AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT INFORMATYKI

Projekt dyplomowy

Indoor positioning system based on Ultra-Wideband technology

System lokalizacji wewnętrznej w oparciu o technologię UWB

Autor: Sebastian Szczepański, Aleksander Wójtowicz
Kierunek studiów: Informatyka
Opiekun pracy: dr hab. inż. Tomasz Szydło, prof. AGH

Kraków, 2023

Contents

1. Project goals and vision

An indoor positioning system (IPS) defines a network of connected devices that are used to track the location of objects in real time. Unlike the more common *Global Navigation Satellite System (GNSS)*, the tracking usually takes place within a building or a contained area and does not rely on satellite technologies, hence being used when GNSS does not provide satisfactory accuracy or coverage. To determine the exact location, fixed reference points, also known as beacons or anchors, are used. Tracked objects are equipped with tags that can be located using various methods. On a physical layer, there is a plethora of technologies, among which are:

Wi-Fi The most common solution is to use wireless access points as anchors and to measure the intensity of *received signal strength indication (RSSI)* [1]. Having obtained the signal strength from the node to several access points, the node's position may be approximated. With the advent of *MIMO* that uses multiple antennas, a new method has emerged — *Angle of Arrival (AoA*).

Bluetooth Bluetooth tracking was previously implemented in a very similar manner to the Wi-Fi RSSI method; however, with Bluetooth 5.1, *Bluetooth Direction Finding* has been released [2]. It enabled two new methods based on the use of an antenna array: *Angle of Arrival (AoA)* and *Angle of Departure (AoD)*.

Ultra-wideband (UWB) UWB is a fairly recent technology for wirelessly transmitting information across a wide bandwidth. It began to appear in consumer electronics circa 2019 [3]. Due to its high bandwidth and low energy usage, it is a perfect fit for battery-powered applications such as digital car keys safe from relay attacks [4], item finders such as Apple AirTags [5], and other short-range applications susceptible to noise but requiring high throughput and reliability.

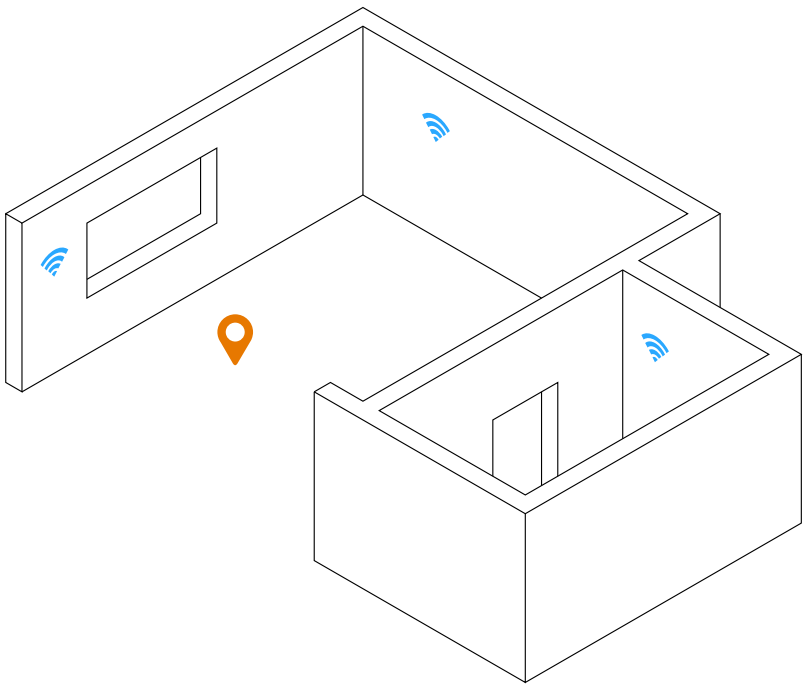

Figure 1: Example of Indoor Positioning System (IPS) using 3 beacons and 1 tag

1.1. IPS applications

Navigation IPS seems to be the perfect fit for navigation systems. Whether it is a store, a museum, or an art gallery, it provides additional guidance to customers and improves their experience. Assuming sufficient accuracy (which would not have been provided by other satellite-based solutions), it could also help the visually impaired.

People tracking Such a system could have been used to track employees and analyze their performance. It could replace traditional RFID-based time tracking solutions.

Room occupancy detection Using IPS as a presence detection system could be useful for smart home automation applications. Based on room occupancy, one could control many appliances, such as the lights, air conditioning, or the heating system. It would also help reduce electricity bills by disabling certain devices in empty rooms.

Collision detection in warehouses The safety of the forklift operator is crucial in a high-risk environment. Introducing an IPS, signal warnings and autobraking features could be implemented to improve the safety of a workplace.

Route optimization Having known frequently taken routes, one can easily create heatmaps and optimize paths. It would also have a substantial impact on commerce, e.g. placing products in commonly visited areas, hence increasing chances of picking them up by customers.

Medical equipment localization Medical equipment is often spread over many rooms and tracking is not trivial. In emergency situations, quick access and location awareness are crucial. Tagging the devices would greatly help to find them and save time that could have been dedicated to patients.

1.2. Concept of the solution

Our goal is to develop an open-source platform for asset tracking applications. We want to create a hardware and protocol-agnostic framework; however, we will focus primarily on providing an implementation for the DWM1001-DEV development kit [6] (nRF52832 SoC with DWM1001C module), using UWB for ranging purposes and Bluetooth Mesh as data transport layer. On top of that, we will create a server application exposing an Application Programming Interface (API) allowing integration with other services and make one as an example. We will also discuss available distance measurement and positioning algorithms, including *Two-Way Ranging (TWR)* and *Time Difference of Arrival (TDoA)*.

1.3. Platform architecture

The platform will consist of four main components:

Library for embedded devices The library will be designed with interoperability in mind. Hardware-specific code will be encapsulated, and a uniform API for high-level ranging operations will be supplied to allow the user to either use our preimplemented integration or to provide a custom implementation for hardware of one's choice. The same should apply to the transport layer protocol, as one might want to use something other than Bluetooth Mesh.

The library will be implemented in the *C* language (since it is the most widely supported language among all embedded platforms) and will be provided as a module for Zephyr RTOS [7], the native development environment for nRF SoCs.

Gateway application To connect a Bluetooth Mesh network with an external web service, a gateway application is required. It will be responsible for relaying messages between the server and individual nodes of the network. This design provides much better flexibility than the naive design, which would require the nodes to communicate with the server directly, using an additional network interface, such as Wi-Fi.

Server application The server is supposed to establish communication with all gateways and expose a unified API for other applications. Among other things, it will be responsible for:

1. aggregating locations of tracked tags and storing historical data,
2. configuring nodes in the networks, such as physical location of the beacons or tags' polling rate,
3. exposing administration tools over the API, such as event viewer and logging, performing firmware upgrades, network setup.

Support for external services By using the underlying API, developers would be able to create services that interconnect directly with the server. They could provide simple integrations with other platforms or complete applications.

We will provide our own service showing tags' locations on a 2D/3D plane. In the case of more advanced systems, developers may want to create a proprietary, more sophisticated solution that provides the required functionality.

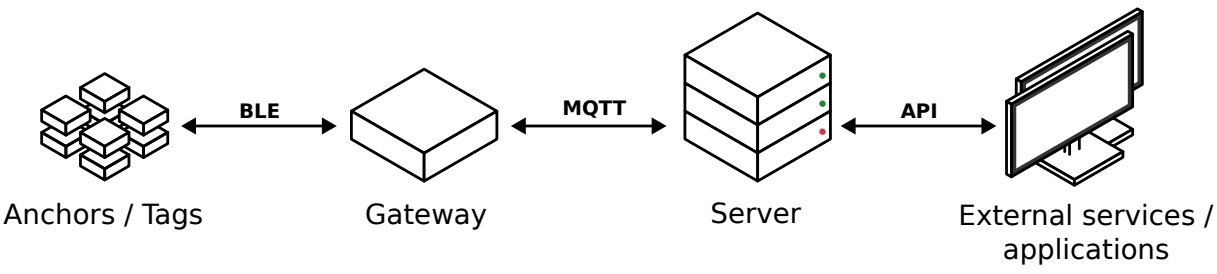

Figure 2: Platform architecture

1.4. Competitive solutions

There are already many companies offering Indoor Positioning Systems, both as generic systems supposed to be integrated with customer applications, or as solutions tailored to customer requirements. Unfortunately, the crucial parts of those solutions are paid and close-sourced, significantly increasing the entry threshold, especially for individuals and small businesses. Our implementation will remain free and open-source, allowing one to build such systems only at a cost of the required hardware.

Popular IPS solutions include:

Pozyx Pozyx offers both the software and hardware required to build an IPS [8]. Pozyx's solutions include custom hardware incorporating the DW1000 IC, the exact same chip our project uses, for ranging purposes. Two kinds of development kits are offered — creator and enterprise grade. Creator-grade anchors can be connected to a PC for further data processing. Anchor-grade anchors are connected to a special gateway over Ethernet, which further communicates with other services. They provide management software, available both on-premise and in the cloud, and analytics software.

Eliko Eliko offers UWB-based hardware with dedicated management software just like Pozyx, although their offering focuses on the configurability of system parameters, so different levels of responsiveness and precision can be achieved [9]. Eliko also claims that their Active-Passive Two-Way Ranging (AP-TWR) algorithm provides many improvements over the standard TWR.

Ubisense Ubisense offers two main products. SmartSpace is a software platform that focuses on processing location data aggregated from many sources, not just IPSs, to provide customizable monitoring and event detection [10]. Ubisense's Real-Time Location System (RTLS), named Dimension4 [11], provides a rather unique localization service with the use of UWB, utilizing both TDoA and AoA measurements.

Figure 3: Pozyx's Value Assessment Validation Package

1.5. Risk analysis

to There are many factors and risks that can prevent a software project from being successful. In addition to common problems that may occur during the development phase, we may also encounter many problems related to the maintenance of the project in the future. We have attempted to identify and assess them in the list below:

Hardware obsolescence At first glance, it seems that our platform is heavily dependent on the hardware used, as in our case, the core of our system consists of UWB-based modules performing ranging between the nodes. We are addressing that problem by developing a generic library, which will not be concerned with the used ranging method. If the production of used modules ceases or new technology that supersedes UWB emerges the market, our platform will most likely be ready for a shift. This future-proofs all other parts of the platform — the code which handles communication between the nodes, the gateway application, and the server application which processes all of the location data. Hardware-bound code, while significant, is a relatively small part of the system.

Software obsolescence Obsolescence of libraries and other dependencies used in our project is beyond our control, but to minimize the risks, we have carefully chosen the technologies that seem to be the most popular now and pose a low risk of becoming out-of-date quickly. Additionally, the applications running on tags and anchors will be based on Zephyr RTOS, which is not only an operating system but also a set of hardware-independent libraries for connectivity and peripherals. This makes our library easily portable to a wide array of microcontrollers and development kits.

Poor project maintenance Developers are not particularly interested in choosing solutions that are not actively maintained. They expect library maintainers to actively respond to questions, bug reports, and feature proposals. This means that the success of our library greatly depends on our active support of the library itself and its users, at least in the initial phase. The fact that our platform will be open-source will simplify the adoption and usage of it, as if a developer encounters some issue or a bug in our platform, it will be much easier for them to narrow it down or even fix it on their own.

2. Functional scope

In this chapter, we discuss the target audience, requirements, and key functionality of our platform through the use of user stories. The requirements include both functional requirements, which define the desired features and functionality of the platform, and non-functional requirements, which ensure that the platform is reliable, scalable, and easy to use. The user stories provide examples of how the platform will be used and the features it will provide to meet the needs of the target audience.

2.1. Target audience

Makers, hobbyists One of our main target audiences are electronics hobbyists and the so-called *makers*. Currently available RTLS solutions are either very expensive and not open-sourced, or even not available to individuals at all. Our platform provides all of the required software to build anything which requires positioning functionality — the software for embedded devices that makes up for the network, and server software which processes events from these devices and provides access to gathered data with APIs.

The software running on the tags is out-of-the-box compatible with Decawave DWM1001-DEV units, which are available on the market for as little as $15 each. This significantly lowers the entry barrier to this kind of technology. In addition, thanks to open-source implementation highly dependent on the modular Zephyr SDK, this software can be easily ported to work with other development kits.

Hobbyists and novice programmers will also benefit from the possibility of inspecting the inner workings of the positioning software to learn, easily debug their solutions, and extend the functionality of the platform we are providing.

Small enterprises High cost of RTLS systems also decreases their availability to small enterprises that have limited funds and operate on a much smaller scale. Our platform will enable them to rapidly prototype and then implement a system that requires asset positioning with low hardware costs, no software fees, and a well-thought-out software architecture that separates hardware-specific code from difficult parts of data processing. Possibly, engineers at these companies, thanks to open-source license, will also be able to extend and adjust the platform to their needs or share their work by contributing to it.

2.2. Functional requirements

Real-time data delivery Tracked tag positions should be delivered to end listeners as quickly as possible. Our platform calculates the positions on tags themselves, which significantly decreases the effort the server has to make; the low-level internal measurements are not being sent to the server (which means less bandwidth used), instead, the server receives already computed 3D positions (which means less processing power used).

The end applications talking to the server can subscribe to Server-Sent Events (SSE), through which they will receive the most up-to-date positioning data instead of repeatedly polling the server, which may be ineffective.

Asset management Assets (gateways, anchors, tags) must be configurable to some extent by the server. It must be possible to control sets of allowed gateways and paired anchors/tags within the server application. Most important parameters, like physical positions of anchors or tags' position reporting rate, must be configurable without physically accessing these devices.

Multiple network handling The server application must be able to handle multiple separate positioning networks. This is required by applications where a single centralized server app is used to manage many RTLS deployments, possibly located at different premises and overseen by different people.

API-oriented approach and modularity The server application must provide programmatic access to all of the application's functionality. Although a dedicated graphical interface is useful, especially during the initial phases of solution development, it should also be possible to manage assets and retrieve positioning data from other applications programmatically. A well-integrated application shouldn't require using the pre-implemented administration panel whatsoever.

Cross-compatibility with other modules and microcontrollers Applications running on gateways, anchors, and tags must be easily portable to other development kits and end devices. We achieved it by implementing them in the *C* language and using Zephyr RTOS' high-level APIs. Besides the code that handles the DW1000 UWB peripheral (which we consider to be one of the most popular ones), there are no ties in the software with specific hardware, making these applications easily portable to other development kits and end devices supported by Zephyr. The required peripherals are Bluetooth Low Energy controller and in case of a gateway, IP networking stack.

2.3. Non-functional requirements

Scalability The system should be designed with scalability in mind. To reach our goals, we have decided to create a system that could be easily scaled horizontally using *Docker Swarm*, or other solutions such as *Kubernetes*. By spinning up multiple containers, we could easily spread incoming requests across a set of different servers, hence improving the average response time and overall responsiveness of the system. Using advanced geographic load-balancing, we could further reduce round-trip time by redistributing application traffic across data centers in different locations.

As opposed to running more servers, we also reduce the total overhead of the network. By calculating positions directly on tags, we can limit the amount of packets sent across the network, hence significantly lowering the used bandwidth per node. Having relieved some additional strain on edge devices (gateways), we can process more traffic while continuously ensuring high performance.

User-friendliness Emphasis should be placed on user-friendliness. This means that APIs should be as simple as possible and well-documented. Software with poor documentation discourages programmers from using it, especially if it is not yet well established on the market. This also applies to parts of the software that end users do not interact with directly — well-documented architecture will aid more experienced users to adjust the inner parts of the platform to suit their needs or even contribute to it.

Ideally, a user guide explaining how to configure the server application and the required hardware should also be provided.

Security To improve the security of subnets, we should implement a dynamic access list on top of the MQTT broker. Each gateway is separated and has no impact on each other. It also helps to monitor traffic and spot the faulty gateway.

Communication over Bluetooth Mesh should also be secure. The gateway should serve the purpose of Mesh Provisioner, which is used to securely pair anchors and tags, which is a process completely independent of the server-side credential configuration.

Open-sourceness We believe that open-sourcing a system targeted at individuals and small enterprises brings many benefits that cannot be overlooked. Our application, being modular at its core, might benefit significantly from the contributions of independent developers. It is much easier to adopt open-source software, as using it does not require getting involved in any business relations with the party offering the software. It is also much easier to use, integrate, and work with open source software, as if documentation of some feature is not clear enough or some behavior unexpected by the user is encountered, it is possible to inspect the source of the application.

2.4. User stories

As we deliver a SDK, not a complete software solution, our *user stories* [12] focus mainly on the development and maintenance of the project, not the usage of the end product. The importance of these user stories was estimated using the MoSCoW method [13].

