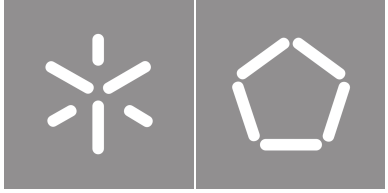
**Universidade do Minho**
School of Engineering

Diogo Miguel Pinto Rio

# Object tracking in industrial environments

October, 2023



**Universidade do Minho**
School of Engineering

Diogo Miguel Pinto Rio

# Object tracking in industrial environments

Master Thesis
Integrated Master's in Informatics Engineering

Work developed under the supervision of:
**Professor Doutor Adriano Moreira**
**Professor Doutor Filipe Meneses**

October, 2023

## STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

_____, _____

(Place) (Date)

_____

(Diogo Miguel Pinto Rio)

# Acknowledgements

First, I would like to thank both of my advisors, Professor Adriano Moreira and Professor Filipe Meneses. Their knowledge and guidance were essential throughout the entire process of writing this thesis.

I would also like to thank my family for their support throughout my entire academic journey.

Lastly, I would like to thank my friends for their invaluable support, friendship, and happy moments. Without them, my academic journey would not be the same.

*"The difference between winning and losing is most often not quitting." (Walt Disney)*

# Resumo

## Seguimento de objetos em ambiente industrial

A sociedade atual está bastante dependente de sistemas de navegação por satélite. Estes sistemas utilizam satélites para fazer a geolocalização de um dispositivo. Um dos exemplos mais conhecidos de um sistema deste tipo é o famoso Global Positioning System (GPS). Devido à atenuação de sinais causada por materiais de construção, um sistema de posicionamento por satélites está limitado a espaços exteriores. Um sistema de posicionamento interior tenta responder a este problema e usa um conjunto de dispositivos que permitem fazer o posicionamento de pessoas ou objetos em espaços interiores. Esta área de estudo tem sido alvo de várias pesquisas nos últimos anos e, recentemente têm sido implementados em vários setores. Por exemplo, na monitorização de idosos que vivem sozinhos, na gestão de material hospitalar, no seguimento de pessoas para fins de segurança e para uma melhor gestão de recursos em grandes armazéns. Embora os sistemas de posicionamento interiores tenham evoluído significativamente nos últimos anos, existem poucos dispositivos móveis (tags) disponíveis para integração com os sistemas. Além disto, as capacidades das tags que existem são limitadas, especialmente no que toca à sua comunicação com sistemas não proprietários. Esta dissertação procura desenvolver e propor uma tag que possa responder a estes problemas.

**Palavras-chave:** Indoor Positioning System, Wi-Fi, CSI, tag

# Abstract


## Object tracking in industrial environments

Today's society relies heavily on Global Navigation Satellite Systems (GNSS). GNSS are systems that use satellites to provide geo-spatial positioning. One example of such a system is the well-known GPS. Satellite-based positioning systems are limited to outdoor use due to the signal attenuation caused by construction materials and other physical objects inside buildings. This makes GNSS unsuitable for locating entities in indoor or underground locations. An Indoor Positioning System (IPS) may include a device (or set of devices) used to locate persons or objects in an indoor environment. The development of this technology has been the subject of years of research and development. In the past decade, positioning systems have been deployed in various fields, including monitoring individuals living alone, managing medical equipment in hospitals, tracking people for security purposes, and better management of resources in large warehouses. Even though indoor positioning technology has evolved significantly in recent years, only a few mobile positioning devices (*tags*) are available for integration. In addition, the capabilities of existing tags are limited, especially in communicating with open systems. This work aims to develop and propose a tag to address some of these issues.



**Keywords:** Indoor Positioning System, Wi-Fi, CSI, tag

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**AOA** Angle of Arrival *(pp. 6, 7, 11–14)*

**AOD** Angle of Departure *(pp. 7, 11, 12)*

**AP** Access Point *(pp. 21, 23, 27, 29, 33, 41, 50, 69, 72, 78, 80, 85, 93, 94, 106)*

**API** Application Programming Interface *(pp. 26, 33, 37)*

**BSSID** Basic Service Set Identifier *(pp. 20, 21, 23, 27)*

**CSI** Channel State Information *(pp. 9, 10, 20, 21, 23, 25–27, 29, 35, 39, 44, 65, 69, 70, 72, 89, 105)*

**CTE** Constant Tone Extension *(p. 12)*

**DOA** Direction of Arrival *(p. 6)*

**ESP-IDF** Espressif IoT Development Framework *(pp. xii, 10, 25, 38–41, 43, 45, 46, 50, 57, 60, 68, 69, 72, 81, 82)*

**FIFO** First In First Out *(pp. 65, 68)*

**GNSS** Global Navigation Satellite Systems *(p. vii)*

**GPS** Global Positioning System *(pp. vi, vii)*

**I2C** Inter-Integrated Circuit *(p. 81)*

**IDE** Integrated Development Environment *(p. 38)*

**IPS** Indoor Positioning System *(pp. vii, 3, 10, 12, 80, 85, 105, 106)*

**JSON** JavaScript Object Notation *(pp. 25–27, 43, 47, 50, 54, 55, 60, 61, 63, 65, 68, 70, 72, 96)*

**kNN** k-nearest-neighbor *(p. 9)*

**MAC** Media Access Control *(pp. 21, 23, 27, 29, 60)*
**MIMO** Multiple Input Multiple Output *(p. 10)*

**NIC** Network Interface Card *(p. 10)*
**NVS** Non-volatile storage *(p. 45)*

**OFDM** Orthogonal Frequency Division Multiplexing *(pp. 9, 10)*

**QPE** Quuppa Positioning Engine *(p. 11)*

**RFID** Radio Frequency Identification *(p. 3)*
**RSS** Received Signal Strength *(pp. 8–10)*
**RSSI** Received Signal Strength Indicator *(pp. 20, 21, 23, 27, 29, 35, 39, 65, 72, 89, 90, 105)*
**RTLS** Real-Time Location System *(pp. 10–12, 14)*

**SMP** smallest M-vertex polygon *(p. 9)*
**SPIFFS** SPI Flash File System *(pp. 46, 50, 55)*
**SQL** Structured Query Language *(pp. xii, 98–100, 102, 104)*
**SSID** Service Set Identifier *(pp. 25, 27, 45, 56)*
**SVM** support vector machine *(p. 9)*

**TDOA** Time Difference of Arrival *(pp. 5–7)*
**TOA** Time of Arrival *(pp. 5–7)*

**UWB** Ultra-Wideband *(pp. 3, 10)*

**VSCode** Visual Studio Code *(pp. xii, 38, 39, 46, 69, 81, 82)*

# 1 Introduction

In this chapter, we will first discuss the context and motivation of the problem, followed by the primary objectives to be achieved with this work. Finally, a summary of this work's structure and how to best navigate it will be presented.

## 1.1 Context & Motivation

In recent years, indoor positioning technologies have been the subject of extensive research and development. Several applications have been developed utilizing indoor positioning systems. These include monitoring individuals living alone, managing medical equipment in hospitals, tracking people for security purposes and managing resources in large warehouses more effectively.

Despite significant advancements in indoor positioning technology in recent years, only a few mobile positioning devices (tags) are currently available for integration. Here, tags refer to devices that are attached to a target to be tracked or for which the position is to be estimated. Additionally, the capabilities of existing tags are limited, especially in terms of communication with open systems.

In today's society, WiFi and Bluetooth are two common technologies used in indoor spaces for various purposes by both individuals and corporations. Due to their popularity, implementing these communications technologies is relatively inexpensive, making them one of the most attractive communication technologies for indoor positioning systems due to their ubiquity and low cost of implementation. In past studies, a service that estimates an entity's indoor location based on WiFi and BLE technologies has already been developed by a research group at the University of Minho.

## 1.2 Objectives

This work aims to develop and evaluate a device that addresses some of the current issues on the market today and which can be integrated with existing WiFi positioning systems. This tag has the primary function of collecting WiFi data from its surroundings and transmitting it to a service able to estimate its location based on the transmitted information. It should be designed using off-the-shelf components, be

compact, and have high energy efficiency. To achieve this, the following sumarized tasks are going to be carried out through the development of this work:

1. Become familiar with the current technologies in the field of indoor positioning systems and gather data on the current state of the art;
2. Conduct a study to identify optimal characteristics for a tag and possible hardware alternatives;
3. Implement a tag using off-the-shelf hardware, creating a firmware that allows access to the radio interface, including the low-level technical data;
4. Analyse the developed solution.

## 1.3 Structure

This work is structured into seven main chapters: Introduction, State of the Art, Problem Statement and System Architecture, System Specification, System Implementation, System Analysis, and Conclusions and Future Work.

The introductory chapter presents a brief overview of the thesis theme context, motivation, and main objectives.

The second chapter provides a review of current indoor positioning technologies and commercial products.

In the Problem Statement and System Architecture chapter, an analysis of the dissertation problem and requirements is made. Moreover, an overview of the general system architecture is introduced.

Following the Problem Statement and System Architecture chapter, we discuss the specifics of the software and hardware used in our solution.

After, in the System Implementation chapter, the specifics of the implementation of our solution are carefully described.

In the System Analysis chapter, we evaluate our solution with a series of tests and make some conclusions based on the results.

Finally, in the Conclusions and Future Work chapter, we review what was accomplished with this work and map out possible ways to improve it.

# 2

# State of the Art

This chapter explores the state of the art in Indoor Positioning Systems. First, we will discuss Indoor Positioning System concepts and techniques. Here, we will describe some techniques used in IPSs to locate a target.

Lastly, market research on currently available commercial products will be conducted. We will discuss the most used technologies and techniques available on the market and how coupled these solutions are with proprietary hardware and software.

## 2.1 Indoor Positioning Systems concepts and techniques

An Indoor Positioning System uses a variety of technologies and techniques to determine the location of a device or a person within a building or other enclosed space. These technologies include Radio Frequency Identification (RFID), Ultra-Wideband (UWB), Bluetooth, and WiFi. In terms of techniques, fingerprinting, triangulation-based, trilateration-based techniques and others are employed.

A key metric of IPSs is accuracy, and different technologies and techniques can provide varying degrees of accuracy. It is also common for the most accurate IPS to be the most expensive, mainly due to the cost of implementing the required infrastructure and special equipment. Furthermore, precision and complexity are important metrics to consider. Table 2.1 depicts differences in technology, techniques, cost, accuracy, precision and complexity of different IPSs.

There are several positioning estimation algorithms available today. To determine a device's location, these algorithms use signal characteristics, such as time, signal strength, angle of arrival, or angle of departure. The following subsections provide an overview of a few estimation techniques.

### 2.1.1 Triangulation and trilateration

Triangulation and trilateration are similar techniques that can be used for indoor location and other applications such as surveying and navigation.

| Solution | Technology | Techniques | Accuracy | Precision | Complexity | Cost |
|---|---|---|---|---|---|---|
| Microsoft RADAR | WLAN | RSS | 3-5m | 50% within 2.5m 90% withing 5.9m | Moderate | Low |
| Horus | WLAN | RSS | 2m | 90% within 2.1m | Moderate | Low |
| Ekahau | WLAN | RSSI | 1m | 50% within 2m | Moderate | Low |
| SmartLOCUS | WLAN Ultrasound | RSS RTOF | 2-15cm | 50% within 15cm | Medium | Medium to High |
| Sappire Dart | UWB | TDOA | <0.3m | 50% within 0.3m | response frequency 0.1Hz-1Hz | Medium to High |

Table 1: Wireless indoor positioning system solutions [19].

In trilateration, the location of a target point is determined by constructing three circles centered on each reference point. The radius of these circles is equal to the distance between the reference and target points. The intersection of these circles corresponds to the target's location. Figure 1 a) illustrates this technique, where R1, R2 and R3 are reference points and P is the target point we want to determine the location of.

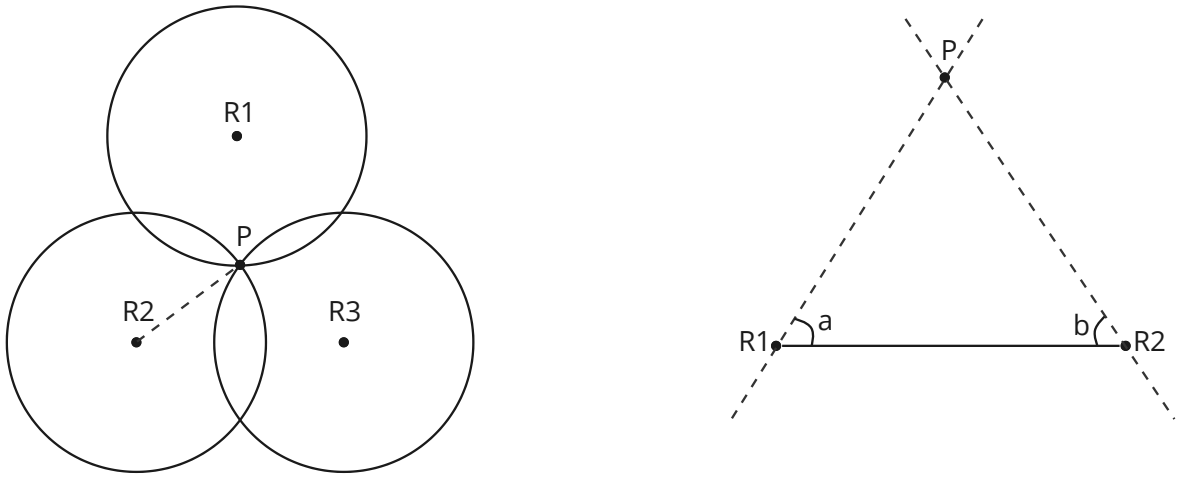
In triangulation, the location of a target point is calculated using the geometric properties of triangles. Assuming we have two reference points, R1 and R2, in a two-dimensional space, and we know their position and the angle a and b between each of them and a third point P, then the location of P will be the intersection of the lines created by the two reference points and their angle to P. When the three points are connected, we will have a triangle, and by using trigonometry principles, we can determine the exact location of the point P. This is illustrated in figure 1 b).

A key difference between triangulation and trilateration is the number of reference points used to determine a device's location. When triangulating a point in 3D space, it is necessary to use three or more reference points, whereas only two are required in 2D space. Trilateration always requires at least three reference points in two-dimensional or three-dimensional space.

Triangulation and trilateration are common techniques used for indoor location systems, and many techniques used in indoor location systems are derived from them. In this context, both techniques involve using multiple wireless signals, such as Bluetooth and WiFi, to determine a device's location. It is worth noting that the location accuracy of a device using these methods depends on the accuracy of the measurements between the device and the reference points. An inaccurate measurement may result in an inaccurate determination of the device's location through triangulation or trilateration. The accuracy of indoor location using these techniques can be affected by several factors, such as the quality and reliability of the wireless signals being used. Moreover, the number and positioning of the reference points, and the

presence of obstacles or interference that can degrade the signals will affect the accuracy of the location.

The following pages provide an overview of some techniques that support trilateration and triangulation.




(a) Trilateration (b) Triangulation

Figure 1: Illustrations of the triangulation and trilateration techniques.

#### 2.1.1.1 Time of Arrival (TOA)

Time of Arrival (TOA) is a trilateration-based technique that can be utilized to determine a device's location using multiple reference points. This method involves measuring the amount of time it takes for a signal to travel between the device and each reference point. The propagation time of a signal is directly proportional to the distance between a receiver and a transmitter, if both are in line of sight. Thus, we can estimate the distance between them and the device by measuring the one-way propagation time at each reference point. Finally, we can estimate the device's location by using the trilateration technique discussed before.

Similarly to trilateration, two-dimensional positioning requires the measurement of TOA from at least three receivers.

There are two main problems associated with this technique. First, the device and the reference points must be equipped with precisely synchronized clocks. Second, it is necessary to add a timestamp to the transmitted signal to calculate the distance it traveled. Moreover, the accuracy of TOA can be affected by the quality and reliability of the clocks being used and the presence of obstacles or interference that can affect the signals [19, 18, 2, 17].

#### 2.1.1.2 Time Difference of Arrival (TDOA)

Time Difference of Arrival (TDOA) works similarly to TOA but involves measuring the difference in the time of arrival of a signal at each reference point rather than the absolute time of arrival.

In general, TDOA techniques can be classified into two categories: multi-node and multi-signal.

In multi-node TDOA, TOA is measured at each reference point and a time difference is calculated between pairs of them. Doing so can define a hyperbola for each pair of reference points on which the device should be located. Where the two hyperbolas intersect is the location of the device. As a result, this technique requires at least three reference points [18]. Figure 2 illustrates this, where P is a device we want to locate and R1, R2 and R3 are reference points.

The disadvantages of this technique are similar to those of the TOA technique described above. This technique requires all reference points to be time synchronized rather than just the device and reference points. In both techniques, the accuracy of the location estimation is affected by the quality and reliability of the clocks and the presence of obstacles or interference that might affect the signals [2]. In contrast, it is not required that a timestamp accompany the device's signal.

For multi-signal TDOA, two different kinds of signals with different propagation speeds are used. To perform this technique, additional equipment is required. The distance between a device and a reference point can be calculated by measuring the time difference of arrival of both signals at a reference point [18].

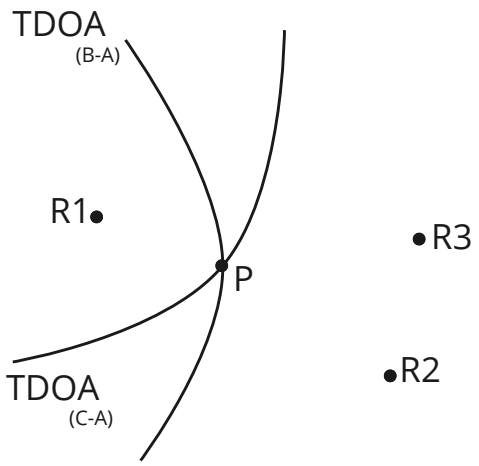



Figure 2: Muti-node time difference of arrival technique. Adapted from [18].

#### 2.1.1.3 Angle of Arrival (AOA)

The Angle of Arrival (AOA) technique utilizes the angles of arrival of signals at multiple reference points to estimate the device's location. Alternatively, this may be referred to as direction finding or Direction of Arrival (DOA). To achieve this, each reference point must be equipped with an antenna capable of determining a signal's angle of arrival. The device's location can be found at the intersection of several pairs of angle direction lines. Using the triangulation technique discussed above, we will be able to

determine the location of that device. This technique is illustrated in figure 3, P is the target device we want to locate, R1 and R2 are reference points and a and b are the angles of arrival of a signal.

Similarly to the requirements for triangulation, AOA needs at least two reference points for 2D positioning. As for three-dimensional space, a minimum of three reference points is required. Furthermore, there is no need for time synchronization, as is the case with TOA and TDOA. In addition, a timestamp is not required to be attached to any signal.

This technique, however, requires large and complex hardware, and location accuracy degrades as the target device moves away from the reference points. This technique depends on the accuracy of angle measurements taken by reference points, so the quality and reliability of the measuring apparatus influence it. Moreover, AOA can be affected by the presence of obstacles or interference that can interfere with the signals. An example would be multipath reflections arriving from misleading directions [17, 19].

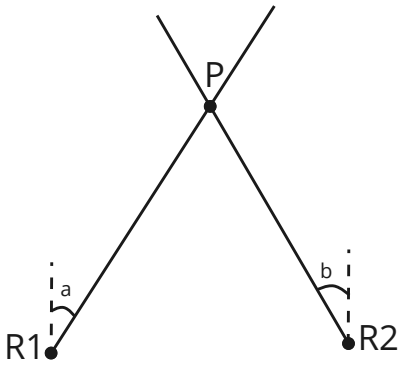



Figure 3: Angle of arrival technique. Adapted from [19].

### 2.1.2 Angle of Departure (AOD)

The Angle of Departure (AOD) is similar to AOA in that it involves measuring the angle of a signal and using that information to calculate a target's location. In AOD, however, we measure the angle at which the signal is transmitted from a reference point to the device.

In order to do this, the reference points must be equipped with an array of antennas. Using its antennas, the reference point broadcasts a data signal, and when each packet sent by the antenna array reaches the receiver's antenna, the distinct distance traversed from the transmitter results in a phase-shifted signal in reference to the previous signal.

Using these data, the device can calculate the angle of departure of each signal and estimate the position of the reference point [23].

## 2.2 Fingerprinting

The fingerprinting technique, also known as scene analysis [19], consists of collecting various radio signals measurements of a space (fingerprints) and estimating the location of a target device based on those measurements. This method relies on the fact that all locations receive unique radio signals that allow them to be distinguished from one another. Fingerprinting is generally more accurate than conventional infrastructure-based indoor positioning techniques such as the triangulation and trilateration approaches discussed previously. In addition, it can take advantage of existing infrastructure, such as WiFi networks, making it an extremely cost-effective technique [14]. The fingerprinting process can be divided into two phases: offline or calibration phase and online or real-time phase.

In the offline phase, a radio map is constructed by collecting fingerprints from multiple known locations. Due to the dynamic nature of radio signals, the amount of data collected in this phase reflects the system's accuracy. Higher levels of detail provide a higher level of accuracy during this phase. Various characteristics of the signals can be recorded in each fingerprint. Received Signal Strength (RSS) is commonly used in fingerprint-based systems. Figure 4 illustrates an indoor map where a device collects fingerprints at various locations. Table 2 is an example of a radio map based on those measurements.

This is the stage at which fingerprinting suffers from one of its major disadvantages. In large buildings, constructing a very detailed radio map is a challenging task due to the large number of fingerprints that must be collected. Moreover, radio maps need to be frequently updated because radio signals are constantly changing due to propagation effects, shifts in building layouts, and changes in the positions and numbers of access points.

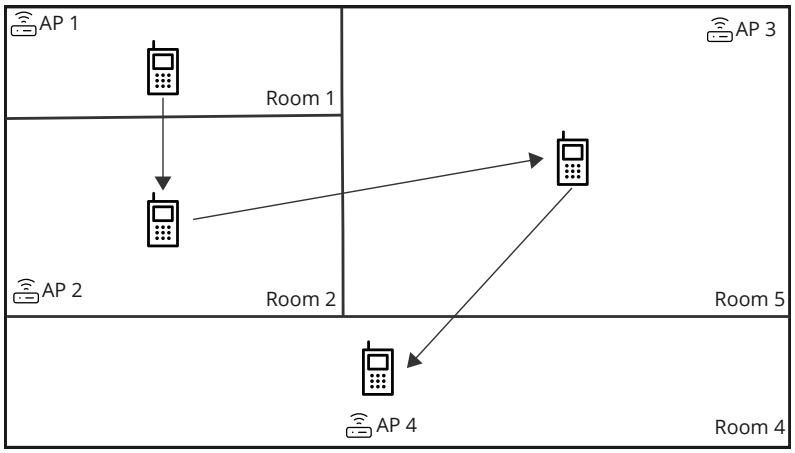



Figure 4: Offline training phase. Adapted from [14].

During the online phase, a positioning algorithm estimates the location of a device by comparing the offline data collected with the current data being gathered by the device. This process is illustrated in figure 5. The device collects a fingerprint at its current location. Then, the positioning algorithm estimates the device's location, based on the radio map obtained in the offline phase.

| Location | RSS(in dBm) | | | |
|---|---|---|---|---|
| | AP 1 | AP 2 | AP 3 | AP 4 |
| Room 1 | -90 | -70 | -60 | -50 |
| Room 2 | -70 | -90 | -55 | -70 |
| Room 3 | -60 | -50 | -85 | -50 |
| Room 4 | -40 | -50 | -60 | -90 |

Table 2: Radio map based on fingerprints taken in the offline phase. Adapted from [14].

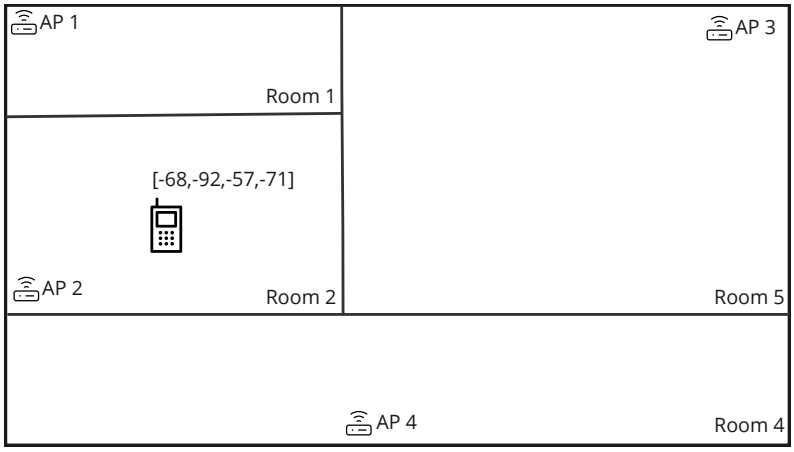



Figure 5: Offline training phase. Adapted from [14].

There are various fingerprint-based algorithms which use different methods to estimate the device's location, such as probabilistic methods, k-nearest-neighbor (kNN), neural networks, support vector machine (SVM), and smallest M-vertex polygon (SMP) [19]. Figure 6 is a schematic diagram of the fingerprinting process.

### 2.2.1 CSI Fingerprinting Systems

As previously stated, RSS fingerprints are used in many indoor localization systems due to their simplicity and low hardware requirements. RSS based approaches, however, have drawbacks. RSS values provide coarse information. In an indoor environment, RSS values are often highly variable over time due to multipath effects. Even stationary devices are susceptible to large location errors due to such high variability [34].

In a WiFi network, signals are propagated through Orthogonal Frequency Division Multiplexing (OFDM) modulation [20]. OFDM is a digital multi-carrier modulation technique. It works by dividing the available bandwidth into multiple narrowband sub-carriers and transmitting data in parallel over each sub-carrier.

Each sub-carrier is orthogonal to one another, meaning that the sub-carriers do not interfere with each other and can be recovered independently.

Channel State Information (CSI) represents fine-grained information about the communication link

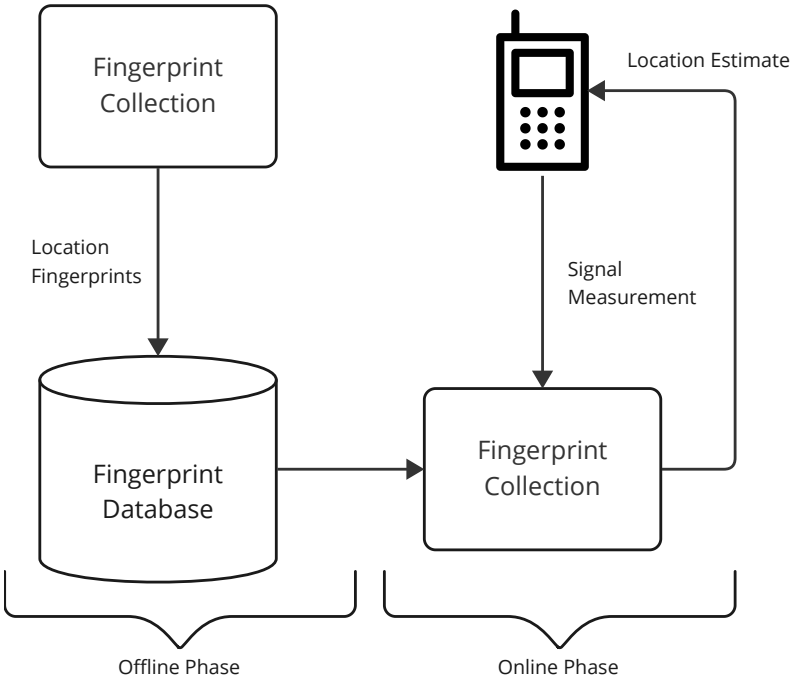



Figure 6: Fingerprint positioning principle diagram. Adapted from [20].

between the transmitter and the receiver. CSI can provide multi-channel subcarrier phase and amplitude information to better describe the signal's propagation path [20]. Studies have shown that CSI data proves to be much more stable than RSS [34].

The collection of CSI data on WiFi network interfaces was not always possible. CSI data was made available only after the implementation of OFDM and MIMO antennas in the IEEE 802.11a/n protocol [20]. And even then, reading these data was not straightforward.

Nowadays, some WiFi Network Interface Cards (NICs) allow the collection of CSI. One such example are the ESP32 microcontrollers, the NIC present in these devices allows the collection of CSI data that can be read using the Espressif IoT Development Framework (ESP-IDF) [11].

CSI has recently become popular in WiFi based high-precision indoor positioning. Since CSI data is more stable and fine-grained than RSS values, more accurate systems can potentially be constructed.

In addition, RSS based fingerprinting and CSI based fingerprinting can be combined to improve positioning accuracy. Jiang et al. [15] and Zhao et al. [7] proposed fingerprinting-based IPSs that use a combination of CSI and RSS values.

## 2.3 Market Research

An increasing number of companies are providing commercial solutions for indoor tracking of entities. Moreover, the technologies and methods these companies use in their IPSs solutions vary. WiFi, Bluetooth and UWB are the most popular technologies for communication. Many of the companies providing IPS currently in the market brand their system as a Real-Time Location System (RTLS). RTLS is a broader term

not necessarily limited to indoor positioning but to the real-time tracking of an entity in general, whether in an outdoor environment or indoors. In this section, some solutions currently present on the market will be presented. Although many of these solutions are branded as RTLS, many only deliver indoor solutions. In cases where an outdoor solution is involved, we will only focus on the indoor aspect of these systems.

### 2.3.1 Quuppa

The Quuppa Intelligent Locating System is a powerful one-size-fits-all RTLS technology platform for location-based services and applications. It provides accurate and real-time tracking for tags and devices using unique Direction Finding methods and advanced proprietary algorithms [24]. It claims sub-meter accuracy, less than one second response time and very low battery consumption. Quuppa offers a proprietary solution that leverages the combination of Bluetooth technology with the Angle of Arrival (AOA) and Angle of Departure (AOD) techniques [12]. The system can be divided into four components:

- Tags, devices and sensors;
- Locators;
- The positioning engine;
- Open APIs.

To implement the system, first, a plan of the number and location of locator devices is made and implemented in an indoor space. Entities are tracked using compatible tags, devices and sensors which send location data to the locators. The data is subsequently computed using the positioning engine to estimate the current entity's location. Finally, the Quuppa open APIs are a convenient and flexible way to connect and interact with the Quuppa Positioning Engine (QPE) to retrieve various types of data about tags, devices and sensors. This system can be used both for computing what is where (location) as well as to answer the question of what is happening over there via a dual-way communication channel over Bluetooth and the Locators.

Moreover, Quuppa allows controlling the level of accuracy by simply altering the number of locators. Thus, in use cases where more accuracy is needed an increase in the number of locators will increase the system's accuracy. The transmission rate for Standard Bluetooth channels is limited to 10 Hz, as per Bluetooth regulations, but the Quuppa system can also be configured to use proprietary channels, on which Quuppa Tags can be set to transmit at up to 50 Hz. Once installed, the Quuppa system typically requires no physical maintenance. The system is constantly monitoring itself and can send an alert if attention is needed. The Locators have inbuilt accelerometers, so if they are moved the system will know. If a Locator is offline, the system will know. In the case of a power failure, the system will even perform an automated recovery.

