpQuadlateration* class, which is initialized for each device, the function *quadlaterate* uses the n coefficient for calculating the position of a device each time all four sensors send the respective signals.

```
1  class BpSink:
2      ...
3      def _init_quadlateration(self):
4          ...
5          self._sink_stream = BpSinkStream(streams_of_streams,
               positions, 1.8, (0.5, 0, 0.5, 0))
6
7  class BpSinkStream:
8      def __init__(self, streams_of_streams_for_laps, positions, n,
           initial_x):
9          ...
10         self._n = n
11         ...
```

```
12      def _merge_rssi_streams(self, lap):
13          ...
14          merged = sz.Stream.merge_rssi_streams(*self.
                _streams_for_laps[lap]) \
15              .sliding_window(2,  return_partial=False) \
16              .map(mergeOutputsToPositioningInput) \
17              .position(self._positions, self._n, self._initial_x)
18
19  class position(sz.Stream):
20      ...
21      def __init__(self, upstream, positions, n, initial_x, *args, **
            kwargs):
22          self._quad = BpQuadlateration(*positions, n)
23
24  class BpQuadlateration:
25      ...
26      def __init__(self, p_A1, p_A2, p_A3, p_A4, n):
27          self.n = n
```

Listing 3.1: Code summary of the n coefficient usage in the *BluePIL* system until initalization of the *BpQuadlateration*

Since the coefficient n is only used in the class *bp_quadlateration*, we opt to remove the n variable from the class *BpSink*, the class *BpSinkStream* and the class *position* to reduce the variable initialisation to the class *BpQuadlateration* only. The proposed solution utilises the common *bp.json* configuration file which also used to configure the location and the ip addresses of the nodes. In this file, we add the parameter n value. This parameter is then read each time the function *quadlaterate* is called so that the positioning always has the latest adjustment of the n coefficient.

Chapter 4

Implementation

This chapter introduces the implementation of *BluePIL 2.0*, taking into account the design decisions made in chapter 3.

4.1 Software

BluePIL was implemented as *Python 3* application, which provides a wide range of libraries and utilities [25]. For *BluePIL 2.0*, the same framework was used to further extend the source code. The enhanced code consists of additional layers to the master-slave architecture of node and sink by encapsulating the *BluePIL* process on the node and restructuring the nodes as listening mini-server. The sink acts as a client that calls the nodes to start the *BluePIL* process, reducing the initialisation time. The data analysis application is not part of the *BluePIL* system itself, but can be operated as an independent sub-element of the system to analyze the current scenario, also implemented in *Python 3*.

4.1.1 Node Configuration

The mini-server socket described in Section 3.4 was implemented using the *Python 3* in-built module *socket*. The benefit of the module was that it allowed quick implementation of a bi-directional communication. It was implemented by opening a Transmission Control Protocol (TCP) port on the node, listening for a client to request connection. It starts by the client, *i.e.* the sink, requesting connection. After the connection is established, the data is sent between the sink and the node as *UTF-8* encoded data packets. These packets are decoded into *Python* string objects for further processing once received by either the node or the sink. Listing 4.1 shows a code snippet out of the node mini-server. This part is responsible for encoding and decoding client messages as well as for following commands according to the message received. Following commands are implemented for the client: *START_NODE, EXIT, KILL.*

```
1  def dataTransfer(self,conn):
2      # big loop to send/receive data
3      while True:
4          # receive
5          data = conn.recv(1024)
6          data = data.decode('utf-8')
7          # split data to separate command from the rest of data
8          data_message = data.split(' ', 1)
9          command = data_message[0]
10         # kills process if new client connects, ensures if something
               goes wrong the process can still be restarted
11         if self.node_process.is_alive():
12             self.node_process.terminate()
13         # different commands
14         if command == 'START_NODE':
15             reply = 'Starting node..'
16             conn.sendall(str.encode(reply))
17             self.node_process = multiprocessing.Process(target=
                   start_node)
18             self.node_process.start()
19             break
20         elif command == 'EXIT':
21             break
22         elif command == 'KILL':
23             self.socket.close()
24             break
25         else:
26             reply = 'Unknown Command'
27             conn.sendall(str.encode(reply))
28     conn.close()
```

Listing 4.1: Data transfer loop of *BluePIL 2.0*

The commands *EXIT* and *KILL* terminate the communication between the sink and the node. *KILL* additionally shuts down the socket on the node mini-server such that no further connections can be established. The *START_NODE* command starts the *BluePIL* node process of the streaming pipeline. Using the *Python* package *multiprocessing*, the mini-server first connects the node starting function to a process entity in line 17 in Listing 4.1 and then starts the process in line 18. The connection to the client is then terminated, but the server continues to wait for a new connection while the node process of the *BluePIL* streaming pipeline starts to wait for the sink to connect. The underlying first version of *BluePIL*, as explained in Section 3.3, is now in progress. If the data collection is done and the system is stopped by the stop signal of the sink, the node process is terminated. The sink client has the possibility to now reconnect to the listing mini-server and restart the process once again. If something went wrong during communication, lines 11 and 12 ensure that the node process is safely terminated as soon as a new client has established a connection. Then, the server is ready to start the node process again. Figure 4.1 shows the different states that the application goes through after it gets started.

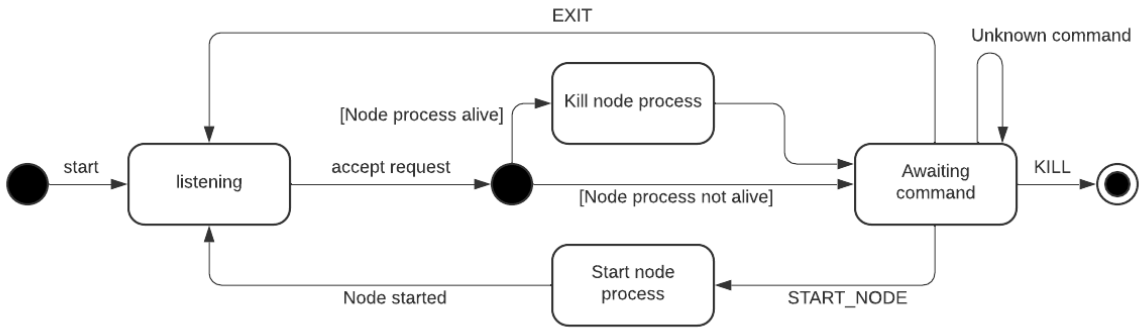

Figure 4.1: State diagram of the listening mini-server

4.1.2 Sink Configuration

The sink device is the counterpart for the node server in the server-client architecture as described in Section 3.4. It is implemented as the main component to be run by the operator to start the *BluePIL* process.

In the first *BluePIL* version, the node configuration message sent to the node had to be manually adjusted by entering the values into the *bp.json* file. Listing 4.2 contains the updated *json* file used for *BluePIL 2.0*, where the node positions and system mode were configured. The newly added parts were the *n_value* in line 3, *number_of_nodes* in line 32 and *true_point* in line 33. The importance of these new values are explained in the Sections 4.1.3 and 4.1.4. Node 1-4 represent the ip and x y location of the deployed node devices used in *BluePIL*. As proposed in Section 3.5, in *BluePIL 2.0* the *bp.json* was connected to a GUI explained next.