No.	Story	Priority
1.	As a project manager, I need to rapidly implement a solution with real-time localization capabilities.	Must
2.	As a software developer, I need a software platform that collects location data.	Must
3.	As a software developer, I don't want to implement positioning software for embedded devices from the ground up.	Must
4.	As a software developer, I need to be able to integrate my application with the positioning platform using one of commonly used API techniques.	Must
5.	As a software developer, I need a detailed documentation so I have no issues integrating the SDK into existing applications.	Must
6.	As a system maintainer, I need to reliably monitor and control my assets from a central server application.	Must
7.	As a system maintainer, I need to be able to remotely configure basic runtime parameters of my assets.	Must
8.	As a software developer, I should be able to easily port the software for embedded devices to other targets.	Should
9.	As a system maintainer, I should be able to provision and connect tracked assets to the network without recompiling firmware running on them.	Should
10.	As a software developer, I should be able to quickly set up an example demonstrating the platform I'm about to use.	Could
11.	As a system maintainer, I should be able to access logs remotely so I can respond to any system issues.	Could
12.	As a system maintainer, I should be able to control the location reporting frequency in order to optimize storage usage by the application.	Could
13.	As a project manager, I would want to have a scalable solution that I can deploy across many facilities.	Would
14.	As a project manager, I would want to be able to deploy the positioning network using a different transport layer than Bluetooth Mesh.	Would
15.	As a project manager, I would want to be able to use different peripherals for positioning purposes such as Bluetooth or Wi-Fi.	Would

Figure 4: User stories

3. Selected realization aspects

This chapter covers design principles, theory of operation and selected implementation details of the software we developed. We discuss several interesting and non-trivial problems — positioning methods in UWB-based RTLS systems, UWB ranging methods, architecture of connectivity of our solution, and structure of backend and embedded applications. Additionally, we introduce the dashboard — an example of a third-party web application utilizing server API we designed.

3.1. Design principles

The project was designed with several principles in mind:

Codebase management First of all, we wanted to avoid using monorepos. Splitting the project by its core features allowed us to create a modular, elastic, and easily extensible solution. By doing so, potential users might opt to replace specific parts with their custom implementation to suit their needs. For ease of navigation and management, we put all the repositories under a common GitHub's organization.

System architecture We spent a significant amount of time on architecture design. Being a fundamental part of our project, we focused on its interoperability, good common practices, and its functionality while being as simple as possible. Throughout the whole development, we strongly relied on *KISS (Keep it simple, stupid!)* principle. By avoiding unnecessary complexity, we were able to come up with a readable, easy-to-follow, and not over-engineered code, which also has a huge impact on its maintainability.

Selection of technologies We are aware that the chosen libraries have a vast impact on development and might determine the project's success. We carefully selected programming languages, frameworks, and libraries according to their relevance, market share, frequency of updates, and predicted support time. Using libraries that could reach *EoL (End of Life)* before the project was even complete was an enormous deal breaker for us.

Docs-as-Code With the docs in the same place as a source code, we can make changes via the standard code review process. It makes documenting code simpler, less demanding, and easier to maintain. Being an open-source project, exhaustive documentation might be a deciding factor for choosing our platform over a competition.

Quality assurance The creation of quality code requires a good test coverage that ensures its reliability. Whenever we refactor some functionality, we do not need to spend extra time manually testing all the edge cases, which also significantly reduces human error rate.

3.2. Principle of operation

3.2.1. Positioning

There are multiple known positioning methods that leverage the physical characteristics of ultra-wideband. The most common methods are Two-Way Ranging (*TWR*) combined with True-Range Multilateration and Time Difference of Arrival (*TDoA*).

Two-Way Ranging with True-Range Multilateration In the TWR method, true distances to multiple base stations (usually at least 4; also called anchors) are measured. These distances are then used to infer the 3D position of the tag. This method of positioning is fairly easy to set up as it does not require any synchronization between anchor devices. On the other hand, the amount of messages exchanged with each positioning attempt is proportional to the number of tag-anchor distance measurements used. This may pose problems with scalability of the system, as one anchor can participate in only one distance measurement attempt at a time.

Time Difference of Arrival In TDoA, *ping* messages are repeatedly broadcasted by tags to all anchors in the positioning network. The differences between the arrival times of these messages at different anchors are then used to infer the position of a tag. This method is much more scalable than TWR, as each positioning attempt requires only a single broadcast message. However, to precisely measure the differences between the arrival times of that message, all anchors must have their clocks synchronized. Various methods can be used, but most of the time anchors are physically connected to a common clock source [14]. This adds much more complexity to the system. In addition, the position must be calculated on a central server, rather than directly on tags. If the tags want to be aware of their own position, it must be sent back from the server.

As we want our system to be easy to set up and use, we have chosen Two-Way Ranging as the primary method of positioning.

3.2.2. Theory behind Ultra-wideband ranging

The system uses ultra-wideband technology due to its unique physical characteristics that make it possible to reliably measure the time of flight (*ToF*) of messages exchanged between the devices. Having precise ToF measurements, distances to stationary anchors can be calculated. These distances are later used in the positioning algorithm.

Compared to other radio technologies, such as Bluetooth or Wi-Fi, radio pulses occupy a very wide frequency band (about 500 MHz), which makes their rising edge very steep [15]. Additionally, these pulses are exceptionally short (around 2 nanoseconds long) [16]. These properties make it possible to precisely register the time of first, non-reflected signal that arrives to the receiver.

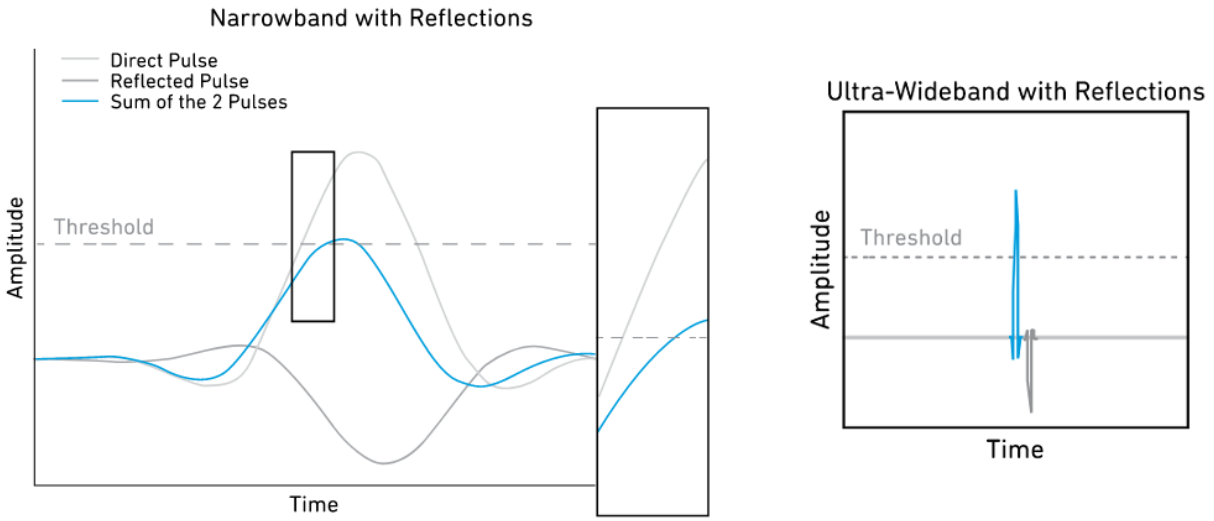

Figure 5: Comparison of signal amplitude over time between narrowband and wideband radios [17]

Two-Way Ranging is a method that uses message transmission and reception times to calculate flight time. There are a few variations, each with its own advantages and disadvantages. The following two are the most popular:

SS-TWR Single-sided TWR is the simplest method. In this scheme, two messages are sent — initial message from tag to anchor, and response message from anchor to tag. The response message includes the measurement of how long it took the anchor to process the initial message and send the response (T_{reply}). Additionally, the tag measures the time between transmission of the initial message and reception of the anchor response (T_{round}).

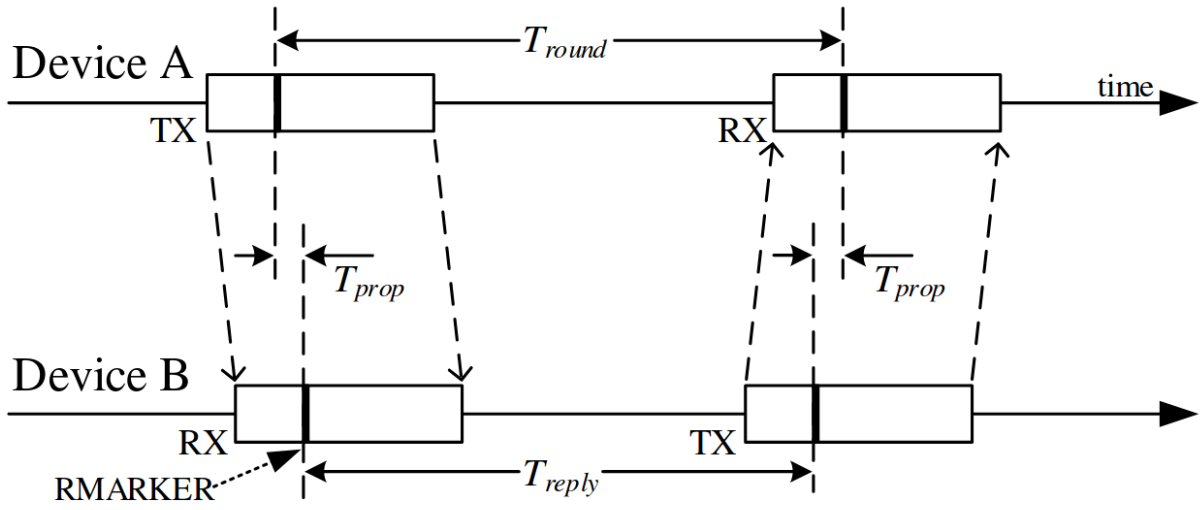

Figure 6: Single-sided Two-Way Ranging [18]

The formula for time of flight is:

$$T_{ToF} = \frac{T_{round} - T_{reply}}{2} \tag{1}$$

As this method requires only two messages to be exchanged, measurements take a very short time. On the other hand, this method is very susceptible to errors due to the drift of both the clock and the frequency between two participating devices [19]. The error can be minimized by incorporating the so-called *carrier integrator value* (register `DRX_CAR_INT`), used to estimate the clock drift rate for which the receiver must compensate [18].

By including this correction, the final formula is as follows:

$$T_{ToF} = \frac{T_{round} - T_{reply} * (1 + offset)}{2} \tag{2}$$

ADS-TWR Asymmetric double-sided TWR is a method in which three messages are exchanged, as in Figure 7.

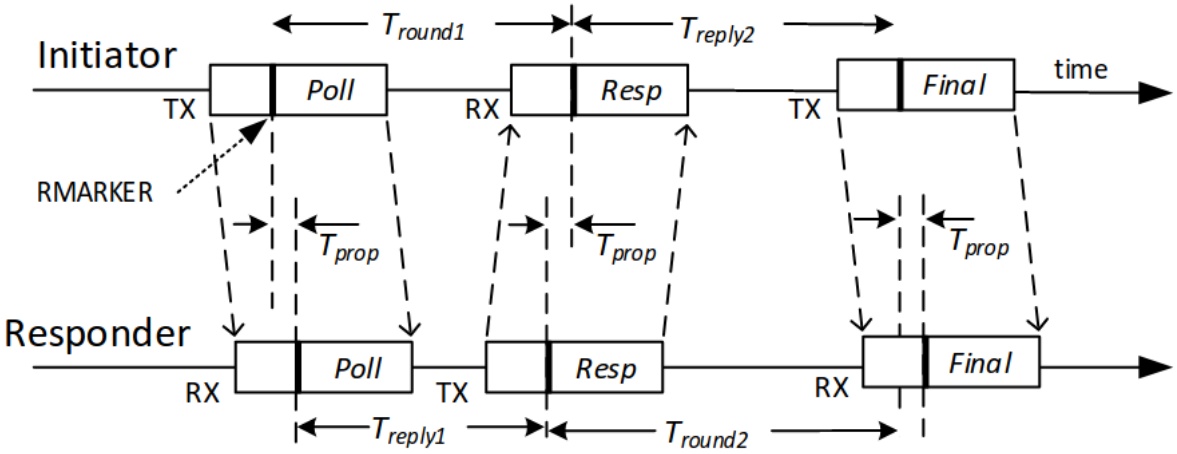

Figure 7: Asymmetric double-sided Two-Way Ranging [19]

Final message carries the following timestamps measured by the tag:

- TX timestamp of the *Poll* message (T_{PollTX})
- RX timestamp of the *Resp* message (T_{RespRX})
- TX timestamp of the *Final* message ($T_{FinalTX}$)

The following timestamps are measured locally by the anchor, which makes the final calculations:

- RX timestamp of the *Poll* message (T_{PollRX})
- TX timestamp of the *Resp* message (T_{RespTX})
- RX timestamp of the *Final* message ($T_{FinalRX}$)

Performing measurements symmetrically by both devices allows to construct a system of expressions that greatly minimizes the measurement error, explained in detail in [20].

$$T_{round1} = T_{RespRX} - T_{PollTX} \tag{3}$$
$$T_{round2} = T_{FinalRX} - T_{RespTX} \tag{4}$$
$$T_{reply1} = T_{FinalTX} - T_{RespRX} \tag{5}$$
$$T_{reply2} = T_{RespTX} - T_{PollRX} \tag{6}$$
$$T_{prop} = \frac{T_{round1} * T_{round2} - T_{reply1} * T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}} \tag{7}$$

The drawback of this approach is that the time of flight measurement is calculated on the anchor, which means that in our case, where the position is calculated by tags, an additional message must be sent with the final measurement. This means that 4 messages in total are exchanged, which is time-consuming and greatly impacts the scalability of the system.

There is also a symmetric variant (SDS-TWR) of this mechanism, which assumes that reply delays are equal in all message exchanges. This method is not widely used as some messages can be processed faster than others; the requirement of equal reply delays artificially extends the time that the whole measurement process takes.

The measured ToF can then be used to estimate the distance between a tag and an anchor. Assuming that all devices are in a line-of-sight environment (that is, the Fresnel zone between two devices is clear of obstacles [16]), the distance can be calculated using the following formula:

$$d = c * T_{ToF} \tag{8}$$

where d is the distance between two radios and c is the speed of light.

3.2.3. Gateway and Bluetooth Mesh for connectivity across devices in a single network

Our system requires all devices to be able to communicate with each other for multiple reasons:

- The server needs to be aware of tags' positions. As they are calculated on tags, these locations must somehow be transmitted to the server.
- Tags and anchors must be configurable and manageable from a centralized application. Nodes of the system must be able to synchronize their settings on boot-up, and configuration updates must be sent from the server on every change. This means that bidirectional communication between the server and the nodes must be implemented.

The naive solution would be to provide IP network access to all devices used in an RTLS and make them communicate directly with the server using a protocol like MQTT, but this approach has several problems:

- IP networking requires relatively expensive hardware. Not only would all devices have to be equipped with some kind of cellular modem or Wi-Fi module, but the deployment of such RTLS would also require setting up an appropriate networking infrastructure. Such infrastructure would include a number of Wi-Fi Access Points or small cellular base stations (*Picocells*) to ensure that the entire area where an RTLS is deployed is covered

by network connectivity. Alternatively, 802.15.4-based protocols, like Thread, could be used, although they are not widely supported yet, and they increase complexity of set-up process (e.g. Thread requires a border router).

- RTLS systems can scale to a large number of anchors and tags. If all of these devices were to connect to the backend server directly (to synchronize node configuration or report location data), it could quickly be overwhelmed by all incoming requests.

- Wireless IP networking usually consumes large amounts of energy due to the nature of wireless networking peripherals being either oriented for high bandwidth (Wi-Fi) or high range (cellular connectivity). Additionally, the multi-layered nature of Internet connectivity incurs additional processing, which also consumes energy.

For these reasons, we chose a different approach — tags and anchors are connected to a gateway device (one per RTLS network) and communicate using Bluetooth Mesh. The gateway offloads some of the data processing that would normally have to be done by the backend server and translates messages sent by nodes to MQTT messages later sent to the backend. It also serves the purpose of a cache — the RTLS configuration is stored by the gateway and later broadcasted to nodes within the network.