#### 2.3.1.1 Tags

Quuppa sells the QT1-1 Tag and QT3-1 Tag, which are designed to be easily attached to a person or object. The QT1-1 Tag is lightweight, waterproof, shockproof and IP67 classified, and it houses a 3-axis accelerometer, a programmable button and an LED [25]. Quuppa QT1-1 Tag can transmit one packet per second 24/7 for three years making roughly 94.6 million transmissions in total. Moreover, the onboard sensors or geofencing areas only activate the tag when needed, extending the battery lifetime by several years [24]. Similarly, the QT3-1 Tag is lightweight, waterproof, shockproof and IP68 classified, and it houses an accelerometer, a programmable button and an LED. This tag is more robust than the QT1-1 Tag, and has a battery life of up to 10 years, making it ideal for industrial environments. Although Quuppa only has two tags in the market, it allows total freedom in tag design. You can choose between multiple Quuppa tag partners [5], integrate the QT1 Tag Module into your device, or design a tag using Quuppa's firmware libraries and schematics. Thus, any Bluetooth-enabled device can be made Quuppa-trackable with some software modifications.

#### 2.3.1.2 Locators

Quuppa offers two options for locator nodes: the Q17 Locator and the Q35 Locator. The former is recommended to be used in smaller systems such as offices. The later is designed to be used in industrial and outdoor environments.

### 2.3.2 U-blox

U-blox offers a portfolio of solutions and services covering all of the technological building blocks required to build RTLS systems. This company's offering of short-range solutions includes the components required to build high-precision indoor positioning solutions utilizing Bluetooth Direction Finding technology. Bluetooth Direction Finding can determine the direction in which radio signals travel from the mobile client to one or several fixed anchor points by leveraging both AOA and AOD techniques [4, 10]. Besides Bluetooth Low Energy modules that can be integrated with custom tags using Bluetooth Direction Finding, U-blox also offers a set of kits that include complete tags, anchor points, and software to create an Indoor Positioning System.

#### 2.3.2.1 XPLR-AOA-1 kit

You can gain first-hand experience with Bluetooth direction finding with the U-blox XPLR-AOA-1 explorer kit. This kit includes an antenna board (C211), a tag (C209), as well as the necessary software for leveraging Angle of Arrival technology. A Bluetooth receiver can detect a moving tag's direction or angle using AOA methods, transmitting a signal with Constant Tone Extension (CTE) appended. Based on the NINA-B406 Bluetooth LE module, the C209 tag will send Bluetooth 5.1 advertisement messages. A NINA-B411 Bluetooth LE module is integrated into the C211 antenna board, which receives messages

and runs the u-connectLocate algorithm to determine the tag's location. The algorithm calculates angles in two dimensions by utilising the entire array of antennas on the C211 board. By itself, the XPLR-AOA-1 kit can detect if an object is approaching a door, track goods passing through a gate, prevent collisions between automated guided vehicles, or follow assets moving within a room with a camera. It is possible to create a positioning system by combining several XPLR-AOA-1 kits and triangulating the directions from three or more C211 boards [16, 38].

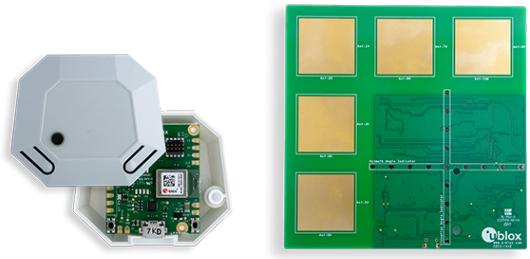


Figure 7: The XPLR-AOA-1 kit [38].

#### 2.3.2.2 XPLR-AOA-2 kit

The more advanced u-blox XPLR-AOA-2 explorer kit includes all the elements necessary to achieve submeter-level position accuracy in indoor environments. The kit includes four u-blox C211 antenna boards that function as anchor points, four u-blox C209 tags for tracking mobile assets, and all software required to leverage AOA techniques and Bluetooth Direction Finding. Similarly to the XPLR-AOA-1 explorer kit, both the tag and the anchor point use the u-blox NINA-B4 Bluetooth 5.1 low energy modules and the u-blox u-connectLocate software to calculate the angles of the incoming radio signals on the C211 antenna boards. An external positioning engine is included in the software package that triangulates the tag's position based on the angles between it and the anchor points. XPLR-AOA-2 is system agnostic and is compatible with various tracking solutions, both local and cloud-based, but is easily integrated with Traxmate's IoT tracking platform. This application allows users to upload a room's floor plan and deploy and configure the antenna boards directly on the map. Once the system has been deployed and configured, live tracking of the tags can be performed [16, 39].

#### 2.3.2.3 XPLR-AOA-3 kit

The XPLR-AOA-3 kit features the latest generation of antenna boards developed by u-blox for Bluetooth direction finding and indoor positioning. It is primarily intended for users to evaluate the newly released

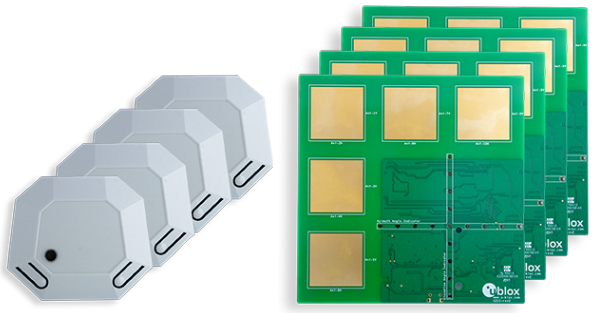
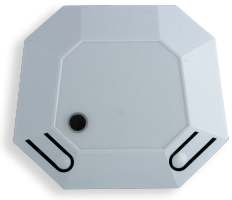


Figure 8: The XPLR-AOA-2 kit [39].

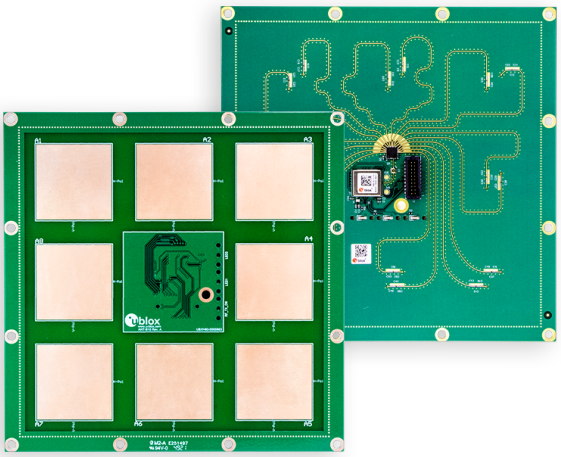
ANT-B10 antenna board and the optimized direction-finding algorithm developed by u-blox. It consists of an ANT-B10 antenna board, an EVB-ANT-1 development board, a C209 AoA tag, and all necessary software for operating the kit and evaluating the u-blox direction-finding solution. There are eight patch antennas on the ANT-B10 antenna board, and a u-blox NINA-B411 Bluetooth 5.1 module that calculates the angle of incoming radio signals using the u-blox's u-connectLocate software. The board is designed to be embedded into commercial products to provide low-power, high precision indoor positioning and to accelerate the evaluation, testing, and commercialization of Bluetooth-based solutions for indoor positioning. Developers can evaluate ANT-B10 antenna boards quickly and easily with the EVB-ANT-1 application board. The system incorporates an NXP RT1061 MCU for configuring and developing direction-finding applications, as well as an Ethernet PHY chip and the u-blox MAYA-W1 WiFi module. In a matter of seconds, EVB-ANT-1 can be connected to ANT-B10 with its off-the-shelf pin header, providing the user with a ready-to-use AOA indoor positioning anchor point. As with the XPLR-AOA-1 kit, the XPLR-AOA-3 kit can be utilized to explore a variety of indoor positioning applications, however, to create a positioning system, several XPLR-AOA-3 kits must be combined to triangulate the directions coming from three or more ANT-B10 antenna boards [40].

### 2.3.3 Azitek

Azitek is a startup company based in Porto, Portugal, specializing in developing of state-of-the-art Real-Time Location System (RTLS), which operate on license-free radio frequencies. Founded in 2019, Azitek was created out of the necessity of having a long-range and accurate Real-Time Location Systems (RTLSs) capable of simultaneously serving endless autonomous vehicles both indoors and outdoors. With Azitek's Asset Management System, companies can better understand their assets and manage them more efficiently. To implement the system, assets are equipped with ultra-low-cost, low-profile Azitek Active IoT tags. Azitek Anchor gateways are then installed in industrial facilities, warehouses, and vehicles.



(a) The C209 tag.



(b) The ANT-B10 antenna board and EVB-ANT-1 application board.

Figure 9: The XPLR-AOA-3 kit [40].

On the Azitek Dashboard, all the data about the assets is displayed, including asset status, location and history. The company's proprietary technology enables tag detection up to 400 meters away with a battery life of six years. Furthermore, each Azitek Gateway can monitor up to 50000 tags in real-time, and its onboard memory prevents data loss in the event of a network outage. The system utilizes a proprietary protocol operating at sub-GHz to avoid interference with WiFi and Bluetooth.

### 2.3.4 STANLEY Healthcare - AeroScout

Stanley Healthcare offers a real-time hospital asset management system called AeroScout. AeroScout automates manual equipment management processes, allowing hospital staff to view the location and status of assets in real-time. Moreover, it is capable of capturing and analyzing data regarding the use of assets. AeroScout tags are attached to equipment and communicate wirelessly with AeroScout's visibility platform, called MobileView. On that platform, asset location and status are automatically logged and monitored [27].


(a) Azitek Active IoT tags.

(b) Azitek Anchor gateways

Figure 10: The Azitek's Asset Management System hardware [3].


Figure 11: The Azitek Dashboard [3].

#### 2.3.4.1 T2s

The T2s tags can be deployed on any standard WiFi network infrastructure, resulting in lower costs and a faster and easier deployment process. These tags can be used to determine the location and status of persons and assets. T2s tags transmit wireless messages to MobileView through WiFi access points. These tags can be worn by staff or attached to various assets, such as medical devices. As a result, patients, staff, and assets can be located accurately in real-time. The T2s Tag is equipped with on-board motion sensors. This motion sensor enables the tag to be configured with different transmission intervals depending on whether it is stationary or in motion. Doing so reduces unnecessary network traffic, and battery life is conserved. T2s tags have a single replaceable battery that can provide power for up to four years. Additionally, MobileView can monitor the battery life of the tags [30].


Figure 12: The AeroScout T2s tag [33].

#### 2.3.4.2 T12

As with the T2s tag, the T12 tag transmits its location using standard WiFi technology. T12 tags, however, are designed to be attached to medical equipment such as infusion pumps, defibrillators, and portable X-ray machines rather than people. Furthermore, the T12 Tag is capable of bi-directional communication, which means it can receive and transmit data. This means that the T12 tag can receive firmware and configuration updates from MobileView, eliminating the need to manually collect, update, and re-deploy the tags. T12 tags are also equipped with a motion sensor to conserve battery energy when they are not in motion. Its user-replaceable battery provides up to 3 years of battery life [28].

#### 2.3.4.3 T12s

T12s tags provide all of the features of T12 tags previously discussed, but they are powered by commercially available CR2 batteries that can be changed easily without unscrewing anything. This provides a battery life of up to two years for the tag [29].


Figure 13: The AeroScout T12 tag [31].


Figure 14: The AeroScout T12s tag [32].

### 2.3.5 Summary

First of all, several different tags are available on the market that use different technologies. However, it seems that BLE-based tags have increased in recent years.

Second, the tags discussed can only be used as part of the ecosystem of their manufacturer. As such, tags cannot be acquired separately and be used as components of other systems.

Regarding the size and autonomy of the tags, there are a number of options available. For the tags presented in this work, the autonomy ranges from two years for STANLEY's Healthcare T2s tag to ten years for the Quuppa QT3-1 tag. The U-blox tags, on the other hand, do not come with a battery included with them.

Table 3 summarises a few aspects of the different tags discussed earlier.

| Tag | Accuracy | Autonomy | Technology |
|---|---|---|---|
| Quuppa's QT1-1 | <1m | 3 years | BLE |
| Quuppa's QT3-1 | <1m | 10 years | BLE |
| Azitek's tag | — | 6 years | — |
| U-blox's C209 | <1m | — | BLE |
| AeroScout's T2s | — | 4 years | WiFi |
| AeroScout's T12 | — | 3 years | WiFi |
| AeroScout's T12s | — | 2 years | WiFi |

Table 3: Summary of some aspects of studied tags.

# 3

# Problem Statement and System Architecture

In this chapter, we will provide a synopsis of the issue this work aims to solve and discuss the system's architecture. First, we will identify the issue and the objective of this work. Then, we will evaluate the functional and non-functional requirements needed to address the problem. Afterwards, an outline of the positioning system structure in which our tag can be incorporated in will be presented. Finally, the primary task of each part of the system will be briefly explained.

## 3.1 The problem

As we saw in chapter 2, only a few offers were developed around WiFi and fingerprinting. Most of the offers currently on the market try to leverage Bluetooth Direction Finding. Additionally, we discussed how Channel State Information (CSI) can be used to improve traditional WiFi and fingerprinting-based indoor positioning systems. The main goal of this work is to create a tag that is capable of collecting and sharing CSI data with a positioning engine. In addition, the tag should also be capable of collecting and sharing traditional fingerprinting data, such as Basic Service Set Identifier (BSSID), and Received Signal Strength Indicator (RSSI).

## 3.2 Requirements

In the previous section, we discussed what the problem is that we are trying to solve. Having that in mind we can identify some functional and non-functional requirements.

First, we need a hardware and software combination that makes the collection of CSI data possible. Moreover, to transmit CSI and other data to a remote positioning engine, the tag needs to be capable of connecting and transmitting over a WiFi network.

Because the tag will be used to track objects or people, it should be relatively small and lightweight.

Finally, the tag should have some degree of customization regarding its configuration. To make it easy for users to change these configurations, the tag should be able to be remotely configured.

Below is a summary of the functional and non-functional requirements we want to achieve.

Functional Requirements:

- Collect fingerprints with BSSID, RSSI and CSI data about the surrounding access points;
- Connect to a WiFi network;
- Send collected fingerprints to a remote positioning engine over WiFi, periodically;
- Be capable of being remotely configured.

Non-functional Requirements:

- Have a good autonomy;
- Be small and lightweight.

## 3.3 General System Architecture

The indoor positioning system our tag could be integrated into consists of one or more tags, access points, a server running a positioning engine, a configuration server, and any device with WiFi capabilities and a web browser. Figure 15 is an illustration of this system.

As discussed before, the tag should integrate an indoor positioning system based on fingerprinting techniques. The positioning engine server is responsible for receiving fingerprints from the tags and inferring their position. For this, as discussed in chapter 2, a radio map is constructed beforehand in the offline phase. By leveraging the data, during the online phase, the positioning engine can estimate a tag's location by using its fingerprints.

Each tag of the system can be configured in various ways. We will discuss this configuration capability in the next chapter. The configuration server is responsible for storing configuration information about each tag in the system. Moreover, it provides these data when requested by the tags.

The tag is connected to a WiFi network and can be configured in two modes, RSSI data or CSI data. In the RSSI data mode, the tag periodically makes scans for different Access Points (APs) in its surroundings for a limited and configurable amount of time. Then it constructs a fingerprint with some of the collected data, including the APs BSSIDs, RSSIs, and Media Access Controls (MACs). This fingerprint is stored in the tag's memory. In the CSI data mode, the tag periodically listens for CSI data packets from communications between it and the currently connected AP. To generate these CSI data packets, the tag sends pings to the AP. The tag generates one fingerprint for each captured CSI data packet containing, among others, the CSI data vector and the connected AP's MAC. Similarly to the RSSI mode, the tag stores these fingerprints in its memory. In both modes, the tag periodically sends the fingerprints stored in memory to the positioning server. The communication mechanisms and data structure of the fingerprints will be discussed in more detail in chapter 4. It is possible to configure various configuration values of the tag in three ways. First, the tag is configured with default values that are stored in a file in non-volatile memory. Editing the values

in this file can change this configuration. Moreover, the tag can be configured via a web interface by any device capable of using a web browser and WiFi connectivity. Finally, every time the tag is initialized, it reads its default configurations file in search for a remote configuration server address. If found, the tag requests the server for configuration information that is then stored in non-volatile memory. The structure of the configuration data and the communication mechanisms regarding configuration will be discussed in chapter 4.

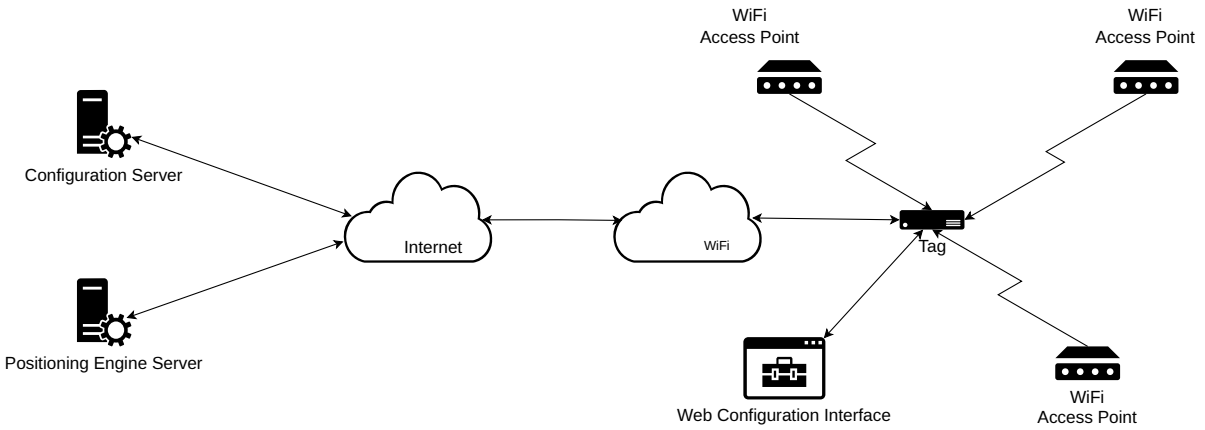



Figure 15: General System Architecture

# 4 System Specification

In this chapter we will discuss the system specification. First, we will explain the software and hardware choices made. Then, we will discuss a data format for configuration files stored in and transmitted to the tag. After, we will examine how the tag communicates with the configuration and positioning engine servers. Lastly, we will show various machine state diagrams for various stages of the tag's lifecycle.

## 4.1 Hardware and Software

In chapter 3, we discussed the goals of this work. Given the requirements, we must choose a combination of hardware and software to accomplish our goals. In the following two sections we will analyse the hardware and software combination used in this work.

### 4.1.1 Hardware

Regarding hardware, we will need a device capable of WiFi connectivity, that is small, lightweight, battery-powered, and energy-efficient. Moreover, the device must be able to detect nearby APs and collect AP specific data, such as RSSI, BSSID, and MAC. Finally, the device needs to be able to collect CSI data from WiFi communications.

Given this set of requirements, we can note that most microcontrollers would be able to satisfy most of the requisites. But currently, only a few of them can collect CSI data from WiFi communications. Espressif Systems makes a set of microcontrollers called ESP32 [6]. They are relatively cheap, small and capable of collecting CSI data. We decided to employ one of these microcontrollers in our work. We decided to use a development board to develop our solution more effectively. A development board is a type of microcontroller commonly used for prototyping. They are usually breadboard friendly, reducing the need for soldering while developing our solution.

We also decided that it would be ideal to add a couple of electronics to our solution, for example LEDs and buttons for specific functions of our tag. For this we would need at least a breadboard, jumper cables, LEDs, resistors and push switches.

Given this, a quick market research revealed that there were affordable kits that included, among others, all the tools we discussed. Finally, we decided to use a kit made by Freenove that is based on the ESP32-WROVER-E microcontroller. Figure 16 shows the kit and the hardware we will use from it. Figure 17 shows the ESP32-WROVER-E microcontroller.


Figure 16: Freenove Started Kit for ESP32 and used hardware from it.

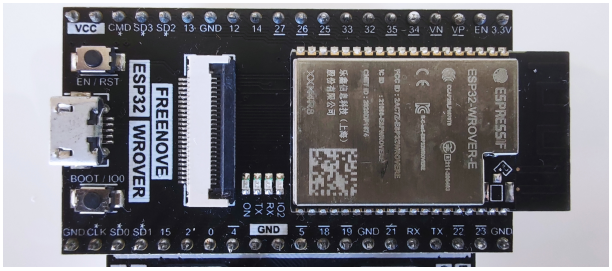



Figure 17: ESP32-WROVER-E microcontroller.

We also have access to an ESP32-WROOM-32D that we will use for testing our solution, as shown in figure 18.

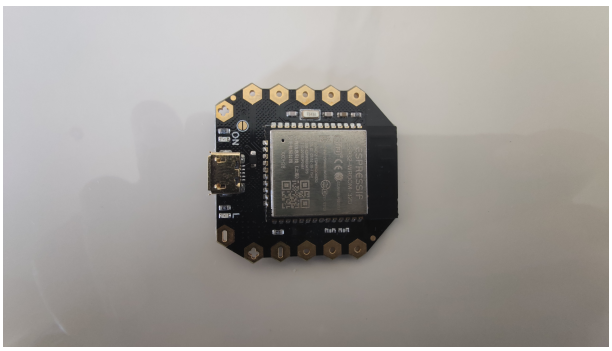


Figure 18: ESP32-WROOM-32D microcontroller.

### 4.1.2 Software

The microcontroller chosen in the previous section does not include any software. Thus, all the necessary software must be constructed to have a working device.

We needed to decide which frameworks and programming languages to use to construct the tag's software. These decisions were highly dependent on the hardware discussed in the previous section. Recall that we decided to use ESP32 microcontrollers because they could collect CSI data. Given this, we were bound to use Espressif's official IoT Development Framework, or ESP-IDF for short, because it was the framework that would allow us to collect CSI data.

ESP-IDF provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++ [13].

For programming languages we decided to use the C programming language for most of the constructed software. We will also use the JavaScript programming language along with HTML and CSS to build a web interface to configure the tag remotely.

## 4.2 Tag Configuration

This section will discuss a data format for the tag configuration.

We want our tag to be configurable. Thus, we must create a data structure to store and communicate such configurations. Since JavaScript Object Notation (JSON) is a widely used file and data interchange format, we will choose this format to structure our configurations. Bellow, in listing 1 we can see an example of such a configuration.

The "TAG_NAME" is the identifier of the tag. The "FINGERPRINTS_SERVER" is a string with an address for a server capable of receiving fingerprints from the tag. The "CONFIG_SERVER" is an address for a server capable of returning a configuration file, in this format, to the tag. The "FINGERPRINT_SERVICE_SLEEP" is the time in milliseconds that the tag stays in a sleep mode after collecting a fingerprint. The "FINGERPRINT_SERVICE_COLLECT" is the number of milliseconds the tag actively collects fingerprints. The "MESSAGE_SERVICE_SLEEP" is the number of milliseconds the tag stays in a sleep mode after sending fingerprints to the fingerprints server. The "QUEUE_SIZE" is the number of fingerprints stored in an internal queue. The "MAX_WIFI_CONNECT_RETRIES" is the maximum number of attempts the tag will try to connect to a WiFi network. The "CSI_MODE" is a boolean for configuring the tag to collect CSI fingerprints if true, or RSSI fingerprints otherwise. Finally, the "WIFI_DETAILS" is an array of objects representing a WiFi network the tag can connect to. Each object comprises a PWD and SSID keys corresponding to the WiFi password and Service Set Identifier (SSID), respectively. We will take a closer look at the application of these values in the subsequent chapter.

```
{
    "TAG_NAME": "tagDiogoRio",
    "FINGERPRINTS_SERVER": "http://server/S02/i2a/i2aSamples.php",
    "CONFIG_SERVER": "http://server:8080/S11/boot",
    "FINGERPRINT_SERVICE_SLEEP": 1000,
    "FINGERPRINT_SERVICE_COLLECT": 1000,
    "MESSAGE_SERVICE_SLEEP": 11000,
    "QUEUE_SIZE": 15,
    "MAX_MESSAGE_ATTEMPTS": 3,
    "MAX_MESSAGE_RETRIES": 3,
    "MAX_WIFI_CONNECT_RETRIES": 3
    "CSI_MODE": true,
    "WIFI_DETAILS": [
      {
        "PWD": "123456710",
        "SSID": "Rede-1"
      },
      {
        "PWD": "123456789",
        "SSID": "Rede-2"
      }
    ]
}
```

Listing 1: Tag configuration file example.

## 4.3 Communication Protocols

This section will discuss the communication protocols used in our solution. First, we will discuss the tag and configuration server communication. Then, we will discuss the communication between the tag and the positioning engine server. Lastly, we will suggest a communication protocol for communicating CSI fingerprints from the tag to the positioning server.

### 4.3.1 Configuration Server

As we saw in chapter 3, the tag must be able to establish communication with remote servers. The tag receives configuration files from a configuration server, and transmits fingerprints to a server that estimates its position. To do both of these tasks, a communication mechanism needed to be implemented.

There were already communication mechanisms from previous works for the configuration and fingerprints servers. Both of the servers have REST Application Programming Interfaces (APIs) that can be used through HTTP requests.

The configuration server can store JSON files for each tag identifier. We saw in the previous section a

JSON data format for configuration files. The configuration server was configured to return a JSON object of the same format when an HTTP POST request is made to the server. The tag must send a body in an x-www-form-urlencoded format along with the POST request. The x-www-form-urlencoded is a format that consists of key-value tuples. The keys and values are encoded in a way where the key-value tuples are separated by '&', with a '=' between the key and the value. An example of a correct body payload can be seen below, in listing 2.

```
1  object_ = { "BSSID":"E8:68:E7:2D:96:08" }
```

Listing 2: Body of a POST request to the configuration server.

We have an 'object_' key and its value is a string that encodes a JSON object with a "BSSID" parameter. The "BSSID" parameter is a unique identifier corresponding to the tag's MAC address.

### 4.3.2 Positioning Engine Server

The fingerprints server receives RSSI fingerprints sent by the tag. It uses the fingerprints to estimate the tag's position internally. Similarly to the configuration server, the fingerprints server receives the fingerprints in an x-www-form-urlencoded formatted body of a POST request. Bellow, in listing 3, we can see a valid example of the body of a POST request.

We have a "scanData" key and its correspondent value is a string that represents a JSON object with multiple values. The "tagName" is a string with the name of the tag sending the fingerprint. The "tagBSSID" corresponds to the MAC address of the tag's physical WiFi interface. The "tagNetwork" represents the SSID of the current network the tag is connected to. The "dataType" parameter instructs the server which data type it is receiving. For RSSI fingerprints this parameter should be populated with "WiFi". The "scanMode" parameter can be "auto" or "manual". The "auto" mode means this fingerprint was collected automatically, and the "manual" mode means the fingerprint was collected manually, that is, after pressing a button in the tag. In our case we will always use the "auto" mode. It is in the "WiFiData" that the fingerprint information lies. It is an array of objects, each containing the BSSID, RSSI, and name of the detected APs during the scan. The name corresponds to the SSID of the networks.

### 4.3.3 CSI Fingerprinting

Our solution also needs to send CSI fingerprints. The fingerprints server previously discussed only supports RSSI fingerprints. Thus, we need to design a data format to send CSI fingerprints. To facilitate further development in the fingerprints server, we will reutilize most of the data format and communication protocols used for RSSI fingerprints. We propose that instead of having a "WiFiData" array, we have a

```
scanData = {
    "tagName": "tagDiogoRio",
    "tagBSSID": "00:11:22:77:88:99",
    "tagNetwork": "fakeOne",
    "dataType": "Wi-Fi",
    "scanMode": "auto",
    "WiFiData": [
        {
            "bssid": "03:6a:01:53:f1:1d",
            "rssi": -52,
            "name": "eduroam"
        },
        {
            "bssid": "11:b8:98:54:3e:a5",
            "rssi": -87,
            "name": "Network1"
        },
        {
            "bssid": "76:56:ab:b6:09:f9",
            "rssi": -78,
            "name": "Network2"
        },
        {
            "bssid": "77:68:86:10:c1:f6",
            "rssi": -61,
            "name": "Network3"
        },
        {
            "bssid": "f5:77:cb:13:6f:7e",
            "rssi": -58,
            "name": "fakeOne"
        }
    ]
}
```

Listing 3: Body of a POST request to the fingerprints server with RSSI data.

```
scanData = {
    "tagName": "tagDiogoRio",
    "tagBSSID": "00:11:22:77:88:99",
    "tagNetwork": "fakeOne",
    "dataType": "CSI",
    "scanMode": "auto",
    "CSIdata": {
        "apMAC": "fa:09:54:3a:8c:55",
        "rssi": -48,
        "channel": 11,
        "secondaryChannel": 0,
        "timestamp": 14061827,
        "data": [-80, 4, 0, 15, 20, 16, 20, 16, 20, 17, 20, 17, 19, 17,
↪  19, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 17, 18,
↪  17, 19, 16, 19, 15, 19, 14, 20, 13, 20, 12, 19, 11, 19, 10, 19, 8,
↪  19, 7, 19, 6, 19, 1, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 1, 2, 5, 11, 5, 11, 4, 12, 4, 12, 4, 13, 5, 14, 5, 15, 5,
↪  15, 5, 16, 5, 16, 5, 17, 5, 17, 5, 17, 5, 18, 6, 18, 6, 19, 6, 19, 7,
↪  19, 8, 19, 9, 19, 10, 20, 11, 20, 12, 20, 12, 20, 13, 20, 14, 20, 8,
↪  9, 32, 39, 33, 38, 34, 38, 34, 37, 35, 37, 36, 35, 36, 34, 37, 34,
↪  37, 33, 37, 33, 38, 33, 38, 34, 38, 34, 38, 34, 36, 34, 35, 35, 33,
↪  35, 32, 36, 30, 37, 28, 37, 26, 36, 23, 36, 21, 36, 19, 36, 17, 37,
↪  13, 37, 9, 36, 6, 37, 1, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 4, 13,
↪  16, 12, 19, 11, 21, 11, 22, 10, 23, 11, 24, 11, 25, 11, 26, 11, 28,
↪  11, 30, 11, 30, 11, 30, 11, 32, 11, 33, 12, 34, 13, 35, 14, 35, 15,
↪  36, 16, 36, 17, 36, 19, 36, 21, 36, 23, 37, 24, 37, 25, 37, 27, 38,
↪  29, 38, 30, 38]
    }
}
```

Listing 4: Body of a POST request to the fingerprints server with CSI data.

"CSIdata" object with various CSI information. Moreover, when sending CSI fingerprints, the "dataType" string shall be populated with "CSI". Listing 4 shows a valid example of our CSI fingerprint data format.

The "CSIdata" object will contain the MAC address of the AP, the WiFi's RSSI, the current primary and secondary channels, a timestamp and a "data" array. The "data" array carries the CSI data. Each number represents a specific information about the WiFi connection [35].

## 4.4 Software Specification

In chapter 3 we described the requirements of our tag in section 3.2. We then gave a general overview of the system and of the tag's responsibilities in section 3.3. With this in mind, we can create a state machine diagram that represents the most important states that our tag can be in. This state machine diagram is depicted bellow, in figure 19.