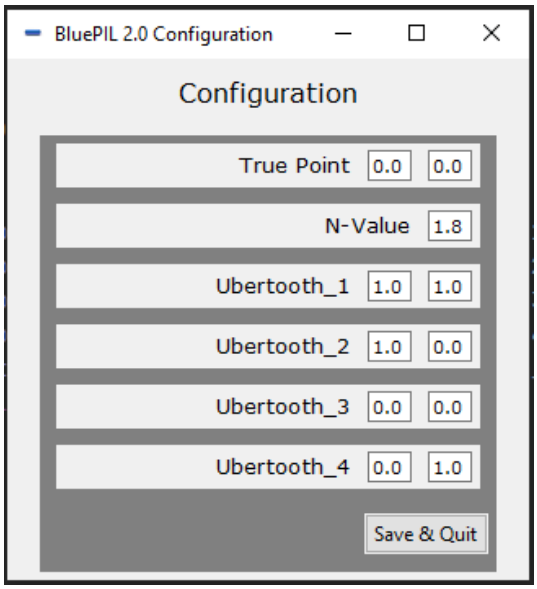

Figure 4.2: Configuration GUI

The GUI is shown in Figure 4.2. It was implemented with *tkinter* [27]. *tkinter* is a standard *Python 3* package for GUI implementation and provides numerous options for interacting with a user by providing buttons, labels, text, text boxes and more. The GUI window, also known as the root window, is a 320 x 320 pixel sized window. Inside the window is the so called frame. Here we implemented several widgets, *i.e.*, the input fields, labels and the button for the configuration. The *Ubertooth_1-4* represent the nodes that are going to be deployed in the scenario with their corresponding Internet Protocol (IP) address and x, y positions in the 2D space. The n_value corresponds to setting the n coefficient. In addition, the *True Point* corresponds to a device actual position to be captured. When the application starts, the GUI reads the *json* configuration file and displays the current settings. The operator of the *BluePIL 2.0* system has the option to change the configurations by writing into the input fields. The *Save & Quit* button was implemented such that it saves the input field values into the *json* configuration file, such that the changes for the systems are applied and passed on.

```
{
    "mode": "POSITIONING",
    "n_value": 1.8,
    "node1": {
        "ip": "192.168.1.101",
        "loc": [
            1.0,
            1.0
        ]
    },
    "node2": {
        "ip": "192.168.1.102",
        "loc": [
            1.0,
            0.0
        ]
    },
    "node3": {
        "ip": "192.168.1.103",
        "loc": [
            0.0,
            0.0
        ]
    },
    "node4": {
        "ip": "192.168.1.104",
        "loc": [
            0.0,
            1.0
        ]
    },
    "true_point": [
        0.0,
        0.0
    ]
}
```

Listing 4.2: bp.json configuration message

Listing 4.3 shows the function that gets called upon pressing the *Safe & Quit* button. It first reads the file into the *conf* variable. Using the *json* package from *Python 3*, we convert the contents of the file into a *Python* dictionary. The contents of the dictionary are edited in lines 4-11. Each input field value is called and converted to a *float* representation as returned in *string*. Then they are stored in the dictionary. In lines 13 to 15, the dictionary is converted and written to the file *bp.json*.

```
def update_json(self):
    conf_file = open("bp.json", "r")
    conf = json.load(conf_file)
    conf_file.close()

    conf['node1']['loc'] = [float(self.d1_cord_one.get()), float(
        self.d1_cord_two.get())]
    conf['node2']['loc'] = [float(self.d2_cord_one.get()), float(
        self.d2_cord_two.get())]
    conf['node3']['loc'] = [float(self.d3_cord_one.get()), float(
        self.d3_cord_two.get())]
    conf['node4']['loc'] = [float(self.d4_cord_one.get()), float(
        self.d4_cord_two.get())]
    conf['true_point'] = [float(self.true_point.get()), float(self.
        true_point_two.get())]
    conf['n_value'] = float(self.n_value.get())

    conf_file = open("bp.json", "w")
    json.dump(conf, conf_file, indent=4)
    conf_file.close()
    app.destroy()
```

Listing 4.3: Save values to *bp.json*

After the configuration stage is done, the client application of the client-server implementation gets started. The implementation is similar to the sink in Subsection 4.1.1 using the *socket* module from *Python 3* to communicate with the sink. The client iterates over every IP stored in the *bp.json* file, connecting to every node server deployed with a dedicated port. Upon successfully connecting, the application awaits a command line input. The commands are implemented analogously to the sink that receives the commands: START_NODE, EXIT, KILL.

Listings 4.4 shows the implementation of the node logic within the connection. This function is called after each connection between the nodes and the sink have been established. When *EXIT* and *KILL* are sent, the connection is terminated. The *START_NODE* command initializes the *BluePIL* sink process. First, the command is sent to the sink which starts the node process on the node server. Secondly, a timer guarantees that the nodes start before the sink, since according to [25], the nodes must be started first. Finally, the process *BluePIL* sink is initialised, which starts the streaming pipeline.

```
def run(self):
    while True:
        command = input('Enter your command: ')
        if command == 'EXIT':
            # Send EXIT request
            self.send(command)
            break
        elif command == 'KILL':
            self.send(command)
            break
        elif command == 'START_NODE':
            self.send(command)
```

```
13              time.sleep(2)
14              print('starting process')
15              start_run_sink()
16              print('Ending process, please restart Application')
17              break
18          else:
19              self.send(command)
20      self.close()
```

Listing 4.4: Sink logic for the client application

4.1.3 Data Analysis Tool for Device Location and Identification

For the data analysis tool described in Section 3.5, we implemented the GUI with *tkinter* [27], the in-build *Python 3* package that provide sufficient tools for visualisation. The tool is a 1350 pixel to 720 pixel sized window application. It contains six input sections within the application which represent the plotting parameters and the constantly updating plot of the received data of the *BluePIL* pipeline. Furthermore, it contains a dynamically updating overview of the sensed device in the area and two fields displaying the current amount of sensed and positioned devices respectively.

The plot was implemented with the *Matplotlib* [28] library. It has been embedded in the *tkinter* GUI as it has a suitable interface built in called *matplotlib.figure*. For the visualisation, we retrieve the data from memory, *i.e.* from the *positions.csv* file, by using the *pandas* [29] data analysis library to do the reading. The library itself was also used to plot the graph which is then added to the *matplotlib.figure*. Listing 4.5 shows for demonstration purposes a simplified and shortened version of the implemented parts dealing with the initialisation of the plot, which would otherwise require several pages to properly visualize them.

```
1  from matplotlib.figure import Figure
2  import pandas as pd
3
4  with open("bp.json") as f:
5      conf = json.load(f)
6  x_room_length = 0
7  y_room_length = 0
8  for i in range(1, 5):
9      sensor = conf[f'node{i}']
10     x_room_length = sensor["loc"][0] if sensor["loc"][0] >
           x_room_length else x_room_length
11     y_room_length = sensor["loc"][1] if sensor["loc"][1] >
           y_room_length else y_room_length
12 col_x = "x"
13 col_y = "y"
14 # room length
15 room_lim_x = (0, x_room_length)
16 room_lim_y = (0, y_room_length)
17 # plot padding
18 plot_padding = 0.5
```

```
19 plot_lim_x = (room_lim_x[0] - plot_padding, room_lim_x[1] +
       plot_padding)
20 plot_lim_y = (room_lim_y[0] - plot_padding, room_lim_y[1] +
       plot_padding)
21
22 df = pd.read_csv('positions.csv')
23 fig = Figure()
24 ax = fig.add_subplot()
25 df.plot(kind="scatter", legend=None, x=col_x, y=col_y, xlim=
       plot_lim_x, ylim=plot_lim_y, alpha=0.3, color="#e6194b", grid=
       True, marker=".", label="Predictions", ax=ax)
26 # plot figure into GUI
27 canvas = FigureCanvasTkAgg(fig, self)
```