Bluetooth Mesh has several properties that led us to choose it for our system:

- It is built on top of Bluetooth Low Energy (*BLE*). An interesting property of the BLE stack is that it is split into two layers: host (upper layer) and controller (lower layer). These can run on either the same or separate hardware interconnected by some interface. Since Bluetooth Mesh is a purely software-based host layer protocol, all applications in our system can use the exact same implementation and APIs. Thanks to that, we were also able to provide a shared implementation of custom Mesh models and surrounding logic to make the devices communicate.

- It does not require setting up an additional networking infrastructure. Bluetooth Mesh establishes a mesh network between all nodes configured as relays. In our case, we can configure all anchors as relays since they are stationary and externally powered. Mesh networks do not require any kind of routing setup; Bluetooth Mesh implements a so-called "managed flood" model in which relays send all messages further, as long as those messages are seen by the relay for the first time. This enables us to cover a large area; theoretically, a Bluetooth Mesh network can span two orders of magnitude larger area than point-to-point Bluetooth Low Energy connections can — the limiting factor is message time-to-live (*TTL*).

- Bluetooth Low Energy hardware is very common and low-cost nowadays. It is supported by both the SoC embedded in the development kits used (nRF52832) and our personal computers that we used to develop this platform on.

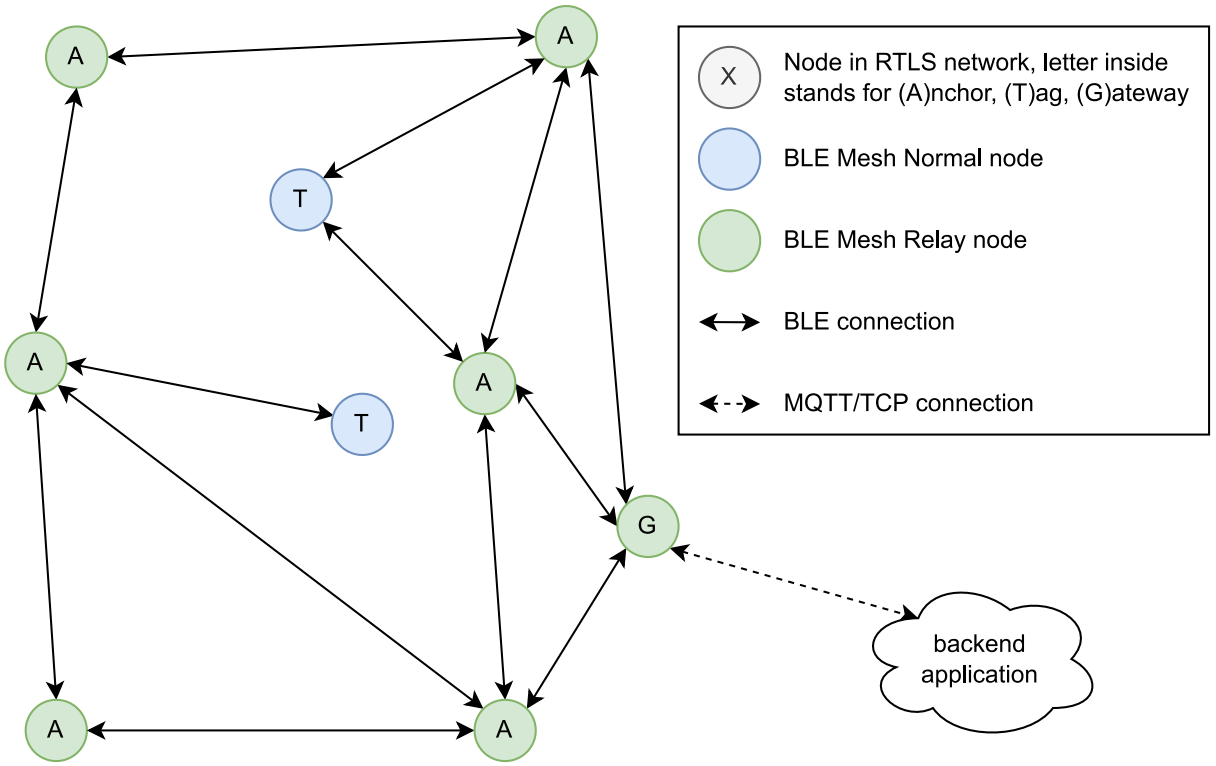

Figure 8: Diagram of Bluetooth Mesh connections in an exemplary RTLS network. Notice that all stationary nodes (gateway and anchors) also serve as a relay in the BLE Mesh network.

3.2.4. MQTT as a lightweight messaging protocol between gateways and a server

MQTT (Message Queue Telemetry Transport) is a lightweight, publish-subscribe network protocol that is designed for communication in low-bandwidth, high-latency, or unreliable networks. It is often used in Internet of Things (IoT) and Machine-to-Machine (M2M) communication, and has become one of the most popular protocols in this field.

In the MQTT architecture, there are three main components:

Broker The central server that receives all published messages and sends them to the subscribed clients.

Publisher A device that sends messages to the broker.

Subscriber A device that receives messages from the broker.

The publish-subscribe pattern used by MQTT allows devices to communicate with each other without the need for direct connections. This makes it easy to scale and add new devices to the network. MQTT is also designed to be lightweight, making it well-suited for resource-constrained devices.

In addition to its use in IoT and M2M communication, MQTT is also used in a wide range of other applications, including home automation, industrial control, and transportation. It is an open protocol, with libraries available for many programming languages, making it easy to implement and use.

The combination of these characteristics makes MQTT an ideal protocol for transmitting messages between gateways and the central server.

3.2.5. Role of the backend application

The backend application, also referred to as a server, plays a vital role in our system. Its primary responsibilities include:

Handling messages from edge devices The server processes messages sent from edge devices via the MQTT broker. This may involve updating the assets' positions, changing the network configuration, or adding new devices.

Gathering and retrieval of tag historical data Historical data could be important for future analysis. It could be utilized by external services to generate statistical data, such as heatmaps, and enhance business decision-making.

Managing network configuration The server maintains the configuration of all connected positioning networks. The configuration data is transmitted to edge devices to ensure that the state of the network is correct and up to date.

Support for external services The server exposes the REST API to external services. It analyzes the incoming request, performs any necessary calculations, and generates a response to send back.

The technology stack, architecture, and included modules are described in detail in **3.4**.

3.2.6. Process sequence diagrams

Adding new networks (gateways) New networks are being added upon user request. The server handles the incoming request, validates it against defined DTO, stores the changes in the database, updates the MQTT broker's access list, and returns the credentials back to the user.

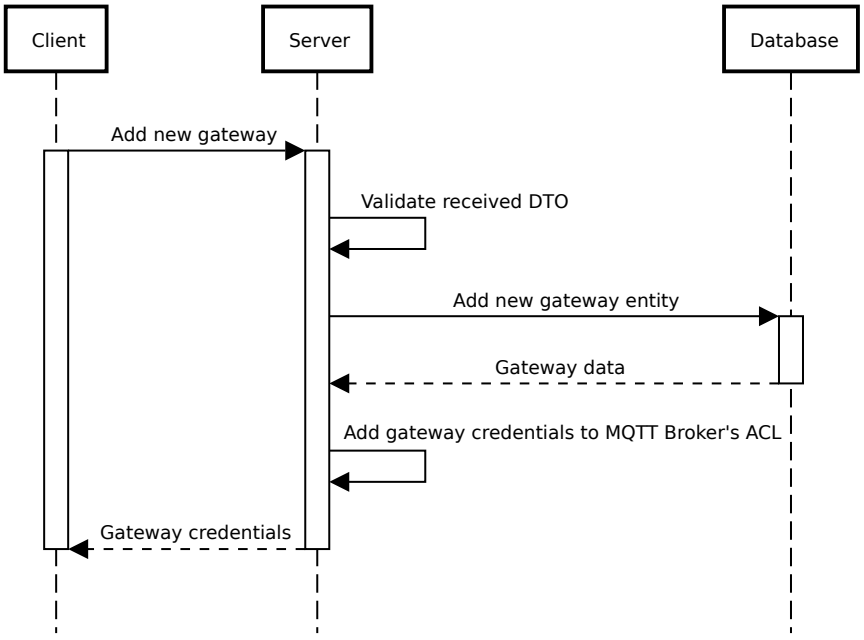

Figure 9: Adding new networks (gateways) sequence diagram

Updating tag configuration The server is responsible for maintaining the proper network configuration. Upon receiving a request to change it, the server updates specific entities in the database and propagates the changes to the appropriate gateways via MQTT messages. If any of these updates have an impact on the tags, the gateway further propagates them over Bluetooth Mesh, as shown in Figure 10.

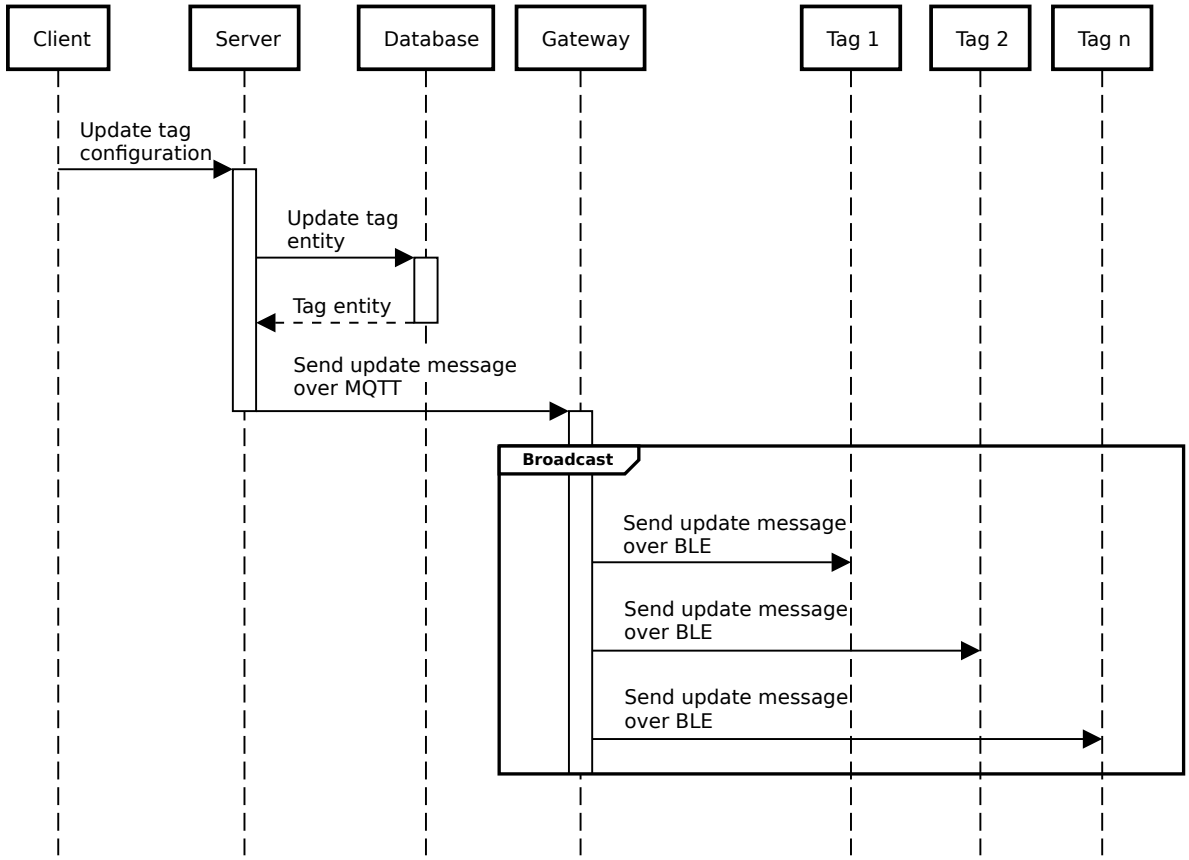

Figure 10: Updating tag configuration sequence diagram

Processing MQTT messages Figure 11 depicts how the server processes incoming MQTT messages, each having its own subscriber. Upon receiving a matching message, the subscriber passes it through middlewares, such as guards, interceptors, pipes, and filters. After validating and transforming the message, the subscriber executes the actual business-related code.

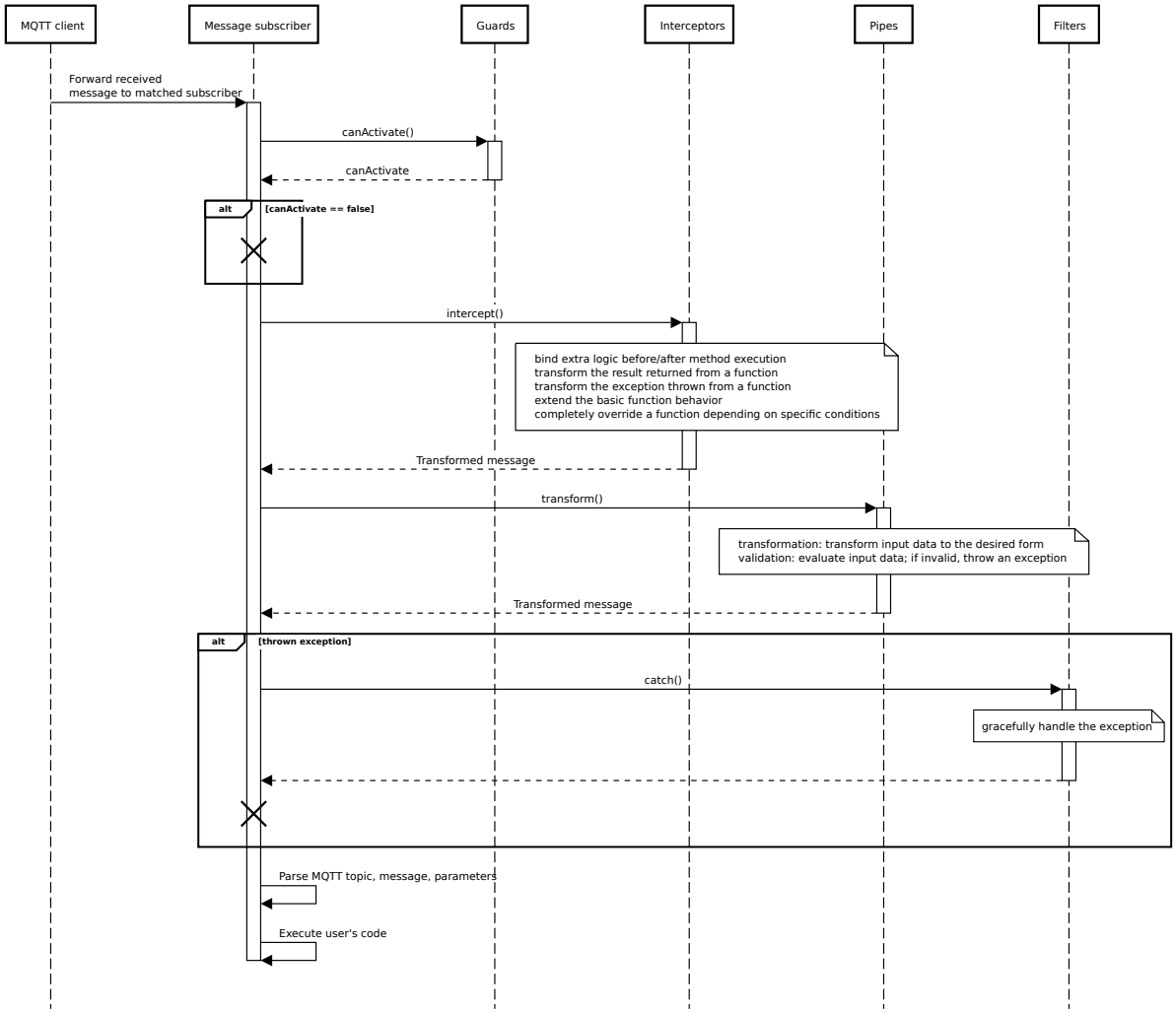

Figure 11: Processing MQTT messages sequence diagram

Receiving assets' positions Figure 12 shows how users obtain the tags' positions. After fetching initial data using a regular HTTP request, the client subscribes to the server's event stream. Once the tag sends updated position data, the server emits a new event reflecting the change.