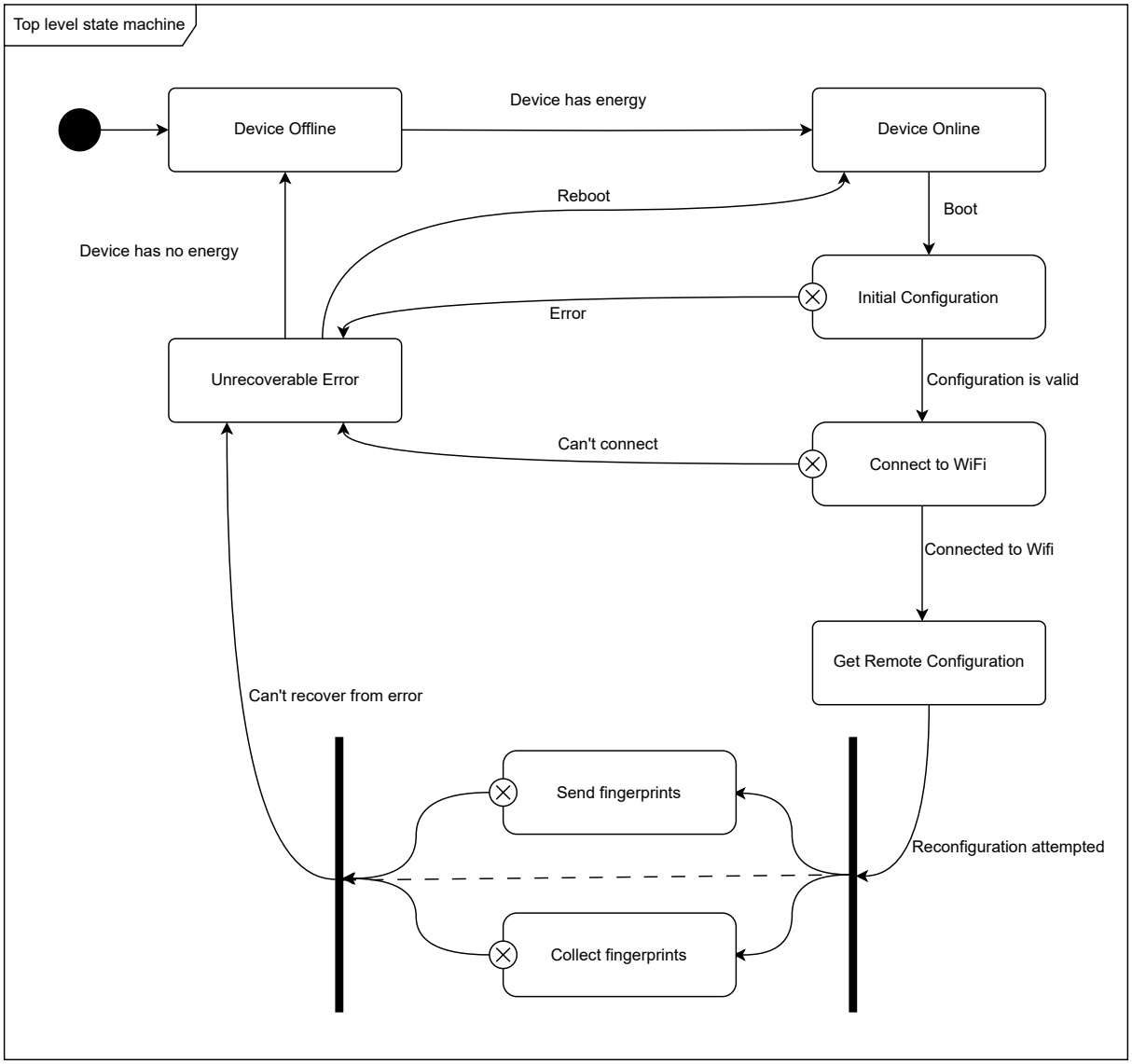



Figure 19: Top level state machine diagram of the tag's system.

First, we have a "Device Offiline" and "Device Online" state. The first, represents a state where the microcontroller has no power or is not connected to it. Naturally, in this state no operations are being performed. The latter, represents the state where energy is provided to the microcontroller, and the microcontroller is able to start performing computations. At this stage, the microcontroller starts performing internal computations known as bootloading. After the bootloading is complete, the microcontroller enters

a stage of application startup, where it starts running the software flashed into its memory.

The "Inital Configuration" state is the state where the tag reads a configuration stored internally and stores a representation of such in memory. It is possible to enter this state after the microcontroller has booted and started the application startup, leaving the "Device Online" state. There are two possible ways out of this state. One is if there is an unrecoverable problem, for example, a corrupt configuration file. Under these circumstances, the tag goes to an "Unrecoverable Error" state. The other, and expected, path is when the configuration file is valid and correctly loaded into the microcontroller's memory. In this case, the tag enters the "Connect to WiFi" state. We have covered a data format to store configuration files in section 4.2. We will dive deeper into this state in subsection 4.4.1.

In the "Connect to WiFi" state, the tag already has the configurations loaded in memory. As we saw in section 4.2, the configurations are composed of an array of WiFi networks the tag can connect itself to. There are two ways out of this state. One is when there is an unrecoverable error, preventing the tag from connecting to a WiFi network. For example, when none of the configured WiFi credentials are valid. When this happens, the tag goes into an "Unrecoverable Error" state. If an error does not occur and the tag can connect to a WiFi network successfully, we enter a "Get Remote Configuration" state.

The "Get Remote Configuration" state is where the tag requests a configuration from a remote server. Note that, whether the tag successfully acquires a configuration from the remote configuration server or not, it can continue operating and will leave this state. For instance, if the tag's request to the remote configuration server is not done successfully, the tag can use its initial configuration. The same can happen if the request is done successfully, but it was not possible to read the configurations for any reason. Given this, after the tag attempts to reconfigure itself with configurations from a remote configuration server, it leaves this state.

After leaving the "Get Remote Configuration" state, the tag enters in two states in parallel. The "Send fingerprints" and "Collect fingerprints" states. The tag will remain in these states indefinitely, leaving only when it faces an unrecoverable error.

In the "Collect fingerprints" state, the tag goes through cicles of collecting fingerprints around it, storing them in memory and sleeping for a configurable amount of time.

In the "Send fingerprints" state, the tag reads all of the fingerprints stored in memory and sends them to a remote server. In this state the tag will send all of the fingerprints it has stored and then enters a sleep state for a configurable amount of time.

Lastly, the "Unrecoverable Error" state, is a state that, as we discussed, is entered when an unrecoverable error happens in other states. After entering this state, the tag stops all computations. There is two possible ways out of this state. When the tag stops having power, it enters the "Device Offline" state. If the tag has power, it will stay in this state indefinitely, until a reboot of the microcontroller is done by removing and applying power again, or by pressing the reboot button in the tag.

We have now discussed at a higher level each of the main states. In the following subsections we will discuss in more detail some of these main states. We will not discuss further the "Online Device", "Offline Device" and "Unrecoverable Error" states, given that they are very simple states and are already

covered above.

### 4.4.1 Initial Configuration

In this section we will discuss the "Initial Configuration" state in more detail. Figure 20 depicts this state machine diagram.

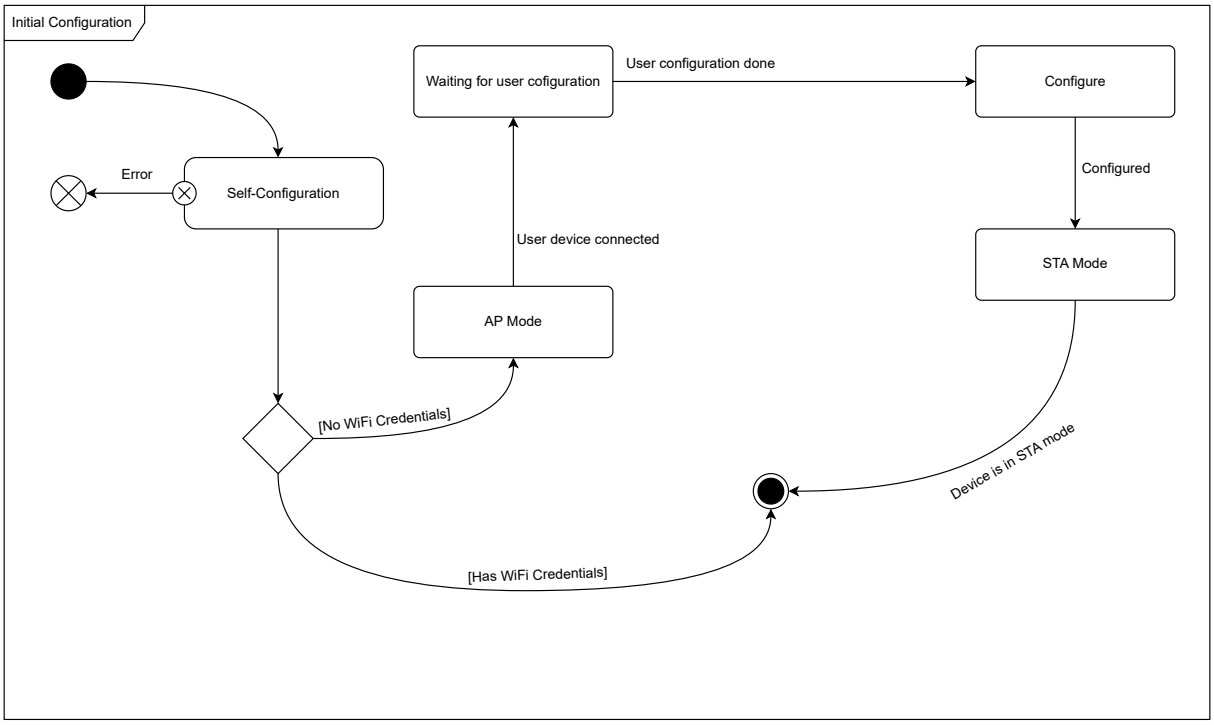



Figure 20: State machine diagram of the Initial Configuration state.

When the tag enters this state, it starts by entering a "Self-Configuration" state. In this state, the tag reads a configuration file stored in non-volatile memory. Figure 21 illustrates this state.

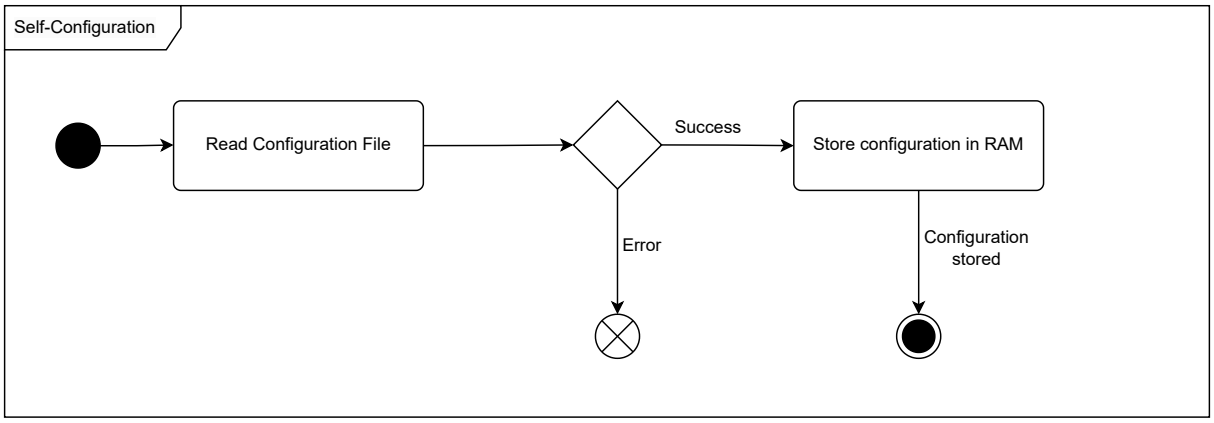



Figure 21: State machine diagram of the Self-Configuration state.

Its possible to leave this state in two ways. One is, if the tag reads the file successfully, the file has the correct data format and the configurations are loaded into memory. Another is if the tag is unable to read the configuration file with success. In the first case, the tag leaves this state and continues the flow of the "Initial Configuration" state. In the latter case, there is an error reading the file and the tag leaves the "Initial Configuration" state to an "Unrecoverable Error" state, as depicted in figure 19.

After leaving the "Self-Configuration" state with success, the tag reads the configurations and tries to find a WiFi network in the configuration file. If the tag can find a WiFi network in the configuration, it exits the "Initial Configuration" state.

If the tag is not able to find a WiFi network, it will then configure itself as an AP and provide a web API for users to edit the tag's configuration. It will then enter the "Waiting for user configuration" state. When a user configures the tag, the tag moves on to a "Configure" state. In this state, the tag will load the configurations provided by the user into memory. Moreover, the tag will store the new configurations in non-volatile memory. After this, the tag proceeds to the "STA Mode" state, where the tag reconfigures itself to be in station mode. When that reconfiguration is complete, the tag leaves the "Initial Configuration" state to the "Connect to WiFi" state, as illustrated in figure 19.

### 4.4.2 Connect to WiFi

In this section we will discuss the "Connect to WiFi" state in more detail. Figure 22 depicts this state machine diagram.

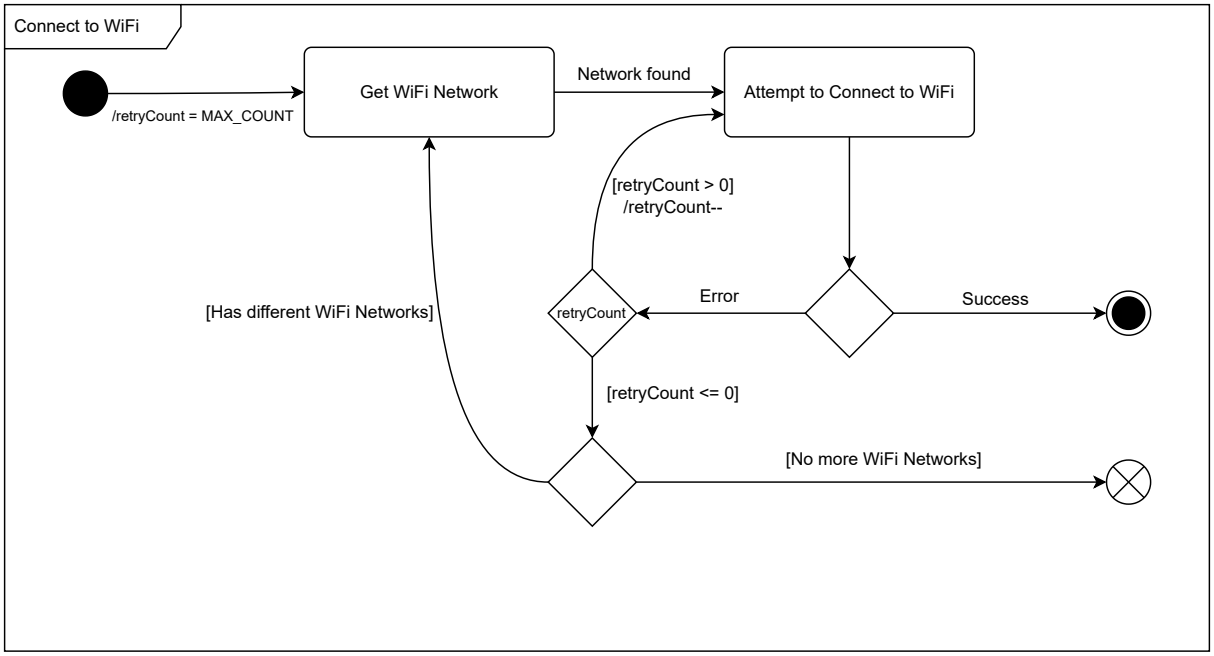



Figure 22: State machine diagram of the Connect to WiFi state.

When the tag enters this state, it starts by entering a "Get WiFi Network" state. Here the tag reads

the configuration file and retrieves the list of WiFi networks. Then it gets the first network on that list and advances to the "Attempt to Connect to WiFi" state. In the "Attempt to Connect to WiFi" state, the tag tries a maximum of "MAX_COUNT" number of times to connect to the WiFi network.

If the tag cannot connect to the WiFi network after "MAX_COUNT" tries, it checks to see if there are more networks it can try. If there are, the tag goes back to the "Get WiFi Network" state. Here, the tag reads the network list again, and gets the next item on that list and the cycle repeats.

If during this cycle the tag connects to a WiFi network successfully, it leaves this cycle to the "Get Remote Configuration" state, as depicted in figure 19. On the other hand, if the tag was not able to connect to any network, it leaves the "Connect to WiFi" state to the "Unrecoverable Error" state.

### 4.4.3 Get Remote Configuration

In this section we will discuss the "Get Remote Configuration" state in more detail. Figure 23 depicts this state machine diagram.

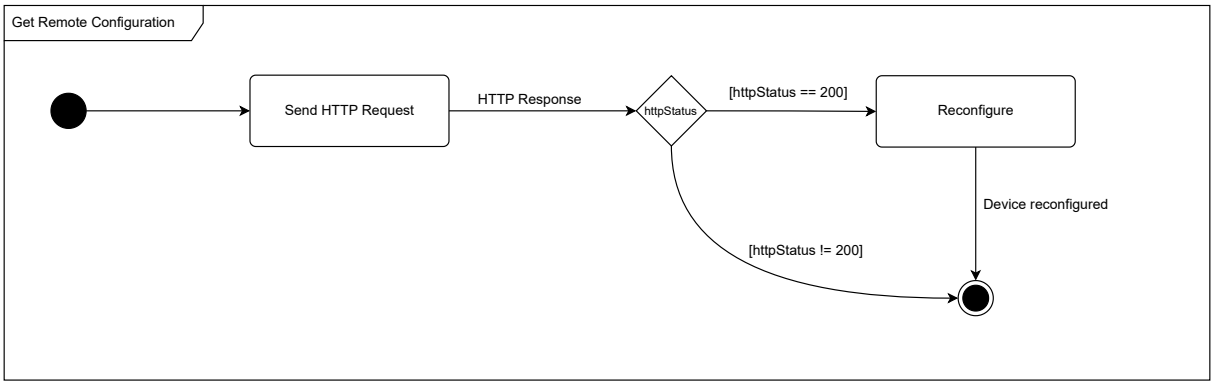



Figure 23: State machine diagram of the Get Remote Configuration state.

When the tag enters this state, it starts by entering a "Send HTTP Request" state. In this state, the tag will try to fetch the configurations from a remote configuration server.

If the tag is able to get the configurations successfully, it will store them in its memory. Moreover, the tag will replace the configuration file stored in its non-volatile memory and update it with these configurations. This way, the tag will use the remote server configurations if rebooted. After this, the tag will leave this state.

On the other hand, if the tag is not able to get the configurations successfully, it will not update any of its configurations and leave this state.

### 4.4.4 Collect fingerprints

In this section we will discuss the "Collect fingerprints" state in more detail. Figure 24 depicts this state machine diagram.

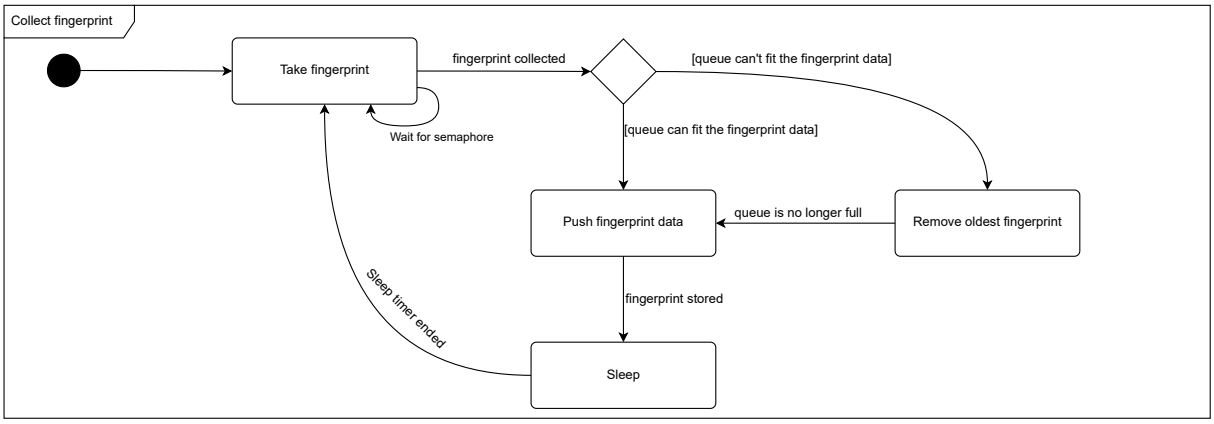



Figure 24: State machine diagram of the Collect Fingerprints state.

When the tag enters this state, it starts by entering a "Take fingerprint" state. In this state, the tag will collect fingerprints around it.

Recall that, the "Collect fingerprints" and "Send fingerprints" states run in parallel. The tag only has one antena and one WiFi interface, so it can't send fingerprints and collect fingerprints at the same time. A semaphore is needed to prevent both states from using the WiFi interface at the same time. In the "Take fingerprint" state, the tag waits for the semaphore to become green before it starts collecting fingerprints.

When the tag is able to collect a fingerprint, it needs to store it in a queue. This queue will have a configurable, but limited size. After exiting the "Take fingerprint" state, the tag checks if there is space available in the queue. If there is, the tag adds the fingerprint to the end of the queue in the "Push fingerprint data" state. If the queue is full, the tag first removes the oldest fingerprint in the queue in the "Remove oldest fingerprint" state, and then adds it to the end of the queue in the "Push fingerprint" state.

Then, the tag will enter a "Sleep" state for a configurable amount of time. After this, the tag will go back to the "Collect fingerprints" state and repeat the cycle.

Note that, even thought the tag can be configured to collect either CSI or RSSI fingerprints, the described process will be the same.

### 4.4.5 Send fingerprints

In this section we will discuss the "Send fingerprints" state in more detail. Figure 25 depicts this state machine diagram.

When the tag enters this state, it starts by entering a "Send fingerprint" state. In this state, the tag will send fingerprints to the server. First, the tag enters the "Read from queue" state. In this state, the tag reads the fingerprints queue, and checks if there is any fingerprint stored in the queue. If the queue is empty, the tag goes to the "Sleep" state. If there are fingerprints in the queue, the tag goes to the "Send HTTP Message" state.

In the "Send HTTP Message" state, the tag sends a fingerprint stored in the queue to the remote

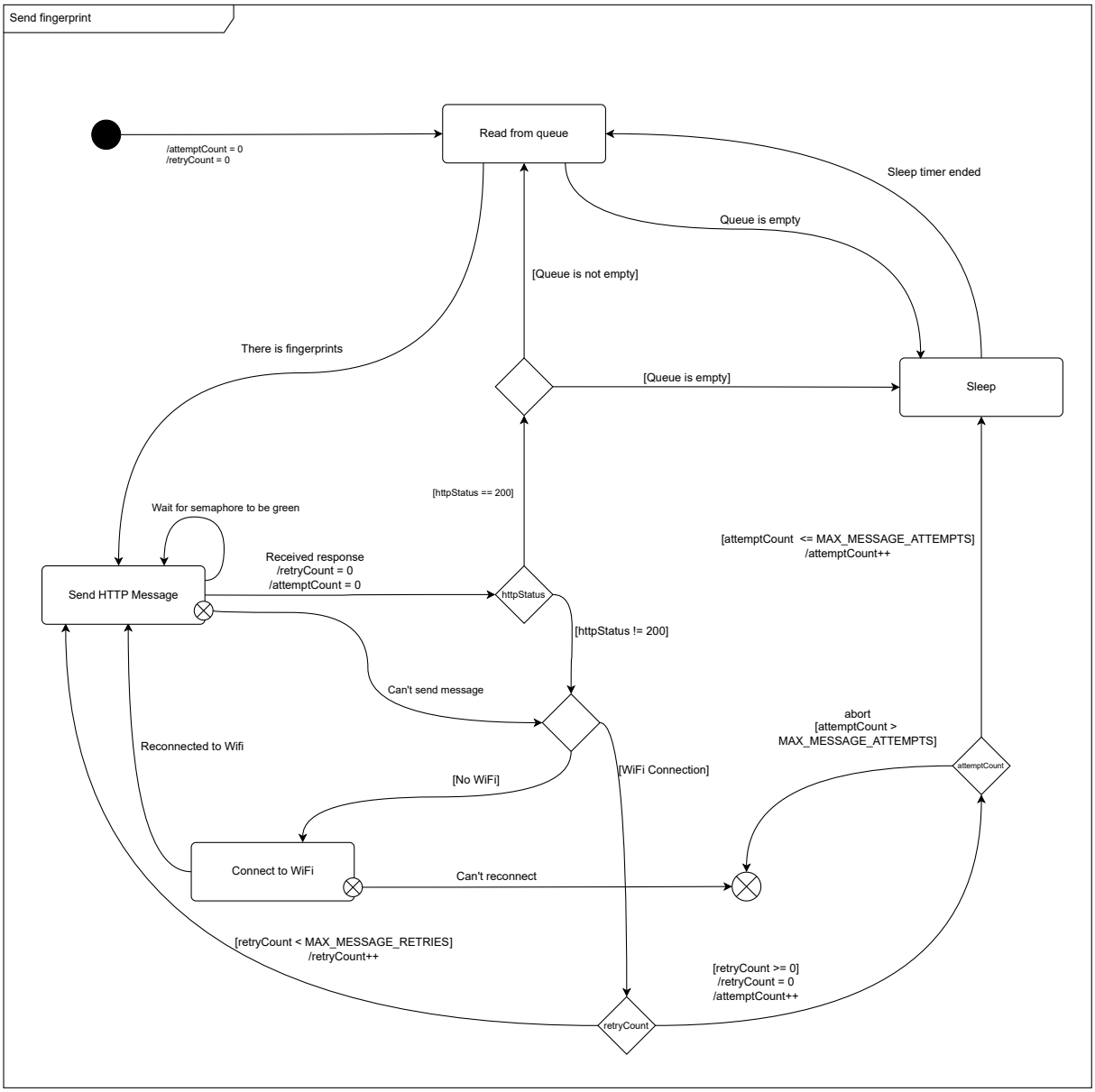



Figure 25: State machine diagram of the Send Fingerprints state.

fingerprints server. This state requires the tag to have access to the WiFi interface. For the same reasons described in the last subsection, there is a semaphore in this state. The tag must wait for the semaphore to be green to send the fingerprints to the server. After the tag attempts to send a fingerprint to the server, it can leave this state for two reasons. One is if there is a response from the server. Another is if the tag could not obtain a response from the server.

If the tag could not receive a response from the server, it could mean the tag lost its WiFi connection. Thus, in this case the tag goes to the "Connect to WiFi" state. We have discussed this state in detail in subsection 4.4.2. The only difference here, is that after leaving the "Connect to Wifi" state with success, the tag goes back to the "Send HTTP Message" state. In contrast, if the tag was unable to reconnect to a WiFi network, the tag leaves the "Send fingerprints" state, to an "Unrecoverable Error" state.

If the tag received a response from the server, it could be a positive response or a negative response. We know that the remote server's API answers with an HTTP status code 200 in a positive scenario. Thus, if the status code of the response is 200, the tag checks if the queue is empty. If it is, all of the fingerprints have been sent to the server, and the tag goes to a "Sleep" state. If there are still fingerprints in the queue, the tag moves again to the "Read from queue" state and repeats the cycle. This cycle will be made untill all of the fingerprints in the queue are sent to the server. Now, if the tag received a negative response from the server, it will check if there is a WiFi connection. If there is not, it will go to the "Connect to WiFi" state as explained above. If there is WiFi, the tag will move to the "Send HTTP Message" and try again to send the fingerprints. It will do this for a maximum number of times. If the maximum number of retries is exceeded, the tag checks the number of attempts made. If the tag also exceeded the maximum number of attempts, it leaves the "Send fingerprints" state to the "Unrecoverable Error" state. If not, the tag will increment the number of attempts and go to the "Sleep" state.

Lastly, the tag will sleep in the "Sleep" state for a limited and configurable amount of time. If the queue isn't empty after this time, the tag will go to the "Read from queue" state and start sending all the fingerprints in the queue to the server again. Otherwise, like previously explained, the tag will go back to the "Sleep" state.

# 5 System Implementation

In this section we will discuss the various steps and decisions regarding the implementation of the tag. First, we will discuss the development environment and tools used. Then, we will examine how the software was implemented to achieve the planned state machine diagrams discussed in the previous chapter. Lastly, we will examine the implementations of some electronics used in our prototype.

## 5.1 Development tools and environment

As we saw in chapter 4, we decided to use the ESP-IDF to develop our tag. The ESP-IDF framework requires us to install some dependencies. To do this, we can install all of the dependencies manually, or we can use extensions for a set of Integrated Development Environments (IDEs) to more easily download and install all of the required dependencies. The ESP-IDF framework highly recommends using the latter approach.

Following the framework's documentation recommendations, we will use this approach for installing the ESP-IDF and required dependencies. To do this we need to choose between two IDEs, Eclipse and Visual Studio Code. We chose VSCode as the IDE for creating the software for this project due to its extensive features and wide range of supported programming languages.

The installation of the ESP-IDF using the ESP-IDF extension for VSCode was very straightforward. The extension uses a GUI that guides the user through the steps of the installation. At the beginning of the installation process, the user has the option of using an "Express" or "Advanced" setup mode. In our installation of the framework we opted to use the "Express" mode for its simplicity.

In order for our application to run on the microcontroller, we need to build the source code and flash it to the microcontroller's memory. We can do this in two ways. We can use the command line and a set of commands, or we can use the functionalities provided by the ESP-IDF extension in VSCode. For its simplicity we always used the latter method. We only need to press a button and the commands will be automatically run for us.

It is also possible to monitor the microcontroller and see real-time printed strings by the microcontroller. This is especially useful for debugging. We can use the command line or the ESP-IDF extension

for VSCode in order to do this. We again, opt to use the VSCode ESP-IDF extension.

## 5.2 Software Development

In section 4.1 of chapter 4 we decided to use ESP32 microcontrollers as the base hardware for our solution. These microcontrollers come with no software needed to meet our requirements. As such, all of the software was made from scratch.

Recall that, in section 4.4 of the same chapter, we described various possible states for the tag with state machine diagrams. The software constructed was mostly based on those state machine diagrams. In the following subsections we will explain how each of the states was translated into source code. To do this we will describe the logic behind it, and illustrate it with relevant source code lines.

Before that, we will first describe how the various parts of our source code were organized in the next subsection.

### 5.2.1 Software Architecture

Keeping in mind the requirements and the state diagrams described in the previous chapter, we created five components.

The "ConfigurationService" component, responsible for configuration-related operations. For example, reading and parsing the configuration file in the "Initial Configuration" state.

The "HttpService" component, responsible for HTTP requests like fetching a configuration from a remote configuration server and sending a fingerprint to a remote server.

The "WifiService" component, responsible for WiFi-related tasks, like connecting and disconnecting from a network.

The "ScanService" component, responsible for collecting RSSI and CSI fingerprints of the environment around the tag.

The "SendService" component, responsible for reading the stored fingerprints and sending them to the server. Moreover, this component is responsible for handling and overcoming potential network problems when sending fingerprints.

The different components can, and do, use functions from other components to accomplish their responsibilities. For example, every component will use the tag's configurations that are stored in the scope of the "ConfigurationService".

The ESP-IDF framework VSCode extension provides a VSCode command we can use to scaffhold a component. Figure 26 shows this command. We created our five custom components using it.