Listing 4.5: Plot initialization into the GUI

Line 4-12 in Listing 4.5 refers to the configuration file *bp.json* sensor coordinates and finds the highest x y values to store them into the *x_room_length* and *y_room_length* respectively. Along with a set 0.5 *plot_padding*, they determine the x and y axes lengths for proper plot visualization. The data to plot is visible at line 22. The *matplotlib.figure* and the *subplot* interface for the *pandas* plot is initialized in line 23 and 24. Based on the tag *ax=ax*, the plot of the positions is added to the figure axes in line 25. Finally, the figure itself is added to the GUI in line 27 with the parameter *self* referring to the *tkinter* class frame holding the widgets.

The *matplotlib.animation.FuncAnimation* was used to implement the data polling discussed in 3. This class creates an animation by repeatedly calling a function. Passed parameter for this class are the figure to be animated and the function to be repeatedly called. Furthermore, the parameter *interval* is set to 500 which equals 500 milliseconds until the function is repeated again. The function to be called in embedded into the GUI class. The main goal is to update the plot with the newly retrieved data by calling *pandas* to re-read the file *positions.csv* and re-render the plot. The number of last retrieved points is programmed to be set in an input field and updated accordingly after 500 milliseconds. In addition, the update function is also responsible for the re-rendering of the following widgets explained in the next sections. The widgets can be seen in Figure 4.3 starting from Kalman Filter to *Sensed Devices*. The n coefficient implementation is further described in Section 4.1.4.

For less outliers, a Kalman Filter was implemented to be optionally applied upon ticking the check box as shown in Figure 4.3. As soon as the box is checked, the animation function will call the Kalman Filter class with the retrieved data as input. The output is then used as the new data set. The Kalman Filter class was implemented in the first version of *BluePIL* to smooth outliers in the data after the experiments. It was reused and integrated into the data analysis tool.

To plot all monitored devices at once, we implemented a *Plot All LAPs* check box that will, if triggered, prompt the animate function to plot not only the selected device but all of positioned devices collected in the *positions.csv* file. First, the animate function parses the different LAPs found in the data file. Then, for every device found, data points will be filtered into a dictionary, with each LAP as the key and the points as its value.

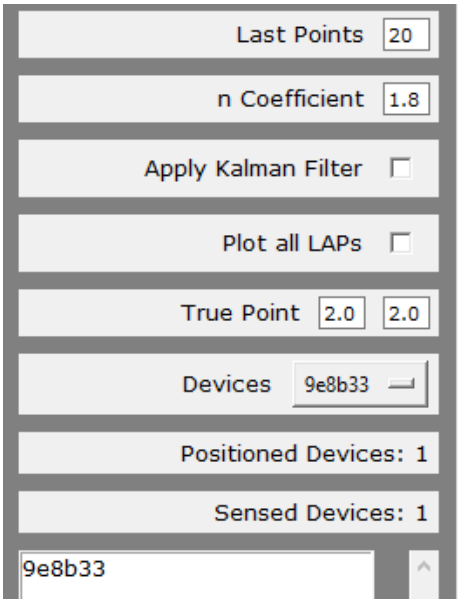

Figure 4.3: Parameter section of the data analysis tool

Furthermore, the selected data range to be displayed and the average, *i.e.* the *Prediction Mean*, will also be calculated before plotting. Finally, each device together with their *Prediction Mean* will be displayed on the graph each with one of the 22 distinctive colors implemented. The labels are updated accordingly.

The *True Point* is referring to the actual position of the device in a scenario. By reading the *bp.json* configuration file, the position of the point is added to the plot. There we implemented also the option to change the true point position if needed by writing into the input fields provided with the label *True Point*. The left input field referring to the x value and the right input field to the y value. The input is automatically adjusted with the next updating cycle.

For selective plotting of devices, we have implemented a drop-down menu where the user can select a LAP to be observed of the positioned devices. By using the *pandas unique()* function, the GUI determines the unique LAPs in a given data frame and updates the drop-down menu accordingly. Additionally, by filtering the *positions.csv* file by the LAP chosen, the animate function display the graph with the LAP data points. Furthermore, the sum of all positioned devices is shown to provide an overview.

Within the first version of the *BluePIL* pipeline, we implemented a small adaptation such that each time a data stream is registered by a sensor, the sink stores the corresponding LAP into a separate file called *laps.csv*. Listing 4.6 provides insight to the respective function. This function is called within the pipeline after a data packet is received from a deployed node. It is responsible for registering a potential newly sensed LAP and merges the streams once all nodes have started a respective stream. Line 3 of Listing 4.6 checks if a stream for this device already exists, else it is added to the *self._stream_for_laps* dictionary. Any additional stream for the same LAP not fall under this condition, as such it is called only once for each LAP. Therefore, we implemented in line 5 to 7 a file read and write section, where the newly found LAP gets appended into the *laps.csv* file. In

the GUI application, this file is read in the animation function so that the number and names of the individual LAPs captured but possibly not yet positioned are displayed as shown in Figure 4.2.

```
1  def _register_rssi_stream(self, lap, idx, stream):
2      print("Registering stream {0} for LAP {1}".format(idx, lap))
3      if lap not in self._streams_for_laps:
4          self._streams_for_laps[lap] = [None] * _NUM_UBERTEETH
5          csvfile = open(f'laps.csv', 'a')
6          csvfile.write(lap +'\n')
7          csvfile.close()
8      self._streams_for_laps[lap][idx] = stream
9      if None not in self._streams_for_laps[lap]:
10         self._merge_rssi_streams(lap)
```

Listing 4.6: Function of the *BluePIL* pipeline to register a data stream

4.1.4 Path-Loss Coefficient

The path-loss coefficient n needs to be adjustable as described in Section 3.5.1. The first version of *BluePIL* passed the n value through several classes though its only used in the *BpQuadlateration* class. Therefore, we erased the n value from the other classes and set the coefficient value within the main location calculation class. Additionally, we removed n as a parameter to be passed to initialize the class. Figure 4.7 shows the two functions of the *BpQuadlateration* class that use the n path-loss coefficient. In the initialization function, the class reads the value from the *bp.json* file, the common configuration file that was configured before by the operator. As the initialization and the calculation are not concurrent, the path-loss coefficient is read once again in line 14-16 for every function call to calculate the location of the device to maintain the latest adaptations. The adaptations can be done in the GUI, as shown in Figure 4.7, by writing the input into the input field of the *n Coefficient*. The animation function reads this input every interval and writes it into the common *bp.json* file.

```
1  class BpQuadlateration:
2      def __init__(self, p_A1, p_A2, p_A3, p_A4):
3          self.pos_A1 = p_A1
4          self.pos_A2 = p_A2
5          self.pos_A3 = p_A3
6          self.pos_A4 = p_A4
7          with open("bp.json", 'r') as f:
8              conf = json.load(f)
9          self.n = conf['n_value']
10
11     def get_func(self, p, rssi):
12         p_x = p[0]
13         p_y = p[1]
14         with open("bp.json", 'r') as f:
15             conf = json.load(f)
```

```
16          self.n = conf['n_value']
17
18
19          def f(x):
20              return (x[0] - p_x)**2 + (x[1] - p_y)**2 - 10**((x[2] -
                    rssi) / (5 * self.n))
21
22          def df(x):
23              return [2*(x[0] - p_x), 2*(x[1] - p_y), (-log(10) / (5 *
                     self.n)) * 10**((x[2] - rssi) / (5 * self.n))]
24
25          return f, df
```

Listing 4.7: Section of the *BpQuadlateration* class for calculating the device location

Chapter 5

Evaluation

5.1 Hardware

The following section contains overview of the hardware used by [25] and the updated components for *BluePIL 2.0*.