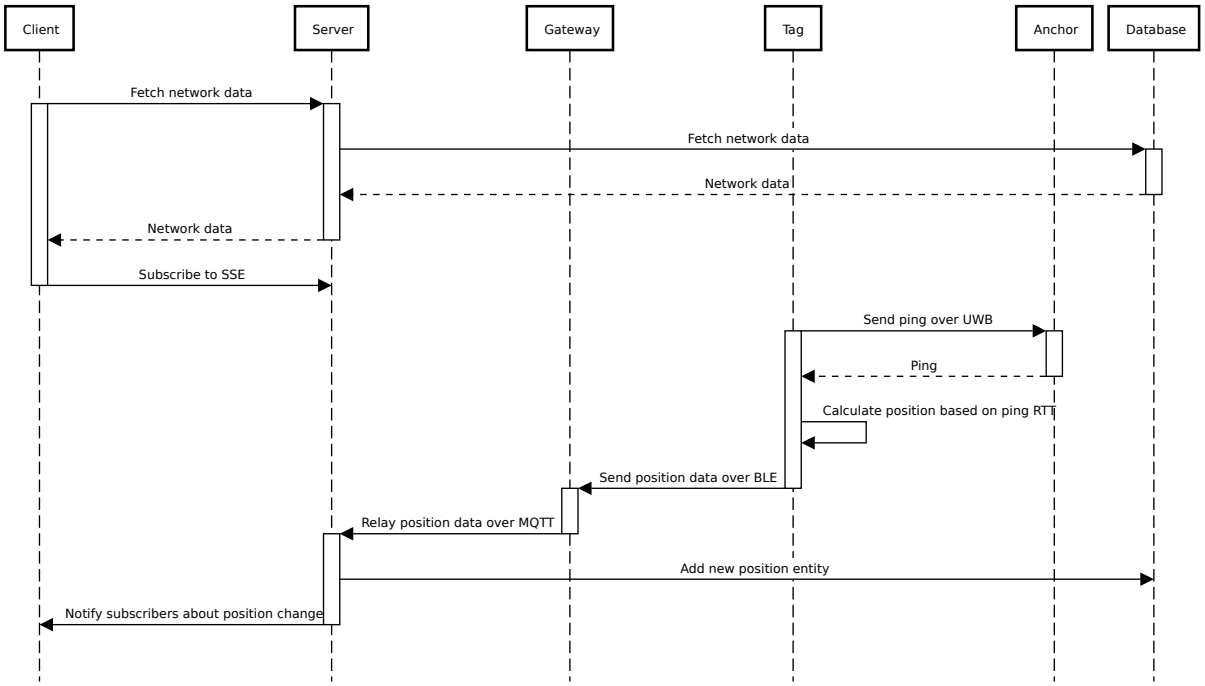

Figure 12: Receiving assets' positions sequence diagram

3.3. Backend application

The backend application is a crucial component of the platform, serving as the interface for external services and coordinating the various functions of the platform. It exposes a REST API, allowing for programmatic access to its functionality via HTTP requests. In addition, it communicates with gateways using the MQTT protocol.

3.3.1. Technology stack

After evaluating possible options, weighing pros and cons, we chose the following technology stack:

Node.js We are using Node.js as a back-end JavaScript runtime environment. It is 00=performant, easy to scale, good for rapid development, and has great community support.

TypeScript JavaScript, being a dynamically typed language, might be unsafe to work with at times. Having a strongly typed programming language such as TypeScript, we were allowed to develop more readable, robust, and less error-prone code.

Nest Nest is a Node.js framework for web applications which combines elements of Object Oriented, Functional, and Functional Reactive Programming. It provides a level of abstraction above common frameworks, such as Express and FastAPI, making them easily swappable. Being heavily inspired by Angular, it solves many architectural issues. It provides a base for our backend application.

PostgreSQL PostgreSQL is an open-source object-relational database system that runs on all major operating systems. It has great documentation and community support. Since we do not have strong opinions or requirements on relational databases, we chose the currently most popular option. Being supported by most ORMs, we are also enabling potential users of our platform to directly access the data available in the database using different technologies.

Mosquitto Mosquitto is an open-source message broker. It implements the MQTT protocol, which is very popular in IoT devices, as it is lightweight and simple to use. We are using Mosquitto to route messages between edge devices (gateways) and servers. Having a separate "room" for every gateway, it works like a direct tunnel offering high security.

Server-sent events Many applications, in order to get the most recent updates in real-time, use the long-polling mechanism or WebSockets. With server-sent events, we push new data in an event-based manner. As opposed to WebSockets, it is unidirectional, easier to implement (no handshakes, no connection upgrade), and has less overhead, which makes it perfect for our use case.

3.3.2. Architecture

Our backend service heavily relies on *Docker*, which provides consistent and isolated environments. Each service runs in its own container, independent of the host system. It ensures the same functionality across different platforms and architectures. To easily spin up multiple containers based on our images, we use *Docker Compose*. We currently use the following images:

postgres An object-relational database management system. It stores our persistent data, such as network configuration, tags' positions, and connected devices.

mosquitto An open source implementation of a server of the MQTT protocol. It is used for communication between the server and the gateway devices.

hyperrtls-backend Actual implementation of the server. It is the most crucial part of our system — it processes incoming messages, notifies subscribers about any changes, and exposes data to external services over the REST API.

traefik Reverse proxy, load balancer, and HTTP cache. It handles incoming traffic and forwards it to the appropriate containers.

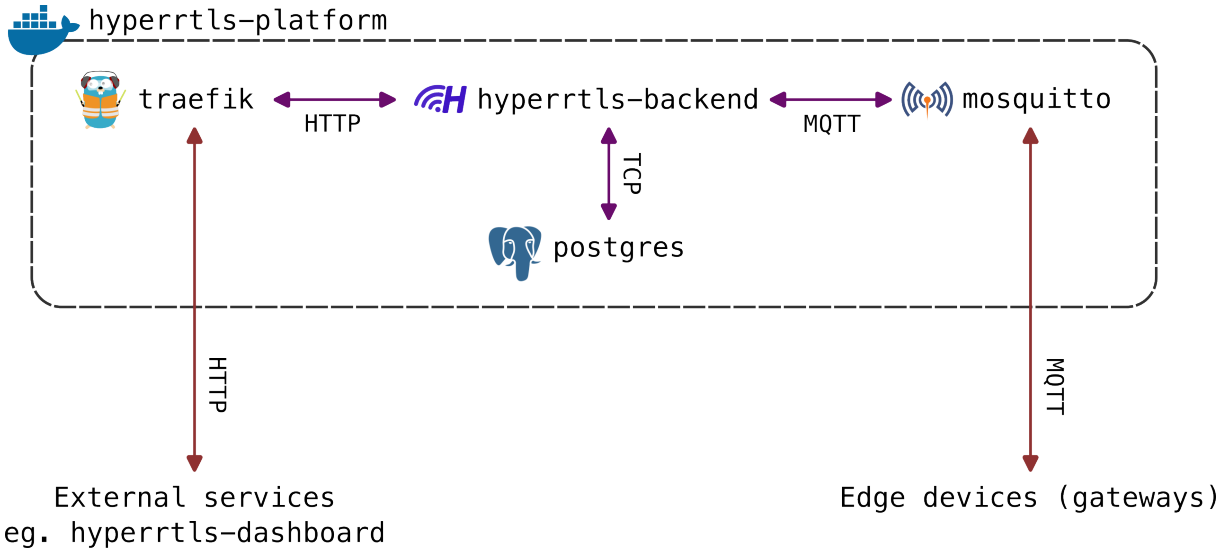

Figure 13: Backend application architecture

3.3.3. Handling network traffic — reverse proxy and SSL

With the aid of Traefik, we created a router that redirects network traffic to appropriate containers, while limiting the visibility of other services that should not be exposed to the outside network. It also manages SSL certificates, which provide secure, encrypted communication.

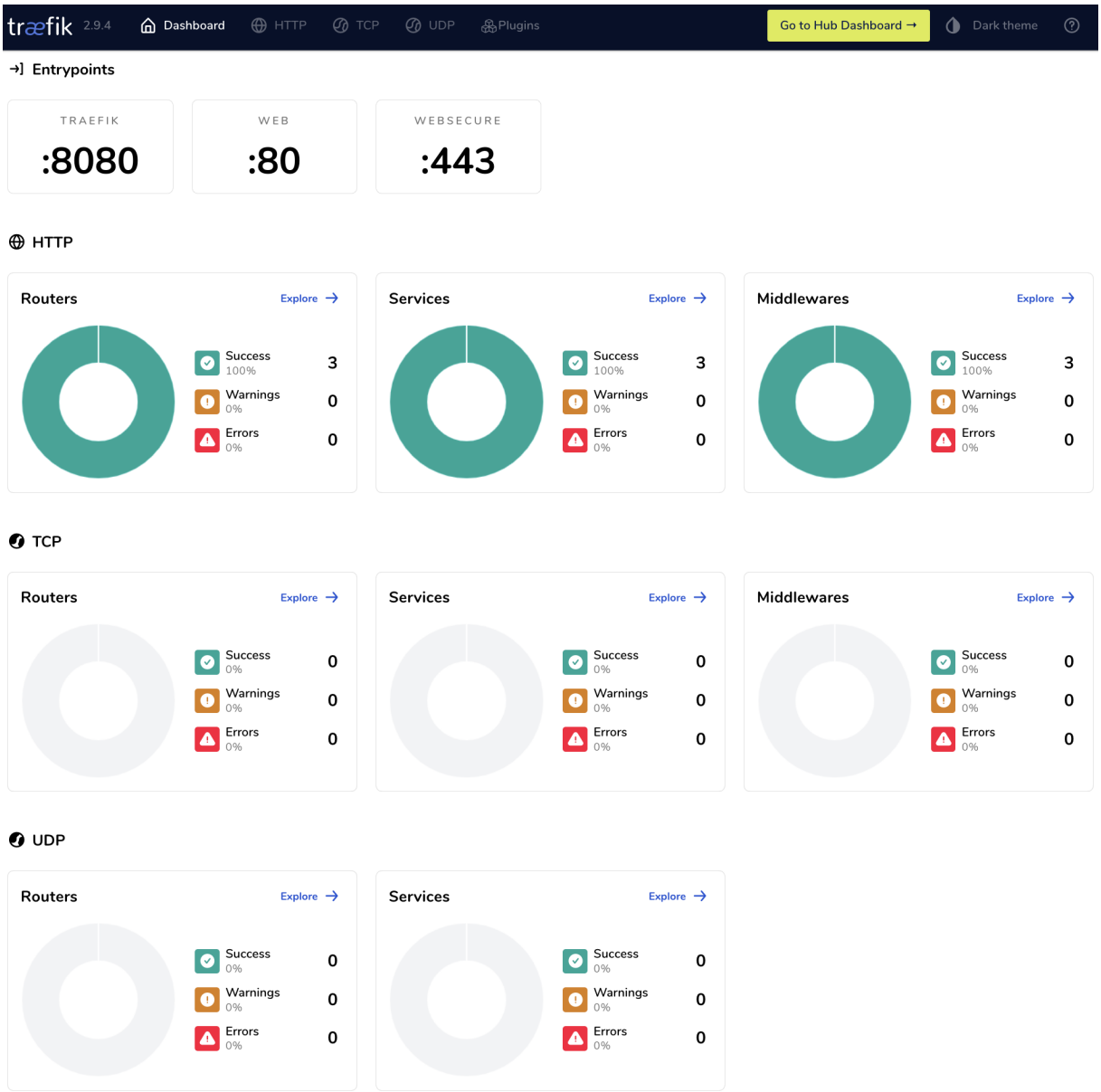

Figure 14: Traefik dashboard

3.3.4. Relational Database

We used PostgreSQL as a RDBMS (Relational Database Management System). We designed four different entities: *gateway*, *anchor*, *tag*, and *tag-position*. Storing the tag positions in a separate table allowed us to model the *1:N* relation, effectively obtaining the tag positions history (multiple timestamped positions per tag). *mikro_orm_migrations* is an internal MikroORM's table that stores the migration history. Migrations help update the database schema, which involves adding, changing, and removing tables or columns, changing constraints, types, and much more. They represent incremental changes to data structures, which are very often reversible.

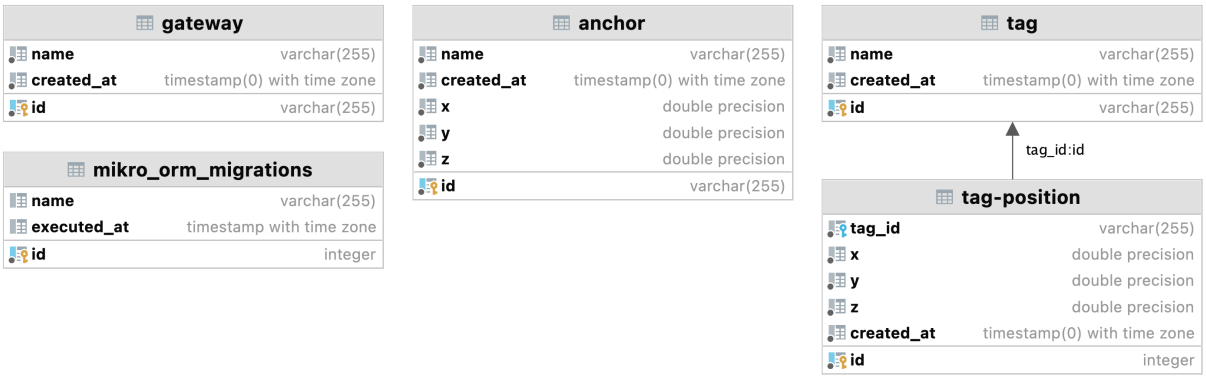

Figure 15: Database diagram

	id	tag_id	x	y	z	created_at ▾ 1
1	6713	tag_1	2.164	0.093	4.524	2022-12-08 14:54:17 +00:00
2	6710	tag_1	2.458	0.676	4.4	2022-12-08 14:54:16 +00:00
3	6708	tag_1	2.116	0.036	4.6	2022-12-08 14:54:16 +00:00
4	6711	tag_1	2.073	-0.211	4.64	2022-12-08 14:54:16 +00:00
5	6712	tag_1	2.041	0.115	4.597	2022-12-08 14:54:16 +00:00
6	6709	tag_1	2.314	0.985	4.246	2022-12-08 14:54:16 +00:00
7	6704	tag_1	2.364	1.38	4.49	2022-12-08 14:54:15 +00:00
8	6707	tag_1	2.678	1.447	4.662	2022-12-08 14:54:15 +00:00
9	6706	tag_1	2.366	1.006	4.446	2022-12-08 14:54:15 +00:00
10	6705	tag_1	2.628	1.961	4.561	2022-12-08 14:54:15 +00:00
11	6703	tag_1	2.249	0.243	4.616	2022-12-08 14:54:15 +00:00
12	6701	tag_1	1.931	0.303	4.781	2022-12-08 14:54:14 +00:00
13	6702	tag_1	2.161	0.436	4.676	2022-12-08 14:54:14 +00:00
14	6698	tag_1	2.099	0.516	4.854	2022-12-08 14:54:14 +00:00
15	6700	tag_1	2.094	0.726	4.774	2022-12-08 14:54:14 +00:00
16	6699	tag_1	2.135	0.773	4.734	2022-12-08 14:54:14 +00:00
17	6697	tag_1	2.071	0.711	4.738	2022-12-08 14:54:13 +00:00
18	6695	tag_1	2.082	0.674	4.807	2022-12-08 14:54:13 +00:00
19	6693	tag_1	2.107	0.906	4.684	2022-12-08 14:54:13 +00:00

Figure 16: Position history as records in the database

3.3.5. Backend's modules

The backend application has been built with modularity in mind. Every core feature has been extracted as a separate module, allowing easier modification, and also enforcing the single-responsibility principle. Throughout development, we provided the most modular solution possible. Each module has its tasks, dependencies and might include other modules as well.