After running the command, a "components" folder was created, and within it, five folders corresponding to each component were also constructed. Moreover, the scaffholding needed for each component was generated. Figure 27, shows the components folder structure. For each component there is a

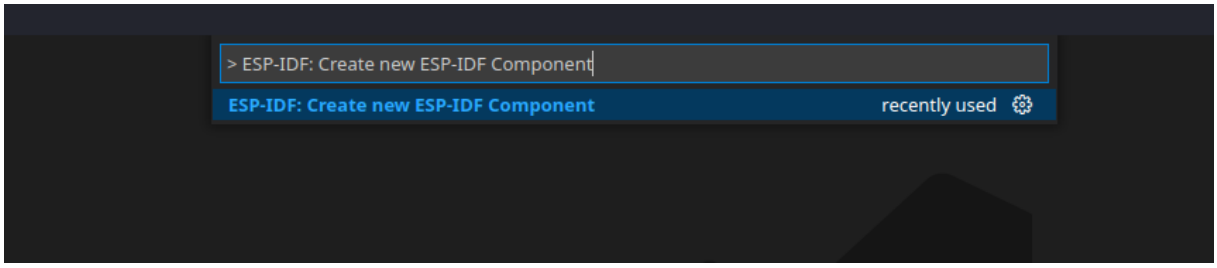



Figure 26: Create new component command.

"CMakeLists.txt" file, with CMake configurations. There is also a header file and a C file for all of the component's logic.

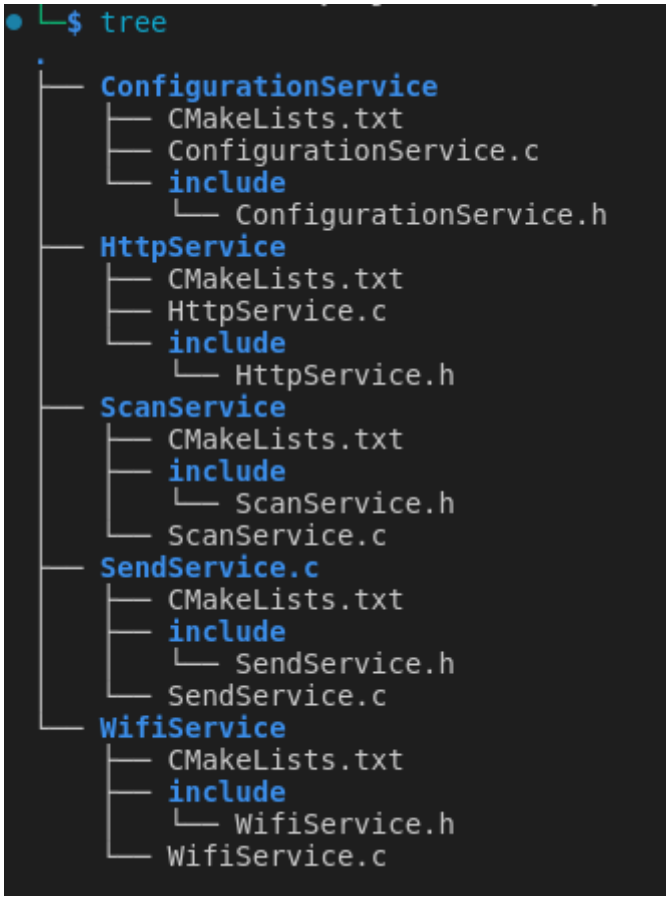



Figure 27: Components folder structure.

### 5.2.2 Post Boot Flow

According to the ESP-IDF framework documentation [1], the application startup flow consists of three phases. The first and second stage bootloader, and the application startup. It will be in the latter stages of application startup that our custom source code flashed into the microcontroller's memory will start to run. Every ESP-IDF project needs to have a "main" file with an "app_main" function. The application

startup layer will end by calling this function, marking the beginning of the execution of the developed source code.

In section 4.4 of chapter 4, we described a top level state machine diagram of our solution. In it, we can see a pipeline of states, from the "Initial Configuration" to the parallel states of "Send fingerprints" and "Collect fingerprints". Moreover, in most of these states, it is possible for an unrecoverable error to occur, making the tag go to an "Unrecoverable Error" state. We will initiate all states with the "app_main" function. Additionally, there will be logic in the main function that will send the tag to an "Unrecoverable Error" state.

The following listing 5, is an excerpt of the main function source code. Note that on lines 3 and 4, we have a comment followed by "(...)". This sequence of characters means that there is source code that was omitted from this excerpt. We will use this notation for the next listings in this paper, in order to simplify our explanations. Note that in the comment above the "(...)" , a brief explanation of what the omitted code is doing is given.

We can see that in line 17, we are calling a "read_local_configuration" function. This function will read the configuration file, stored in non-volatile memory of the tag. This corresponds to the "Self-Configuration" state of the "Initial Configuration" state described in chapter 4, section 4.4.1.

The "read_local_configuration" function returns an "esp_err_t" type value, which is a data type that represents error codes. It's a standardized way of handling and conveying error information within the ESP-IDF framework. Moreover, this data type is used to indicate the success or failure of functions and operations. When a function returns an "esp_err_t" data type that is equal to "ESP_OK", it means it ran with success [8].

We can see that in line 17, we have an if statement that checks if the "read_local_configuration" function did not run with success. If it did not, the operations inside the if statement are executed. These operations correspond to setting up the device in AP mode, as shown in figure 20. Conversely, following the same idea of the same figure, if the "read_local_configuration" function runs with success, we continue to the next state. We will better examine the source code behind these functions in subsection 5.2.3.

Note that, in lines 20 and 22, we are calling an "ESP_LOGE" macro. This is a macro used for logging error messages to the serial output. It's part of the logging system provided by ESP-IDF to help developers diagnose issues and monitor the behavior of their applications. There are 5 types of macros like this, to log error, warning, information, debug, and verbose messages. In the development of our application, we will use the "ESP_LOGE" and "ESP_LOGI" macros to log error and information messages, respectively. Figure 28 is an example of these messages being logged to the serial output. In red we can see the error messages and in green the information messages.

On line 27 we start the next state following the "Initial Configuration" state, the "Connect to WiFi" state. The "wifi_connect_from_config" function will initiate this state and try to connect the tag to a WiFi network. Again, this function also returns an "esp_err_t" data type. Here, the function is wrapped in an "ESP_ERROR_CHECK" macro.

```
1  int app_main(void)
2  {
3      // Initialize and setup board pins
4      // (...)
5
6      // Install ISR service and register ISR handler for push button
7      gpio_set_intr_type(PUSH_BUTTON_PIN, GPIO_INTR_POSEDGE);
8      gpio_install_isr_service(ESP_INR_FLAG_DEFAULT);
9      gpio_isr_handler_add(PUSH_BUTTON_PIN, scan_button_isr_handler,
   ↪  NULL);
10
11     // Initialize NVS and SPIFFS
12     // (...)
13
14     // Initialize WiFi interface, TCP/IP stack and default event loop
15     // (...)
16
17     if (read_local_configuration() != ESP_OK)
18     {
19         // Initiate AP mode and web server
20         ESP_LOGE(TAG, "Failed to read local configuration");
21         wifi_init_softap();
22         ESP_LOGI(TAG, "Starting Configuration server...\n");
23         setup_server();
24         return 1;
25     }
26
27     ESP_ERROR_CHECK(wifi_connect_from_config());
28
29     http_get_remote_configuration();
30
31     // Create queue, semaphore, and task parameters; Initialize I2C
   ↪  and MPU6050
32     // (...)
33
34     // Create tasks
35     xTaskCreatePinnedToCore(scan_task, "Scan task", 4096,
   ↪  scanTaskParams, 10, &scanTaskHandle, 0);
36     xTaskCreatePinnedToCore(send_task, "Send task", 4096,
   ↪  sendTaskParams, 10, &sendTaskHandle, 1);
37     xTaskCreate(setApModeConfigTask, "Ap mode task", 4096, NULL, 1,
   ↪  &setApModeConfigTaskHandle);
38     return 0;
39 }
```

Listing 5: Tag app_main function of the project's main file.

```
I (865) WIFI_SERVICE: Attempting to connect to Rio...
I (875) phy_init: phy_version 4670,719f9f6,Feb 18 2021,17:07:07
I (995) wifi:mode : sta (40:22:d8:e9:40:84)
I (995) wifi:enable tsf
I (995) WIFI_SERVICE: Connecting...
E (3415) WIFI_SERVICE: Connection to AP failed!
I (3415) WIFI_SERVICE: Retrying to connect to the AP
E (5825) WIFI_SERVICE: Connection to AP failed!
I (5825) WIFI_SERVICE: Retrying to connect to the AP
E (8235) WIFI_SERVICE: Connection to AP failed!
I (8235) WIFI_SERVICE: Retrying to connect to the AP
E (10645) WIFI_SERVICE: Connection to AP failed!
I (10645) WIFI_SERVICE: Failed to connect to SSID: Rio
```

Figure 28: Information and error logging examples.

The "ESP_ERROR_CHECK" macro is used for error checking and handling. The function in the macro must return an "esp_err_t" data type. If the function returns something different than "ESP_OK", the macro will log the error code, location, and failed statement to serial output. Moreover, it will cause the program to halt. When this happens, the tag goes to the "Unrecoverable Error" state, as illustrated in figure 19. After the program is halted, the only way to get out of this state is to reboot the tag or power it off [9].

If the "wifi_connect_from_config" function returns "ESP_OK", it means that the tag is connected to a WiFi network and it can move to the next state.

The "http_get_remote_configuration" function, in line 29, starts the "Get Remote Configuration" state. Here, the tag will try to get configurations from a remote server and there is no error handling. If the function isn't successful, the tag will keep using the configurations it generated in the "Initial Configuration" state, as discussed in chapter 4. The logic behind this function will be better examined in section 5.2.5.

Finally, lines 35 and 36 will initiate the "Collect fingerprints" and "Send fingerprints" states respectively. For this we will create two tasks using the "xTaskCreatePinnedToCore" function. This function is from the FreeRTOS multitasking library, that is included with the ESP-IDF framework. It is used to create a new task and assign it to a specific core of the microcontroller. In this case we are assigning the "scan_task" to core 0 and the "sendTask" to core 1.

We have now covered the initialization function of our solution. We saw how the source code correlates to the top level state machine diagram described in section 4.4 of chapter 4. We will analyse in more detail the inner workings of each of the functions explained above in the following subsections.

### 5.2.3 Initial Configuration

In this subsection we will dive deeper into the implementation of the "Inicial Configuration" state. In this state the tag will read a configuration file stored in its non-volatile memory into runtime memory. In chapter 4, section 4.2, we presented a data format for the tag's configurations. Now we need to turn each key in the JSON object into a C data structure to store them in memory. This way, we can use

the configuration values easily and without having to read the configuration file multiple times, which is computationally expensive.

In order to do this, we first need to define what C data type each of the keys values will be. Table 4 shows each of the configuration values, with a brief description of their use, their units, and their C data type representation.

| Name | Description | Unit | C type |
|---|---|---|---|
| TAG_NAME | Name of the tag. | – | char* |
| FINGERPRINTS_SERVER | Path to the remote server that receives fingerprint data in the format $host : port$. | – | char* |
| CONFIGURATION_SERVER | Path to the remote server that sends configurations to the tag in the format $host : port$. | – | char* |
| FINGERPRINT_SERVICE_SLEEP | Amount of time the tag sleeps after collecting a fingerprint | ms | unsigned short |
| FINGERPRINT_SERVICE_COLLECT | Amount of time the tag is collecting fingerprints. | ms | unsigned short |
| MESSAGE_SERVICE_SLEEP | Amount of time the tag sleeps after sending a message | ms | unsigned short |
| QUEUE_SIZE | Maximum amount of fingerprints stored in a queue. | – | unsigned short |
| MAX_MESSAGE_ATTEMPTS | Maximum amount of times the tag enters the sleep state after failing to send a message with a WiFi connection | – | unsigned short |
| MAX_MESSAGE_RETRIES | Maximum amount of times the tag retries to send a particular message. | – | unsigned short |
| MAX_WIFI_CONNECT_RETRIES | Maximum amount of times the tag tries to connect to a particular WiFi network. | – | unsigned short |
| CSI_MODE | Flag that configures the tag to collect CSI fingerprints if true. If false the tag should collect RSSI fingerprints. | – | bool |
| WIFI_DETAILS | This a list of pairs of WiFi SSID and WiFi passwords for the tag to connect to. | – | 2 x char** |

Table 4: Configurable variables.

As we can see in the table, the tag name will be represented by a "char*" pointer, and all the configuration values that are numbers will be represented by unsigned shorts. The unsigned short data type can hold values from 0 to 65535. Our solution only allows these range of numbers as configuration file values. We restrict the values to this range because the range is more than enough for our use cases. Moreover, this way, we can save memory, which is always good when working with small devices like our tag's microcontroller. Also, note that the configuration values that represent a temporal value, should be in milliseconds. The "CSI_MODE" is a boolean that is true for collecting CSI fingerprints. The C representation of this configuration will be a bool C type, that only uses 1 byte of memory. The

```
1  struct ConfigurationVariables
2  {
3    char *TAG_NAME;
4    char *FINGERPRINTS_SERVER;
5    char *CONFIGURATION_SERVER;
6    unsigned short FINGERPRINT_SERVICE_SLEEP;
7    unsigned short FINGERPRINT_SERVICE_COLLECT;
8    unsigned short MESSAGE_SERVICE_SLEEP;
9    unsigned short QUEUE_SIZE;
10   unsigned short MAX_MESSAGE_ATTEMPTS;
11   unsigned short MAX_MESSAGE_RETRIES;
12   unsigned short MAX_WIFI_CONNECT_RETRIES;
13   bool CSI_MODE;
14   char **WIFI_SSID;
15   char **WIFI_PWD;
16   unsigned short WIFI_ARRAY_SIZE;
17 };
18
19 extern struct ConfigurationVariables configs;
```

Listing 6: Tag configuration struct definition.

"WIFI_DETAILS" configuration is an array of objects with an SSID and a password. To represent this in C we will use two "char**" pointers, meaning we will use two arrays of strings. One array will contain the SSIDs and the other will contain the passwords. The positions in the arrays will be the same for the password and SSID of each object.

To represent all of this in a structured way in C, we will use a struct, shown in listing 6. This struct contains all of the configuration values in the data types described above. Moreover, this struct will be global and shared across all of the components. The "WIFI_ARRAY_SIZE" is an extra variable that represents the size of the "WIFI_SSID" and "WIFI_PWD" arrays. This will be calculated at runtime and does not need to be provided in the configuration file.

As previously discussed, the configuration file for the tag's initial configuration needs to be stored in non-volatile memory. We could do this in various ways. First, we could use an external storage like an SD Card and store the file in it. This would require us to purchase and integrate an SD Card Module, along with its software dependencies into our solution.

Another way, would be to use the Non-volatile storage (NVS) library, already integrated in the ESP-IDF framework. The NVS library is designed to store key-value pairs in flash [21]. This approach would mean, that we would have to be restricted to using the key-value pair data format provided by the NVS library. Moreover, it is recommended to store small values using NVS. Our configuration values are relatively small, but we want our tag to serve a web configuration interface. This means we need to store web-related source code, like HTML, CSS, and JavaScript files in the future. This can't be stored in NVS.

```
1 nvs,      data, nvs,     ,        0x6000,
2 phy_init, data, phy,     ,        0x1000,
3 factory,  app,  factory, ,        1M,
4 storage,  data, spiffs,  ,        500K
```

Listing 7: The partitions.csv partition file.

Lastly, SPI Flash File System (SPIFFS) can also be used to store files. SPIFFS is a file system designed for use with SPI NOR flash memory commonly found in ESP32 microcontrollers. It is a lightweight, efficient file system that allows the storage and management of files in the flash memory of these devices. One disadvantage of using SPIFFS, is that it does not support directories [26]. It produces a flat structure, but this should not be an issue for our use case.

Given the possibilities, we decided that SPIFFS would be the technology that would be better aligned with our requirements.

SPIFFS is not enabled out of the box in ESP-IDF. There is a couple ways to enable and use SPIFFS. Since we are already using VSCode with the ESP-IDF extension, as discussed in section 5.1, we will enable and use it using these two tools. First, we need to manually declare a partition for it. We created a file, named "partitions.csv" based on the built-in "Single factory app, no OTA" partition file [22] and added an SPIFFS section, as listing 7 shows.

ESP-IDF has a configuration menu for various configurations and settings of ESP-IDF projects. Using VSCode and the ESP-IDF extension, we can easily access them in the button shown in figure 29.

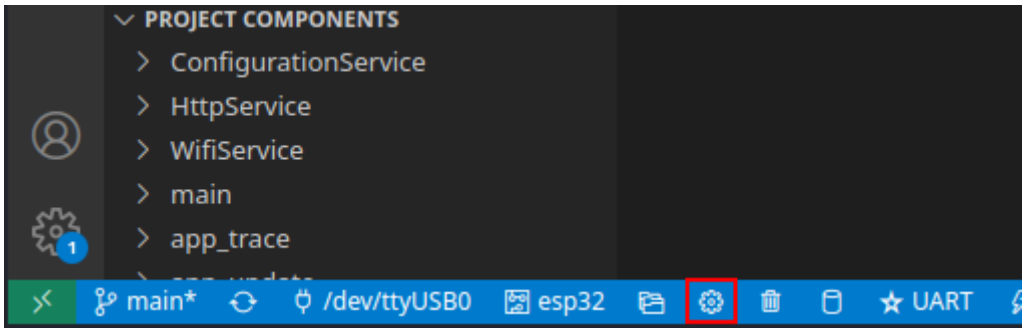



Figure 29: Configuration menu button.

In this configuration menu, we need to edit the configurations of the project and set a custom partition table as shown in figure 30.

A folder also needs to be created where the files we want to upload to the tag's SPIFFS partition should live. We named this folder "spiffs_data", and added the default configuration file, following the configuration data format discussed on the previous chapter with the name "defaults.json".

Finally, the last step of the SPIFFS installation is to edit our main "CMakeLists.txt" file and add the command shown in listing 8. Now, every time we build our source code and flash it to the microcontroller's memory, the files stored in the "spiffs_data" folder will also be uploaded into the microcontroller.

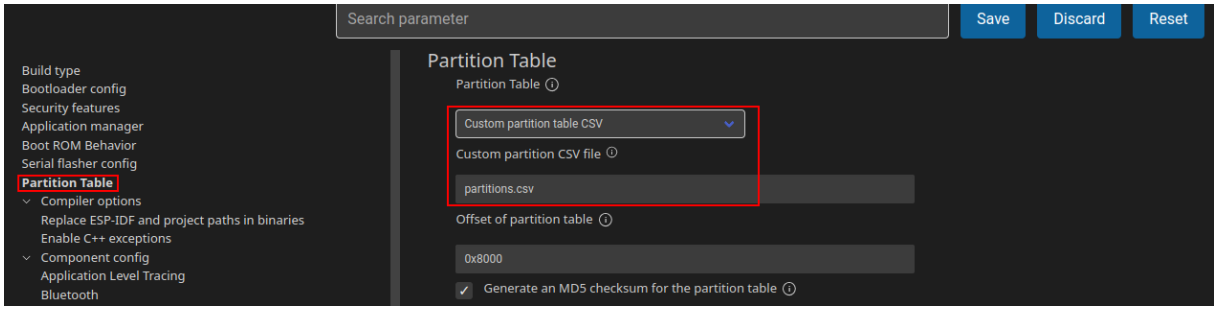



Figure 30: Partitions configurations changes.

```
1 spiffs_create_partition_image(storage ../spiffs_data FLASH_IN_PROJECT)
```

Listing 8: Main CMakeLists.txt file.

We saw in the previous subsection, that the "Initial Configuration" state is initialized by the "read_local_configuration" function in the main file. The "read_local_configuration" function is part of the "ConfigurationService" component discussed in subsection 5.2.1. This function will call a "read_configs_from_file" function with the path of the configuration file as an argument, as we can see in listing 9.

```
1  esp_err_t read_local_configuration(void)
2  {
3      return read_configs_from_file("/spiffs/defaults.json");
4  }
```

Listing 9: read_local_configuration function of the "ConfigurationService" component.

The "read_configs_from_file" function is shown in listing 10. This function will open a file in the path provided as an argument and read its contents. The contents will then be sent to a "parse_JSON_and_store_in_configs" function at line 27. Note that here, we are giving the path to the default configuration file as an argument.

On line 27 we can also see an "ESP_GOTO_ON_ERROR" macro. This macro requires us to define an "esp_err_t" variable in our function like we did on line 3. The macro will check the return value of the "parse_JSON_and_store_in_configs" function. If the value is not "ESP_OK", it will set "esp_err_t" to the return value of the function and perform a "goto" the provided part of the function. In our case, this macro will goto "end" and perform a cleanup of allocated memory. Lastly, the macro will also log the message provided.

The "parse_JSON_and_store_in_configs" function will parse a configuration JSON provided in its arguments, read the configuration values, and store them in runtime memory, using the struct discussed before. This function is shown in listing 11 bellow.

To help us parse and read the configuration JSON, we will use the well-known cJSON library. This

```
esp_err_t read_configs_from_file(char *filename)
{
    esp_err_t ret = ESP_OK;

    FILE *f = NULL;
    long len = 0;
    char *data = NULL;

    /* open in read binary mode */
    f = fopen(filename, "rb");
    if (f == NULL)
    {
        ESP_LOGE(CONFIG_SERVICE_TAG, "File does not exist!");
        return ESP_ERR_NOT_FOUND;
    }
    /* get the length */
    fseek(f, 0, SEEK_END);
    len = ftell(f);
    fseek(f, 0, SEEK_SET);

    data = (char *)malloc(len + 1);

    fread(data, 1, len, f);
    data[len] = '\0';
    fclose(f);

    ESP_GOTO_ON_ERROR(parse_JSON_and_store_in_configs(data), end,
↪  CONFIG_SERVICE_TAG, "Failed to parse local configurations!");

    goto end;

end:
    free(data);
    return ret;
}
```

Listing 10: read_local_configuration function of the ”ConfigurationService” component.

```
esp_err_t parse_JSON_and_store_in_configs(char *text)
{
    // Logs if the tag is already self configured
    // (...)

    // Declare variables
    // (...)

    json = cJSON_Parse(text);
    if (!json) {
        // Error, log error and set err to ESP_FAIL
        // (...)
    }
    else {
        // Parsing successful, get values and store in configs
        tag_name = cJSON_GetObjectItem(json, "TAG_NAME");
        if (cJSON_IsString(tag_name) && (tag_name->valuestring != NULL))
        {
            configs.TAG_NAME = strdup(tag_name->valuestring);
        }
        else
        {
            // If not self configured, set err to ESP_FAIL (as it is a critical error)
            if (!self_configured)
                err = ESP_FAIL;
            ESP_LOGE(CONFIG_SERVICE_TAG, "Can't read TAG_NAME");
        }

        // Read and store other configurations
        // (...)

        // Print json
        // (...)
    }

    // Set self configured flag to true
    if (!self_configured)
        self_configured = 1;

    // Free memory
    // (...)
}
```

Listing 11: parse_JSON_and_store_in_configs function of the "ConfigurationService" component.

library comes as part of the ESP-IDF framework, so no additional installation is needed to use it. On lines 14 to 20 we can see how we read the "TAG_NAME" variable and store it in our global configs variable. Note, on line 24 we have an if statement. The "self_configured" variable, is a global variable to the "ConfigurationService" component. It is used to check if there is already a valid configuration in runtime memory. Recall that, in section 4.4 of chapter 4, we discussed that if the configurations from the remote configuration server are not valid, we will keep the previous valid configuration. These lines of code are what ultimately makes that happen. If "self_configured" is false and we can't get a configuration variable, the "parse_JSON_and_store_in_configs" function will return an error. Otherwise, if the "self_configured" variable is true, it means that there is already a valid configuration in memory. Thus, the function keeps the last valid configuration value and keeps trying to read other configuration values without returning an error. In the "Initial Configuration" state, the "self_configured" variable will be false. On lines 37-38, we set the "self-configured" variable to a true value. If the program reaches these lines without returning, it means that the reading of the configuration JSON was successful and the configs struct is populated.

Finally, if the "parse_JSON_and_store_in_configs" is successful, the tag will go to the next state with its configuration loaded in runtime memory.

If the "read_local_configuration" function is not successful and does not return "ESP_OK", the tag will provide a web interface where users are able to change its configurations, as discussed in subsection 4.4.1 of chapter 4. To do this, the tag first needs to configure its WiFi interface to be in AP mode. Then, an HTTP server needs to be setup. This is done in lines 18-25 of listing 5.

The "wifi_init_softap" function is part of the "WifiService" component and is shown in listing 12. It configures the tag in AP mode and the IP address that users need to connect to, to access the web interface with their devices. To configure the tag in AP mode, we first must ensure there is no WiFi connection and that the interface is not currently actively running in STA mode. We do this in lines 5-6. Then, we need to set the tag's WiFi interface in AP mode and provide some configurations. We do that in lines 8 to 23, where we configure the tag to be an open AP, named "ESP32-AP" and only allow one connection at a time. Using this approach, we could run into the potential problem of having multiple tags in AP mode, thus having multiple WiFi networks called "ESP32-AP". Since the tags will only be in AP mode when expressly done by the user or in the case of a problem with the default configuration files, this will not be a big issue in practice. Users will need an IP address where they can access the web server. To solve this, we will set the tag to have a static IP of "192.168.1.1". This way, users can access the web server by navigating to that IP on their web browsers. We do this on lines 25-31. Now that our configurations are set, we call the "esp_wifi_start" function to start the WiFi interface, on line 35.

After the tag is in AP mode, we need to setup the HTTP server. This is done by calling the "setup_server" in line 23 of listing 5. The "setup_server" is shown in listing 13.

We needed a place to store HTML, CSS and JavaScript source code so the tag could serve a web page. It is possible to minimize the HTML, CSS and JavaScript code into one single HTML file. We have discussed before that we use SPIFFS to store the tag's default configuration file. We will use the same approach to store a HTML file. A HTML file with a simplistic UI was made and stored in the "spiffs_folder"

```
esp_err_t wifi_init_softap(void)
{
    esp_err_t ret = ESP_OK;

    esp_wifi_disconnect();
    esp_wifi_stop();

    ESP_GOTO_ON_ERROR(esp_wifi_set_mode(WIFI_MODE_AP), end,
↪  WIFI_SERVICE_TAG, "esp_wifi_set_mode(WIFI_MODE_AP) failed");

    esp_netif_t *esp_netif_ap = esp_netif_create_default_wifi_ap();

    wifi_config_t wifi_ap_config = {
        .ap = {
            .ssid = "ESP32-AP",
            .ssid_len = strlen("ESP32-AP"),
            .max_connection = 1,
            .authmode = WIFI_AUTH_OPEN,
            .pmf_cfg = {
                .required = false,
            },
        },
    };
    esp_wifi_set_config(WIFI_IF_AP, &wifi_ap_config);

    esp_netif_ip_info_t ipInfo;
    IP4_ADDR(&ipInfo.ip, 192, 168, 1, 1);
    IP4_ADDR(&ipInfo.gw, 192, 168, 1, 1);
    IP4_ADDR(&ipInfo.netmask, 255, 255, 255, 0);
    esp_netif_dhcps_stop(esp_netif_ap);
    esp_netif_set_ip_info(esp_netif_ap, &ipInfo);
    esp_netif_dhcps_start(esp_netif_ap);

    ESP_LOGI(WIFI_SERVICE_TAG, "wifi_init_softap finished.");

    ESP_GOTO_ON_ERROR(esp_wifi_start(), end, WIFI_SERVICE_TAG,
↪  "esp_wifi_start() failed");

    goto end;

end:
    return ret;
}
```

Listing 12: wifi_init_softap function of the "WifiService" component.

```
1  httpd_handle_t setup_server(void)
2  {
3
4      html = readFromFile("/spiffs/index.html");
5      configJSON = readFromFile("/spiffs/defaults.json");
6
7      httpd_config_t config = HTTPD_DEFAULT_CONFIG();
8      httpd_handle_t server = NULL;
9
10     if (httpd_start(&server, &config) == ESP_OK)
11     {
12         httpd_uri_t uri_get = {
13             .uri = "/",
14             .method = HTTP_GET,
15             .handler = get_req_handler,
16             .user_ctx = NULL};
17
18         httpd_uri_t submit = {
19             .uri = "/submit",
20             .method = HTTP_POST,
21             .handler = submit_handler,
22             .user_ctx = NULL};
23
24         httpd_uri_t uri_get_configs = {
25             .uri = "/configs",
26             .method = HTTP_GET,
27             .handler = get_configs_handler,
28             .user_ctx = NULL};
29
30         httpd_register_uri_handler(server, &uri_get);
31         httpd_register_uri_handler(server, &submit);
32         httpd_register_uri_handler(server, &uri_get_configs);
33     }
34
35     return server;
36 }
```

Listing 13: setup_server function of the "ConfigurationService" component.

of our project. The web page interface can be seen in figure 31. Just like the default configuration file, the HTML file will also be uploaded to the tag.

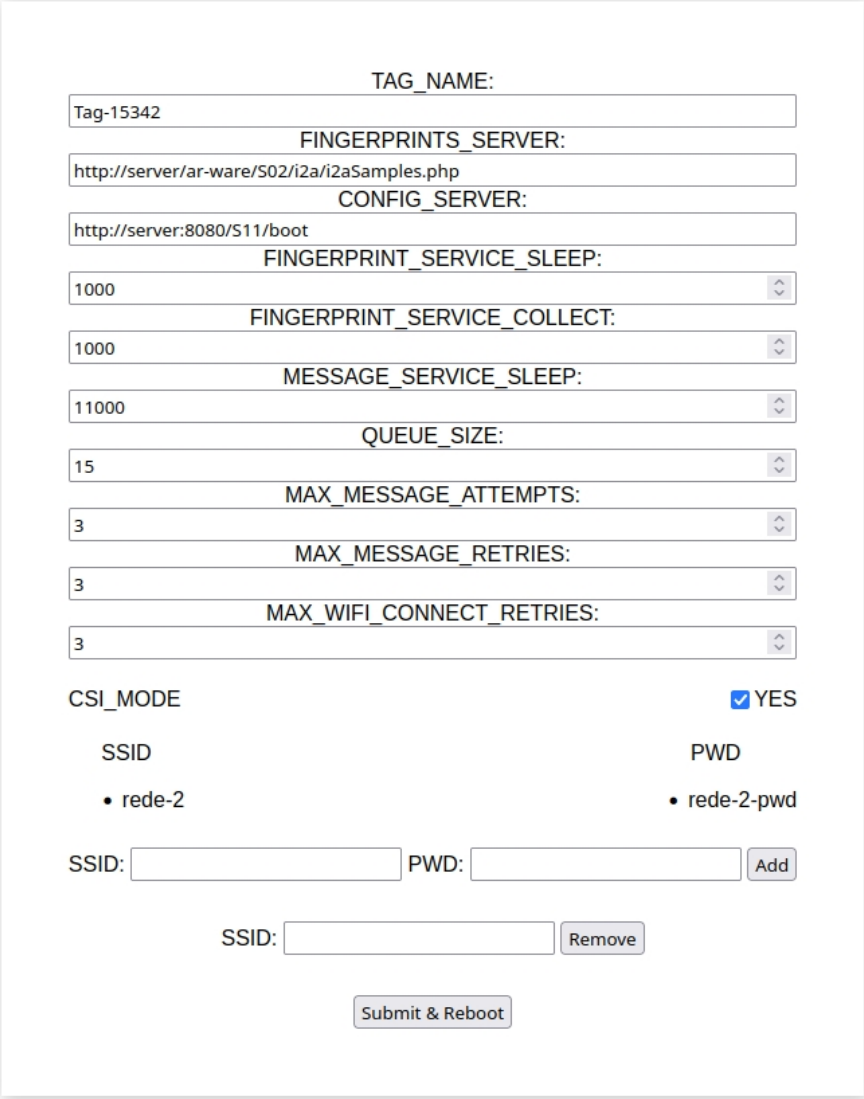



Figure 31: Configuration web interface.