In Table 5.1 there is a summary of the new hardware choices for *BluePIL 2.0* to meet the requirements in contrast to previous version. The Ubertooth One devices acting as the sensors of the *BluePIL* system allow to identify the devices LAP, which are normally hidden in the lower layer of the Bluetooth protocols. They still suit our requirements at a comparably low cost. They are capable of sniffing the 79 BTBR/EDR channels and can send the captured packet via USB to the host systems with the RSS values for positioning. Firmware, and libraries are also provided by Project Ubertooth to enable the host system to use Ubertooth.

Category	**BluePIL**	**BluePIL 2.0**
Sensor	Four Ubertooth One Devices	Four Ubertooth One Devices
Node Device	Four Asus Tinkerboards (Debian Linux)	Four Raspberry Pi Zero W's (Raspberry OS)
Sink Device	MacBook Pro	Microsoft Surface Book 2
Network	Gl-iNet Mifi Smart Router	Netgear AirCard 790 Mobile Hotspot
Powerbank	Four Fresh n Rebels Powerbanks 3000 mAh	

Table 5.1: Hardware for *BluePIL* and *BluePIL 2.0* in comparison

Instead of using *Asus Tinkerboards* [30] we opt for the less energy consuming *Raspberry Pi Zero W's* [31] that are still capable of processing *BluePIL*. The reason is due to requirement four to provide a more mobile and simple deployable system. Originally, the nodes were powered by longer cables, limiting mobility and deployability. The benefit of having a less power absorbing device is that they can be supplied by powerbanks more efficient. The *Raspberry Pi Zero W's* are best suited because they are inexpensive, have a small size and do not consume much power. With a cost of 12.90 CHF [32] (at the time of writing this thesis) it contains a single ARM1176JZF-S 1 GHz core, 512 MB RAM, 11 b/g/n WLAN to connect to the network and Bluetooth 4.1 as well as BLE. The processing power in sufficient for the consumption of *BluePIL*. The only drawback is that the

Raspberry Pi Zero W have a micro-USB port instead of a USB 2.0 port to connect the Ubertooth sensor. To overcome this factor, we used a micro USB to USB 2.0 adapter. The nodes had a *Raspberry Pi OS* [33] installed on them, the official supported operating system for *Raspberry Pi's*.

The powerbanks enable the system to be more mobile and deployable due to the nodes devices no longer requiring cables. The *Fresh n Rebel* powerbank [34] with 3000 mAh offer a cheap variant with whom the nodes can be power supplied for longer periods of time. Figure 5.1 shows one of the node devices that were deployed for experimentation.

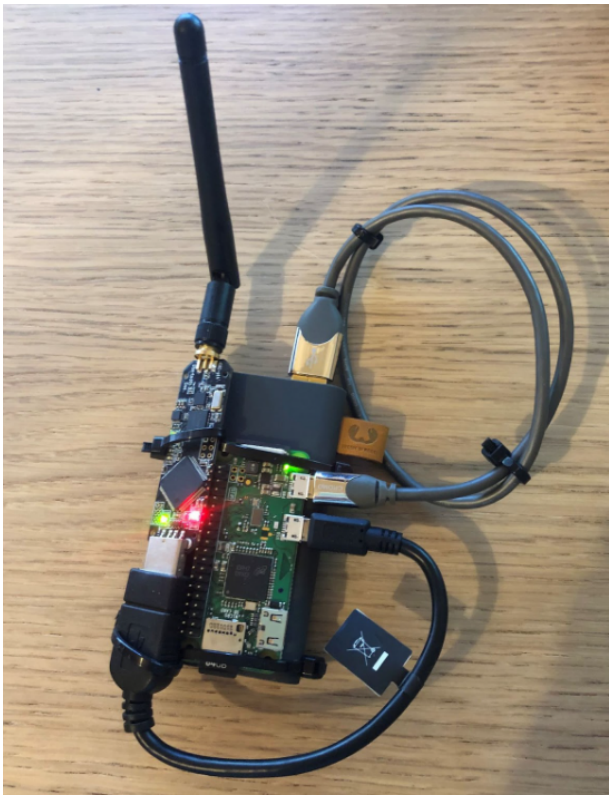

Figure 5.1: Node Device

As far it goes for the sink, it was convenient to use a *Microsoft Surface Book 2* [35] with 16GB of LPDDR3 memory and a Intel i7-8650U 4.2 GHz quad-core processor. This is sufficient for its usage as the sink as it has better hardware than the 2017 *Macbook Pro* [36] of [25]

Instead of using a *GL-iNet Mifi Smart Router* [37], a *Netgear Mobile Hotspot* [38] was used, which also creates a wireless network for the application and is portable to any place. To ensure time synchronisation between the nodes and the sink, a SIM Card with internet access was integrated into the portable router.

Since the *Raspberry Pi OS* is a *Debian-based*[39] system, many fundamental processes are shared. Therefore, the system provides *Crontab*, a utility application to effectively schedule a routine background job. To further improve deployment, the nodes are implemented

to start the mini-server after the system boots. This eliminates the step of establishing a SSH connection and starting the mini-server application on the nodes.

Each of the nodes 1 to 4 is programmed to always request the same IP addresses 192.168.1.101 to 192.168.1.104 respectively. They have been given a static IP address to simplify deployment. Instead of determining the IP addresses each time they are connected to a network, they receive the requested IP addresses unless the network prohibits it.

5.1.1 Scenario

The three experiments were conducted in a warehouse using a 4 m by 4 m measured square field as shown in figure 5.2. In each corner one of the *Raspberry* nodes were deployed. A warehouse offers few environmental influences, *i.e.* a constant temperature, little ultraviolet light (UV) and less signal propagation as there are no walls near the experiment, which is important when working with Bluetooth signals and in line with [25] suggestions for future experiments.

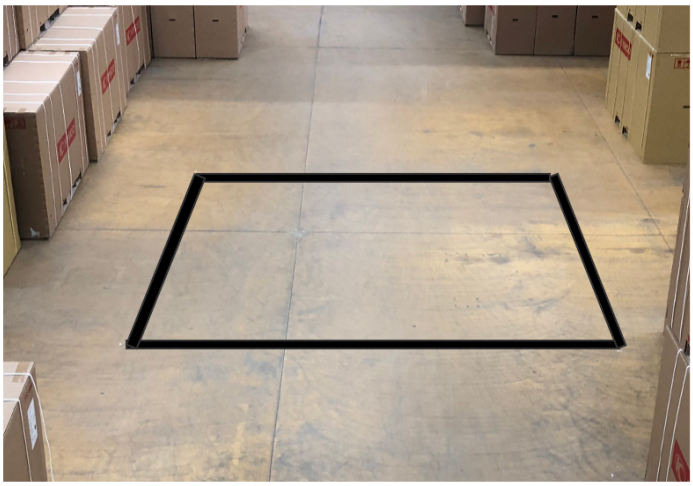

Figure 5.2: Experiment setup

After the nodes have been deployed and the scenario parameters, *i.e.*, the position of the nodes, the n coefficient and the *True Point* of the device has been set. following three experiments have been conducted.