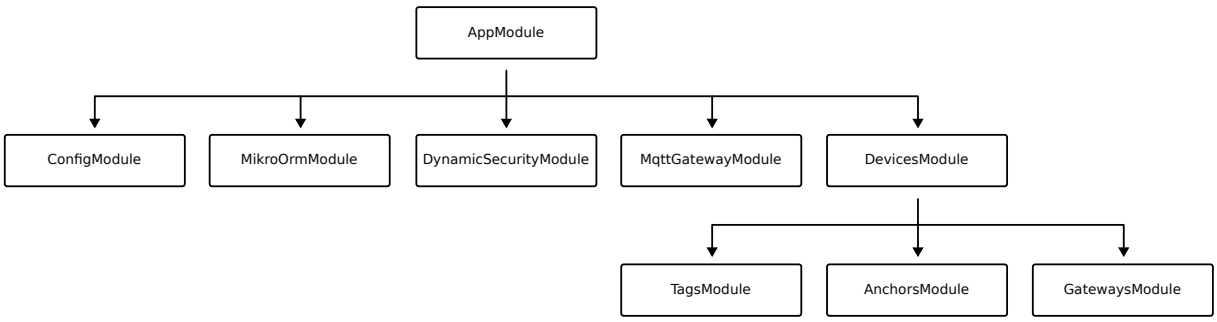

Figure 17: Backend's modules diagram

AppModule AppModule is an entry point of the entire application. It initializes other top-level modules, establishes a connection to the Mosquitto broker, database, loads the configuration from environment files, and also handles logging.

```
@Module({
  imports: [
    ConfigModule.forRoot({...}),
    MikroOrmModule.forRootAsync({
      useFactory: async (
        configService: ConfigService<EnvironmentVariables, true>,
      ) => ({...}),
      inject: [ConfigService],
    }),
    DynamicSecurityModule.registerAsync({
      useFactory: async (
        configService: ConfigService<EnvironmentVariables, true>,
      ) => ({...}),
      inject: [ConfigService],
    }),
    MqttGatewayModule.registerAsync({
      useFactory: async (
        configService: ConfigService<EnvironmentVariables, true>,
      ) => ({...}),
      inject: [ConfigService],
    }),
    DevicesModule,
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

Listing 1: AppModule definition

ConfigModule ConfigModule is responsible for loading the configuration data from environmental files or variables. It is set to report any missing or invalid configuration options during run-time, which ensures a valid application state and operation. It uses *class-transformer* and *class-validator* packages under the hood.

```
DynamicSecurityModule.registerAsync({
  useFactory: async (
    configService: ConfigService<EnvironmentVariables, true>,
  ) => ({
    brokerUrl: configService.get('DYNSEC_BROKER_URL'),
    clientConfig: {
      username: configService.get('DYNSEC_BROKER_USERNAME'),
      password: configService.get('DYNSEC_BROKER_PASSWORD'),
    },
  }),
  inject: [ConfigService],
})
```

Listing 2: Usage of ConfigService in a module register method

MikroOrmModule MikroORM is a TypeScript ORM for Node.js. It is based on Data Mapper, Unit of Work, and Identity Map patterns. It natively supports MongoDB, MySQL, MariaDB, PostgreSQL, and SQLite databases. MikroORM also supports the repository design pattern. We created a separate repository for each and every entity, resulting in easier-to-read and more manageable code.

```
public async update(
    id: string,
    data: Pick<EntityData<TagEntity>, 'name'>,
) {
    const tagEntityReference = this.tagRepository.getReference(id);
    wrap(tagEntityReference).assign(data);
    await this.tagRepository.flush();
}
```

Listing 3: Simple update operation in tags' controller

DevicesModule DevicesModule is aggregating other device-related modules, such as *TagsModule*, *AnchorsModule*, and *GatewaysModule*. Each of those modules has its own controllers, services, and dependencies. Its usage is optional; however, it may contain shared functionality and provide a higher-level API.

```
@Module({
  imports: [TagsModule, AnchorsModule, GatewaysModule],
  controllers: [],
  providers: [],
  exports: [],
})
export class DevicesModule {}
```

Listing 4: DevicesModule definition

DynamicSecurityModule DynamicSecurityModule is an abstraction layer over Mosquitto's *Dynamic Security Plugin*. It provides role-based authentication and access control features such as access control lists. All the rules can be updated whilst the broker is running. It uses a dedicated topic-based API. Our implementation is fully asynchronous and type-safe.

```
public async onApplicationBootstrap() {
    this.logger.log('Initializing module...');

    await this.dynsecController.init();

    const [roles, groups, clients] = await Promise.all([
        this.dynsecService.listRoles({}),
        this.dynsecService.listGroups({}),
        this.dynsecService.listClients({}),
    ]);

    this.logger.log(
        `Detected ${roles.totalCount} roles, ${groups.totalCount}
           groups, and ${clients.totalCount} clients in total`,
    );

    this.logger.log('Successfully initialized module');
}
```

Listing 5: DynamicSecurityModule init method

MqttGatewayModule MqttGatewayModule is used to send and receive messages from the MQTT broker. It uses metaprogramming to simplify usage. For easier message routing, we have developed a custom topic pattern matcher and other utility functions. We have also provided a custom validation pipe specifically for MQTT messages.

```
export class TagPositionPayload {
  @IsNumber({}, { each: true })
  position: [number, number, number];
}

@MqttGateway('tags')
@UsePipes(new MqttValidationPipe())
export class TagsMqttController {
  constructor(private readonly tagsService: TagsService) {}

  @MqttSubscribe('+tagId/position')
  async onPositionMessage(
    @Topic('tagId') tagId: string,
    @Payload() payload: TagPositionPayload,
  ) {
    await this.tagsService.onPositionMessage(tagId, payload.position)
       ;
  }
}
```

Listing 6: Example usage of MqttGatewayModule

TagsModule / AnchorsModule / GatewaysModule These modules are used to interact with devices in the network. Each device type has its own entity (representation in the database), separate event bus, controllers, and services. Devices define which HTTP requests and MQTT messages they handle.

```
export abstract class BaseDeviceEntity {
  constructor(id: string, name: string) {
    this.id = id;
    this.name = name;
  }

  @PrimaryKey({ autoincrement: false })
  @Check({ expression: 'LENGTH(id) <= 16' })
  id!: string;

  @Property()
  name!: string;

  @Property()
  createdAt?: Date = new Date();
}
```

Listing 7: Base device entity schema

```
export type PositionEventData = {
  position: [number, number, number];
};

@Injectable()
export class TagsEventBus extends BaseDevicesEventBus {
  constructor() {
    super('tag');
  }

  public emit(
    eventType: 'position',
    deviceId: string,
    eventData: PositionEventData,
  ): void;
  public emit(
    eventType: string,
    deviceId: string,
    eventData: Record<string, unknown>,
  ): void {
    super.emit(eventType, deviceId, eventData);
  }
}
```

Listing 8: Tags' event bus

3.4. Embedded software

Software for embedded devices used in the system (gateways, anchors, and tags) was developed using Zephyr Project, an open source embedded software development platform consisting of:

- a real-time operating system (*RTOS*) that provides means for developing multithreaded applications,
- a hardware abstraction library (*HAL*), which is a set of common APIs used to control hardware available on different kinds of microcontrollers that is normally controlled by accessing vendor-specific registers,
- a configuration system based on Kconfig and devicetree, inherited from the Linux kernel project,
- a distribution of the ports of the *GCC* compiler that supports the latest C and C++ standards for all supported hardware architecture,
- a set of additional tooling for libraries and project management, such as `west` command-line tool, which is used for changing Zephyr version, building projects, flashing applications to target hardware, and debugging.

As many parts of the applications for these three devices are common, the code is organized into a single source tree. We leverage:

- devicetree, to configure the code which accesses specific hardware available on target devices,
- Kconfig, to appropriately configure the RTOS, additional libraries, and the application,
- CMake, to selectively compile only the sources required by a given application.

3.4.1. Solution architecture

Figure 18 depicts the part of the platform that runs on embedded devices. The components of our `hyperrtls-embedded` subproject that are sourced from external sources are colored blue.

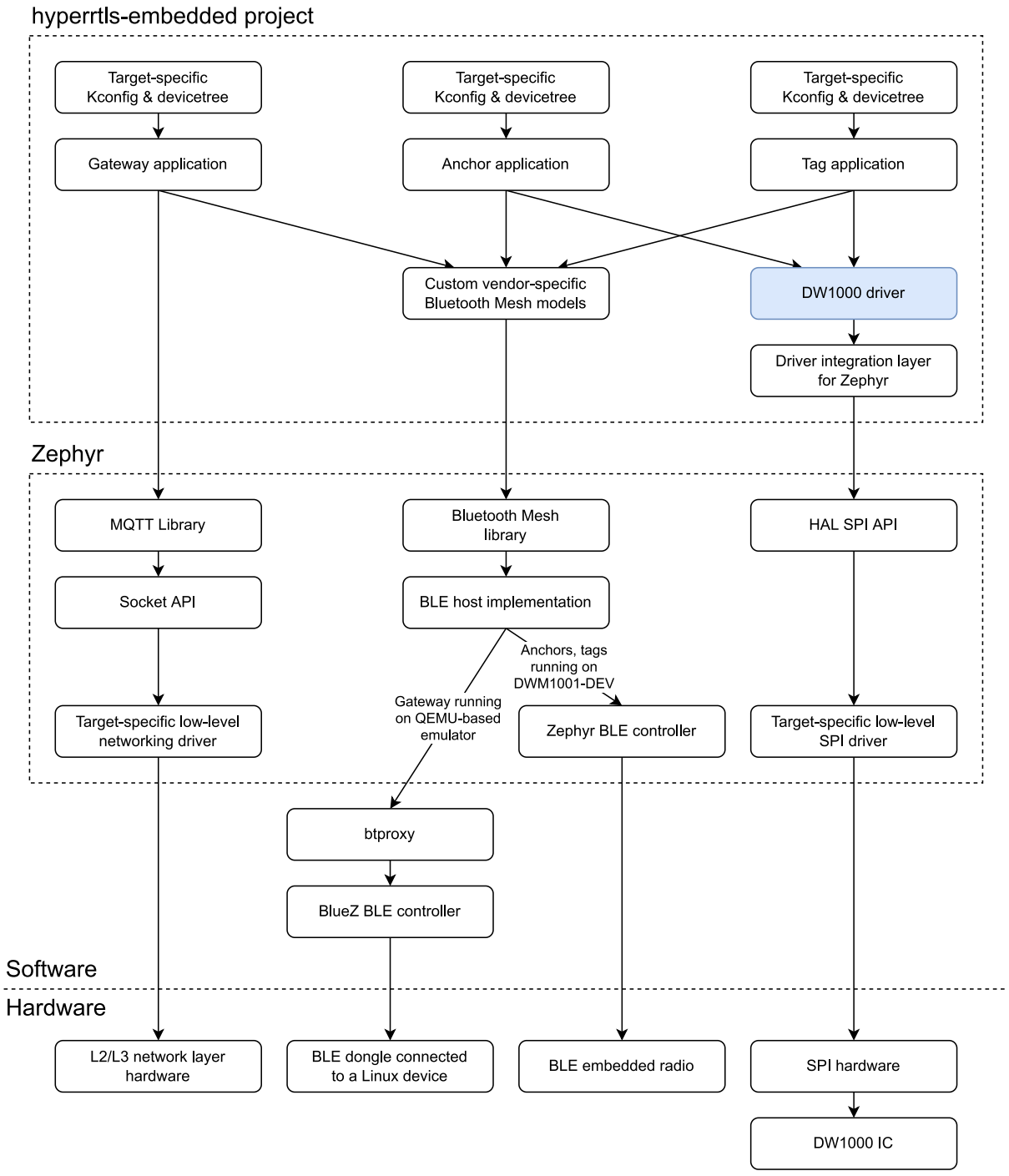

Figure 18: Embedded solution architecture

The applications shown in Figure 19 have the following roles:

Tag application The tag application runs on devices that are being localized. Tags frequently measure their distance to anchors using the DWM1001C module, calculate their own position, and send it to the gateway over Bluetooth Mesh.

Anchor application Anchors are taking part in the distance measurement process; they answer tags' ranging messages. Like tags, they are also connected over Bluetooth Mesh to the gateway to enable remote configuration.

Gateway application The gateway application is an intermediary in communication between nodes (tags, anchors) and the backend application. It communicates with it using the MQTT protocol. Gateway application requires both Bluetooth Mesh and IP stacks to be available; during the development process we were running the gateway application in a QEMU emulator with access to the host's Internet connectivity and the Bluetooth controller. The possibility of running Zephyr applications in an emulator with access to real hardware is an exceptionally useful feature of Zephyr [21].

3.4.2. Selected implementation details

DW1000 driver Decawave (now Qorvo) does not provide a Zephyr-compatible driver for the core component of our system, which is the DW1000 UWB IC. Implementing a driver from the ground up for such a complex peripheral was not considered. There are open-source implementations that are compatible with platforms such as STM32, nRF5 SDK, or Apache Mynewt. Thankfully, all calls to platform-specific functions were abstracted away into separate source files and headers, making it quite easy to port the library, which we did.

We started by copying the version of the driver compatible with STM32 HAL and later reimplemented platform-specific functions, including those for:

- SPI bus configuration,
- SPI read and write operations,
- interrupt locking,
- module rebooting.

For example, the original implementation of the `writetospi()` function targeting STM32 HAL looks as in Listing 9 (original code reformatted and comments removed).

```
int writetospi(uint16 headerLength,
               const  uint8 *headerBuffer,
               uint32 bodyLength,
               const  uint8 *bodyBuffer) {
    decaIrqStatus_t stat = decamutexon();

    while (HAL_SPI_GetState(&hspi1) != HAL_SPI_STATE_READY);

    HAL_GPIO_WritePin(DW_NSS_GPIO_Port, DW_NSS_Pin, GPIO_PIN_RESET);
    HAL_SPI_Transmit(&hspi1, (uint8_t *)&headerBuffer[0],
       headerLength, HAL_MAX_DELAY);
    HAL_SPI_Transmit(&hspi1, (uint8_t *)&bodyBuffer[0], bodyLength,
       HAL_MAX_DELAY);
    HAL_GPIO_WritePin(DW_NSS_GPIO_Port, DW_NSS_Pin, GPIO_PIN_SET);

    decamutexoff(stat);

    return 0;
}
```

Listing 9: writetospi() DW1000 driver implementation for STM32 HAL

The same function, ported to the Zephyr API, looks as in Listing 10.

```
int writetospi(uint16 headerLength,
               const  uint8 *headerBuffer,
               uint32 bodyLength,
               const  uint8 *bodyBuffer) {
    decaIrqStatus_t irq_stat = decamutexon();

    int res = spi_write_dt(&dw1000_spi_bus, &(struct spi_buf_set) {
        .buffers = (struct spi_buf[]) {
            {
                .buf = (uint8_t *)headerBuffer,
                .len = headerLength
            },
            {
                .buf = (uint8_t *)bodyBuffer,
                .len = bodyLength
            }
        },
        .count = 2
    });

    decamutexoff(irq_stat);

    return res;
}
```

Listing 10: writetospi() DW1000 driver implementation for Zephyr HAL

DW1000 ranging and timing issues DW1000 IC uses an internal, 64-GHz, 48-bit-wide clock to register reception timestamps and precisely schedule the time at which the messages are sent. That clock's frequency also determines a lower bound of distance measurement precision — one clock cycle takes about 15.63 ps; during that time light covers a distance of around 4.7 mm.

For practical reasons, instead of trying to measure the actual processing time and trying to write it into the transmitted messages (which itself is a process that incurs a delay that is difficult to accurately measure), the so-called *delayed transmission mode* is used to artificially extend the processing time to some predefined value, which we can write into the message.

These delays are not trivial to determine. They should be as low as possible to minimize errors due to possible clock and frequency difference drift. On the other hand, if these delays are too low, the MCU cannot process and schedule a response message on time. In such cases, DW1000 reports an error and cancels the transmission.

Finding the correct transmission delays was a lengthy process. The original samples shipped together with the DW1000 IC driver were prepared for a STM32F105RC MCU with a 72 MHz CPU and an 18 MHz SPI bus. nRF52832 MCU in development kits we used has a 64 MHz CPU and 8 MHz maximum SPI frequency, a considerably lower value. Additionally, we are using a completely different compiler and RTOS from the one used in Decawave's examples, so even if the core logic of the distance measurement scheme is implemented similarly, runtime performance might differ.

Transmission delays in example applications are usually in the 100-300 μs range. In our implementation, we had to increase these delays **by a factor of about 3**.

Multi-anchor Ultra-wideband ranging in practice Initially, we have considered implementing both SS-TWR and ADS-TWR ranging methods, as the latter, at least theoretically, can provide more accurate measurements, although sacrificing scalability. In practice, when we were testing the examples provided in the DW1000 driver, the SS-TWR method with clock offset correction gave almost as precise measurements as ADS-TWR. As the latter method is much worse in terms of performance (with SS-TWR we manage to reliably take about 150 measurements per second, with ADS-TWR only about 40), we decided to go forward with SS-TWR only.

Problems arose when we tried to measure the distance to multiple anchors. As radio communication, due to its nature, is broadcast, all anchors were receiving messages from the tag trying to find the distance to just one of them. This situation had to be handled by anchors through verification of addressing fields in the message and possible reinitialization of internal anchor logic. This, in turn, dramatically affected the performance of the system; between every single measurement, we needed to add a 5 ms delay to give anchors enough time to reject the frame and start listening again.

This problem was later solved by implementing *frame filtering* — DW1000 can be ordered to automatically ignore received frames it is not a recipient of, without any additional action taken by the MCU. That way we can quickly perform multiple distance measurements, which are later averaged to compensate for the noise.

Multilateration algorithm We have successfully implemented a positioning algorithm that runs directly on the tags. We have considered numerous multilateration methods, including:

1. equation system-based method — method proposed in [22] solves the problem of finding an intersection of spheres, centered on the position of the anchors, with the radii of the distance measured to these anchors.

2. linear regression-based method [22], where a system similar to that of the first method is constructed, although any number of measurements (at least 4) can be used to improve the result.

3. non-linear optimization-based method, where the minimized objective function is defined as the sum of squares of differences between the measured distance from tag to anchor, and the distance between the solution and the anchor [23].

Method no. 1 has been ruled out because it turned out, that due to noisy measurements, on some occasions the equation system becomes inconsistent. This situation is especially prominent in cases where the measured distances have a negative error. From a geometric point of view, the spheres we are trying to find an intersection point of are too small to actually intersect. On the other hand, if all measurements have some positive error (as in our case, since factory-programmed antenna delay correction is usually a little low), then the positioning error is different for every choice of 3 distance measurements used to solve the system with two solutions. This problem could be potentially solved by solving such an equation system for every triplet of anchors and averaging all solutions, but then again, for some triplets, there could be no solution at all.

Precision of method no. 3 was very promising [24], although the Gauss-Newton optimization algorithm is iterative and could require a large number of iterations to converge to some acceptable solution. Additionally, the objective function is not convex, thus the algorithm might find only a local minimum of the objective function.

We chose method no. 2, as it was fairly easy to implement and has good precision. It also always produces some kind of result, even though the distance measurements are imperfect. Furthermore, if the error of all measurements is similar, then in the center of the anchor system, the errors cancel out.

For matrix operations (multiplication and inverse), we used the Zephyr Scientific Library (*zscilib*) [25], which is a scientific computing library for applications based on Zephyr OS.

Bluetooth Mesh Implementation-wise, Bluetooth Mesh is quite straightforward to use with the APIs Zephyr provides. As there are no standard Bluetooth Mesh models that we could use to implement communication in our system, we had to define our own. A declaration of a model of a gateway in our system is shown in Listing 11.