The HTTP server needs to be configured to return the HTML file when browsers do a GET HTTP request. The web interface should show the current configurations loaded in memory. We can do this in two ways. One is to add the configurations to the HTML and then send it to the browser when requested. Another is to create a path, that browsers could use to fetch the configurations and then internally display them to the user. Since our configurations are dynamic, it will be much simpler to do the latter approach. Otherwise, we would have to create complex logic to edit our HTML file. It is much simpler, to create within our HTML file, a JavaScript function that will fetch the tag's configurations from the HTTP server, parse them and mount the values in the UI. Thus, the tag's HTTP server will need to return the tag's configurations when there is a GET HTTP request to a specific path. We will set this path as "/configs".

```
1  esp_err_t get_req_handler(httpd_req_t *req)
2  {
3      return httpd_resp_send(req, html, HTTPD_RESP_USE_STRLEN);
4  }
5
6  esp_err_t get_configs_handler(httpd_req_t *req)
7  {
8      return httpd_resp_send(req, configJSON, HTTPD_RESP_USE_STRLEN);
9  }
```

Listing 14: get_req_handler and get_configs_handler functions of the "ConfigurationService" component.

Moreover, the tag's configurations are sent in the body of the HTTP response. They are sent in a JSON that follows the data format discussed in section 4.2 of chapter 4.

The HTTP server will also need to receive a POST HTTP request with configurations in its body to reconfigure the tag. We also need to define a specific path for this. We will set this path to "/submit". The configurations sent should also follow the data format for configurations already discussed. The logic to create the configuration JSON and send them to the tag's HTTP server needs to be run in the browser. Thus, we included in the HTML file, JavaScript functions that will construct the JSON with the user's input configurations and do the "POST" request with it.

In listing 13, we start the HTTP server in lines 7-10. Then, in lines 11-33 we register handlers functions for returning the HTML file on a GET request with the "/" path, the configurations on a GET request with the "/configs" path and a POST request with configurations to the "/submit" path. On lines 4-5 we use the "readFromFile" function to read the HTML file and the configs file. The "readFromFile" is very similar to the "read_local_configuration" discussed before, but it will return what was read instead of calling the "parse_JSON_and_store_in_configs" function. The HTML file data will be stored in the "html" variable and the configuration file will be stored in a "configJSON" variable. Both of these variables are global to the "ConfigurationService" component.

The "get_req_handler" will handle GET requests to the "/" path and answer with the HTML file read. The web browsers will internally read and parse the HTML file and construct the web page. The "get_configs_handler" will handle GET requests to the "/configs" path and answer with the configuration file read. The "get_req_handler" and "get_configs_handler" functions are shown in listing 14.

The "submit_handler" will handle POST requests to the "/submit" path and configure the tag using the configuration JSON provided. The logic that parses the user input and makes a POST request is going to be running in web browsers. The source code for this is included in the HTML page provided by the tag. The "submit_handler" function is shown in listing 15.

To read the payload of a POST request, we need to first allocate memory to store it. The contents of the POST request will not have the null termination character. In C, a null termination character is required to signal the end of a string, which is stored in memory in the form of a char pointer. Thus,

```
1  esp_err_t submit_handler(httpd_req_t *req)
2  {
3      char content[req->content_len + 1];
4
5      int ret = httpd_req_recv(req, content, req->content_len);
6      content[req->content_len] = '\0';
7
8      if (ret <= 0)
9      { /* 0 return value indicates connection closed */
10         /* Check if timeout occurred */
11         if (ret == HTTPD_SOCK_ERR_TIMEOUT)
12         {
13             httpd_resp_send_408(req);
14         }
15         return ESP_FAIL;
16     }
17
18     ESP_LOGI(CONFIG_SERVICE_TAG, "content: %s\n", content);
19     parse_JSON_and_store_in_spiffs(content);
20
21     /* Send a simple response */
22     const char resp[] = "Configuration was sucessfull!";
23     httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
24
25     esp_restart();
26     return ESP_OK;
27 }
```

Listing 15: submit_handler function of the "ConfigurationService" component.

we need to allocate memory equal to the payload's size plus one. This is done in lines 3-6. In line 3 the memory is allocated and in line 5 we use the "httpd_req_recv" function to store the payload in the allocated space. Then, in line 6 we add the termination character. At this point we have the configuration JSON stored in the "content" variable. Now we can use the "parse_JSON_and_store_in_spiffs" function. This function will parse the configuration JSON and replace the default configuration file in SPIFFS. We will better examine this function later in section 5.2.5. After this, a simple response is sent to the web browser and the tag is rebooted using the "esp_restart" function. Note that, since the default configuration file was replaced with the provided configurations, when the tag reboots and enters the "Initial Configuration" state, it will configure itself with the user provided configurations. This flow is not exacly as planned in the state machine diagram presented in figure 20. However, rebooting the device after replacing the default configuration file with the new configurations simplifies the source code logic while having the same outcome.

### 5.2.4 Connect to WiFi

In this section we will analyse the implementation of the "Connect to WiFi" state. Following a successful initial configuration of the tag, the tag will now try to connect itself to a WiFi network in this state.

We saw in subsection 5.2.2 that the "Connect to WiFi" state is initialized by the "wifi_connect_from_config" function. This function is part of the "WifiService" discussed in subsection 5.2.1 and is shown in listing 16.

The "wifi_connect_from_config" function loops through the arrays of SSIDs and passwords of the configs struct and tries to connect to each WiFi network. It will stop when it connects to a WiFi network, or if it loops through the both arrays and is unable to connect to a WiFi network. In the first case the function will return "ESP_OK", and in the latter it will return "ESP_FAIL". In the latter case, as we discussed before, the program will halt, and the tag will go to an "Unrecoverable Error" state.

```
1  esp_err_t wifi_connect_from_config()
2  {
3
4      for (int i = 0; i < configs.WIFI_ARRAY_SIZE; i++)
5      {
6          ESP_LOGI(WIFI_SERVICE_TAG, "Attempting to connect to %s...",
   ↪  configs.WIFI_SSID[i]);
7          wifi_connect(configs.WIFI_SSID[i], configs.WIFI_PWD[i]);
8          if (connected)
9              break;
10     }
11
12     if (connected)
13         return ESP_OK;
14     else
15         return ESP_FAIL;
16 }
```

Listing 16: wifi_connect_from_config function of the "WifiService" component.

In lines 8 and 12 we have a "connected" variable. This is a global variable to the "WifiService" component that is true when the tag is connected to a WiFi network.

The "wifi_connect_from_config" calls the "wifi_connect" function in its for loop with the SSID and password of each WiFi network in the configs struct. The "wifi_connect" function has the responsibility of connecting to a specific WiFi network. An excerpt of the "wifi_connect" function can be found in listing 17.

The ESP32 WiFi drivers use an event-based logic for many of its features. For connecting to a WiFi network, we need to configure the WiFi driver with a configuration. In this configuration we can provide an SSID and password for the network we want to connect to. We do this on lines 11-18. After setting up the

```
1  void wifi_connect(char *ssid, char *pwd)
2  {
3      s_wifi_event_group = xEventGroupCreate();
4
5      esp_event_handler_instance_t instance_any_id;
6      esp_event_handler_instance_t instance_got_ip;
7
8      esp_event_handler_instance_register(WIFI_EVENT, ESP_EVENT_ANY_ID,
   ↪  &wifi_event_handler, NULL, &instance_any_id);
9      esp_event_handler_instance_register(IP_EVENT, IP_EVENT_STA_GOT_IP,
   ↪  &wifi_event_handler, NULL, &instance_got_ip);
10
11     wifi_config_t wifi_config;
12     memset(&wifi_config, 0, sizeof(wifi_config_t));
13
14     strncpy((char *)wifi_config.sta.ssid, ssid, 32);
15     strncpy((char *)wifi_config.sta.password, pwd, 64);
16
17     esp_wifi_set_mode(WIFI_MODE_STA);
18     esp_wifi_set_config(WIFI_IF_STA, &wifi_config);
19     esp_wifi_start();
20
21     EventBits_t bits = xEventGroupWaitBits(s_wifi_event_group
   ↪  WIFI_CONNECTED_BIT | WIFI_FAIL_BIT, pdFALSE, pdFALSE, portMAX_DELAY);
22
23     // Check the bits of the event group and print a success or error
   ↪  message
24     // (...)
25
26     // Clean up
27     // (...)
28 }
```

Listing 17: wifi_connect function of the "WifiService" component.

configurations, we can start the WiFi driver by calling the ESP-IDF's "esp_wifi_start" function. The WiFi driver will then start with the provided configurations. Moreover, it will emit events to a default event loop [35]. We can handle these events by registering our own handler functions with the default event loop. This is done in lines 8 and 9. There, we are registering that our "wifi_event_handler" function will be called when a "WIFI_EVENT" or an "IP_EVENT" is launched to the default event loop. The "wifi_event_handler" function is shown in listing 18.

On line 3 of listing 17 we call the "xEventGroupCreate" function and store its results in a "s_wifi_event_group"

```
1  void wifi_event_handler(void *arg, esp_event_base_t event_base,
2                          int32_t event_id, void *event_data)
3  {
4      if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START)
5      {
6          esp_wifi_connect();
7      }
8      else if (event_base == WIFI_EVENT && event_id ==
   ↪  WIFI_EVENT_STA_DISCONNECTED)
9      {
10         ESP_LOGE(WIFI_SERVICE_TAG, "Connection to AP failed!");
11         if (s_retry_num < configs.MAX_WIFI_CONNECT_RETRIES)
12         {
13             esp_wifi_connect();
14             s_retry_num++;
15             ESP_LOGI(WIFI_SERVICE_TAG, "Retrying to connect to the
   ↪  AP");
16         }
17         else
18         {
19             xEventGroupSetBits(s_wifi_event_group, WIFI_FAIL_BIT);
20             s_retry_num = 0;
21             connected = 0;
22         }
23     }
24     else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP)
25     {
26         // Prints success message ans set s_retry_num to 0
27         // (...)
28         connected = 1;
29         xEventGroupSetBits(s_wifi_event_group, WIFI_CONNECTED_BIT);
30     }
31 }
```

Listing 18: wifi_event_handler function of the "WifiService" component.

variable. The "s_wifi_event_group" is a global variable of the "WifiService" component. The "xEventGroupCreate" is a FreeRTOS function that creates a new RTOS event group, and returns a handle by which the newly created event group can be referenced [36]. After, in line 21 we call the "xEventGroupWaitBits" function. The "xEventGroupWaitBits" function will wait for the provided bits to be set on the provided event group [37]. In our case we provide to the "xEventGroupWaitBits" function the "WIFI_CONNECTED_BIT" and "WIFI_FAIL_BIT" bits and the previously created event group handle.

As stated before, our "wifi_event_handler" will be called whenever there is "WIFI_EVENT" or "IP_EVENT" sent to the default event loop. Events sent to the default event loop have a more specific event id, which

is provided as an argument to our handler. In line 4 of listing 18, we are handling a "WIFI_EVENT" with a "WIFI_EVENT_STA_START" id. Here, we call the "esp_wifi_connect" function. This function attempts to connect to the WiFi network configured in the "wifi_connect" function. If it connects to the WiFi network with success, the WiFi driver will emit an "IP_EVENT" with an "IP_EVENT_STA_GOT_IP" id. In that case, our handler will be called again and we will handle that event on lines 24-31. There, we set the "connected" global variable to 1 (true), and set the "WIFI_CONNECTED_BIT" on the RTOS event group with the "xEventGroupSetBits" function.

On the other hand, if the WiFi driver is not able to connect to the WiFi network, a "WIFI_EVENT" with id "WIFI_EVENT_STA_DISCONNECTED" will be sent to the default event loop. We handle this in lines 8-22. Recall that in subsection 4.4.2 of chapter 4, we discussed a retry loop for each network connection attempt. We've implemented this here using a global variable to the "WifiService" component, called "s_retry_num". We will check if we have exceeded the maximum configured retry attempts in line 11. If we did, the "WIFI_FAIL_BIT" will be set on the RTOS event group with the "xEventGroupSetBits" function on line 19. Moreover, the "connected" variable will be set to 0 (false) and the "s_retry_num" variable will be reset to 0. On the other hand, if the maximum number of retries is not exceeded, we will call the "esp_wifi_connect" function and try to reconnect to the network. The "esp_wifi_connect" will then emit an event, that will again be handled by our handler in one of the if statements described.

Going back to the "wifi_connect" function, when the handler sets either the "WIFI_CONNECTED_BIT" or the "WIFI_FAIL_BIT" in the RTOS event group. Our function will resume and we will return to the "wifi_connect_from_config" function. If at this point we are connected to a WiFi network, the "connected" variable will be true, and we break out of the for loop and return "ESP_OK". Otherwise, the for loop cycle will continue, until the tag is either connected to a WiFi network, or it runs out of WiFi networks to attempt to connect to. In the latter case, the function will return "ESP_FAIL" which will ultimately place the tag in an "Unrecoverable Error" state.

### 5.2.5 Get Remote Configuration

In this subsection we will look into the various pieces of software that make the "Get Remote Configuration" state implementation possible. In this state, the tag attempts to fetch configuration data from a remote configuration server.

In subsection 5.2.2 we discussed that the "http_get_remote_configuration" starts the "Get Remote Configuration" state. The "http_get_remote_configuration" can be found in listing 19, and it is part of the "HttpService" component.

Notice that, in contrast to most of the functions discussed before, the "http_get_remote_configuration" function returns void. This follows our state machine diagram for this state and what was discussed before in section 4.4 and subsection 4.4.3 of chapter 4. The function does not need to return a "esp_err_t" data type because it will never go to the "Unrecoverable Error" state due to itself. If something goes wrong, the tag will simply keep the configurations it had before. Nevertheless, in case of insuccess,

```
1  void http_get_remote_configuration(void)
2  {
3      char *json_text;
4
5      if (http_get_configuration_data(&json_text) == ESP_OK)
6      {
7          read_remote_configuration(json_text);
8
9          free(json_text);
10     }
11     else
12     {
13         ESP_LOGE(HTTP_SERVICE_TAG, "Failed to get remote configuration,
   ↪  using default configs");
14     }
15 }
```

Listing 19: http_get_remote_configuration function of the "HttpService" component.

there will be logged a message to the serial output, as we can see in lines 11-14.

On line 5 we use the "http_get_configuration_data" function to get configuration data from the configured remote configuration server in the format discussed in section 4.2 of chapter 4. After, in line 7, the tag will attempt to parse the configuration data and reconfigure itself. An excerpt of the "http_get_configuration_data" is shown in listings 20 and 21.

Recall that in section 4.3 of chapter 4, we discussed how the communication with the remote configuration server should be made. We will need to send a POST request with the tag's MAC address in a specific format to the remote configuration server. In lines 6-7 we allocate memory for the MAC address and store in it the MAC address of the tag using the "get_mac_address" function. The "get_mac_address" function is part of the "WifiService" component. It simply stores the MAC address of the tag in the char pointer given as an argument. Then, on lines 10-12, we create the body of the POST request with the format discussed in section 4.2 of chapter 4. On line 17 we call the "esp_http_client_open" function from the ESP-IDF framework. This function will open a connection with the server and write all header strings to it. Then we need to write the POST body to this connection socket. That is done in line 24. Afterwards, the server should respond with a configuration JSON, in the format discussed before in chapter 4. The tag will read it using the "esp_http_client_read" from ESP-IDF on line 39. Finally, we will allocate space for the received data and point the "data" pointer argument to it.

Now, if this all happens with success, we will continue the execution path of the "http_get_remote_configuration" function and call the "read_remote_configuration" in line 7 of listing 19. The "read_remote_configuration" function is shown in listing 22. It will simply call the "parse_JSON_and_store_in_spiffs" function as we can see on line 5.

```
1  esp_err_t http_get_configuration_data(char **data)
2  {
3      // Initialize HTTP client and local variables. Set the request URL
   ↪  to the remote configuration server in the configs struct.
4      // (...)
5      // GET MAC
6      char *mac = (char *)malloc(sizeof(char) * 18);
7      get_mac_address(mac);
8
9      // Format post data
10     ssize_t bufsz = snprintf(NULL, 0, "object_={\"BSSID\":\"%s\"}",
   ↪  mac);
11     char *post_data = malloc(bufsz + 1);
12     snprintf(post_data, bufsz + 1, "object_={\"BSSID\":\"%s\"}", mac);
13
14     ESP_GOTO_ON_ERROR(esp_http_client_set_method(client,
   ↪  HTTP_METHOD_POST), end, HTTP_SERVICE_TAG, "Failed to get remote
   ↪  config");
15     esp_http_client_set_header(client, "Content-Type",
   ↪  "application/x-www-form-urlencoded");
16
17     _err_t err = esp_http_client_open(client, strlen(post_data));
18     if (err != ESP_OK)
19     {
20         ESP_LOGE(HTTP_SERVICE_TAG, "Failed to open HTTP connection:
   ↪  %s", esp_err_to_name(err));
21     }
22     else
23     {
24         int wlen = esp_http_client_write(client, post_data,
   ↪  strlen(post_data));
25         if (wlen < 0)
26         {
27             ESP_LOGE(HTTP_SERVICE_TAG, "Write failed");
28         }
29         content_length = esp_http_client_fetch_headers(client);
```

Listing 20: http_get_configuration_data function of the "HttpService" component. (Part 1)

The "parse_JSON_and_store_in_spiffs" function will parse the JSON provided as argument and create a valid JSON to store in non-volatile memory. An excerpt of this function is shown in listing 23. In lines 13-19, the tag checks if the "TAG_NAME" variable exists in the provided JSON. If it does not, or it is of an invalid type, a correct "TAG_NAME" variable will be added with the value of the current configuration in the "configs" struct. The process for the remaining configuration values will be the same. At

```
30          if (content_length < 0)
31          {
32              ESP_LOGE(HTTP_SERVICE_TAG, "HTTP client fetch headers
    ↪  failed");
33          }
34          else
35          {
36              const int response_length =
    ↪  esp_http_client_get_content_length(client);
37              output_buffer = malloc(response_length + 1);
38
39              int data_read = esp_http_client_read_response(client,
    ↪  output_buffer, response_length);
40
41              output_buffer[response_length] = '\0';
42
43              if (data_read >= 0)
44              {
45                  // Log status and received data
46                  // (...)
47
48                  // Store post payload in data
49                  char *temp = malloc(strlen(output_buffer) + 1);
50                  strcpy(temp, output_buffer);
51                  *data = temp;
52              }
53              else
54              {
55                  ESP_LOGE(HTTP_SERVICE_TAG, "Failed to read response");
56              }
57          }
58      }
59
60      goto end;
61
62  end:
63
64      // Free resources
65      // (...)
66      return ret;
67
```

Listing 21: http_get_configuration_data function of the "HttpService" component. (Part 2)

```
1  esp_err_t read_remote_configuration(char *jsonText)
2  {
3      esp_err_t ret = ESP_OK;
4
5      ESP_GOTO_ON_ERROR(parse_JSON_and_store_in_spiffs(jsonText), end,
   ↪  CONFIG_SERVICE_TAG, "Can't read remote configuration! Using
   ↪  defaults...");
6
7      goto end;
8
9  end:
10     return ret;
11 }
```

Listing 22: read_remote_configuration function of the "ConfigurationService" component.

the end of this process, we will have a JSON with valid configuration values. This JSON is given to the "parse_JSON_and_store_in_configs" function, that, as we saw before, parses the JSON and loads the JSON's configurations into the "configs" struct. Note that, only the valid configuration values present in the JSON received from the server will be updated.

On line 28 if listing 23 we call the "write_configs_to_file" function from the "ConfigurationService" component. This function will write to the path provided a configuration file in the format discussed in 4.2 of chapter 4. This configuration file represents the current state of the "configs" struct. Note that, we are writing to the same path as the default configuration file. This will replace that file with this new one. This way, when the tag reboots, it will use the configuration received from the server. In essence, this makes the remote server configurations the new default configurations.

### 5.2.6 Collect fingerprints

In this subsection we will analyse the relevant source code that will make the "Collect fingerprints" state implementation possible. In this state, the tag collects fingerprints and stores them in runtime memory.

Recall that in section 4.4 of chapter 4 we discussed that the "Collect fingerprints" and "Send fingerprints" states would run in parallel. We also argued that since the tag only has one antenna and one WiFi interface, a semaphore would be needed to stop both states from using the WiFi interface at the same time. At first, an implementation was made with no semaphore to test our hypothesis, and we were correct. Thus a semaphore was implemented. We also planned to use a queue to store fingerprints. Since the "Collect fingerprints" state runs in parallel with another state that needs to read from the same queue, a thread safe implementation was needed.

In subsection 5.2.2 we described that we created a task that would run the "scan_task" function in

```
1  esp_err_t parse_JSON_and_store_in_spiffs(char *text)
2  {
3      // Declare variables
4      // (...)
5
6      json = cJSON_Parse(text);
7      if (!json)
8      {
9          ESP_LOGE(CONFIG_SERVICE_TAG, "Error before: [%s]\n",
   ↪  cJSON_GetErrorPtr());
10         ret = ESP_FAIL;
11     }
12     else
13     {
14         tag_name = cJSON_GetObjectItem(json, "TAG_NAME");
15         if (!(cJSON_IsString(tag_name) && (tag_name->valuestring !=
   ↪  NULL)))
16         {
17             cJSON_AddStringToObject(json, "TAG_NAME",
   ↪  strdup(configs.TAG_NAME));
18             ESP_LOGE(CONFIG_SERVICE_TAG, "Can't read TAG_NAME, using
   ↪  latest valid value...");
19         }
20         // Test if other configuration values exist and are valid, if
   ↪  not, use the config values stored in the configs struct
21         // (...)
22
23         // Print json to serial output and update configs
24         out = cJSON_Print(json);
25         parse_JSON_and_store_in_configs(out);
26
27         // Store json in defaults file
28
   ↪  ESP_GOTO_ON_ERROR(write_configs_to_file("/spiffs/defaults.json",
   ↪  out), end, CONFIG_SERVICE_TAG, "Can't write remote configuration to
   ↪  file...");
29     }
30     goto end;
31
32   end:
33     // Free memory and return
34     // (...)
35 }
```

Listing 23: parse_JSON_and_store_in_spiffs function of the "ConfigurationService" component.

core 0 of the tag's microcontroller. The "scan_task" function is shown in listing 24.

The "scan_task" is part of the "ScanService" component and is a while loop that will run forever. To implement a semaphore, we leveraged the FreeRTOS binary semaphore. Binary semaphores can be used to restrict the execution of blocks of code if the semaphore can not be "taken". In line 19 we call the "xSemaphoreTake" function. The tag will try to "take" the semaphore for a period of time. If the semaphore does not become available during that period, it will not run the block of code in lines 20-31. On the other hand, if the "scan_task" is able to "take" the semaphore, it will enter the code block and perform its computations. When the "scan_task" enters the code block, it prevents other functions from "taking" the semaphore. Thus, at the end of our computations we have to "give" the semaphore. We do that on line 27 with the "xSemaphoreGive" function.

In subsection 4.4.4 of chapter 4 we discussed the implementation of a queue to store fringerprints that will be sent to the fingerprints server at some point in the "Send fingerprints" state. In order to do this we, naturally, need a queue. We will use a FreeRTOS implementation of a queue. They follow an First In First Out (FIFO) methodology and are thread-safe. This queue is created in the "main" function of the tag's software.

The tag needs to be capable of collecting RSSI or CSI fingerprints based on its configurations. If the tag is configured in CSI mode, it needs to run some initialization code to be able to collect CSI data. This is done in lines 7-12, where we check our "configs" struct if the tag is configured in CSI mode. We will discuss the initialization code later in this subsection. In lines 22-25 the tag will run the appropriate scan based on the current configurations. We will explain the implementation of the collection of CSI and RSSI fingerprints in the following subsubsections.

#### 5.2.6.1 Implementation of RSSI fingerprint collection

At line 25 of listing 24 we call the "scan_rssi" function. This function is part of the "ScanService" component and is responsible for performing an active scan of WiFi networks around the tag.

It receives the queue handler for the queue where the fingerprints will be stored. The "scan_rssi" function can be found in listings 25 and 26.

In line 3 of listing 25 we declare and create a "jsonArray" variable. This variable represents a JSON array. It is created using the "cJSON_CreateArray" function from the cJSON library.

Recall that, in section 4.3 of chapter 4 we discussed a data format for the RSSI fingerprints we need to send to the fingerprints server. In listing 3 of that section we can see that we have in lines 7-33 an array of objects. Each object represents a WiFi network found in the scan. The "scan_rssi" function will populate the "jsonArray" variable with a cJSON representation of objects like this.

At the time of writing this work, almost all of the ESP32 devices only supported the 2.4GHz WiFi band. Only the very recent ESP32-C5 microcontroller is capable of using the 5 GHz band. Since we did not have this type of microcontroller, our solution will only support 2.4GHz WiFi connections. The 2.4GHz band is divided into 13 channels. The tag needs to scan every channel to find WiFi networks around it. Recall that

```
1  void scan_task(scan_parameters_t *params)
2  {
3      ESP_LOGI(SCAN_SERVICE_TAG, "%s", "Scan task is running\r\n");
4
5      esp_ping_handle_t ping_handle = NULL;
6
7      if (configs.CSI_MODE)
8      {
9          ESP_LOGI(SCAN_SERVICE_TAG, "CSI MODE ENABLED");
10         scan_csi_init(params->xQueue);
11         ping_handle = scan_ping_router_init();
12     }
13
14     while (1)
15     {
16         gpio_set_level(params->scan_led_pin, true);
17         ESP_LOGI(SCAN_SERVICE_TAG, "LOOP scan task");
18
19         if (xSemaphoreTake(params->xSemaphore,
↪  params->semaphore_wait_time))
20         {
21
22             if (configs.CSI_MODE)
23                 scan_csi(ping_handle);
24             else
25                 scan_rssi(params->xQueue);
26
27             xSemaphoreGive(params->xSemaphore);
28
29             gpio_set_level(params->scan_led_pin, false);
30             vTaskDelay(configs.FINGERPRINT_SERVICE_SLEEP /
↪  portTICK_PERIOD_MS);
31         }
32         else
33         {
34             ESP_LOGE(SCAN_SERVICE_TAG, "Failed to take semaphore for
↪  scaning");
35             gpio_set_level(params->scan_led_pin, false);
36         }
37     }
38 }
```

Listing 24: scan_task function of the "ScanService" component.

```
1  void scan_rssi(QueueHandle_t queue)
2  {
3      cJSON *jsonArray = cJSON_CreateArray();
4      char *out = NULL;
5
6      uint16_t ap_count = 0;
7
8      const uint32_t scanTime = configs.FINGERPRINT_SERVICE_COLLECT / 13;
9
10     wifi_scan_config_t scan_config = {
11         .ssid = NULL, // Set to NULL to scan all SSIDs
12         .bssid = NULL, // Set to NULL to scan all BSSIDs
13         .channel = 0, // Set to 0 to scan all channels
14         .show_hidden = true, // Set to true to scan hidden SSIDs
15         .scan_type = WIFI_SCAN_TYPE_ACTIVE, // Set the scan type
   ↪  (active or passive)
16         .scan_time.active.min = scanTime / 2, // Minimum active scan
   ↪  time per channel
17         .scan_time.active.max = scanTime // Maximum active scan time
   ↪  per channel
18     };
19
20     ESP_LOGI(SCAN_SERVICE_TAG, "Starting RSSI Scan...");
21     esp_wifi_scan_start(&scan_config, true);
22
23     esp_wifi_scan_get_ap_num(&ap_count);
24
25     wifi_ap_record_t ap_info[ap_count];
26     memset(ap_info, 0, sizeof(ap_info));
27
28     esp_wifi_scan_get_ap_records(&ap_count, ap_info);
```

Listing 25: scan_rssi function of the "ScanService" component. (Part 1)

```
29
30      for (int i = 0; (i < ap_count); i++)
31      {
32          cJSON *jsonObject = cJSON_CreateObject();
33          char *mac = (char *)malloc(sizeof(char) * 18);
34          sprintf(mac, "%02X:%02X:%02X:%02X:%02X:%02X",
↪   MAC2STR(ap_info[i].bssid));
35          cJSON_AddStringToObject(jsonObject, "bssid", mac);
36          cJSON_AddNumberToObject(jsonObject, "rssi", ap_info[i].rssi);
37          cJSON_AddStringToObject(jsonObject, "name", (char
↪   *)ap_info[i].ssid);
38          cJSON_AddItemToArray(jsonArray, jsonObject);
39
40          free(mac);
41      }
42
43      if (uxQueueSpacesAvailable(queue) <= 0)
44      {
45          ESP_LOGE(SCAN_SERVICE_TAG, "No space, removing first
↪   element...");
46          cJSON *jsonValue = NULL;
47
48          xQueueReceive(queue, &(jsonValue), (TickType_t)5);
49          cJSON_Delete(jsonValue);
50      }
51      xQueueSend(queue, &jsonArray, (TickType_t)0);
52
53      // Print jsonArray and cleanup
54  }
```

Listing 26: scan_rssi function of the "ScanService" component. (Part 2)

in section 4.2 of chapter 4, we discussed a "FINGERPRINT_SERVICE_COLLECT" variable. This variable is the time the tag will scan for WiFi networks. The ESP-IDF WiFi interface only allows us to specify a minimum and maximum scan time per channel. Thus we need to divide our configuration value by the 13 channels to get our scan time per channel. This is done at line 8 of listing 25. In lines 10-18 we create a configuration variable for the WiFi interface scan. In there, we set the maximum scan time per channel to the computed value at line 8, and we set the minimum scan time per channel to half that value. The scan is started at line 21 using the "esp_wifi_scan_start" function. In lines 30-41 of listing 26 we create a representation of a JSON object in the format described above and add it to the "jsonArray". After this, the "scan_rssi" function will check the queue for space. If there is no space, it will remove the first element of the queue and delete it. This is done at lines 43-50. Note that, since the queue follows an FIFO methodology, the first element of the list is the oldest. Then, the "jsonArray" is added to the

queue at line 51. Lastly, the constructed "jsonArray" is printed to serial output and the allocated memory is freed.

#### 5.2.6.2 Implementation of CSI fingerprint collection

In order to collect CSI data packets, there needs to be some kind of WiFi traffic between the AP the tag is connected to and the tag itself. With our current implementation, there is no traffic generated while the tag is scanning for fingerprints. This would mean that little to no CSI fingerprints would be collected.

One way to overcome this issue is to configure the tag to be in sniffer mode, also called, promiscuous mode. In promiscuous mode, the tag will also collect CSI data packets destined to other devices. This would mean there would be a higher chance of collecting CSI data when doing a fingerprint scan. This solution would work in a scenario with a lot of devices generating traffic. But it will not work in a scenario where there are little to no other devices besides the tag and the AP.