5.2 Experiment 1: Evaluation of the Automation

5.2.1 Experimental Setup

In a first experiment, we evaluated the effectiveness of the automation of *BluePIL 2.0*. We, therefore, measured the time it takes for the operator to start the data collection pipeline.

We started the time measurements after physically setting up the devices and configuring them, *i.e.*, setting IP, position and n coefficient as well as the *True Point* in the *bp.json* file. This was chosen because the physical setup are different in each scenario and therefore not meaningful. Additionally, the part to enhance the system focuses on the software implementation mainly. Since configuration is part of the *BluePIL 2.0* initialization steps, we set three different timers.

1. Measured the time once the client application is started.
2. Measured the time after the configuration GUI is terminated.
3. Set after the connection between the sink and the nodes is established and the *START_NODE* command is sent to the nodes just before the sink is started, initialising the pipeline stream for data collection.

The timers one to three thus indicate the time that the implemented server-node architecture needs to fulfil its purpose. To setup was repeated for 30 times.

5.2.2 Results

Figures 5.3 5.4 and 5.5 show the results of the first experiment in *box-plots*. Overall, we have achieved an average of 7.37 seconds to launch the whole *BluePIL 2.0* application with a median of 7.09 seconds. In addition, the time required for server-client communication was 5.17 seconds on average and 4.89 seconds on median. The outliers in Figure 5.4 show that the network problems between the nodes and the sink can sometimes cause delays in communication. Nevertheless, they remain within reasonable limits as the process never exceeded 8 seconds. The configuration time was a simple *Safe Quit* procedure, as the configuration was already done beforehand, causing an average of 2.2 seconds and a median of 2.16 seconds additional time for the setup, as it was embedded in the start-up process.

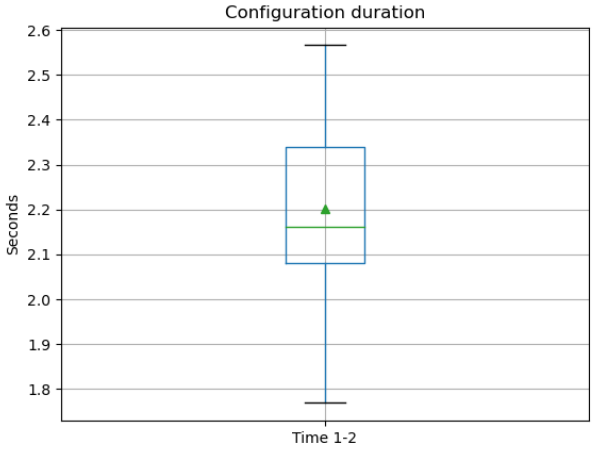

Figure 5.3: Time between first and second timer

Figure 5.4: Time between second and third timer

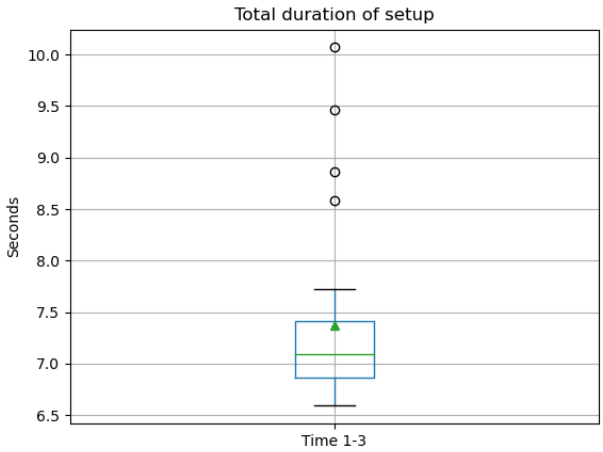

Figure 5.5: Time between first and third timer

5.3 Experiment 2: Evaluation of the Path-Loss Coefficient Adjustment

5.3.1 Experimental Setup

For the second experiment, we ran the data analysis tool during data collection of *BluePIL 2.0*. The goal of the experiment was to evaluate the GUI section that is responsible to change the n coefficient during the process. The data analysis tool was therefore ran for two minutes, changing the n coefficient every 20 seconds manually within the GUI. The time was set short because the implementation does not allow for other results within the update cycles, although two minutes is still a considerable amount to show its effectiveness.

To ensure that the coefficient was changed, the function *bpquadlaterate*, which calculates the location, has been modified to output the current timestamp and the coefficient used for the calculation if a positioning has happened. The timestamps are then compared to determine if the correct coefficient value was used for the respective 20-second interval.

5.3.2 Results

The results in Table 5.2 show the parameter adjustment during the positioning in different time-intervals and how many times it was applied coordinately.

Time	Coefficient	Total calculations	Correctly used coefficient
0-20 seconds	3.0	9	9
20-40 seconds	2.0	5	5
40-60 seconds	1.8	4	4
60-80 seconds	2.2	5	5
80-100 seconds	2.7	1	1
100-120 seconds	1.5	8	8

Table 5.2: Coefficient analysis ran for two minutes

5.4 Experiment 3: Evaluation of the Data Analysis Tool

5.4.1 Experimental Setup

A third experiment was done to evaluate the *BluePIL* data analysis tool, *i.e.* the GUI implemented to visualize the data collected. Two phones were placed into the square field in the warehouse connected to two headsets via Bluetooth. One was set to the coordinates *[2,2]*, the other to *[1,3]*. In addition, the *True Point* was set to first device. The experiment ran for 5 minutes without interruption with a n coefficient of 1.8. The time was set to 5 minutes to collect sufficient data to display in the GUI and the n coefficient was set to 1.8 as this was suggested by [8] for a grocery store which has similar conditions to our scenario. We have to take into account that the n value setting does not play an important role for this experiment as it is for evaluating the GUI. For the data analysis, we apply the Kalman Filter upon the data for each device to eliminate outliers. Firstly, we plot all the devices positioned and secondly the first device at coordinates *[2,2]*. The aim is to show how the system benefits from a data analysis tool by analysing the user interface, but not to improve the data collection itself.

5.4.2 Results

Figure 5.6 shows the GUI after 5 minutes of deployment. The system has detected the two devices in the room as expected, as the underlying system has not been changed. Each

device is plotted with the 10 last points received of the data set. The first positioned device represented in red and the second device in green. Without having to manually analyse the data, the predictions are displayed on the plot, so we know where the device is located. It detecting over 29 different devices near the scenario. In Figure 5.7, we decided to only plot the device *ae7bbb* that should be near the *True Point*. The prediction mean reached an overall mean error of 0.8041 meters as calculated and displayed by the tool on the button left of the graph for the mean of the last 10 points.