```
struct hrtls_model_gw_location {
    float x;
    float y;
    float z;
} __packed;

typedef void hrlts_model_gw_loc_push_handler_t(uint16_t sender_addr,
   const struct hrtls_model_gw_location *location);

struct hrtls_model_gw_handlers {
    hrlts_model_gw_loc_push_handler_t *push;
};

#define HRTLS_MODEL_GW_ID 0x0001

#define HRTLS_MODEL_GW_NODE_REG_OPCODE BT_MESH_MODEL_OP_3(0x01,
   HRTLS_COMPANY_ID)

#define HRTLS_MODEL_GW_LOC_PUSH_OPCODE BT_MESH_MODEL_OP_3(0x02,
   HRTLS_COMPANY_ID)
#define HRTLS_MODEL_GW_LOC_PUSH_LEN sizeof(struct
   hrtls_model_gw_location)

#define HRTLS_MODEL_GW(handlers) \
  BT_MESH_MODEL_VND_CB( \
    HRTLS_COMPANY_ID, \
    HRTLS_MODEL_GW_ID, \
    hrtls_model_gw_ops, \
    NULL, \
    handlers, \
    NULL)

extern const struct bt_mesh_model_op hrtls_model_gw_ops[];
```

Listing 11: Bluetooth Mesh gateway model declaration

The declaration above is later used to compose a Bluetooth Mesh element, which is a composition of models. The elements compose a node in Bluetooth Mesh, as shown in Listing 12.

```
static struct bt_mesh_model vnd_models[] = {
    HRTLS_MODEL_GW(&gw_handlers)
};

static struct bt_mesh_model sig_models[] = {
    BT_MESH_MODEL_CFG_SRV,
    BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub)
};

static struct bt_mesh_elem elements[] = {
    BT_MESH_ELEM(0, sig_models, vnd_models)
};

static const struct bt_mesh_comp comp = {
    .cid = HRTLS_COMPANY_ID,
    .elem = elements,
    .elem_count = ARRAY_SIZE(elements)
};

void bt_ready(int err) {
    // [...]
    err = bt_mesh_init(&prov, &comp);
    // [...]
}
```

Listing 12: Bluetooth Mesh gateway model registration

The last step is to provide handler methods that will process messages received by the model. Implemention of such a handler is shown in Listing 13.

```
static int handle_message_loc_push(struct bt_mesh_model *model,
                                   struct bt_mesh_msg_ctx *ctx,
                                   struct net_buf_simple *buf) {
    struct hrtls_model_gw_handlers *handlers = model->user_data;
    struct hrtls_model_gw_location location;

    if (buf->len != sizeof(location)) {
        LOG_ERR("Loc push incoming buffer has unexpected length: %"
           PRIu16, buf->len);
        return -1;
    }

    LOG_INF("Loc push message received");

    memcpy(&location, net_buf_simple_pull_mem(buf, sizeof(location)),
        sizeof(location));
    if (handlers->push) {
        handlers->push(ctx->addr, &location); // application-defined
           callback receives the message
    }

    return 0;
}

const struct bt_mesh_model_op hrtls_model_gw_ops[] = {
    { HRTLS_MODEL_GW_LOC_PUSH_OPCODE, BT_MESH_LEN_EXACT(
       HRTLS_MODEL_GW_LOC_PUSH_LEN), handle_message_loc_push},
    BT_MESH_MODEL_OP_END
};
```

Listing 13: Bluetooth Mesh gateway model handler definitions

3.4.3. Hardware requirements of embedded software

Thanks to the modular and hardware-agnostic nature of Zephyr, we were able to provide software that is easy to port to other embedded hardware, rather than the one used in the development process. The following list summarizes the requirements for devices used in our system:

- Gateway
  - TCP/IP networking stack with appropriate Zephyr support driver, such as: Wi-Fi controller, cellular modem, ethernet controller, OpenThread stack
  - Bluetooth Low-Energy controller
  - interface compatible with Zephyr Shell such as UART
- Anchor / tag
  - Bluetooth Low-Energy controller
  - DW1000 IC-based module, such as DWM1001C available on DWM1001-DEV development kits
  - SPI bus to communicate with the DW1000 IC.

3.5. Dashboard application

Dashboard application is an example of a third-party web application using the API exposed by the backend. It enables visualization of a positioning network located in a residential space. The application displays the positions of tags on a three-dimensional model of a room.

We used the following technologies:

React React is a JavaScript library that allows developers to build interactive user interfaces. Its reactive design allows it to automatically update the UI in response to changes in the data, making it a popular choice for building single page applications (SPAs) where the UI is updated dynamically as the user interacts with the application.

Next.js Next.js is a React framework for building web applications that provides a set of tools and configurations for tasks such as routing, data fetching, asset management, and search engine optimization. It simplifies the process of building React-based web applications by handling many of the common challenges and concerns that developers encounter.

Material UI Material UI is a library of React components that follows the principles of Google's Material Design. It offers a wide range of customizable and extensible components that make it easy to implement user interfaces quickly and effectively.

Three.js + React Three Fiber + Drei Three.js is a lightweight, general-purpose 3D library that includes a high-performance WebGL renderer. React Three Fiber is a wrapper library that enables developers to declaratively build 3D scenes using React components. Drei is a collection of utility functions that are designed to be used in conjunction with React Three Fiber to make it easier to work with Three.js and React.

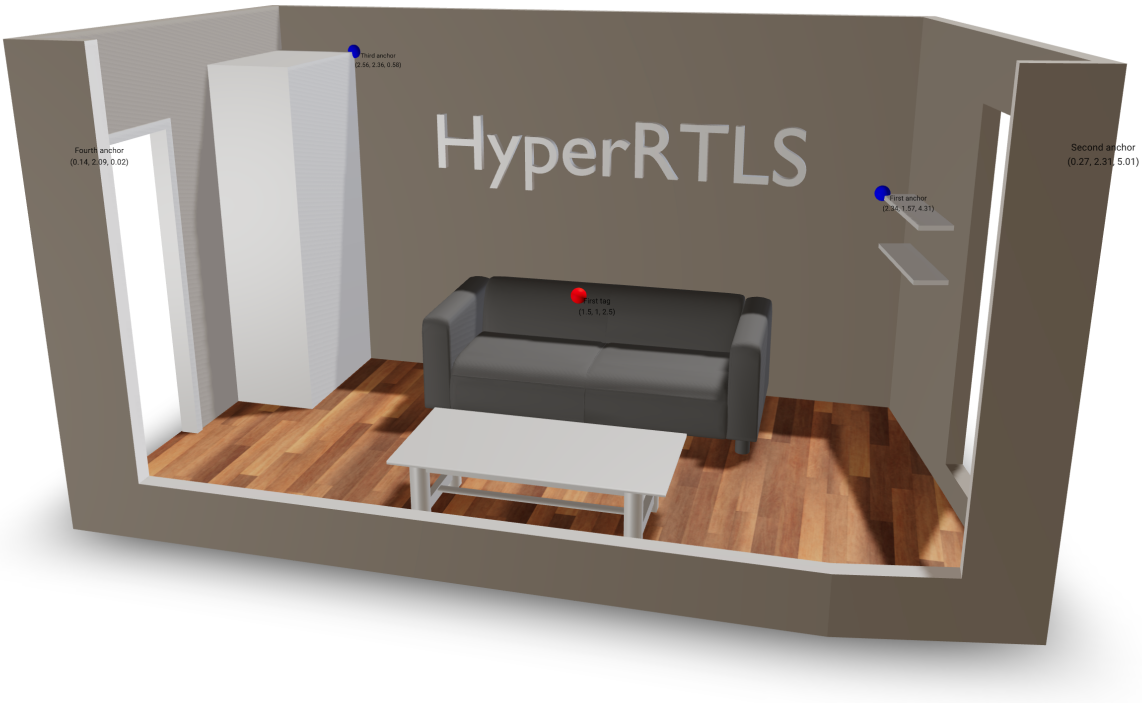

Figure 19: Dashboard view featuring real-time asset tracking and custom 3D model

4. Work organization

The objective of our project was both development and research.

Development From a development perspective, our goal was to create an open-source framework/platform that would allow others to easily develop a solution that includes a Real-Time Location System (RTLS). This would enable them to take advantage of the benefits of RTLS technology without having to invest significant time and resources in development. Overall, our goal was to create a platform that would make it easier for developers to create RTLS solutions and contribute to the growth and advancement of this important technology.

Research From a research standpoint, while we were aware that it is possible to develop such a system using Ultra-Wideband (UWB) technology as there were numerous solutions available in the industry, we were not certain about how we wanted to implement positioning functionality itself. There are various aspects to consider, such as the communication method used in the network, ranging methods and algorithms, and multilateration techniques, each of which comes with its own unique advantages and disadvantages. Therefore, we conducted research on these aspects while also developing our solution.

We were aware that the project we were undertaking was not simple and that there were numerous factors to consider. Our primary concern was to avoid being stuck with technology that did not meet our needs. We began development with a thorough research process, examining various frameworks, libraries, and algorithms and assessing their strengths and weaknesses. For every feasible realization aspect that we considered, we created a simple proof-of-concept application to test its functionality and compatibility with our requirements. Following these steps, we were confident in our choices and began developing the final product. Throughout the project, we used both iterative and incremental development with rapid prototyping prior to each iteration.

4.1. Team members and their role

Our team consisted of two members who were responsible for specific areas of the project based on their expertise and previous experience.

Sebastian Szczepański, a full-stack web developer, focused on creating the backend service and dashboard application, as well as solving architectural issues and developing a continuous integration/continuous delivery workflow.

Aleksander Wójtowicz, an embedded software developer, was responsible for developing software for embedded devices, including gateway, anchor, and tag applications, as well as researching and implementing ranging and multilateration algorithms.

Besides being fairly competent in the fields we were responsible for, we also have basic knowledge in the other's area of expertise. We were able to work together on certain problems and review each other's work. In his spare time, Sebastian also works on microelectronics and embedded software, while Aleksander has some experience in developing backend applications.

4.2. Used tools

Throughout the project, we used the following tools:

GitHub We utilized GitHub as a code hosting platform that provided a range of features to support our development process. The platform features a version control system that allows us to track and manage changes to our codebase over time, as well as collaboration capabilities that enable us to work together effectively. Specifically, we utilized pull requests to review and discuss each other's code, GitHub Actions for automating our CI/CD pipeline, and the issues feature to track and discuss progress, problems, and ideas. These tools helped us maintain high-quality code, stay organized and focused, and collaborate efficiently as a team.

Discord & Messenger We used Discord and Messenger as free and popular communication tools. Discord, in particular, provided a range of features to facilitate collaboration and problem solving, including the ability to make audio and video calls, share screens, and collaborate in real time. Messenger, on the other hand, was used for more basic communication needs such as texting. We used these tools to stay connected and communicate effectively throughout the project.

Visual Studio Code Visual Studio Code is a highly effective and versatile code editor that offers a range of features to support efficient code development. It is lightweight, free of charge, and can be easily customized with a variety of add-ons to transform it into a full-fledged integrated development environment. It was our primary choice for code development due to its ease of use, powerful capabilities, and wide range of customization options.

Email & WebEx Email was our primary method of communication with the project supervisor throughout the project. It allowed us to share updates on our progress, seek feedback, and stay on track by focusing on the most important aspects of the project. We also utilized WebEx to host periodic video calls, during which we discussed ideas and the realization of various aspects of the project. These forms of communication enabled us to stay connected and work closely with the supervisor to ensure the successful completion of the project.

Microsoft Teams We used Microsoft Teams to submit successive chapters of the thesis. The class notebook within the platform provided guidelines and assignments that we were required to complete and submit. This helped us track our progress and focus on the most critical aspects of the project. Using Microsoft Teams, we were able to stay organized and ensure that we met all the necessary requirements and deadlines for the thesis.

SequenceDiagram.org SequenceDiagram.org is an online tool that enables the creation of UML sequence diagrams, which are graphical representations of interactions between various parts of a system. We used it to create diagrams explaining specific parts of our code and how they should interact with each other. It helped to better understand and visualize the relationships and dependencies within our codebase.

Inkscape Inkscape is a free and open-source graphics editor that is primarily used for creating and editing vector images. We created a range of vector graphics that were then embedded in our project paper using the Scalable Vector Graphics (SVG) format. This allowed us to ensure that the included graphics were of high quality and remained readable regardless of the screen or print resolution. It helped us to present our ideas and findings in the paper.

Blender & Fusion 360 Blender and Fusion 360 are powerful 3D modeling and design tools that enabled us to create floor plans and 3D models for our project. These models were used to visualize the positions of tracked entities in the dashboard, providing a more accurate representation of their locations.

Overleaf Overleaf is a cloud-based LaTeX editor that provides a variety of collaboration and productivity features. It includes a rich history of all changes made to the document, allowing team members to easily review and track the progress of the project. The platform also includes the ability to work on the document simultaneously with other team members. Using Overleaf, we were able to eliminate the need to set up a local LaTeX compiler and manually share document files, greatly improving our workflow and reducing the overall time spent writing the paper.

IEEE Xplore / arXiv / ResearchGate Throughout the project, we conducted research and sought relevant scientific sources to support our work. We used a variety of digital libraries, including IEEE Xplore, arXiv, and ResearchGate, to find relevant literature and sources. Among these, we found IEEE Xplore to be especially valuable as it primarily consists of technical literature. Using these digital libraries, we were able to easily access a wide range of sources and information that helped us better understand the project.

4.3. Tracking progress, work planning

To ensure that our work was well organized and managed efficiently, we used the Issues and Projects features on GitHub. These tools allowed us to track and plan our work in a structured and systematic way. Using GitHub Issues, we were able to create detailed descriptions of the tasks and projects we were working on, track our progress, and document any ideas, issues, or challenges that arose. The Projects feature provided a visual representation of our work, allowing us to better understand the status and priorities of different tasks and projects. This enabled us to plan and manage our work more effectively, leading to a more efficient and successful outcome. Overall, the use of these tools helped us maintain a high level of organization and productivity throughout the project.

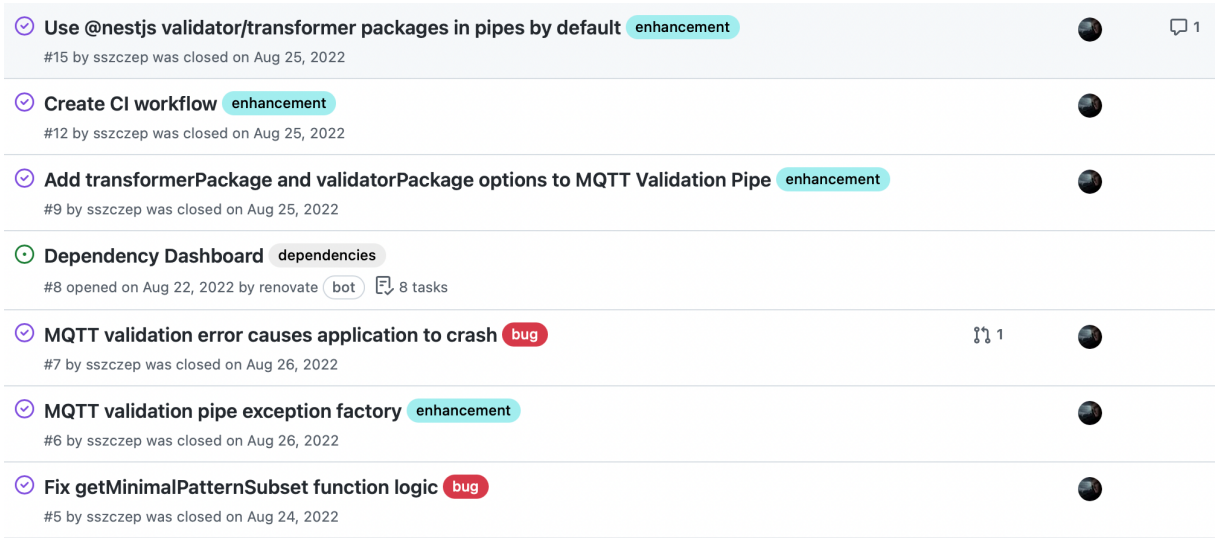

Figure 20: List of created tasks during development of the backend application

4.4. Code management

We employed a modified version of the *GitFlow* methodology. This is a widely recognized Git branching model that involves the use of multiple branches, each with a specific purpose. One key aspect of the GitFlow model is that it promotes a clean and organized commit history by enforcing a strict branching and merging process.

main The main branch holds the production-ready version of the application. To release a new version, we merge directly from the development branch into the main branch. As part of this process, new images may be pushed to the Docker registry if necessary. This method allows us to efficiently release updates without the need for a dedicated release branch, as opposed to the standard *GitFlow* methodology.

development The development branch is where the work on new features and changes takes place. It is used to integrate code from feature branches that are created to develop specific features or changes.

feat According to the GitFlow model, feature branches should be created directly from the development branch. Once a feature has been implemented, it should be merged back into the development branch. It is important to note that feature branches should never interact directly with the main branch, as this can cause disruptions to the production-ready codebase.

hotfix Hotfix branches are used to quickly resolve issues in the main branch without going through the whole GitFlow process. These branches are similar to feature branches in that they are used to implement and test a specific change, but they are based on the main branch rather than the development branch. After the issue has been fixed, the hotfix branch should be merged back into both the main and development branches. This ensures that the fix is incorporated into both the current production version and the version under development.