Another solution is to actively generate traffic while collecting CSI data packets. One way to do this is to ping the AP the tag is connected to. This will generate a consistent number of fingerprints, based on how many pings are sent to the AP. Moreover, it will not be affected by the lack of devices and traffic around the tag. It is important to note that sending pings to the AP could affect the AP's performance negatively.

Given the advantages of the latter approach over the first, and the fact that, if there is not a large volume of pings being sent to the AP, the performance lost is negligible, we will follow the latter approach in our implementation. Note that we could also have a combination of both approaches. Using the promiscuous mode would still introduce a non-deterministic number of CSI fingerprints that would be collected at any given time. We decided to not use this approach, since it would cause more uncertainty in the system.

By default, the ESP32 WiFi interface does not collect CSI data. To do that, the ESP-IDF project configuration needs to be changed. Similarly to what we did in subsection 5.2.3, we will use the ESP-IDF extension of VSCode to allow the microcontroller to collect CSI data. Again, we can open the configuration menu in the button depicted in figure 29. Afterwards, we navigate to the "Components config" section and then to the "Wi-FI" subsection. In there, we enable the "Wi-Fi CSI (Channel State Information)" option, as depicted in figure 32.

After having the project's configurations set, we can start implementing the CSI fingerprint collection logic. The ESP-IDF allows us to collect CSI data using a callback function. This callback function will be called every time the microcontroller receives a CSI packet and the collection of CSI data is enabled.

As discussed before, when the tag is configured to collect CSI fingerprints, additional initialization logic needs to be implemented. This is done in lines 7-12 of listing 24. In line 10 of the same listing, the "scan_csi_init" function is called.

The "scan_csi_init" function is shown in listing 27. Here we configure the WiFi interface to collect all possible CSI data. Moreover, it is in this function that we set the callback to handle CSI data, using

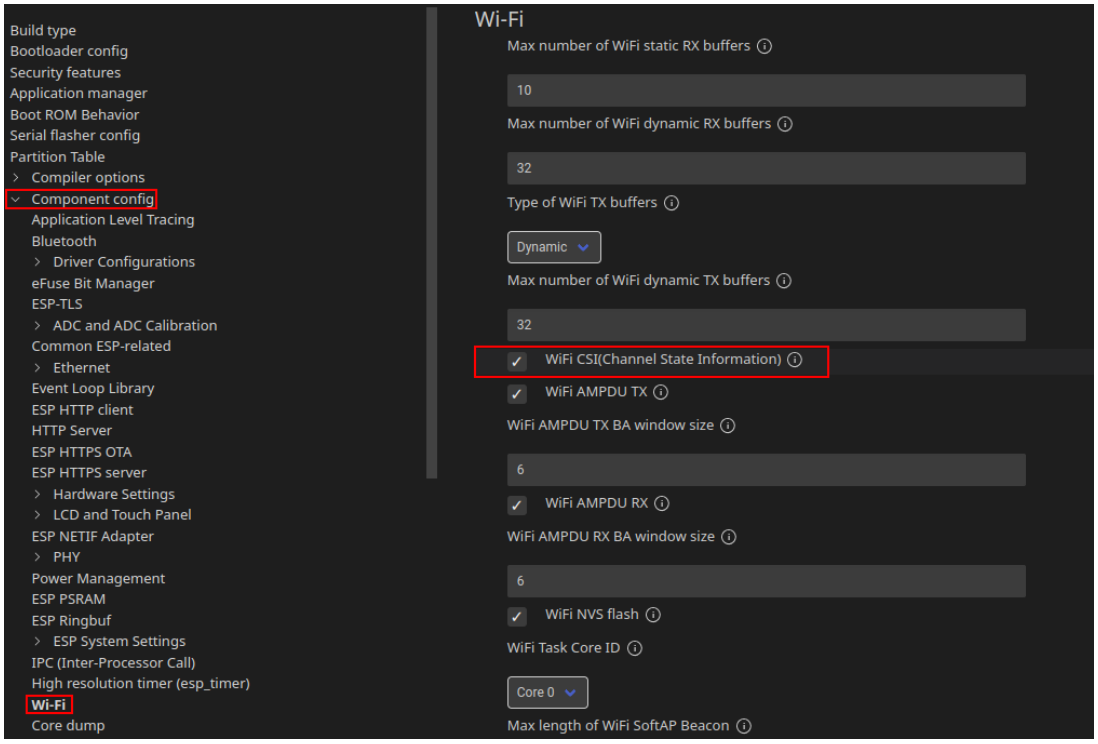



Figure 32: CSI configuration changes.

```
1  void scan_csi_init(QueueHandle_t queue)
2  {
3      wifi_csi_config_t csi_config = {
4          .lltf_en = true,
5          .htltf_en = true,
6          .stbc_htltf2_en = true,
7          .ltf_merge_en = true,
8          .channel_filter_en = true,
9          .manu_scale = false,
10         .shift = false,
11     };
12
13     esp_wifi_set_csi_config(&csi_config);
14     esp_wifi_set_csi_rx_cb(scan_csi_cb, queue);
15 }
```

Listing 27: scan_csi_init function of the "ScanService" component.

the "esp_wifi_set_csi_rx_cb" in line 14. An excerpt of the "scan_csi_cb" callback function is shown in listing 28.

In the "scan_csi_cb", a JSON object representing a CSI fingerprint will be constructed using the cJSON library. This object follows the format discussed in subsection 4.3.3 of chapter 4 for CSI fingerprinting. Listing 4 is an example of this. Each time a CSI data is captured and the "scan_csi_cb"

```
1  void scan_csi_cb(QueueHandle_t queue, wifi_csi_info_t *info)
2  {
3
4      const wifi_pkt_rx_ctrl_t *rx_ctrl = &info->rx_ctrl;
5
6      cJSON *jsonValue = NULL;
7      jsonValue = cJSON_CreateObject();
8
9      char *apMac = (char *)malloc(sizeof(char) * 18);
10     sprintf(apMac, MACSTR, MAC2STR(info->mac));
11
12     cJSON_AddStringToObject(jsonValue, "apMAC", apMac);
13     cJSON_AddNumberToObject(jsonValue, "rssi", rx_ctrl->rssi);
14     cJSON_AddNumberToObject(jsonValue, "channel", rx_ctrl->channel);
15     cJSON_AddNumberToObject(jsonValue, "secondaryChannel",
   ↪  rx_ctrl->secondary_channel);
16     cJSON_AddNumberToObject(jsonValue, "timestamp",
   ↪  rx_ctrl->timestamp);
17
18     cJSON *dataArray = cJSON_CreateArray();
19
20     for (int i = 1; i < info->len; i++)
21     {
22         cJSON *number = cJSON_CreateNumber(info->buf[i]);
23         cJSON_AddItemToArray(dataArray, number);
24     }
25
26     cJSON_AddItemToObject(jsonValue, "data", dataArray);
27
28     if (uxQueueSpacesAvailable(queue) <= 0)
29     {
30         ESP_LOGE(SCAN_SERVICE_TAG, "No space, removing first
   ↪  element...");
31         cJSON *jsonValue = NULL;
32
33         xQueueReceive(queue, &(jsonValue), (TickType_t)5);
34         cJSON_Delete(jsonValue);
35     }
36
37     xQueueSend(queue, &jsonValue, (TickType_t)0);
38
39     // Print constructed data and cleanup memory
40     // (...)
41 }
```

Listing 28: scan_csi_cb function of the "ScanService" component.

function is called, it is given an "info" argument. The "info" argument is a pointer to a struct of type "wifi_csi_info_t" that contains, among others, all of the information needed to construct the JSON object. This is done in lines 6-26.

Note that, in listing 27, when setting the CSI callback to the "scan_csi_cb" function, we also gave a queue handler to the "esp_wifi_set_csi_rx_cb" function. This queue handler will be given as a parameter to the "scan_csi_cb" when it is called. After constructing the JSON representation of the CSI fingerprint, we will use this queue handler to add the fingerprint to the queue. This logic is very similar to what was discussed before for RSSI fingerprints. We first check if the queue is full, and if it is, the oldest element is removed. Then, the CSI fingerprint is added to the queue. This is done in lines 28-37 of listing 28.

After executing the "scan_csi_init" function, the tag is configured to capture CSI data and call the "scan_csi_cb" callback function described above. But it will not start collecting CSI fingerprints packets at that point because the WiFi interface still needs to be set to collect CSI data packets. This will be done in line 23 of listing 24 and will be explained later.

Recall that we decided to create traffic in the network using pings to be able to collect CSI data consistently. Following the "scan_csi_init" function, we will configure a ping service that will generate this traffic. This is done using the "scan_ping_router_init" function in line 11 of listing 24. The "scan_ping_router_init" function is shown in listing 29.

The "scan_ping_router_init" will configure a service provided by the ESP-IDF to ping devices. In lines 5-8, we configure this service to send an infinite amount of pings every second. Afterwards, in lines 10-13, we check the IP of the AP the tag is currently connected to and configure the service to send the ping to that address. The "scan_ping_router_init" will return a ping handler configured as described. We can use this ping handler afterwards to start and stop the service from sending pings, note that at this point of the execution flow, no pings are being sent.

The "scan_csi" function in line 23 of listing 24, starts the collection of CSI fingerprints flow. This function is shown in listing 30.

At the start of the "scan_csi" function, the ping service is started by calling the "esp_ping_start" function with the ping handler created before when initializing the ping service. Immediatly following this, the WiFi interface is set to collect CSI data packets using the "esp_wifi_set_csi" function. At this point, the ping service is generating traffic in the network and that is triggering the "scan_csi_cb" callback function explained before. Recall that we have defined a configuration value for the time the tag takes to collect fingerprints. By using the "vTaskDelay" function, we will suspend the execution of the "scan_csi" function for the configured amount of time to collect fingerprints. Both the ping service and the "scan_csi_cb" callbacks are executed in different tasks. Thus, the tag will be collecting CSI fingerprints while the "scan_csi" function is suspended. When the time to collect fingerprints ends, the tag resumes the execution of "scan_csi" function and stops sending pings and collecting CSI data. This is done in lines 13-14 of listing 30.

This concludes our explanation of the two different fingerprint scans implemented. After executing either the RSSI scan flow, or the CSI scan flow, the tag will enter a sleep state where it does not make

```
esp_ping_handle_t scan_ping_router_init()
{
    esp_ping_handle_t ping_handle = NULL;

    esp_ping_config_t ping_config = ESP_PING_DEFAULT_CONFIG();
    ping_config.count = ESP_PING_COUNT_INFINITE;
    ping_config.interval_ms = 1000;
    ping_config.data_size = 1;

    esp_netif_ip_info_t local_ip;

    esp_netif_get_ip_info(esp_netif_get_handle_from_ifkey("WIFI_STA_DEF"),
    &local_ip);
      ping_config.target_addr.u_addr.ip4.addr =
    ip4_addr_get_u32(&local_ip.gw);
      ping_config.target_addr.type = ESP_IPADDR_TYPE_V4;

    esp_ping_callbacks_t cbs = {0};
    esp_ping_new_session(&ping_config, &cbs, &ping_handle);

    return ping_handle;
}
```

Listing 29: scan_ping_router_init function of the "ScanService" component.

```
void scan_csi(esp_ping_handle_t ping_handle)
{
    esp_ping_start(ping_handle);

    esp_wifi_set_csi(true);

    ESP_LOGI(SCAN_SERVICE_TAG, "Starting to collect CSI data...");

    vTaskDelay(configs.FINGERPRINT_SERVICE_COLLECT /
    portTICK_PERIOD_MS);

    ESP_LOGI(SCAN_SERVICE_TAG, "Stopping to collect CSI data...");

    esp_ping_stop(ping_handle);
    esp_wifi_set_csi(false);
}
```

Listing 30: scan_csi function of the "ScanService" component.

new scans for a limited amount of time at line 30 of listing 24. Lastly, after the configured sleep time, the tag will perform new scans, since we are inside an infinite while loop. Note that, on lines 16, 29 and 35 of listing 24, a LED is being turned on and/or off. We will discuss this further in section 5.3.

### 5.2.7 Send fingerprints

In this subsection we will analyse the implementation of the "Send fingerprints" state. In this state, the tag sends all of the fingerprints present in the queue to a remote fingerprint server.

In subsection 5.2.2 we see that the "Send fingerprints" state is started by the "xTaskCreatePinnedToCore" function in line 36 of listing 5. This function creates a task that runs the "send_task" function in core 1 of the tag's microcontroller. An excerpt of the "send_task" function can be found in listings 31 and 32.

Recall the state machine diagram of this state in figure 25. We have a "normal" cycle where all of the fingerprints in the queue are sequentially sent to the server. There is also a cycle for when something goes wrong and the tag can't send a fingerprint at first. We will first analyse the "normal" flow of this state, omitting the error handling flow for now.

On line 15, the tag checks if there are fingerprints stored in the queue. If there is not, the tag goes to a sleep state for the amount of time configured in the previous states. This happens on line 62. If there are fingerprints, the tag will try to "take" the semaphore in line 21. If the tag is unable to "take" the semaphore, it will print an error message and go back to the beginning of the while loop in lines 50-54. Note that, the while loop is infinite. This means the tag will keep trying to get the semaphore until it is successful. On line 21 we also check if the "semaphore_taken" variable is true. This variable is set to true after the tag is able to "take" the semaphore and is set to false when the tag "gives" the semaphore. This is needed because our logic assumes the tag is running in an infinite loop and, in some cases, goes back to the beginning of the loop after "taking" the semaphore. Since the semaphore was taken and not released, without this variable, the tag would not be able to run the operations inside the if statement again.

After successfully "taking" the semaphore, the tag will send the first fingerprint in the queue to the configured remote fingerprints server by calling the "http_post_fingerprints" at line 27. If that is successful, the tag removes the fingerprint from the queue and releases the allocated memory for it. The "http_post_fingerprints" function is part of the "HttpService" and is very similar to the "http_get_configuration_data" function described above in listings 20 and 21. The only difference is the different body, which will follow the format discussed in section 4.3 of chapter 4.

The tag will now check if there are more fingerprints in the queue at line 44. If there are, it will go back to the beginning of the queue and use the same logic described above to send another fingerprint. This is possible due to the "semaphore_taken" variable described above. The tag will keep this cycle until there are not fingerprints in the queue to be sent. When this happens, the tag will "release" the

```
1  void send_task(send_parameters_t *params)
2  {
3      const char *pcTaskName = "Send task is running\r\n";
4      ESP_LOGI(TAG, "%s", pcTaskName);
5
6      bool semaphore_taken = false;
7      unsigned short attempts_counter = 0;
8      unsigned short retries_counter = 0;
9
10     while (1)
11     {
12
13         cJSON *jsonArray = NULL;
14
15         if (uxQueueMessagesWaiting(queue) > 0)
16         {
17             // There is fingerprints to send
18             gpio_set_level(SEND_LED_PIN, true);
19             ESP_LOGI(TAG, "LOOP send task");
20
21             if (semaphore_taken || xSemaphoreTake(xSemaphore,
↪  SEMAPHORE_WAIT_TIME))
22             {
23                 semaphore_taken = true;
24
25                 xQueuePeek(queue, &(jsonArray), (TickType_t)0);
26                 ESP_LOGI(TAG, "Sending data to server...");
27                 if (http_post_fingerprints(jsonArray) == ESP_OK)
28                 {
29                     // Remove from queue
30                     xQueueReceive(queue, &(jsonArray), (TickType_t)0);
31
32                     // Set counter to 0 in case they were incremented
33                     attempts_counter = 0;
34                     retries_counter = 0;
35                     cJSON_Delete(jsonArray);
36                 }
37                 else
38                 // Reconnect to wifi and resend fingerprint logic
39                 // (error handling flow)
40                 // (...)
```

Listing 31: send_task function of the "SendService" component. (Part 1)

```
41                  ESP_LOGE(TAG, "Queue size: %d",
    ↪  uxQueueMessagesWaiting(queue));
42
43                  // if queue is not empty continue sending fingerprints
44                  if (uxQueueMessagesWaiting(queue) > 0 &&
    ↪  attempts_counter <= 0)
45                  {
46                      continue;
47                  }
48              }
49              else
50              {
51                  ESP_LOGE(TAG, "Failed to take semaphore for sending");
52                  gpio_set_level(SEND_LED_PIN, false);
53                  continue;
54              }
55
56              xSemaphoreGive(xSemaphore);
57              semaphore_taken = false;
58
59              gpio_set_level(SEND_LED_PIN, false);
60          }
61
62          vTaskDelay(configs.MESSAGE_SERVICE_SLEEP / portTICK_PERIOD_MS);
63      }
64  }
```

Listing 32: send_task function of the "SendService" component. (Part 2)

semaphore and set the "semaphore_taken" variable to false on lines 56-57. Afterwards, the tag enters the sleep state for the configured amount of time at line 62.

Now that we have analysed the "normal" flow, we will now take a look at the error handling code that was omitted from listing 31. This code block would start at line 39 of listing 31. The code block is shown in listing 33.

If the "http_post_fingerprints" function returns a value different than "ESP_OK", the tag will enter this code block. First, it will check if the tag is connected to any WiFi network using the "is_wifi_connected" function. The "is_wifi_connected" function is part of the "WiFiService" component and returns true if the tag is connected to a WiFi network of false otherwise.

If the tag is connected, we will increase "retries_counter" and print an error message. Then we'll go back to the beginning of the while loop. This happens in lines 10-15 of listing 33. The "retries_counter" variable keeps track of how many retries were attempted to send the same fingerprint. On line 4 we check if the number of retries exceeded the configured maximum number of retries. If it did, the "retries_counter"

```
{
    if (is_wifi_connected())
    {
        if (retries_counter >= configs.MAX_MESSAGE_RETRIES)
        {
            retries_counter = 0;
            attempts_counter++;
        }
        else
        {
            retries_counter++;
            ESP_LOGE(TAG, "Failed to send fingerprints to server... Retrying");
            ESP_LOGE(TAG, "Retry nº %d", retries_counter);
            continue;
        }

        if (attempts_counter >= configs.MAX_MESSAGE_ATTEMPTS)
        {
            ESP_LOGE(TAG, "Failed to send fingerprints to server");
            esp_system_abort("Failed to send fingerprints to server (MAX_MESSAGE_ATTEMPTS reached)");
        }

        ESP_LOGE(TAG, "Retry nº %d, Attempt nº %d", retries_counter, attempts_counter);
    }
    else
    {
        ESP_LOGE(TAG, "No Wifi connection... Reconnecting");
        wifi_disconnect();
        esp_err_t connected = wifi_connect_from_config();
        if (connected == ESP_OK)
        {
            ESP_LOGI(TAG, "Reconnected!");
            continue;
        }
        else
        {
            ESP_LOGE(TAG, "Failed to reconnect");
            esp_system_abort("Failed to reconnect");
        }
    }
}
```

Listing 33: Reconnect and retry code block of the send_task function.

variable will be set to 0 and the "attempts_counter" variable will be incremented in lines 5-8. Afterwards, the tag will leave this block of error handling and go into the sleep state started by line 62 of listing 32. The "attempts_counter" variable controls how many times the tag tries to send a fingerprint and fails after the configured maximum number of retries. If the tag exceeds the configured maximum number of attempts, it will go to the "Unrecoverable Error" state. This happens at lines 17-21 of listing 33. There, we call the "esp_system_abort" function instead of using the "ESP_ERROR_CHECK" macro used before. This is because the "send_task" is running on a FreeRTOS task, which is essentially a thread. Using the "ESP_ERROR_CHECK" macro here would only kill this thread and not halt the program completely. We chose to halt the program completly, because if the tag can't communicate with the remote fingerprints server, having the "scan_task" running would, in practice, achieve nothing.

If the "is_wifi_connected" function returns false, the tag will try to reconnect itself to any configured WiFi network. To do this, we will use the exact same functions and logic used in the "Connect to WiFi" state, which was already analized in subsection 5.2.4. If the tag reconnects to a WiFi network, it will go back to the beginning of the while loop and try to send the fingerprints again. Otherwise, it will go to an "Unrecoverable Error" state. This happens at lines 35-39 of listing 33.

### 5.2.8 AP mode button

During the development of our solution, we noticed it would be good to have a way of reconfiguring the tag when it is running. We already have a flow where the tag creates a web interface. Users can connect to it with any device that has a browser and WiFi connectivity capabilities. This flow was discussed in subsection 5.2.3. We added a physical button to our solution that, when clicked, would set the tag in AP mode and serve the same web configuration interface. To do this, we will use the same logic used and explained before.

An ISR was used to trigger the logic that places the tag in AP mode and serves the web interface. An ISR is a subroutine that is automatically executed in response to a specific hardware interrupt. The ISR service is initiated in line 7 of listing 5. Then we register the "scan_button_isr_handler" to be called when GPIO "PUSH_BUTTON_PIN" is interrupted, in line 8 of the same listing. In practice, when we push the button connected to the "PUSH_BUTTON_PIN", the "scan_button_isr_handler" will be called. The "scan_button_isr_handler" is shown in listing 34.

The "scan_button_isr_handler" function includes a software debouncer that will ensure there is no more than one button press registered per "DEBOUNCE_DELAY_MS" milliseconds.

Along with the "scan_task" and "send_task" tasks, a "setApModeConfigTask" was also initiated at line 37 of listing 5. The "setApModeConfigTask" is shown in listing 35. Note that, at line 5 of listing 35, the "vTaskSuspend" function with a "NULL" parameter is called. The "vTaskSuspend" function is part of the FreeRTOS and suspends a task given as a parameter. If "NULL" is given to the "vTaskSuspend" task, the caller task will be suspended. Thus, when the "setApModeConfigTask" task is started, it will be immediately suspended.

```
void IRAM_ATTR scan_button_isr_handler(void *arg)
{

    uint64_t current_time = esp_timer_get_time();
    uint64_t time_diff = current_time - scan_button_last_press_time;
    scan_button_last_press_time = current_time;

    if (time_diff >= (DEBOUNCE_DELAY_MS * 1000))
    {
        int gpio_level = gpio_get_level(PUSH_BUTTON_PIN);
        if (gpio_level == 0)
            return;

        xTaskResumeFromISR(setApModeConfigTaskHandle);
    }
}
```

Listing 34: scan_button_isr_handler function of the project's main file.

```
void setApModeConfigTask(void *params)
{
    while (1)
    {
        vTaskSuspend(NULL);
        vTaskSuspend(sendTaskHandle);
        vTaskSuspend(scanTaskHandle);
        wifi_init_softap();
        ESP_LOGI(TAG, "Starting Configuration server...\n");
        setup_server();
    }
}
```

Listing 35: setApModeConfigTask function of the project's main file.

At line 14 of listing 34, we call the "xTaskResumeFromISR" function. The ""xTaskResumeFromISR" is part of the FreeRTOS and it resumes a suspended task given as a parameter. In our case, we will resume the "setApModeConfigTask" task. Note that, in listing 35, the "setApModeConfigTask" was suspended at line 5. Thus, when pressing the button, the "setApModeConfigTask" will resume, and it will suspend the "send_task" and "scan_task" tasks, place the tag in AP mode and serve the configuration server in lines 6-10 of listing 35.

### 5.2.9 Accelerometer

Having in mind the implementation of the tag done so far, the tag will always be making fingerprinting and sending cycles even if it is stationary. If the tag is stationary, there is not much added value for the tag to constantly collect and send fingerprints to the positioning engine, since the inferred position will remain the same. With this in mind, we could make our solution more power-efficient by slowing down the collection and sending of fingerprints cycles when the tag is stationary.

In order to detect if the tag is stationary, an accelerometer sensor will be added to our solution. An accelerometer is a device that measures the acceleration of a body relative to its resting frame. By adding an accelerometer to our tag, we can detect when the tag is moving by analyzing its acceleration at any given moment. If there is no acceleration detected, it is likely the tag is not in motion. Note that, following this implementation is far from perfect. By having no acceleration detected by the accelerometer attached to the tag does not directly mean the tag is not in motion. Picture a scenario, where the tag is in a perfectly balanced conveyor belt, traveling at a constant speed. Since there are no changes in speed, there is no acceleration, but the tag is still moving in relation to an observer's reference frame.

Our world is far from perfect, and in practice, the tag will never be in motion in a perfect conveyor belt at a perfect constant speed. There are going to be bumps, turns and other factors that will output an acceleration.

The opposite is also true. The accelerometer could measure an acceleration at a given point, but, in practice, the tag could remain stationary. For example, if the tag is stationary on a table and we violently tap the table, an acceleration will be measured by the tag's accelerometer, but the tag remained in the same position.

If we could accurately detect that the tag is not in motion, stopping the tag from doing its collection and sending fingerprints cycles would be very power-efficient. But, as we saw above, our approach to detecting movement could yield false positives. Thus, instead of stopping the cycles all together, we decided to slow the cycles down. This way, even if the tag detects it is stationary but it is not, it will not impact the IPS in a serious way, since it will be collecting and sharing fingerprints with the remote positioning server anyways, but at a slower pace.

Included with the acquired Freenove Starter Kit discussed in chapter 4, there was an MPU6050 module. The MPU6050 sensor module includes, among others, a 3-axis accelerometer. We will discuss this module better in section 5.3, but it is important to note that this module gives an acceleration reading

```
1  void start_i2c(void)
2  {
3      i2c_config_t conf;
4      conf.mode = I2C_MODE_MASTER;
5      conf.sda_io_num = (gpio_num_t)13;
6      conf.scl_io_num = (gpio_num_t)14;
7      conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
8      conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
9      conf.master.clk_speed = 100000;
10     conf.clk_flags = 0;
11     i2c_param_config(I2C_NUM_0, &conf);
12     i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0);
13 }
```

Listing 36: start_i2c function of the "Accelerometer" component.

for each tridimensional axis in g-force (g). Moreover, the module supports the common Inter-Integrated Circuit (I2C) protocol that allows it to communicate with the tag's microcontroller. The I2C protocol allows multiple "peripherals" or "slaves" to communicate with one or more "controllers" or "masters". To implement the logic that will read the MPU6050 module's accelerometer readings, infer if the tag is stationary and slow the tag's main cycles down, we decided to create and add an ESP-IDF component to our solution. To do this, we will use the same approach discussed previously in subsection 5.2.1 of this chapter. The newly added component was named "Accelerometer".

In order to read the MPU6050 module's accelerometer readings, we need to enable and initialize the I2C protocol in the tag's microcontroller. Moreover, we need to register the microcontroller as a "master" and set the pins used for I2C protocol communications. This is done by calling the "start_i2c" function in the "app_main" function of our source code. The "start_i2c" is shown in listing 36. After having the I2C protocol correctly configured, we can now start to communicate with the MPU6050 module. Communication using the I2C protocol can be time-consuming to implement. Since the MPU6050 module is commonly used, we researched if there was any library we could include in our solution that would simplify our implementation.

The ESP-IDF provides a component registry, where open-source components can be shared and easily integrated. With a quick search in this registry, we found a component made specifically to simplify the communication with MPU6050 modules.

The component can be installed in various ways, but we will install it using the ESP-IDF extension for the VSCode IDE. First, we need to open the ESP-IDF components registry using a VSCode command, as shown in figure 33. Then, we use the search bar and search for "mpu6050". The search results should include a component named "espressif/mpu6050". Clicking on that search result will open a page with information about the component and an "Install" button as shown in figure 34. To install the component, we just need to click the "Install" button and the component will be automatically downloaded

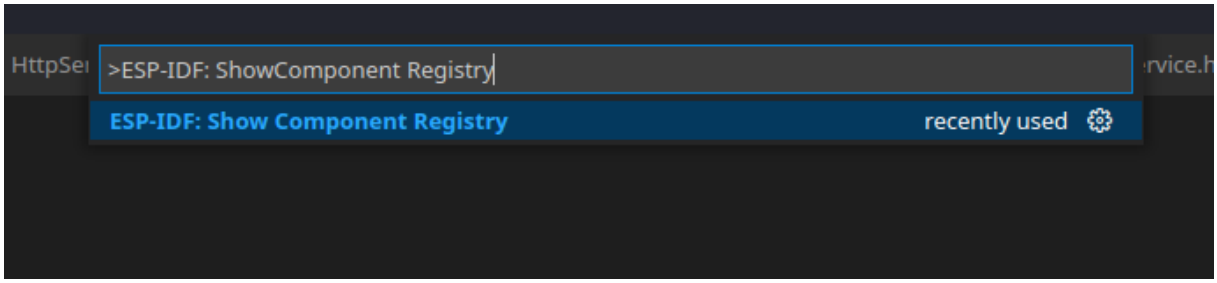



Figure 33: VSCode command to open the ESP-IDF registry.

and installed. To use it from this point forward, we only need to include the "mpu6050.h" header file in our source code.

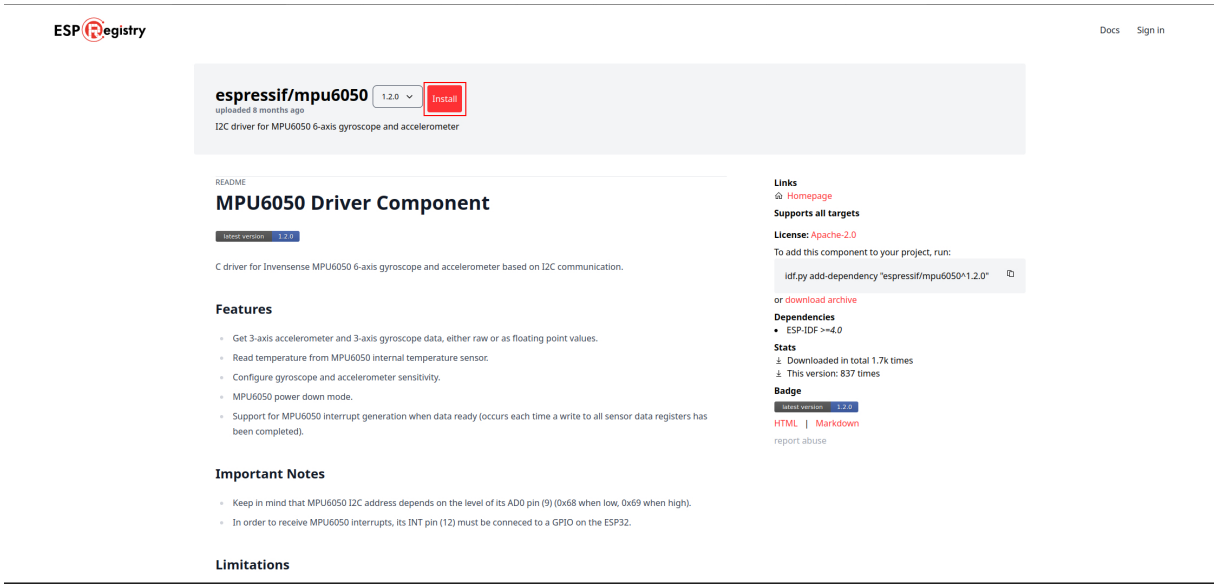



Figure 34: MPU6050 component registry page.

Using functions from this library, a "mpu6050_init" function was created that contains the initialization logic needed prior to accelerometer readings. The "mpu6050_init" is shown in listing 37.