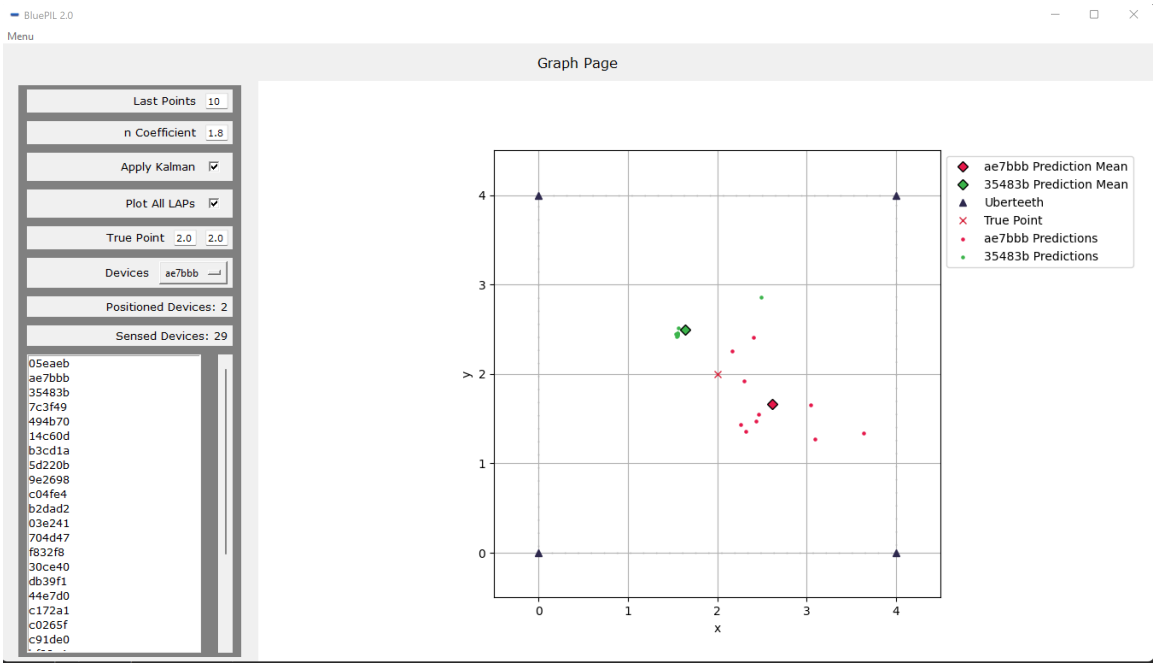

Figure 5.6: GUI after 5 minutes, plotting all devices

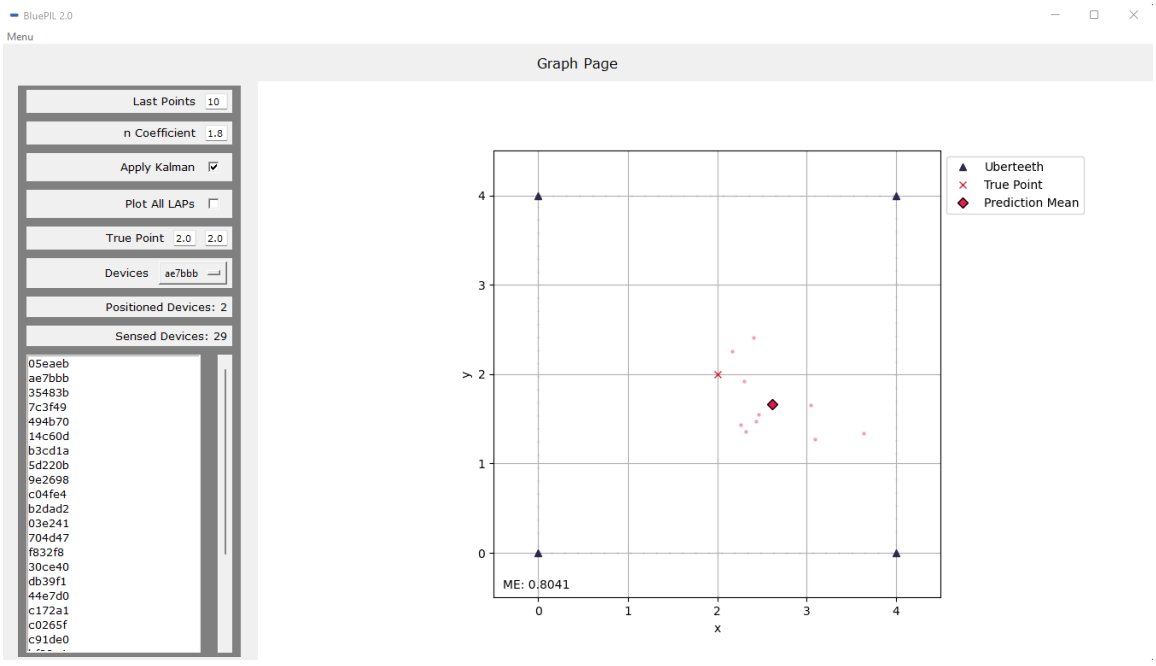

Figure 5.7: GUI after 5 minutes, plotting *ae7bbb*

5.5 Discussion

The three experiments conducted show a broad overview over the enhanced *BluePIL* system. The aim of this thesis was firstly to reduce the deployment time and increase mobility for future data collection. Secondly, making the path-loss coefficient n adjustable during data collection and thirdly to plot the data near-real-time to enhance the understanding of the scenario and adjust the n coefficient accordingly.

In the first experiment *BluePIL 2.0* performed with a full deployment time of less than 8 seconds on average. If we consider that the configuration is already done before, it is capable to connect the devices in around 5 seconds on average, which is successfully faster than deploying the application via a SSH connection. It should be taken into account that the author who carried out the experiments knows the manual steps and commands in detail, so the execution could be faster than that of an average person, although we must consider that they only consist of two easy-to-learn commands. The client-server architecture made the deployment flexible and facilitated stopping and restarting the application in contrast to the previous version. The use of the new hardware in *BluePIL 2.0* improved the mobility of the system and the connected *powerbanks* made them available for scenarios where no electricity is available. In addition, cabling is no longer required, which reduce the overall workload. The *Raspberry Pi Zero W's* proved themselves as viable alternatives to the *Asus Tinkerboards*.

The second experiment showed that the n coefficient is now adjustable to any value given by the operator. It opens up the possibility of running several scenarios in different environments to determine the best possible coefficient. By visually analysing the data analysis tool, the accuracy can be improved by testing with different coefficient or setting them to a known one. Interestingly, the amount of calculations within the scenario changed drastically between 1 to 9 calculation per 20 seconds. This can be attributed to the fact that perhaps one of the nodes was not able to collect data properly and therefore there were less calculations.

The third experiment showed the benefit of having a GUI plotting the current data. It was capable of representing the last points received of the two devices deployed in the scenario. The time that would have been required to post-analyze the data is gone. Additionally, malicious behaviour of the system can be detected during collection and maybe prevented. Although the sample size shown was relatively small, the accuracy was better than the experiments performed by [25], suggesting that environmental conditions with fewer factors, *i.e.* temperatures around 18°C and no walls, strongly favour the localization, although the aim was not to test or improve the accuracy. Despite the fact that the experiments took place in a warehouse, the system was very efficient and detected over 29 different devices in the vicinity of the scenario in the third experiment, even though the nearest offices and personas were several metres away behind walls.

In the following the requirements described in Section 3.1 are compared to the performance of *BluePIL 2.0*:

R1 Automating the deployment and operation of *BluePIL* to reduce the number of manual steps required in its bootstrap: The requirement is satisfied. The overall manual steps required were reduced significantly.