```
git checkout main
git checkout -b develop
git checkout -b feat/feature_name
# push changes to feat/feature_name branch
git checkout development
git merge feat/feature_name
git checkout main
git merge development
# oops, clients reported an issue with the latest release
# that needs to be fixed ASAP
git checkout -b hotfix/hotfix_name
# push fixes to hotfix/hotfix_name branch
git checkout development
git merge hotfix/hotfix_name
git checkout main
git merge hotfix/hotfix_name
```

Listing 14: Example of our modified GitFlow methodology using git command line interface

4.5. Continuous Integration

To facilitate efficient development, we created a GitHub Actions workflow that was dedicated to testing and publishing updated Docker images to the registry. This workflow was triggered whenever changes were pushed to the codebase and automatically ran tests to ensure that the code was functioning correctly. The updated Docker images were then tagged according to the branch they were pushed to, allowing users to easily update their containers using semantic versioning. This process greatly reduced the number of steps required to release a new version of the software, making it faster and easier for us to deploy updates. In addition, we released images for both the *linux/amd64* and *linux/arm64* architectures to ensure maximum compatibility for our users. Overall, the use of this workflow and the release of images for multiple architectures greatly improved the ease and efficiency of our development process.

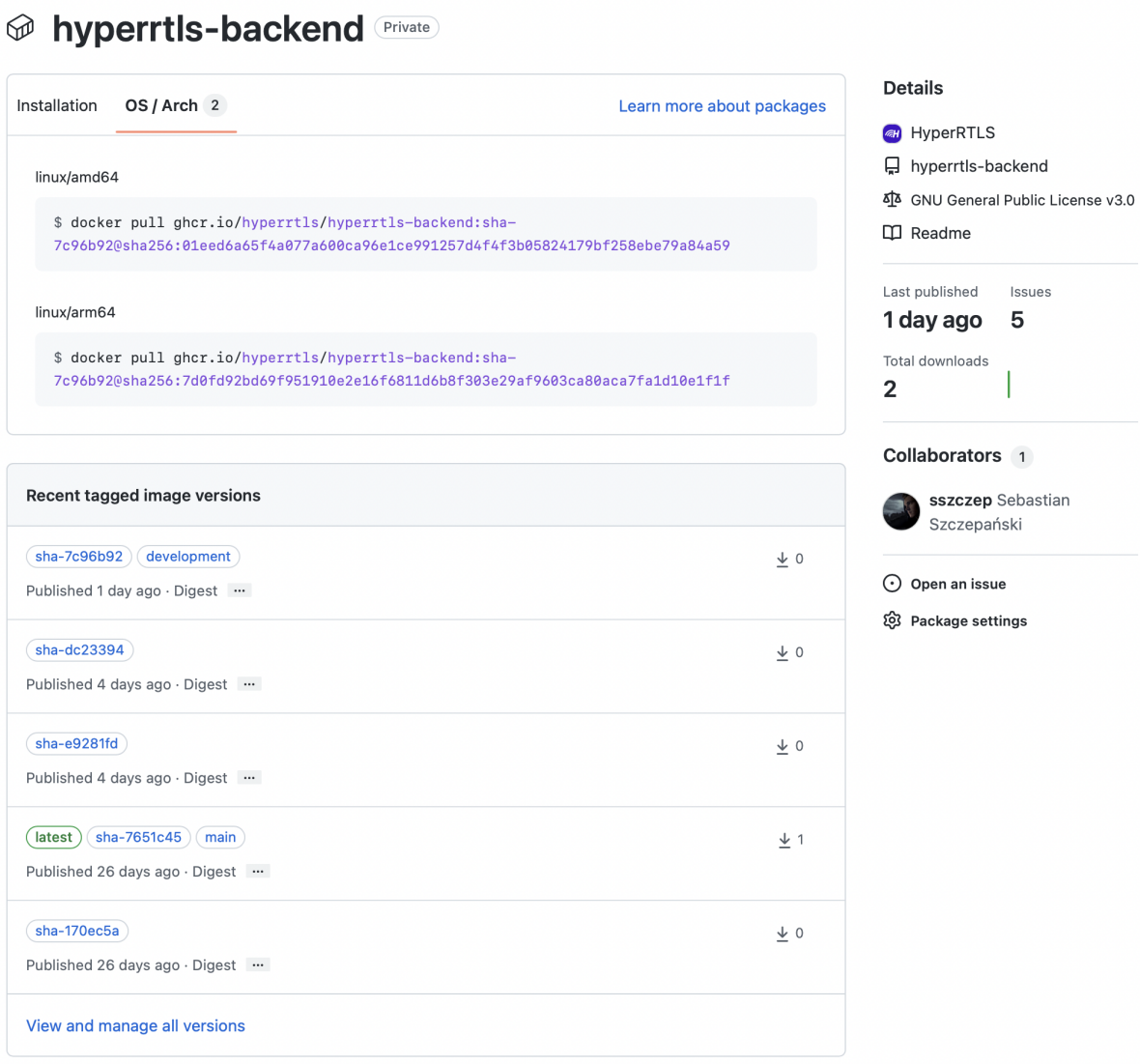

Figure 21: Released Docker images using a CI pipeline

5. Project results

The previous chapters of the thesis have outlined the goals of our project, the intended audience and use cases, the specific areas of focus, and the tools and workflow we employed. In this concluding chapter, we will reflect on the project results, evaluate how successfully we achieved the objectives, and identify any gaps or issues. We will also discuss ideas for future improvements.

5.1. Current state of the project

5.1.1. Ranging

As described in **Chapter 3**, the process of distance measurement between tags and anchors is a key element of the positioning system. We ultimately chose the SS-TWR distance measurement method, which turned out to be accurate enough for our needs and easier to implement than ADS-TWR. Figure 22 depicts a violin plot comparing real distance to multiple distance measurement samples. The measurements were taken indoors, in a living room.

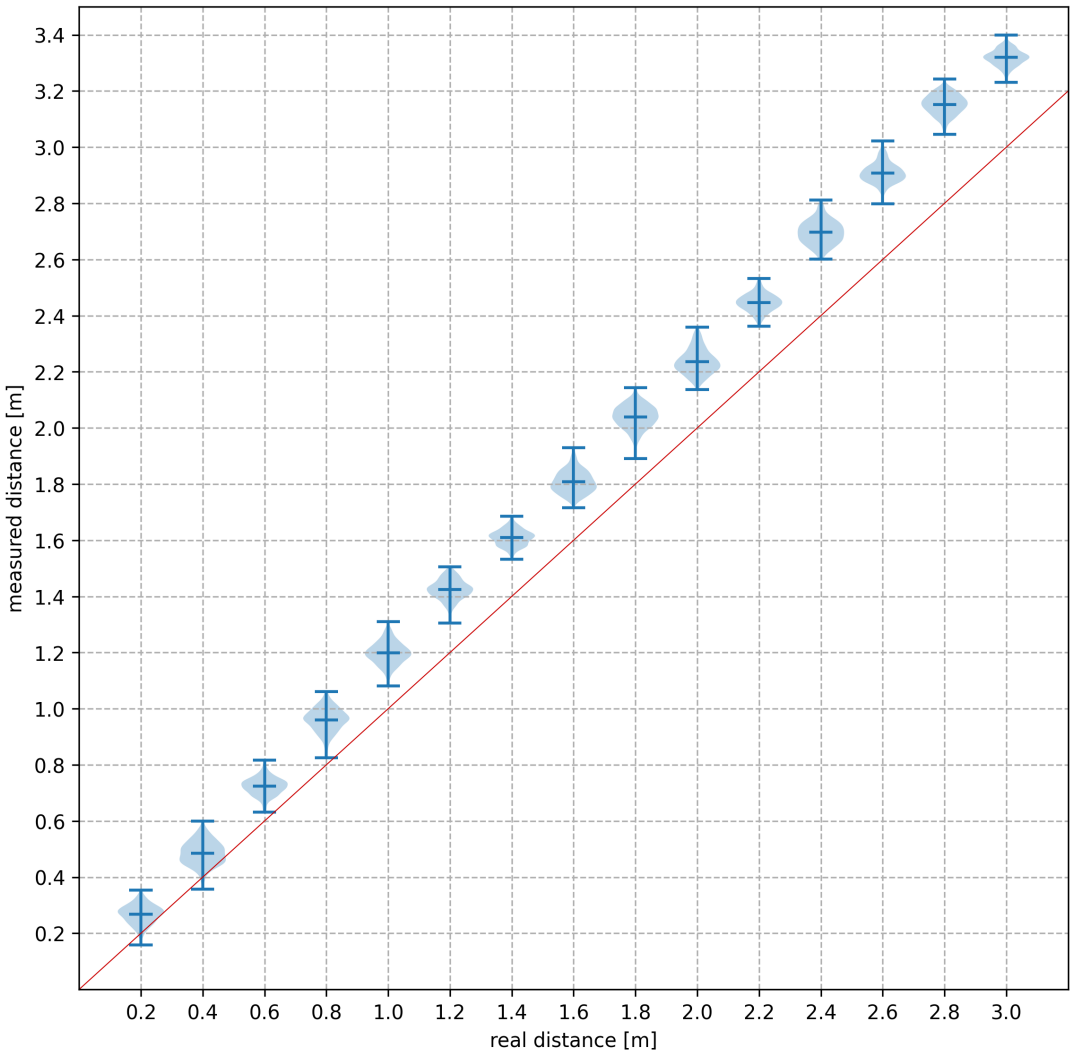

Figure 22: Real distance vs measured distance. The width of each shape represents the density of the distance measurement distribution.

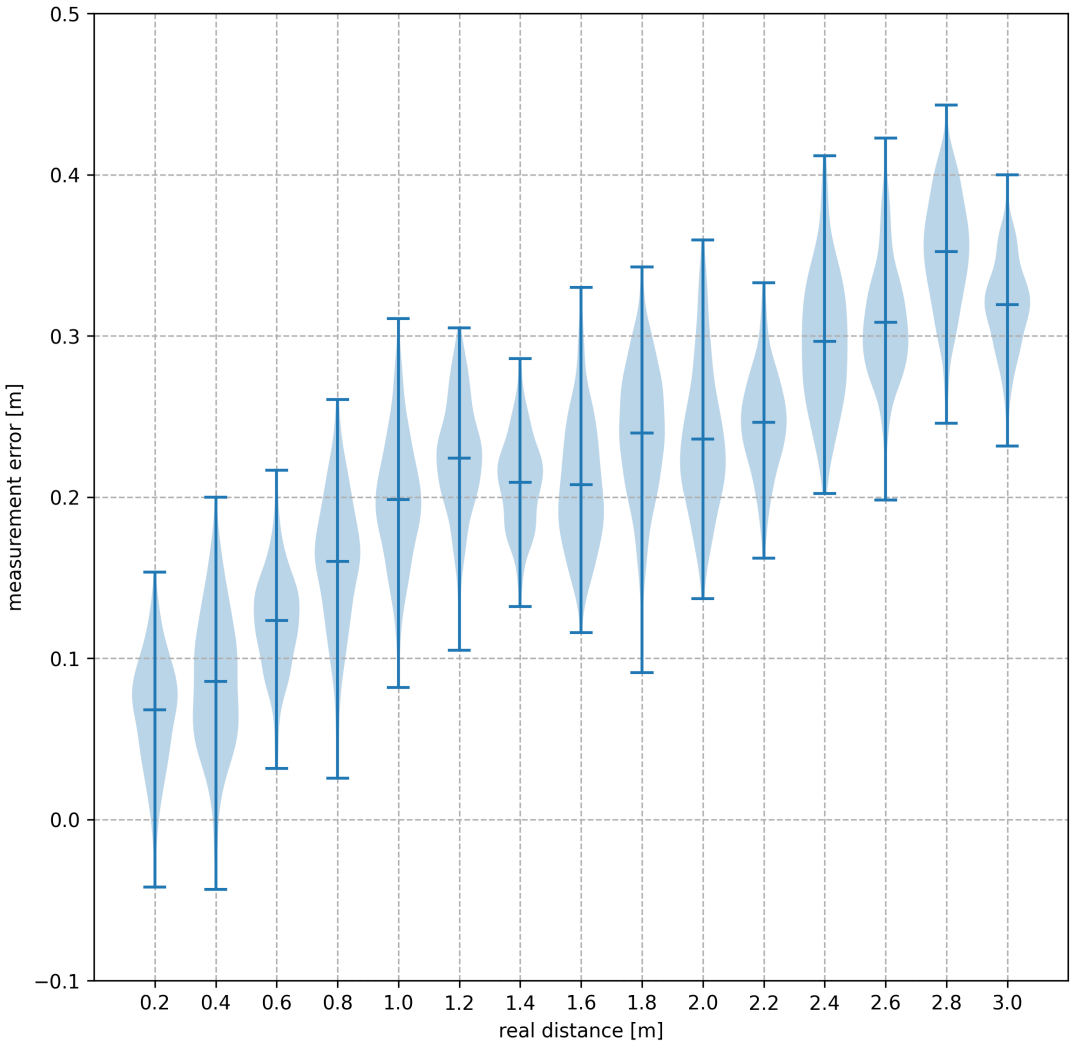

Figure 23: Real distance vs distance measurement error. The width of each shape represents the density of the distance measurement error distribution.

We have randomly selected an anchor and tag pair, and taken about 500 distance measurement samples for each 20 cm step in range from 20 to 300 cm. As seen in these violin plots, the noise of distance measurements is contained in the -10 to +10 cm range. The error seems to be correlated to true range — the mean of error for 20 cm real distance is just 8 cm, while for 300 cm real distance it is 32 cm.

The results are affected by the antenna delay, which may be slightly different for every manufactured unit. An estimate of correction value for antennas used in DWM1001C modules is pre-configured by the manufacturer. It is generally a little too low, giving a positive error in distance measurement. Higher precision can be achieved by individual calibration of units used in the system, although that is outside the scope of our work.

The API provided in the tag and anchor application allows one to modify the ranging method, so we may consider implementing others in the future.

5.1.2. Positioning

The implemented multilateration algorithm performs up to our expectations. During the tests in the setting presented in **5.1.3** section, most of the time the position estimation error was less than 20 cm. The system is less precise in cases where the line of sight between a tag and an anchor is obstructed.

Precision was greatly improved by averaging out five consecutive distance measurements. Much better results can be achieved with proper correction of antenna delays and use of, for example, the Extended Kalman Filter (*EKF*) [26].

We have measured the accuracy of our sample positioning network by putting a tag on a stand at eye level. The tests were conducted indoors, in a living room, specifically in the setting presented in **5.1.3** section.

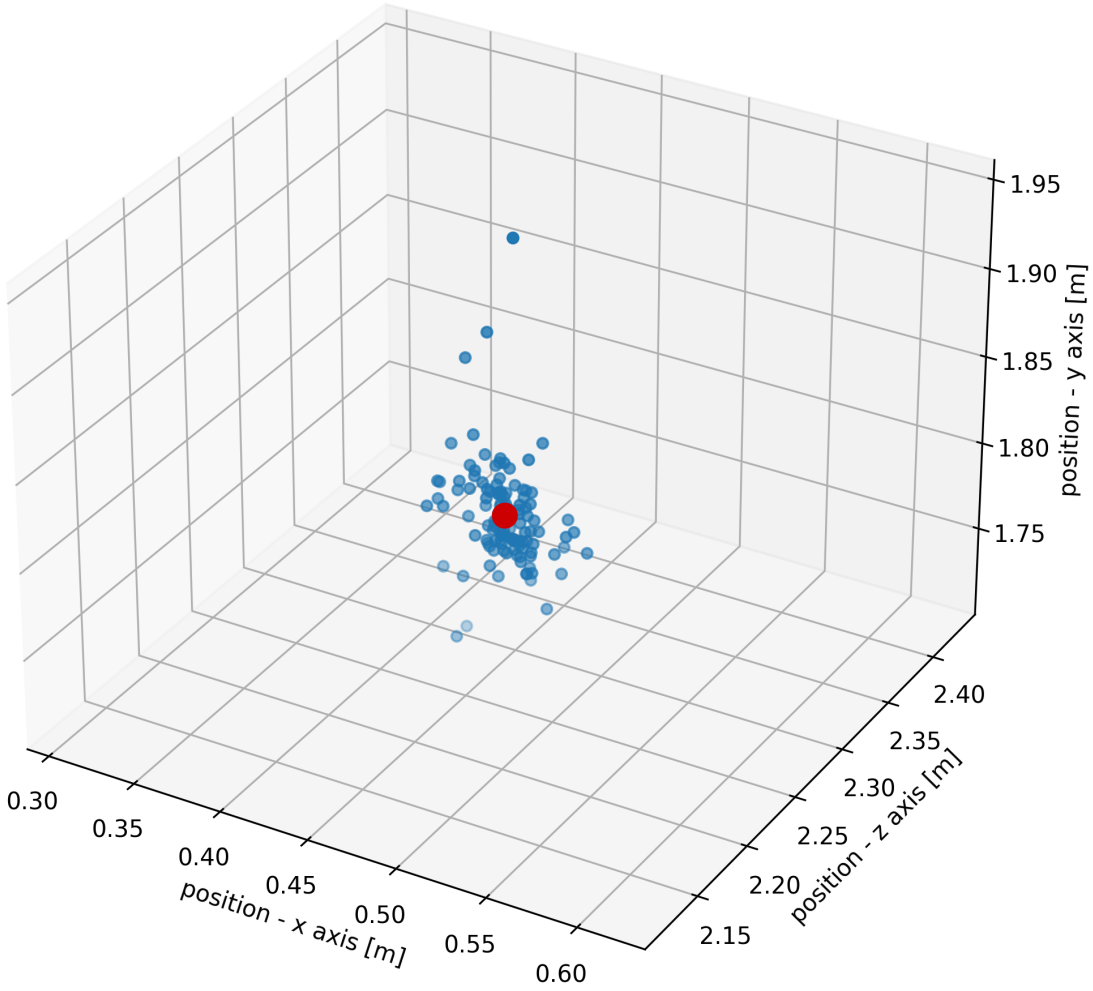

Figure 24: Scatterplot of estimated position. The true location is shown as red dot.

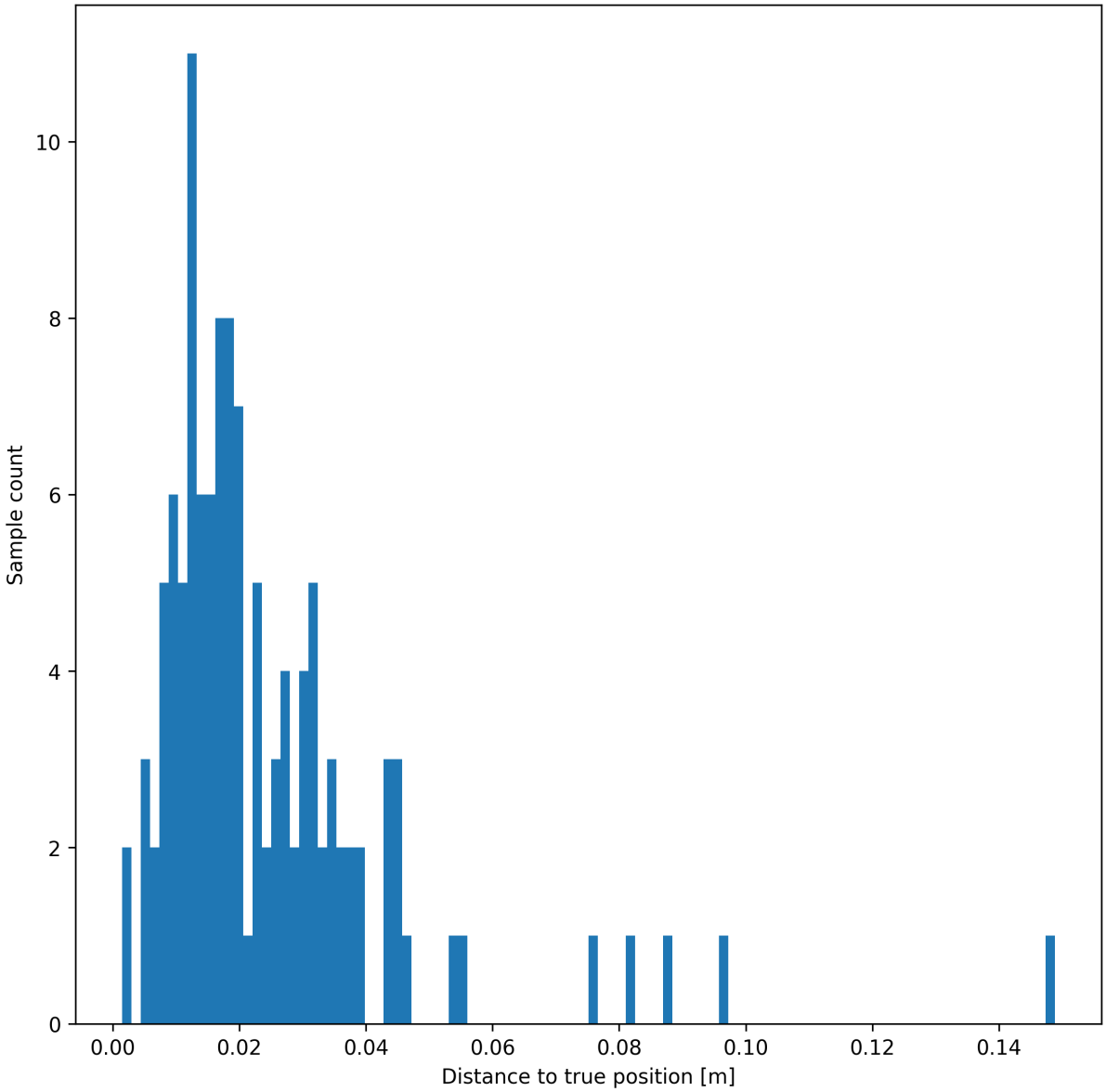

Figure 25: Position estimation error — histogram of euclidean distances between estimated and true position.

As shown in Figure 25, surprisingly, the accuracy of position estimation is much better than of raw distance measurements, which have a positive error. We suspect that in scenarios where the tracked tag is located in the center of the system, distance measurement errors are canceling each other out.

5.1.3. Dashboard

Using *Three.js*, *React Three Fiber*, and *Drei*, we were able to render a 3D model of a residential room designed in *Blender*. Having connected to server's event stream (using Server-Sent Events), we received tags' updated positions in real time. Anchors and tags are represented by blue and red spheres, respectively.

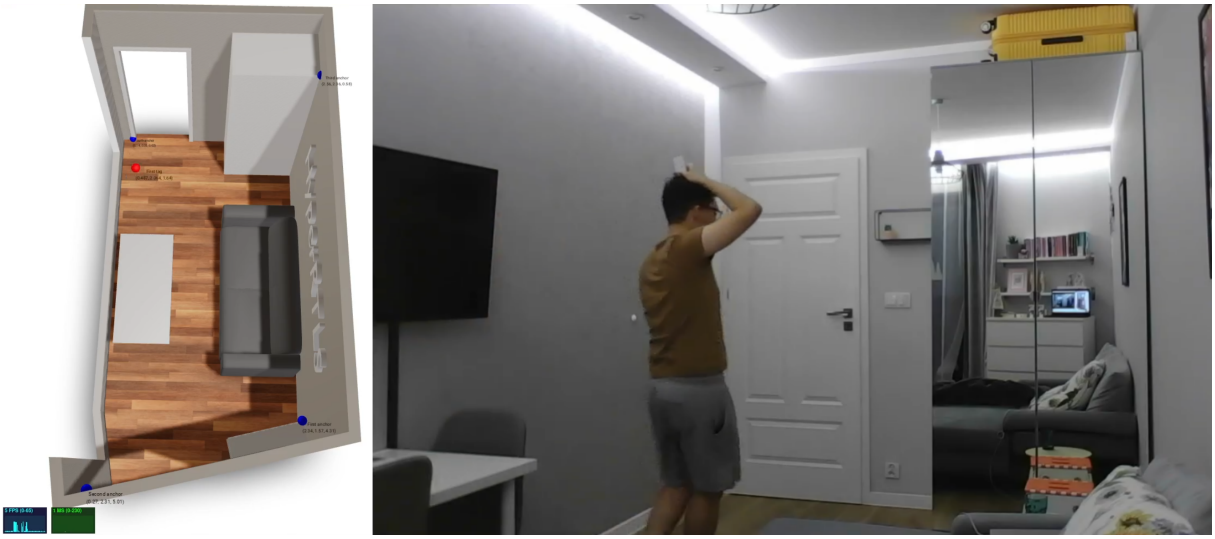

Figure 26: Simultaneous view of dashboard and Aleksander holding a tag on top of his head