In lines 3-5 we use functions from the "espressif/mpu6050" component installed previously to initialize the MPU6050 module. Then in lines 7-14, an ESP-IDF framework timer is registered that will call a "mpu6050_read" function every 5ms. Lastly, on lines 16-17, we store the current values of the "FINGERPRINT_SERVICE_SLEEP" and "MESSAGE_SERVICE_SLEEP" configuration values in two global variables. In order to slow down the tag's cycles when no movement is detected, the tag's configurations are going to be modified with increased sleep timers. The initial values need to be stored to revert the configurations back to its original form when the tag is in movement again.

The "mpu6050_read" function will make a reading of the MPU6050 module's accelerometer. Then, it will infer if the tag is moving or not and adjust the configuration values according to the situation. The "mpu6050_read" function is shown in listing 38.

```
1  void mpu6050_init()
2  {
3      mpu6050_dev = mpu6050_create(0, MPU6050_I2C_ADDRESS);
4      mpu6050_config(mpu6050_dev, ACCE_FS_2G, GYRO_FS_500DPS);
5      mpu6050_wake_up(mpu6050_dev);
6
7      const esp_timer_create_args_t cal_timer_config = {
8          .callback = mpu6050_read,
9          .arg = NULL,
10         .name = "MPU6050 timer",
11         .skip_unhandled_events = true,
12         .dispatch_method = ESP_TIMER_TASK};
13     esp_timer_create(&cal_timer_config, &cal_timer);
14     esp_timer_start_periodic(cal_timer, 5000); // 5ms
15
16     ORIGINAL_FINGERPRINT_SERVICE_SLEEP =
   ↪  configs.FINGERPRINT_SERVICE_SLEEP;
17     ORIGINAL_MESSAGE_SERVICE_SLEEP = configs.MESSAGE_SERVICE_SLEEP;
18 }
```

Listing 37: mpu6050_init function of the "Accelerometer" component.

First, the "mpu6050_get_acce" function from the "espressif/mpu6050" component is used to make a reading of the MPU6050 module's accelerometer. As briefly discussed before, the MPU6050 module's accelerometer makes g-force readings in a tridimensional axis. Thus, the reading returns three values that represent the measured g-forces in each axis. We can represent the measured values as axis-aligned vectors with a length equal to the measured value. Now, these three vectors can be added to form a single vector of acceleration. By calculating the magnitude of the acceleration vector, we have the total acceleration that the tag is under at each reading. We can define a threshold of acceleration that we consider the tag to be stationary. Notice that there are error margins in measurements, and there is always the acceleration of gravity present in the measurements. Thus, we can not simply assume the tag is in movement if the magnitude of the acceleration vector is larger than zero. After some empirical research, we set 1.15g as the acceleration that the tag will be considered to be in movement. Note that the accelerometer is always experiencing 1g of force due to gravity, thus the threshold is in practice 0.15g. These calculations are done in the "isMoving" function that is called in line 8 of listing 38. The "isMoving" function is shown in listing 39 and returns "ESP_OK" if the tag is considered to be in motion or "ESP_FAIL" if not.

Our approach to slow the tag's main cycles will be to replace the original values of the "FINGERPRINT_SERVICE_SLEEP" and the "MESSAGE_SERVICE_SLEEP" configurations. These values will be doubled when no motion is detected. Doing this will increase the time the tag remains in the sleep state in both cycles, making it more power-efficient. When this happens, we will say the tag is in "lazy" mode.

```
1  void mpu6050_read(void *pvParameters)
2  {
3      mpu6050_get_acce(mpu6050_dev, &acce);
4
5      uint64_t current_time = esp_timer_get_time();
6      uint64_t time_diff = current_time - lastLazyModeSwitchTime;
7
8      if (isMoving(acce.acce_x, acce.acce_y, acce.acce_z) == ESP_OK &&
   ↪  inLazyMode == true)
9      {
10         ESP_LOGI(TAG, "ITS MOVING!!! Detected G force: %f , Reverting
   ↪  lazy mode configs...", sqrt(acce.acce_x * acce.acce_x + acce.acce_y *
   ↪  acce.acce_y + acce.acce_z * acce.acce_z));
11
12         revertLazyModeConfigs();
13         inLazyMode = false;
14         lastLazyModeSwitchTime = current_time;
15     }
16     else if (inLazyMode == false && (time_diff >= (60 * 1000000) ||
   ↪  lastLazyModeSwitchTime == 0))
17     {
18         ESP_LOGI(TAG, "No Movement Detected. Detected G force: %f ,
   ↪  Setting lazy mode configs...", sqrt(acce.acce_x * acce.acce_x +
   ↪  acce.acce_y * acce.acce_y + acce.acce_z * acce.acce_z));
19         inLazyMode = true;
20         setLazyModeConfigs();
21         lastLazyModeSwitchTime = current_time;
22     }
23 }
```

Listing 38: mpu6050_readfunction of the "Accelerometer" component.

A global variable named "inLazyMode" is added by the "Accelerometer" component. This variable is a boolean that is true when the tag is in "lazy" mode. This variable is initialized as false.

In lines 8-16 of listing 38, we have an if statement that checks if the tag is in motion. If it is, and the "inLazyMode" variable is true, it means the tag was stationary in the past and the tag's main cycle sleep values were increased. Thus, since motion was now detected, the configuration values needed to be reverted to its original values. This is done by using the "revertLazyModeConfigs" function at line 12.

If the "isMoving" function returns "ESP_FAIL" at line 8, then no movement is detected and the tag is considered to be stationary. In lines 17-22, the tag is set in "lazy" mode. The "setLazyModeConfigs" function will replace the "FINGERPRINT_SERVICE_SLEEP" and the "MESSAGE_SERVICE_SLEEP" configuration values with the double of its originally configured values.

```
esp_err_t isMoving(float x, float y, float z)
{
    const float threshold = 1.15;

    const float vectorMagnitude = sqrt(x * x + y * y + z * z);

    if (vectorMagnitude > threshold)
    {
        return ESP_OK;
    }
    else
    {
        return ESP_FAIL;
    }
}
```

Listing 39: isMoving function of the "Accelerometer" component.

As discussed, the "mpu6050_read" function is called every 5ms. In order for the tag to not be constantly switching from "normal" to "lazy" mode, a debouncer mechanism was added. After leaving "lazy" mode, the tag can only enter "lazy" mode again after one minute. In contrast, there is no debouncing mechanism to set the tag back to the normal mode. This is done to give the IPS as much accuracy as possible when the tag is in motion.

## 5.3 Hardware

The focus of our work was to have a working prototype, capable of collecting WiFi fingerprints and sharing them with a remote server. This is done with the use of an ESP32 microcontroller and the custom source code explained in the past section. However, we decided to improve our prototype using some electronic hardware. As briefly discussed in subsection 5.2.8, we added a button to our prototype. When pressed, it allows users to place the tag in AP mode, and access its web interface to reconfigure the tag. Moreover, we added two LEDs. One will be on when the tag is in the "Send fingerprints" state and not sleeping. Another, that will be on when the tag is in the "Collect fingerprints" and not sleeping. This will aid users to visually understand what the tag is doing at the moment. An MPU6050 module was also added to make the tag more power-efficient. It is used to help detect if the tag is stationary. If it is, the sleep time of the "Send Fingerprints" and "Collect Fingerprints" states will be increased. Finally, a 9V battery was added to make our tag mobile and battery-powered.

In order to integrate the various electronics with the microcontroller, we are going to use electronic wires, a push switch, resistors, a 9V battery adapter, an extension board and a breadboard. All of these were included in the development kit discussed in subsection 4.1.1 of chapter 4. The MPU6050 module

was also included. Figure 35 shows the implemented circuit diagram and figure 37 shows the microcontroller and the other electronics assembled in a breadboard.

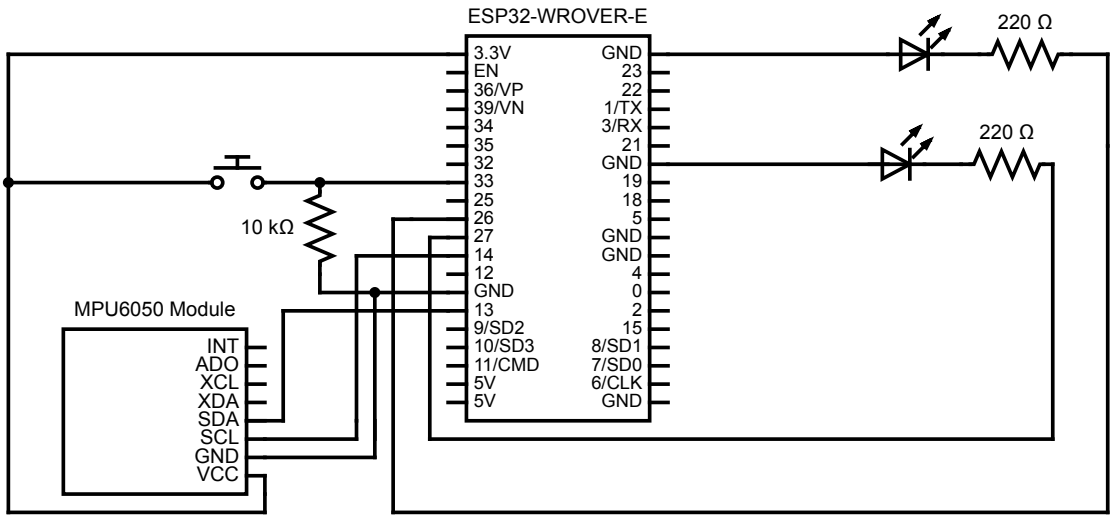



Figure 35: Circuit diagram of the tag's prototype.

An LED emits light when an electric current passes through it. In our solution, we are going to use a red and a yellow LED. The red LED should operate at 1.8V with a 20mA current. The yellow LED should operate at 2.2V with a 20mA current. The GPIO pins on our microcontroller output 3.3V. Thus, to not damage the LEDs, a resistor is needed. We can easily calculate the resistance needed for this resistor using Ohm's law. For the yellow LED, we would need a resistor of 55Ω. The lowest resistor included with the acquired kit is 220Ω, thus, this are the resistors we will use in our solution. The resistence is more than needed, so the LEDs don't light as much as they could. But in practice, it is still visible if an LED is on or off. The LEDs will be connected to the 27 and 26 GPIOs as shown in figure 35.

To implement a push switch on our circuit, we also need to have a pull-up or pull-down resistor. This is because digital logic circuits can be in one of three states: high, low, or floating. When a GPIO pin is not pulled to a high or low logic level, it is in a floating state. It is neither in a high or low logic state, and the microcontroller might unpredictably interpret the input value as either a logical high or a logical low. Pull-up and pull-down resistors help to solve this issue by pulling the GPIO pin to a logical high or logical low, respectively. In our implementation, we will use a pull-down resistor as shown in figure 35. This means that when the switch is open, the 33 GPIO pin will detect a logical low. When the switch is closed, a logical high will be detected instead. A 10kΩ resistor was used in our implementation. Switches inherit a mechanical fenomenon called bounce. When a push switch is pressed, two pieces of metal come into contact with each other. In practice, the pieces of metal are not perfectly flat or aligned. They make and break contact multiple times before the push switch gets enough force for the metal pieces to be firmly connected. To solve this issue, a debouncer needs to be implemented. A debouncer will

guarantee that when the user presses the switch, the microcontroller detects only one press. There are hardware debouncers and software debouncers. In our solution, we decided to use a software debouncer as described in subsection 5.2.8, thus no additional hardware was needed.

The MPU6050 module is a 6-axis Motion Tracking Device. It combines a 3-axis Gyroscope and a 3-axis Accelerometer all in a small package. Additionally, it includes a temperature sensor. The MPU6050 module is shown in figure 36. To communicate with microcontrollers, the MPU6050 module has an

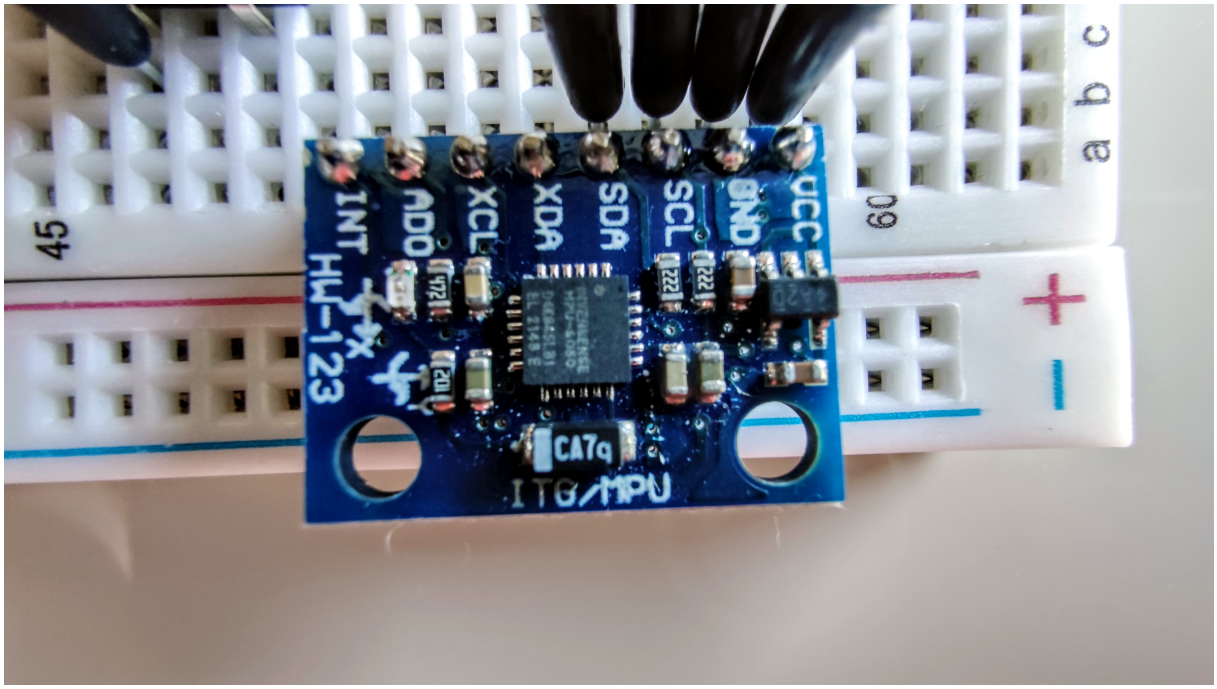



Figure 36: The MPU6050 module in the breadboard.

I2C bus interface. To use the module, there are four connections that need to be made with jumper cables. First, power needs to be supplied to the module's VCC pin. By consulting the module's data sheet, we can see it has an operation voltage in the 2.375V-3.46V range. Thus, we need to supply 3.3V to the module's VCC pin. Next, the GND pin of the module should be connected to a GND pin in the microcontroller. Afterwards, the necessary pins for the I2C protocol need to be connected. In subsection 5.2.9, we analysed a function that initializes the I2C protocol on the microcontroller's side. There, the 13 and 14 pins were assigned to be the SDA and SCL pins, respectively. We need to connect the SDA pin of the module to pin 13 of the microcontroller, and the SCL pin of the module to pin 14 of the microcontroller. The other pins in the module should be left unconnected.

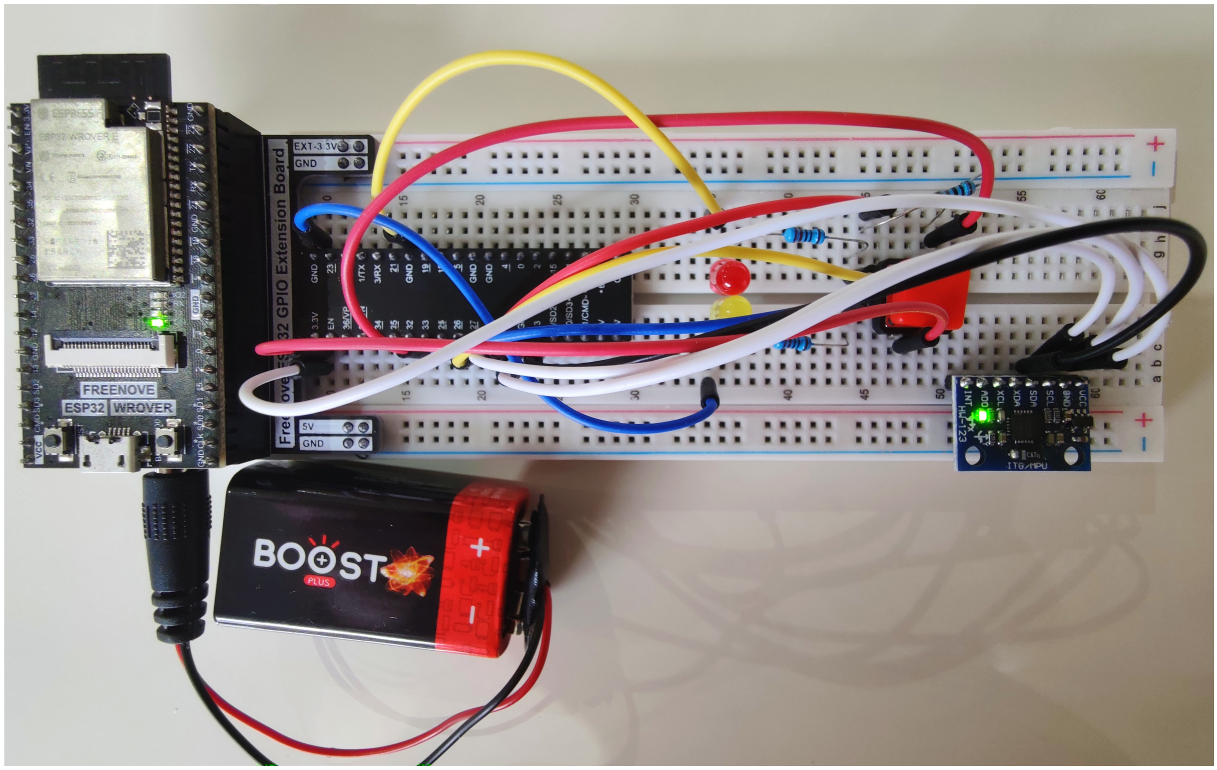



Figure 37: The final assembled circuit in the breadboard.

# 6

# System Analysis

In this chapter, we will evaluate the established solution through a series of test cases. First, we will construct test cases to evaluate the solution. Afterwards, we will review the outcomes of these tests.

## 6.1 Test Cases

After having developed the solution, it is important to test and assess its performance and robustness. Ideally, our solution will run in ideal scenarios, but we also added logic to handle less than ideal scenarios as well. In the following subsections we will create a set of test cases that will ensure the operation of the developed solution.

### 6.1.1 Test Case 1: Collection of fingerprints

Given that the collection of fingerprints is the main objective of this work, this test case will ensure that this objective was correctly accomplished.

Recall that in order for the tag to enter the "Collect fingerprints" state, it had to go through the "InitialConfiguration" and "Connect to WiFi" states with success. Thus, this test will ensure that the tag can correctly read and use configurations, connect itself to a WiFi network, and transmit fingerprints to a remote server.

The implemented tag can collect RSSI or CSI fingerprints as previously discussed. Thus, to ensure that both types of fingerprint collection are working, this test case will test both. First, the tag will be configured to collect RSSI fingerprints and then the tag will be configured to collect CSI fingerprints.

As discussed, there is a server already constructed from other works capable of receiving RSSI fingerprints. We can check if the server's database has fingerprints from our tag to show that the fingerprints are being sent correctly. As for the CSI fingerprints, the server has no support as of the writing of this work for them. Even if the fingerprints correctly follow the sugested format and protocol in subsection 4.3.3 of chapter 4, the server won't store them. Thus, to assert the correct behavior of the collection of CSI fingerprints, a custom server needs to be created. This server needs to receive the CSI fingerprints

and show the received contents to assert the correctness of this flow. With this in mind, a simple server was created using JavaScript that will simply print the body of HTTP requests sent to it. We will call this server the custom echo server. By configuring the tag to send fingerprints to this server and analyzing what was printed by it, we can confirm if the tag is working correctly. Moreover, this approach can be used along with checking the fingerprints server database when testing the collection of RSSI fingerprints.

After successfully connecting to a WiFi network, the tag will try to fetch configurations from the configured configuration server. In this test case, the tag will be configured with an invalid configuration server. This way, we can guarantee that the tag is using the default configurations flashed into its memory. Furthermore, this will test if the tag can use its default configurations if it can't get one from a remote configurations server.
In order to run the test case, the following pre-conditions must be met:

1. There is a WiFi network with credentials equal to the WiFi configuration value of listing 40, 41 and 42;
2. There is an echo HTTP server running that can be accessible by the tag at the address configured in listing 41 and 42;
3. The fingerprints server is online and can be accessed by the tag.

The test case steps are as follows:

1. Configure the tag's default configuration file with the values present in listing 40;
2. Turn the tag on;
3. Wait a couple of minutes;
4. Query the remote fingerprints server's database;
5. Reconfigure the tag's default configuration file with the values present in listing 41;
6. Turn the tag on;
7. Wait a couple of minutes;
8. Check the logs from the custom echo server;
9. Reconfigure the tag's default configuration file with the values present in listing 42;
10. Turn the tag on;
11. Wait a couple minutes;
12. Check the logs from the custom echo server.

For this test case to be successful, the following expected results must be met:

1. The fingerprints server database must have fingerprints sent by the tag stored in it's database;

2. The data format in the logs printed by the custom echo server at step 8 should follow the data format presented in listing 3;

3. The data format in the logs printed by the custom echo server at step 12 should follow the data format presented in listing 4.

```
{
  "TAG_NAME": "tagTestCase1A",
  "FINGERPRINTS_SERVER":
↪ "http://fingerprints-server/ar-ware/S02/i2a/i2aSamples.php",
  "CONFIG_SERVER": "",
  "FINGERPRINT_SERVICE_SLEEP": 1000,
  "FINGERPRINT_SERVICE_COLLECT": 1000,
  "MESSAGE_SERVICE_SLEEP": 11000,
  "QUEUE_SIZE": 15,
  "MAX_MESSAGE_ATTEMPTS": 3,
  "MAX_MESSAGE_RETRIES": 3,
  "MAX_WIFI_CONNECT_RETRIES": 3,
  "CSI_MODE": false,
  "WIFI_DETAILS": [
    {
      "PWD": "123456710",
      "SSID": "TestCase1Network"
    },
  ]
}
```

Listing 40: Default configuration file to be flashed for the first part of test case 1.

```
{
  "TAG_NAME": "tagTestCase1B",
  "FINGERPRINTS_SERVER": "http://192.168.67.16:8080",
  "CONFIG_SERVER": "",
  "FINGERPRINT_SERVICE_SLEEP": 1000,
  "FINGERPRINT_SERVICE_COLLECT": 1000,
  "MESSAGE_SERVICE_SLEEP": 11000,
  "QUEUE_SIZE": 15,
  "MAX_MESSAGE_ATTEMPTS": 3,
  "MAX_MESSAGE_RETRIES": 3,
  "MAX_WIFI_CONNECT_RETRIES": 3,
  "CSI_MODE": false,
  "WIFI_DETAILS": [
    {
      "PWD": "123456710",
      "SSID": "TestCase1Network"
    }
  ]
}
```

Listing 41: Default configuration file to be flashed for the second part of test case 1.

```
{
  "TAG_NAME": "tagTestCase1C",
  "FINGERPRINTS_SERVER": "http://192.168.67.16:8080",
  "CONFIG_SERVER": "",
  "FINGERPRINT_SERVICE_SLEEP": 1000,
  "FINGERPRINT_SERVICE_COLLECT": 1000,
  "MESSAGE_SERVICE_SLEEP": 11000,
  "QUEUE_SIZE": 15,
  "MAX_MESSAGE_ATTEMPTS": 3,
  "MAX_MESSAGE_RETRIES": 3,
  "MAX_WIFI_CONNECT_RETRIES": 3,
  "CSI_MODE": true,
  "WIFI_DETAILS": [
    {
      "PWD": "123456710",
      "SSID": "TestCase1Network"
    }
  ]
}
```

Listing 42: Default configuration file to be flashed for the third part of test case 1.

### 6.1.2 Test Case 2: Tag can serve a web interface

This test case will ensure that the tag is able to serve a web interface for users to reconfigure its configurations.

Recall that the tag should serve this web page in two scenarios. One is if there is invalid or no default configurations when the tag is in the "InitialConfiguration" state. In this case, the tag should automatically configure itself in AP mode and serve the web configuration interface. Another case is when the tag is in the "Collecting fingerprints" and "Send fingerprints" parallel states and a user presses the button to place the tag in AP mode. After this, the tag should stop collecting and sending fingerprints and provide the web configuration server to the user. In both cases, the tag should be able to reconfigure itself with the configurations given by the user via the web configuration interface.
In order to run the test case, the following pre-conditions must be met:

1. There is an echo HTTP server running that can be accessible by the tag at the address shown in figure 38 for the "FINGERPRINTS_SERVER" configuration value;
2. The fingerprints server is online and can be accessed by the tag.

The test case steps are as follows:

1. Configure the tag's default configuration file with the values present in listing 43;
2. Turn the tag on;
3. Connect to the WiFi network provided by the tag;
4. Navigate to "http://192.168.1.1" on a browser;
5. Configure the tag with the values shown in figure 38 using the web interface;
6. Wait a couple minutes;
7. Query the remote fingerprints server's database;
8. Press the tag's button;
9. Connect to the WiFi network provided by the tag;
10. Navigate to "http://192.168.1.1" on a browser;
11. Configure the tag with the values shown in figure 39 using the web interface;
12. Wait a couple minutes;
13. Check the logs from the custom echo server.

For this test case to be successful, the following expected results must be met:

1. The tag must create an open WiFi network on steps 3 and 9;
2. The web interface must be accessible on steps 4 and 10;
3. There should be fingerprints sent by the tag stored in the fingerprints server database on steps 7;
4. There should be logs of fingerprints sent by the tag in the custom echo server on step 13;

```
1 {
2   "TAG_NAME": "tagTestCase2A"
3 }
```

Listing 43: Default configuration file to be flashed for the first part of test case 2.

### 6.1.3 Test Case 3: Tag can reconnect to a WiFi network

This test case will ensure that the tag is able to reconnect to a WiFi network if it loses connection to it.

Recall that the tag should be able to reconnect to a WiFi network from its configurations if it loses connection with the previously connected network. In order to test this, we need an AP that can be easily switched off. After the correct operation of the tag for a few minutes, we will turn the AP off. The tag will lose connection and should retry to connect itself. In the meantime, we turn on the AP and check if the tag reconnected to it and resumed its normal operation. In various stages of the implementation of our solution, there were instructions to print debug logs to the serial output. We can follow some of these logs to ensure the tag is correctly following the intended flow.
In order to run the test case, the following pre-conditions must be met:

1. There is a WiFi network with credentials equal to the WiFi configuration values of listing 44;
2. The WiFi network of the previous point must be easily switchable on and off.

The test case steps are as follows:

1. Configure the tag's default configuration file with the values present in listing 44;
2. Wait a couple minutes;
3. Switch the WiFi network off;
4. Check the logs printed by the tag to the serial output;

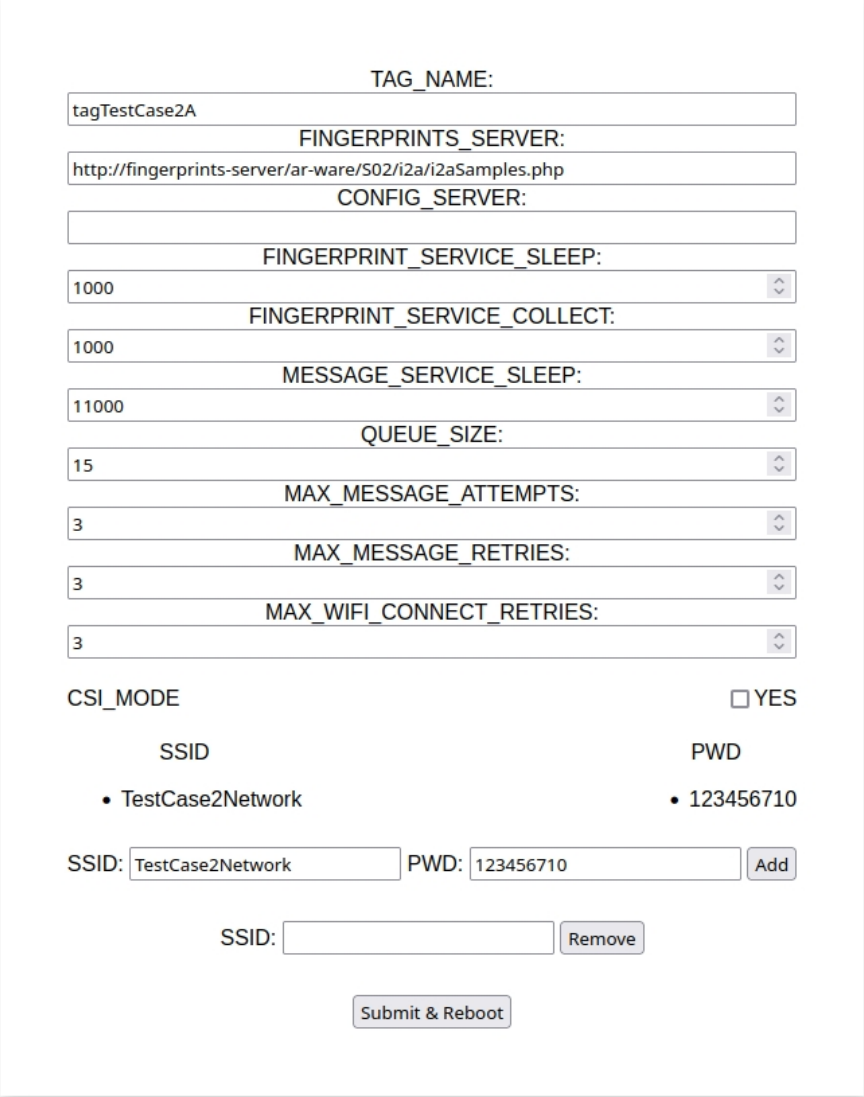



Figure 38: The configuration values to be inserted at step 5.

5. Turn the WiFi network on;
6. Check the logs printed by the tag to the serial output;
7. Wait a couple minutes;
8. Query the remote fingerprints server's database;

For this test case to be successful, the following expected results must be met:

1. The tag should print "No Wifi connection... Reconnecting" to the serial output on step 4;
2. The tag should print a "Reconnected!" to the serial output on step 6;
3. There should be fingerprints sent by the tag stored in the fingerprints server database on step 7.