R2 The system must be able to plot the data near-real-time: The requirement is satisfied. The GUI fetches the data real-near-time with an updating function interval of 0.5 seconds and represents the data accordingly.

R3 Parameters must be adjustable for better visualisation of the plot: This requirement is satisfied. Last points, *True Point*, Kalman Filter as well as plotting different devices at once are possible parameters to adjust the plotting.

- **The n path-loss coefficient should be adjustable during data collection:** This requirement is satisfied. The n coefficient is updated within a 0.5 second interval during data collection when adjusted.

R4 Prototype must enable mobility of experiments allowing to be easily deployable in different scenarios: This requirement is satisfied. The new smaller nodes with the connected powerbanks increase mobility for different scenarios.

In regards to the goals of this work, on which the requirements are based, we can derive the following:

- **Automation**: The automation was successfully reduced the deployment time for many future scenarios. The nodes scripts starting on bootstrap and the server-client architecture provide much utility for the deployment.
- **Path-loss coefficient**: The path loss coefficient n can now be changed to any value during data collection by the operator in the implemented GUI or in the *bp.json* file responsible for configuration.
- **Analysis**: The analysis GUI provides a near-real-time plotting of the collected data and help understand the scenario given, visualizing each device that got positioned and sensed. Furthermore, it can be used as a viable information to modify the n coefficient.

Chapter 6

Final Considerations

6.1 Summary and Conclusion

This thesis extends the system called *BluePIL*, which passively locates and identifies Bluetooth devices. The aim was to address the problem of the system's lengthy deployment time, limited mobility and lack of utility within the data collection. The objectives of the goals were achieved. The new version called *BluePIL 2.0*, enables faster deployment of the system with a server-client architecture that automates the start-up of the *BluePIL* pipeline. In addition, the new software and hardware configuration makes it easier to deploy by using a WiFi LAN to synchronize with a sink and mobile nodes based on powerbanks instead of connecting cables. During the data collection, the plot of the collected data helps to analyze the scenario on screen. The tool offers the possibility to set the n coefficient to a value suitable for the scenario and also allows the operator to choose a true point for the setting. The data is plotted within a updating interval to be considered near-real-time.

Overall, the updated version developed in this thesis offered a significant improvement in deployment and operation while maintaining low-performance hardware and near real-time credibility. Nonetheless, the specific aspects concerning the localization algorithm still needs to be optimised as the tracking is not precise, although setting better environmental conditions helped to improve the localization in the scenario carried out. Interestingly, the system proves to be good at detecting devices without positioning them, which could be useful for other applications. The data analysis tool is a useful first step for real-time analytics in the context of this work, although in a real-world scenario a marketing analytics company would likely end up implementing a much larger cloud-based analysis application that receives data from several different pipelines of *BluePIL* deployments simultaneously, calculates various behaviours and creates running records. Though this scale would have gone beyond the scope of this thesis. However, the implemented client-server architecture would support this type of setup well.

Some difficulties encountered in the implementation of *BluePIL 2.0* were the lack of commented code in the underlying source code, which had to be figured out manually by trial and error to get a better understanding of how the system works. In addition, the

node devices with their operating systems required the acquisition of in-depth knowledge about remote connection and setup, as well as learning new operating system calls. The n coefficient also difficult to be observed, as the tests carried out during implementation did not show success for better localization. However, it must be said that the aim of the work was not to achieve better accuracy, but to provide the tools to do so. There were also several approaches to solve the problem of internet connectivity for time synchronisation between the nodes so that the system no longer needs it, but after some discussion, the system would probably be used in a real world scenario where internet access is not an issue, as it is certainly provided. In this sense, the focus was put on more important aspects for *BluePIL*. Initially, the aim of this thesis was rather to experiment with *BluePIL* in different scenarios to find better calibration parameters for each. As it turned out, the calibration was about the n coefficient, but since it is difficult to measure the impact, and public spaces with many people often require consent, the goal was changed.

6.2 Future Work

The system implemented opens new possibilities for future work. Extensive testing in different scenarios are more feasible to do and allow possibly improving the accuracy. *BluePIL* runs with a definitive localization algorithm. By exploring new localization algorithms, the predictive calculator could be evaluated to see if a new algorithm might prove to be better. Concerning the internet connectivity issue to synchronize the clock, most certainly there would be a way to synchronize the time between the sink and the nodes within a network without internet connectivity, though it is still questionable if this is applicable for real-world scenario as most networks deployed provide internet access. In Experiment 3, the last points retrieved from a device are a useful source of information, even though they may not represent the current live prediction, as the device may move away from the scenario or simply stop sending Bluetooth signals. The last 10 points remain on the plot as no more data is received. Therefore, a more complex design could be implemented where the data is plotted according to its last timestamp within a certain range, e.g. the last 5 seconds. Through the time synchronization issues at first, this thesis has not treated this subject as the time has not allowed it. Furthermore, a repetition of Experiment 3 in a more crowd dense area might indicate other utilities, as the system has already indicated at being good at sensing Bluetooth devices. Future work may move in the direction of identifying and sensing Bluetooth devices by using proximity to determine which node the devices are closest to in a much larger area. This could already give clues as to which people regularly walk past or spend time at certain nodes, e.g. a shelf in a market. This would be interesting as it works with less precise localization as environmental factors often prevent this anyway.

Bibliography

[1] Rafael Hengen Ribeiro et al. "ASIMOV: a Fully Passive WiFi Device Tracking". In: *2021 IFIP Networking Conference (IFIP Networking)*. IEEE. 2021, pp. 1–3.

[2] Bruno Rodrigues et al. "BluePIL: a Bluetooth-based PassIve Localization Method". In: *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2021, pp. 28–36.

[3] Bruno Rodrigues et al. "CCount: Correlating RFID and Camera Data for High Precision Indoor Tracking". In: *University of Zurich, Department of Informatics, Tech. Rep* 1 (2022).

[4] Bruno Rodrigues et al. "LaFlector: a Privacy-preserving LiDAR-based Approach for Accurate Indoor Tracking". In: *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE. 2021, pp. 367–370.

[5] Faheem Zafari, Athanasios Gkelias, and Kin K. Leung. "A Survey of Indoor Localization Systems and Technologies". In: *IEEE Communications Surveys Tutorials* 21.3 (2019), pp. 2568–2599.

[6] Luca Mainetti, Luigi Patrono, and Ilaria Sergi. "A Survey on Indoor Positioning Systems". In: *2014 22nd international conference on software, telecommunications and computer networks (SoftCOM)*. IEEE. 2014, pp. 111–120.

[7] Great Scott Gadgets. *Ubertooth*. `https://github.com/greatscottgadgets/ubertooth`. 2020. (Visited on Jan. 20, 2022).

[8] J.B. Andersen, T.S. Rappaport, and S. Yoshida. "Propagation measurements and models for wireless communications channels". In: *IEEE Communications Magazine* 33.1 (1995), pp. 42–49.

[9] *Bluetooth core specification v5.2*. v5.2. Bluetooth SIG. Dec. 2019.

[10] Bluetooth SIG. *Vision and Mission*. `https://www.bluetooth.com/about-us/vision/`. 2022. (Visited on Jan. 20, 2022).

[11] Bluetooth SIG. *Market Update 2021*. `https://www.bluetooth.com/bluetooth-resources/2021-bmu/`. 2021. (Visited on Feb. 17, 2022).

[12] Great Scott Gadgets. *Ubertooth*. `https://github.com/greatscottgadgets/ubertooth`. 2022. (Visited on Mar. 24, 2022).

[13] Texas Instruments. *CC2400 Datasheet*. `https://www.ti.com/lit/ds/symlink/cc2400.pdf`. 2008. (Visited on Mar. 22, 2022).

[14] Georg Carle and Corinna Schmitt, eds. *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Winter Semester 12/13*. Vol. NET-2013-02-1. Network Architectures and Services (NET). Feb. 2013.

[15] Hacker Warehouse. *Ubertooth One*. https://hackerwarehouse.com/product/ubertooth-one/. 2022. (Visited on Mar. 24, 2022).

[16] V Fox et al. "Bayesian filtering for location estimation". In: *IEEE pervasive computing* 2.3 (2003), pp. 24–33.

[17] Guoquan Li et al. "Indoor positioning algorithm based on the improved RSSI distance model". In: *Sensors* 18.9 (2018), p. 2820.

[18] DP-3T. *DP3T - Decentralized Privacy-Preserving Proximity Tracing*. https://github.com/DP-3T/documents. 2020. (Visited on Mar. 18, 2022).

[19] Lorenz Schauer, Martin Werner, and Philipp Marcus. "Estimating Crowd Densities and Pedestrian Flows Using Wi-Fi and Bluetooth". In: MOBIQUITOUS '14. 2014, 171â177.

[20] Mathias Versichele et al. "Pattern mining in tourist attraction visits through association rule learning on Bluetooth tracking data: A case study of Ghent, Belgium". In: *Tourism Management* 44 (2014), pp. 67–81.

[21] Alaa Alhamoud et al. "Presence detection, identification and tracking in smart homes utilizing bluetooth enabled smartphones". In: *39th Annual IEEE Conference on Local Computer Networks Workshops*. 2014, pp. 784–789.

[22] Imad Afyouni et al. "Passive BLE Sensing for Indoor Pattern Recognition and Tracking". In: *Procedia Computer Science* 191 (2021), pp. 223–229.

[23] Livealytics. *Livealytics - We make Live Experience measurable*. https://www.livealytics.com. 2022. (Visited on Mar. 18, 2022).

[24] Bruno Bastos Rodrigues. *Inosuisse Funds Cooperation between the University of Zurich and Livealytics*. https://www.csg.uzh.ch/csg/en/news/PasWITS-Research-Project0.html. Mar. 2020. (Visited on Mar. 17, 2022).

[25] Cyrill Halter. "BluePIL: Fully Passive Identification and Localization of Bluetooth Devices in Near-Real-Time". MA thesis. University of Zurich, Aug. 2020.

[26] Xiaojie Zhao et al. "Does BTLE measure up against WiFi? A comparison of indoor location performance". In: *European Wireless 2014; 20th European Wireless Conference*. 2014, pp. 1–6.

[27] Python Software Foundation. *tkinter - Python interface to Tcl/Tk*. https://docs.python.org/3/library/tkinter.html. 2022. (Visited on Feb. 20, 2022).

[28] The Matplotlib Development team. *Matplotlib: Visualization with Python*. https://matplotlib.org/. 2022. (Visited on Feb. 20, 2022).

[29] The pandas development team. *pandas*. https://pandas.pydata.org/. 2022. (Visited on Feb. 22, 2022).

[30] ASUSTeK Computer Inc. *Tinker Board*. https://tinker-board.asus.com/product/tinker-board.html. 2022. (Visited on Feb. 23, 2022).

[31] Raspberry Pi. *Raspberry Pi Zero W*. https://www.raspberrypi.com/products/raspberry-pi-zero-w/. 2022. (Visited on Mar. 1, 2022).

[32] Pi-Shop.ch - Totonic GmbH. *Raspberry Pi Zero W - EDU*. https://www.pi-shop.ch/raspberry-pi-zero-w. 2022. (Visited on Mar. 2, 2022).

[33] Raspberry Pi. *Raspberry Pi OS*. https://www.raspberrypi.com/software/. 2022. (Visited on Mar. 2, 2022).

[34] Fresh 'n Rebel. *POWERBANK 3000 MAH*. https://freshnrebel.com/uk/powerbank-3000-mah/2pb3000ig/. 2022. (Visited on Mar. 3, 2022).

[35] Microsoft. *Surface Book 2 features*. https://support.microsoft.com/en-us/surface/surface-book-2-features-d752c78d-d1fc-c483-c80d-8343e68ad96b. 2022. (Visited on Feb. 21, 2022).

[36] Apple. *MacBook Pro (13-inch, 2017, Four Thunderbolt 3 ports) – Technical Specifications*. https://support.apple.com/kb/sp755?locale=en_US. 2022. (Visited on Mar. 20, 2022).

[37] GL Technologies Microuter Technologies. *MEET GL-MiFi*. https://www.gl-inet.com/products/gl-mifi/. 2022. (Visited on Mar. 21, 2022).

[38] NETGEAR. *NETGEAR 4G LTE Mobile Hotspot*. https://www.netgear.com/uk/home/mobile-wifi/hotspots/ac810/. 2022. (Visited on Mar. 21, 2022).

[39] Debian. *Debian*. https://www.debian.org/index.en.html. 2022. (Visited on Mar. 19, 2022).

Abbreviations

BLE	Bluetooth Low-Energy
BTBR	Bluetooth Basic Rate
BTEDR	Bluetooth Enhanced Data Rate
CSV	Comma-Separated Values
FHSS	Frequency Hopping Spread Spectrum
GFSK	Gaussian Frequency-Shift Keying
ISM	Industrial, Scientific and Medical
IoT	Internet-of-Things
IP	Internet Protocol
SSH	Secure Shell
SIG	Special Interest Group
TCP	Transmission Control Protocol

List of Figures

List of Tables

Listings

Appendix A

Contents of the Repository

The repository contains the following content:

- Root Directory: Contains both the start application for the node server and the sink client application, the data analysis tool, the configuration file, a demo video and the requirements file for the sink
- node Directory: Contains the node code of the *BluePIL* system
- sink Directory: Contains the sink data stream logic of the pipeline
- node_setup Directory: Contains the updated requirement file for the *Raspberry Pi Zero W*
- data Directory: Contains the data of the experiments

Appendix B

Installation Guidelines

B.1 Installation of Dependencies

BluePIL 2.0 requires Python v3.8 to be installed as well as the dependencies. Dependencies for the sink are located in the Root Directory in requirements.txt and for the nodes in the `node_setup/install.sh`. The Ubertooth One sensors must be connected to each of the *Raspberry Pi Zero W's* via USB.

B.2 Running the Application

Before running the node application, each device's *IP* address, if not already configured, has to be adjusted in the *bp.json* file accordingly. The *BluePIL 2.0* node mini-server can be started by running `python raspberry_server.py`. The sink application is started by running `python run_sink.py`. After starting the sink application, the configuration of the scenario has to be set:

- `True Point`: x, y coordinates of the to be sensed device
- `N-Value`: the n coefficient value
- `Ubertooth_1-4`: x, y coordinates of the *Uberteeth* within the scenario

After the configuration the streaming pipeline can be started by typing `START_NODE`. Demo.mp4 in the Root Directory shows a short demonstration of the setup.