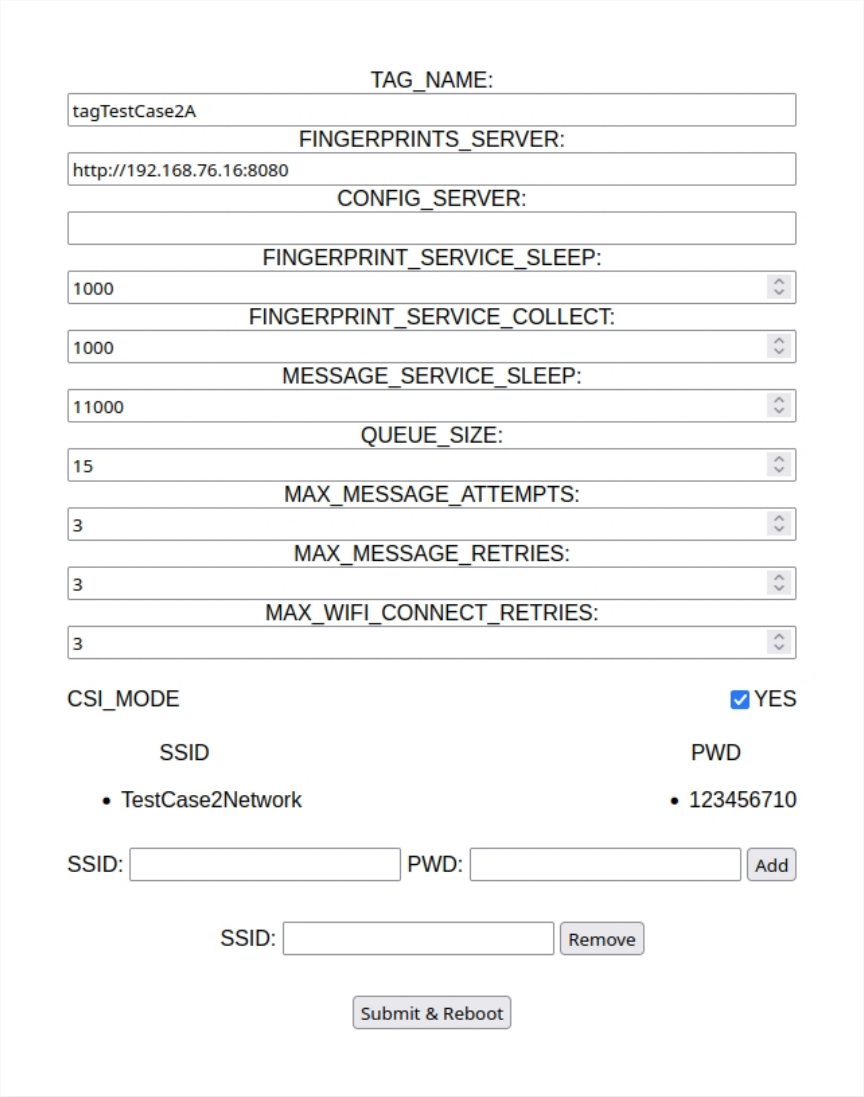



Figure 39: The configuration values to be inserted at step 11.

### 6.1.4 Test Case 4: Tag can receive configurations from a remote server

This test case will make sure the tag can reconfigure itself using the remote configurations server.

Recall that the tag should be able to receive configurations from a remote configuration server and reconfigure itself. We already have a configuration server operational from previous works. The current configurations server returns the configurations present in listing 45. Note that the configuration provided by the remote configuration server does not have the configurations keys capitalized in the returned JSON object. Our tag should be able to ignore the capitalization and use the configurations anyway. Moreover, there is missing and invalid configurations. Our tag should be able to ignore the invalid configurations and fallback to the default configurations for missing configurations.

If we configure the tag's default configurations differently, we can check if the tag reconfigures itself with the remote configuration server configurations. To do this, the default configurations should have

```
{
  "TAG_NAME": "tagTestCase3",
  "FINGERPRINTS_SERVER":
↪ "http://ils.dsi.uminho.pt/ar-ware/S02/i2a/i2aSamples.php",
  "CONFIG_SERVER": "",
  "FINGERPRINT_SERVICE_SLEEP": 1000,
  "FINGERPRINT_SERVICE_COLLECT": 1000,
  "MESSAGE_SERVICE_SLEEP": 11000,
  "QUEUE_SIZE": 15,
  "MAX_MESSAGE_ATTEMPTS": 3,
  "MAX_MESSAGE_RETRIES": 3,
  "MAX_WIFI_CONNECT_RETRIES": 30,
  "CSI_MODE": false,
  "WIFI_DETAILS": [
    {
      "PWD": "123456710",
      "SSID": "TestCase3Network"
    }
  ]
}
```

Listing 44: Default configuration file to be flashed for test case 3.

the correct address for the remote configuration server.
In order to run the test case, the following pre-conditions must be met:

1. There is a WiFi network with credentials equal to at least one of the WiFi configuration values of listing 45 and 46;
2. The fingerprints server is online and can be accessed by the tag;
3. The configuration server is online and can be accessed by the tag.

The test case steps are as follows:

1. Configure the tag's default configuration file with the values present in listing 46;
2. Wait a couple minutes;
3. Query the remote fingerprints server's database.

For this test case to be successful, the following expected results must be met:

1. There should be fingerprints sent by the tag stored in the fingerprints server database with associated tag name "tagDiogo".

```
{
    "fingerprint_service_sleep": 3000,
    "max_message_attempts": 10,
    "max_wifi_connect_retries": 3,
    "fingerprints_server":
↪  "http://fingerprints-server/ar-ware/S02/i2a/i2aSamples.php",
    "queue_size": 5,
    "config_server": "http://configs-server:8080/S11/boot",
    "max_message_retries": 25,
    "tag_name": "tagDiogo",
    "wifi_details": [
        {
            "pwd": "123456710",
            "ssid": "Rio"
        },
        {
            "pwd": "password123",
            "ssid": "rede-2"
        },
        {
            "pwd": "password123",
            "ssid": "rede-3"
        }
    ],
    "fingerprint_service_collect": 3000,
    "message_service_chunks": 30,
    "message_service_sleep": 2000
}
```

Listing 45: Configurations returned by the remote configuration server.

## 6.2 Tests Results

In this section we will discuss the outcome of the test cases described in the last section. As discussed in subsection 4.1.1 of chapter 4, we have two microcontrollers we can test the developed solution in. The test cases were made twice, once using the ESP32-WROVER-E and another using the ESP32-WROOM-32.

### 6.2.1 Test Case 1: Collection of fingerprints

This test case was completed with success and met the expected results using both microcontrollers. The evidence we are going to present refers to the test made using the ESP32-WROVER-E microcontroller.

To meet the first expected result, the fingerprints server database was queried using the SQL command present in listing 47.

```
{
  "TAG_NAME": "tagTestCase4",
  "FINGERPRINTS_SERVER":
↪ "http://fingerprints-server/ar-ware/S02/i2a/i2aSamples.php",
  "CONFIG_SERVER": "http://configuration-server:8080/S11/boot",
  "FINGERPRINT_SERVICE_SLEEP": 1000,
  "FINGERPRINT_SERVICE_COLLECT": 1000,
  "MESSAGE_SERVICE_SLEEP": 11000,
  "QUEUE_SIZE": 15,
  "MAX_MESSAGE_ATTEMPTS": 3,
  "MAX_MESSAGE_RETRIES": 3,
  "MAX_WIFI_CONNECT_RETRIES": 3,
  "CSI_MODE": false,
  "WIFI_DETAILS": [
    {
      "PWD": "password123",
      "SSID": "rede-2"
    }
  ]
}
```

Listing 46: Default configuration file to be flashed for test case 4.

```
SELECT tagName, tagBSSID , tagNetwork , scans.dataType , scanMode ,
↪ serverTimestamp , bssid , rssid FROM scans  INNER JOIN fingerprints
↪ ON scans.id=fingerprints.scan_id WHERE scans.id=(SELECT max(scans.id)
↪ FROM scans ORDER BY id DESC) ORDER BY fingerprints.scan_id DESC;
```

Listing 47: The SQL command used to query the fingerprints server database.

This SQL command will query the database for the last fingerprint received and show a table with relevant information. The result of the query is shown in figure 40. We can see from figure 40 that our tag sent the fingerprints displayed, given that the "tagName" column matches the "TAG_NAME" configuration value present in listing 40. Moreover, the timestamps in the "serverTimestamp" column match the time that the test case was run. We can further confirm the validity of the expected result by using the debug logs printed by the tag to the serial output. Figure 41 shows a log where we can see a fingerprint sent by the tag that directly matches the data shown in figure 40. We conclude that the first expected result was met.

To meet the second and third expected results, we need to analyze the logs printed by the custom echo server. Figure 42 shows the last log printed by the echo server at step 8 of the test case. By analyzing this log, we can confirm that the data sent by the tag follows the data format presented in listing 3. Thus, the second expected result was met with success. Figure 43 shows the last log printed by the

| tagName | tagBSSID | tagNetwork | dataType | scanMode | serverTimestamp | bssid | rssid |
|---|---|---|---|---|---|---|---|
| tagTestCase1A | 40:22:D8:E9:40:84 | TestCase1Network | 1 | 0 | 2023-10-14 12:23:50 | 58:FC:20:B7:64:F2 | -83 |
| tagTestCase1A | 40:22:D8:E9:40:84 | TestCase1Network | 1 | 0 | 2023-10-14 12:23:50 | 58:FC:20:B7:64:F0 | -83 |
| tagTestCase1A | 40:22:D8:E9:40:84 | TestCase1Network | 1 | 0 | 2023-10-14 12:23:50 | B0:BB:E5:E1:FB:54 | -81 |
| tagTestCase1A | 40:22:D8:E9:40:84 | TestCase1Network | 1 | 0 | 2023-10-14 12:23:50 | A2:91:8D:7C:72:F2 | -46 |

4 rows in set (0.073 sec)

Figure 40: The result of the SQL query to the fingerprints server database.

```
I (87347) HTTP_SERVICE: Sending FP to server:
 {
        "tagName":      "tagTestCase1A",
        "tagBSSID":     "40:22:D8:E9:40:84",
        "tagNetwork":   "TestCase1Network",
        "scanMode":     "auto",
        "dataType":     "Wi-Fi",
        "WiFiData":     [{
                        "bssid":        "A2:91:8D:7C:72:F2",
                        "rssi": -46,
                        "name": "TestCase1Network"
                }, {
                        "bssid":        "B0:BB:E5:E1:FB:54",
                        "rssi": -81,
                        "name": "NOS-FB54"
                }, {
                        "bssid":        "58:FC:20:B7:64:F0",
                        "rssi": -83,
                        "name": "MEO-B764F0"
                }, {
                        "bssid":        "58:FC:20:B7:64:F2",
                        "rssi": -83,
                        "name": "MEO-WiFi"
                }]
}
```

Figure 41: The log printed by the tag that matches the fingerprint shown in figure 40

echo server at step 12 of the test case. By analyzing both this log and listing 4, we can conclude they follow the same data format. Thus, the third expected result was met with success.

Concluding, given that all of the expected results were met, the test case was run successfully.

### 6.2.2 Test Case 2: Tag can serve a web interface

This test case was completed with success and met the expected results using both microcontrollers. The evidence we are going to present refers to the test made using the ESP32-WROVER-E microcontroller.

The first and second expected results were achieved with success. It was possible to connect to the WiFi network provided by the tag and access the web interface.

At step 7, we queried the fingerprints server database using the SQL command shown in listing 47. The result of the query is shown in figure 44. We can conclude the data in figure 44 was sent by our tag by analizing the "tagName" and "serverTimestamp" columns. Also, there is a log that matches the data shown in figure 44 printed by the tag. This log is shown in figure 45. Thus, we can conclude the third expected result was also met.

Before running the test case, the custom echo server was started and it printed no logs until step 12.


Figure 42: The last log printed by echo server at step 8 of test case 1.

At step 13, there was various logs printed by the custom echo server. One of such logs is shown in figure 46. Given this, we conclude the fourth expected result was met with success.

Concluding, this test case was run successfully given that all of the expected results were met.

### 6.2.3 Test Case 3: Tag can reconnect to a WiFi network

This test case was completed with success and met the expected results using both microcontrollers. The evidence we are going to present refers to the test made using the ESP32-WROVER-E microcontroller.

The first and second expected results were achieved with success. Figure 47 and 48 show the printed logs by the tag that meets both expected results.

At step 8, the fingerprints server database was queried using the SQL command shown in listing 47. The result of the query is shown in figure 44. By analizing the "tagName" and "serverTimestamp" columns, we can conclude our tag sent this fingerprints after step 5.

Given that all of the expected results were met, the tag passed this test case successfully.

### 6.2.4 Test Case 4: Tag can receive configurations from a remote server

This test case was completed with success and met the expected results using both microcontrollers. The evidence we are going to present refers to the test made using the ESP32-WROVER-E microcontroller.

The expected result of this test case was to check if the tag reconfigured itself with the remote configuration server configurations. Initially the tag was configured to be named "tagTestCase4" and the remote configuration server returns a configuration where the tag is named "tagDiogo".


Figure 43: The last log printed by echo server at step 12 of test case 1.

| tagName | tagBSSID | tagNetwork | dataType | scanMode | serverTimestamp | bssid | rssid |
|---|---|---|---|---|---|---|---|
| tagTestCase2A | 40:22:D8:E9:40:84 | TestCase2Network | 1 | 0 | 2023-10-14 17:48:55 | 58:FC:20:B7:64:F2 | -88 |
| tagTestCase2A | 40:22:D8:E9:40:84 | TestCase2Network | 1 | 0 | 2023-10-14 17:48:55 | B0:BB:E5:E1:FB:54 | -82 |
| tagTestCase2A | 40:22:D8:E9:40:84 | TestCase2Network | 1 | 0 | 2023-10-14 17:48:55 | 52:C5:24:32:94:30 | -43 |

3 rows in set (0.023 sec)

Figure 44: The result of the SQL query to the fingerprints server database.

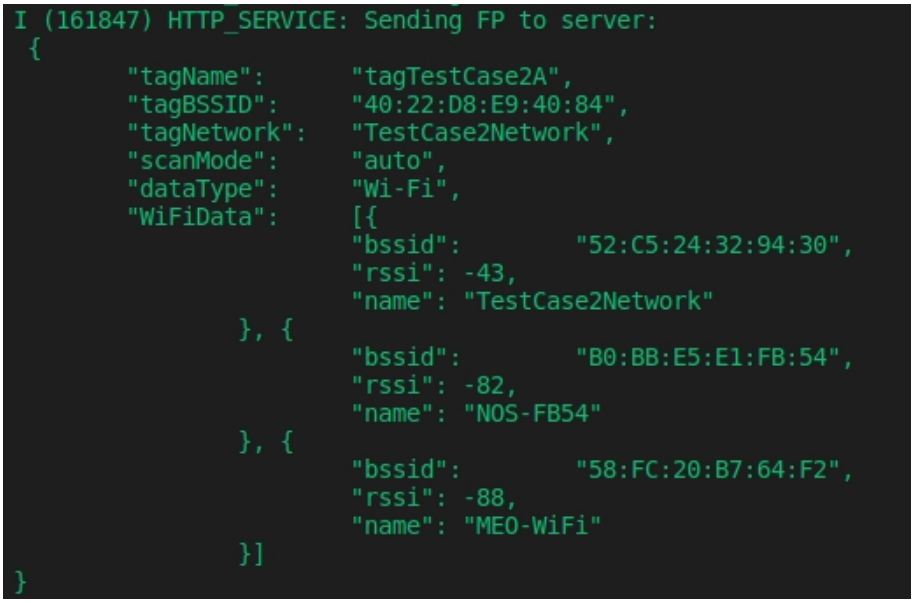



Figure 45: The log printed by the tag that matches the fingerprint shown in figure 44

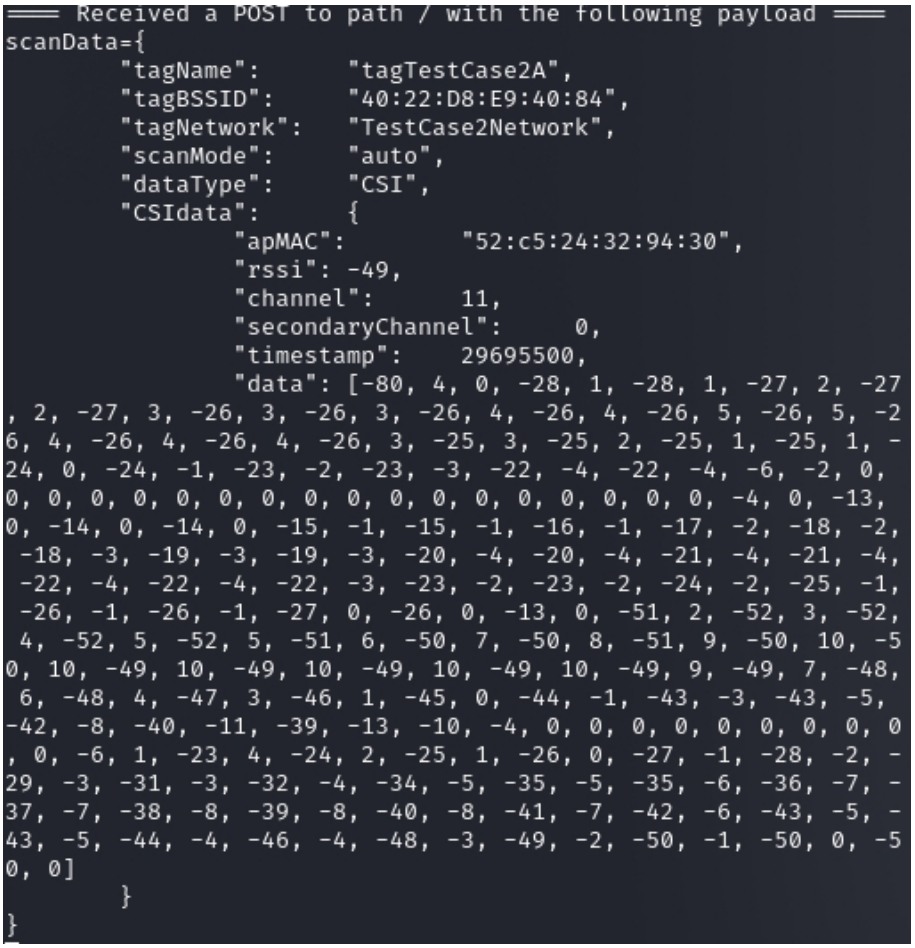



Figure 46: The last log printed by echo server at step 13 of test case 2.

```
E (32847) HTTP_CLIENT: Connection failed, sock < 0
E (32857) HTTP_SERVICE: Failed to open HTTP connection: ESP_ERR_HTTP_CONNECT
E (32857) HTTP_SERVICE: Failed to send FP to server
W (32867) wifi:Haven't to connect to a suitable AP now!
E (32877) SEND_SERVICE: No Wifi connection... Reconnecting
```

Figure 47: The "No Wifi connection... Reconnecting" log printed by the tag.

```
I (51417) wifi:AP's beacon interval = 102400 us, DTIM period = 2
I (52387) esp_netif_handlers: sta ip: 192.168.175.231, mask: 255.255.255.0, g
w: 192.168.175.224
I (52387) WIFI_SERVICE: got ip:192.168.175.231
I (52387) WIFI_SERVICE: Connected to ap SSID: TestCase3Network
I (52397) SEND_SERVICE: Reconnected!
```

Figure 48: The "Reconnected!" log printed by the tag.

| tagName | tagBSSID | tagNetwork | dataType | scanMode | serverTimestamp | bssid | rssid |
|---|---|---|---|---|---|---|---|
| tagTestCase3 | 40:22:D8:E9:40:84 | TestCase3Network | 1 | 0 | 2023-10-14 18:49:46 | 58:FC:20:B7:64:F2 | -85 |
| tagTestCase3 | 40:22:D8:E9:40:84 | TestCase3Network | 1 | 0 | 2023-10-14 18:49:46 | 58:FC:20:B7:64:F0 | -84 |
| tagTestCase3 | 40:22:D8:E9:40:84 | TestCase3Network | 1 | 0 | 2023-10-14 18:49:46 | B0:BB:E5:E1:FB:54 | -81 |

3 rows in set (0.018 sec)

Figure 49: The result of the SQL query to the fingerprints server database.

At step 3, the fingerprints server database was queried using the SQL command shown in listing 47, the result is shown in figure 50. By analizing the "tagName" column, we can see that the database is storing fingerprints that are associated with a tag named "tagDiogo". Thus, we can conclude the tag was reconfigured with the remote configuration server configurations.

Concluding, since the tag met the expected result with success, the tag passed this test case successfully.

| tagName | tagBSSID | tagNetwork | dataType | scanMode | serverTimestamp | bssid | rssid |
|---|---|---|---|---|---|---|---|
| tagDiogo | 40:22:D8:E9:40:84 | rede-2 | 1 | 0 | 2023-10-14 22:00:56 | D4:F9:8D:50:E0:79 | -94 |
| tagDiogo | 40:22:D8:E9:40:84 | rede-2 | 1 | 0 | 2023-10-14 22:00:56 | 58:FC:20:B7:64:F2 | -81 |
| tagDiogo | 40:22:D8:E9:40:84 | rede-2 | 1 | 0 | 2023-10-14 22:00:56 | 58:FC:20:B7:64:F0 | -81 |
| tagDiogo | 40:22:D8:E9:40:84 | rede-2 | 1 | 0 | 2023-10-14 22:00:56 | B0:BB:E5:E1:FB:54 | -76 |
| tagDiogo | 40:22:D8:E9:40:84 | rede-2 | 1 | 0 | 2023-10-14 22:00:56 | FE:6B:10:16:D4:17 | -27 |

5 rows in set (0.027 sec)

Figure 50: The result of the SQL query to the fingerprints server database.

# 7

# Conclusions and Future Work

In this final chapter, we will discuss the results of our work. We will also outline potential avenues for future work.

## 7.1 Conclusions

The main goal of this work was to develop a device capable of integrating a WiFi fingerprinting-based IPS. The device should be capable of not only collecting traditional fingerprinting data, but also CSI data. Furthermore, the device should be able to connect to a WiFi network and be remotely configured. Having good autonomy and being small and lightweight were also goals we set out to achieve.

We believe these goals were achieved with success. A device was constructed that could collect RSSI and CSI fingerprints. It was successfully integrated with an IPS constructed in previous works and can be remotely configured. Furthermore, we added the ability for users to configure the device on site, using a web interface. The device was made with components that are both lightweight and small. Regarding autonomy, during the design phase of this project, we already had power savings in mind. For example, the tag was designed to have sleeping states when sending and collecting fingerprints. In contrast to simply having the tag constantly collecting and sending fingerprints, our sleep states will ensure the tag can be more power-efficient. Moreover, since the sleep timers are user configurable, the tag can be more or less power-efficient based on the user's requirements. An accelerometer module was also added to further improve the autonomy of the tag.

## 7.2 Future Work

Admittedly, the developed solution was made using a prototyping development where the various electronics where assembled on a breadboard. To have the tag in a production environment, the soldering of components as well as a container to house them would be needed. This was not done in this work and is something that could be accomplished in future works. Moreover, there was testing of the solution done with this work. But the tests were done in a controlled environment and were aimed at making

sure the most important features of the developed solution were working correctly. Further testing of the developed solution could be done in more production-like environments.

During the market research phase of this work, we saw that the majority of the solutions currently available are using Bluetooth Direction Finding technologies. Our solution could be improved to be able to also integrate IPSs that use such technologies in the future.

From the very beginning of the development phase of our solution, autonomy was always a big concern. However, we did not evaluate the total autonomy of our solution. This is not simple, because the developed solution has various user configurations that can change the overall autonomy of the tag. A study of optimal configurations and improvements to the developed solution regarding autonomy could also be made in the future.

A web configuration interface provided by the tag to the user was implemented in our solution. The web interface has a very simple user interface that could be improved to be more intuitive and user-friendly. Moreover, the tag will create an AP network with a static name, which is hard to tell apart in a production environment with multiple tags. Thus, our solution could be improved by having a way to differentiate the tags when they are in AP mode.

# Bibliography


[1] *Application Startup Flow - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/startup.html. (Visited on 2023-08-28) (cit. on p. 40).

[2] S. M. Asaad and H. S. Maghdid. "A Comprehensive Review of Indoor/Outdoor Localization Solutions in IoT era: Research Challenges and Future Perspectives". In: *Computer Networks* 212 (2022), p. 109041. issn: 1389-1286. doi: https://doi.org/10.1016/j.comnet.2022.109041. url: https://www.sciencedirect.com/science/article/pii/S1389128622001918 (cit. on pp. 5, 6).

[3] *Asset Management*. url: https://azitek.io/asset-management (visited on 2022-12-24) (cit. on p. 16).

[4] *Bluetooth Indoor Positioning*. 2020-12. url: https://www.u-blox.com/en/technologies/bluetooth-indoor-positioning (visited on 2022-12-17) (cit. on p. 12).

[5] *BlueUp*. https://www.blueupbeacons.com/index.php?page=products_quuppa. (Visited on 2023-03-29) (cit. on p. 12).

[6] *Chipsets | Espressif Systems*. https://www.espressif.com/en/products/socs. (Visited on 2023-10-21) (cit. on p. 23).

[7] X. Dang et al. "A novel passive indoor localization method by fusion CSI amplitude and phase information". In: *Sensors* 19.4 (2019), p. 875 (cit. on p. 10).

[8] *Error Code and Helper Functions - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_err.html. (Visited on 2023-08-28) (cit. on p. 41).

[9] *Error Handling - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/error-handling.html. (Visited on 2023-08-28) (cit. on p. 43).

[10] *Getting started with RTLS*. 2021-10. url: https://www.u-blox.com/en/blogs/insights/rtls-getting-started (visited on 2022-12-17) (cit. on p. 12).

[11] S. M. Hernandez and E. Bulut. "Performing WiFi sensing with off-the-shelf smartphones". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, pp. 1–3 (cit. on p. 10).

[12] *How Quuppa is Proving the New Direction of Bluetooth Location Services*. https://www.bluetooth.com/blog/how-quuppa-is-proving-the-new-direction-of-bluetooth-location-services-and-remote-sensing-and-monitoring/. 2022-03. (Visited on 2023-03-29) (cit. on p. 11).

[13] *IoT Development Framework I Espressif Systems*. https://www.espressif.com/en/products/sdks/esp-idf. (Visited on 2023-08-10) (cit. on p. 25).

[14] B. Jang and H. Kim. "Indoor positioning technologies without offline fingerprinting map: A survey". In: *IEEE Communications Surveys & Tutorials* 21.1 (2018), pp. 508–525 (cit. on pp. 8, 9).

[15] Z.-P. Jiang et al. "Communicating is crowdsourcing: Wi-Fi indoor localization with CSI-based speed estimation". In: *Journal of Computer Science and Technology* 29.4 (2014), pp. 589–604 (cit. on p. 10).

[16] P. Karlsson. *Getting started with Bluetooth for high precision indoor positioning*. url: https://content.u-blox.com/sites/default/files/Indoor-positioning-Getting-started-u-blox-WhitePaper.pdf (visited on 2022-12-17) (cit. on p. 13).

[17] T. Kim Geok et al. "Review of indoor positioning: Radio wave technology". In: *Applied Sciences* 11.1 (2020), p. 279 (cit. on pp. 5, 7).

[18] A. Kulaib et al. "An overview of localization techniques for wireless sensor networks". In: *2011 international conference on innovations in information technology*. IEEE. 2011, pp. 167–172 (cit. on pp. 5, 6).

[19] H. Liu et al. "Survey of Wireless Indoor Positioning Techniques and Systems". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37.6 (2007), pp. 1067–1080. doi: 10.1109/TSMCC.2007.905750 (cit. on pp. 4, 5, 7–9).

[20] W. Liu et al. "Survey on CSI-based indoor positioning systems and recent advances". In: *2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. IEEE. 2019, pp. 1–8 (cit. on pp. 9, 10).

[21] *Non-Volatile Storage Library - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html. (Visited on 2023-08-29) (cit. on p. 45).

[22] *Partition Tables - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/partition-tables.html. (Visited on 2023-08-29) (cit. on p. 46).

[23] G. Pau et al. “Bluetooth 5.1: An Analysis of Direction Finding Capability for High-Precision Location Services”. In: *Sensors* 21.11 (2021). issn: 1424-8220. doi: 10.3390/s21113589. url: https://www.mdpi.com/1424-8220/21/11/3589 (cit. on p. 7).

[24] *Products - Quuppa*. https://www.quuppa.com/overview/. 2023-10. (Visited on 2023-03-29) (cit. on pp. 11, 12).

[25] *QT1-1 Tag User Manual*. https://quuppa.com/product-documentation/manuals/q/QT1-1/topics/QT1-1_user_manual.html. (Visited on 2023-03-29) (cit. on p. 12).

[26] *SPIFFS Filesystem - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html. (Visited on 2023-08-29) (cit. on p. 46).

[27] STANLEY. *AeroScout Asset Management*. Brochure. 2022. url: https://www.stanleyhealthcare.com/sites/stanleyhealthcare.com/files/2022-02/DOC-12-85007-AN_Asset%20Management%20Solution%20Overview%202022.pdf (visited on 2022-12-28) (cit. on p. 15).

[28] STANLEY. *T12 Tag*. Data Sheet. 2022. url: https://www.stanleyhealthcare.com/sites/stanleyhealthcare.com/files/2018-10/AeroScout%20T12%20Asset%20Tag%20Data%20Sheet.pdf (visited on 2022-12-29) (cit. on p. 17).

[29] STANLEY. *T12s Tag*. Data Sheet. 2022. url: https://www.stanleyhealthcare.com/sites/stanleyhealthcare.com/files/2021-02/T12s%20Asset%20Tag%20Data%20Sheet.pdf (visited on 2022-12-29) (cit. on p. 17).

[30] STANLEY. *T2s Tag*. Data Sheet. 2022. url: https://www.stanleyhealthcare.com/sites/stanleyhealthcare.com/files/2018-10/AeroScout%20T2s%20Tag%20Data%20Sheet.pdf (visited on 2022-12-29) (cit. on p. 17).

[31] *T12 Tag | AeroScout RTLS | STANLEY Healthcare Products*. url: https://www.stanleyhealthcare.com/products/aeroscout-t12-tag (visited on 2022-12-29) (cit. on p. 18).

[32] *T12s Tag | AeroScout RTLS | STANLEY Healthcare Products*. url: https://www.stanleyhealthcare.com/products/aeroscout-t12s-asset-tag (visited on 2022-12-29) (cit. on p. 18).

[33] *T2s Tag | AeroScout RTLS | STANLEY Healthcare Products*. url: https://www.stanleyhealthcare.com/products/aeroscout-t2s-tag (visited on 2022-12-29) (cit. on p. 17).

[34] X. Wang et al. “CSI-based fingerprinting for indoor localization: A deep learning approach”. In: *IEEE transactions on vehicular technology* 66.1 (2016), pp. 763–776 (cit. on pp. 9, 10).

[35] *Wi-Fi Driver - ESP32 - ESP-IDF Programming Guide latest documentation*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html. (Visited on 2023-08-29) (cit. on pp. 29, 57).

[36] *xEventGroupCreate() - Create a FreeRTOS event group*. /xEventGroupCreate.html. (Visited on 2023-08-29) (cit. on p. 58).

[37] *xEventGroupWaitBits() - Wait for a bit (flag) or bits in an FreeRTOS event group*. /xEventGroupWaitBits.html. (Visited on 2023-08-29) (cit. on p. 58).

[38] *XPLR-AOA-1 kit*. 2021-06. url: https://www.u-blox.com/en/product/xplr-aoa-1-kit (visited on 2022-12-17) (cit. on p. 13).

[39] *XPLR-AOA-2 kit*. 2021-06. url: https://www.u-blox.com/en/product/xplr-aoa-2-kit (visited on 2022-12-17) (cit. on pp. 13, 14).

[40] *XPLR-AOA-3 kit*. 2022-03. url: https://www.u-blox.com/en/product/xplr-aoa-3-kit (visited on 2022-12-17) (cit. on pp. 14, 15).