5.1.4. REST API

We implemented all endpoints required for correct platform functionality. All of them feature data validation, proper error handling, and sending appropriate status codes. They allow CRUD-like operations (Create, Read, Update, and Delete) on tags, anchors, and gateways. Incoming data validation is based on DTO (Data Transfer Object) schemas, which define the payload shape, parameters types, and other restrictions. In the future, we hope to host a complete documentation page that contains all the endpoints and their usage examples.

We also heavily rely on Server-Sent Events, thanks to which we were able to achieve near-instant position updates.

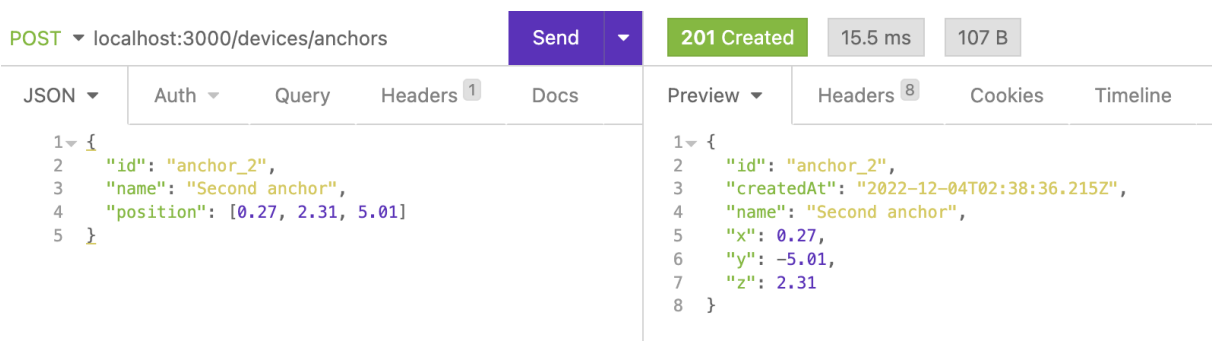

Figure 27: Creation of new anchor device by sending a HTTP request

5.2. Ideas for future consideration

There are few features that we were unable to implement at the time or were postponed for further releases. We hope to continue to work on the project and maintain it. We believe that it might fill the gap created by the absence of open source projects.

Documentation As mentioned in earlier chapters, a detailed documentation should be of utmost priority for every open-sourced project. We should write an installation guide, describe all the endpoints, and tell about the code itself. All this should be done before the project's public launch.

Test suites Being able to automatically and deterministically test changes pushed to the project significantly simplifies contribution and spotting errors. Having a good test suite is also beneficial when refactoring the code to avoid breaking changes. Being an open-sourced project, we should aim for at least 90% code coverage.

Dashboard functionality While we wanted to keep the dashboard as simple as possible, we think it should showcase more functionality and allow for more. We should add asset management system and administrative tools. Ideally, one should be able to configure the whole system using the dashboard only, not needing to manually send REST requests.

GraphQL GraphQL is a query language for APIs, which solves many issues with REST. Applications might ask for exactly what they need and nothing more. GraphQL would facilitate the retrieval and modification of resources, as users would not need to memorize all available endpoint and query options. It would coexist with the current REST API to provide support for more traditional applications.

Scalability We have tried to develop as scalable solution as possible, however, there is still a lot to cover. As of now, we have only containerized all the services and are able to balance the load between them. To further improve system scaling, we should implement health checks, geographic load balancing, and replication and synchronization of remote databases.

Alternative ranging techniques There are more advanced ranging techniques known than discussed in the *Selected realization aspects* chapter. For example, it is possible to send the first message in SS-TWR and ADS-TWR schemes just once, addressing multiple anchors. The anchors then, depending on which index of addressing array they are found, can cooperatively postpone their response to avoid any collisions. This way, in the SS-TWR scheme, to find distances to n anchors, only $n + 1$ messages could be used, instead of $2 * n$. This could improve the performance of the system.

Alternative multilateration techniques The current implementation of the multilateration algorithm can accept only 4 measurements. As there may be more anchors needed in the system, the current implementation only chooses the four closest anchors. To include more measurements in the linear regression model, the Moore-Penrose inverse (instead of the traditional inverse) of a matrix must be found. Algorithms for calculating the pseudoinverse of a matrix are very resource extensive; when we tried to use zscilib's `zsl_mtx_pinv()` method (which uses SVD and QR methods underneath), we ran into out-of-memory errors — the resources on nRF52832 MCU are very limited, the size of RAM is just 64 kB. As this matrix depends only on the chosen set of anchors against

which we are measuring distances, some computation offloading and caching mechanisms could be considered, e.g. to calculate these matrices only on the gateway, which has large computational resources, and cache these matrices both in gateways and tags, as most of the time the tags are positioning themselves against the same set of anchors.

Anti-collision mechanism We have not addressed the problem of communication channel collisions between multiple tags trying to find their positions. UWB and other radio communication technologies are not capable of processing multiple incoming messages simultaneously. If two tags try to find their position at the same time, both tags and anchor fail immediately because of the reception of unexpected messages, and wait for the next positioning attempt. There are some works [27] discussing anti-collision mechanisms for UWB-based communication systems.

5.3. Summary

After nearly a year of research and development, we are proud of what we have accomplished. Although we have not managed to implement all the features specified in the project requirements, we are satisfied with the results. Our platform is definitely not *production-ready*, but objectively speaking, the core work has been completed. It should be noted that the development of such a system usually requires a lot of time, funding, research, and other resources that we could not afford or justify. Most importantly, this engineering thesis is a proof that it is possible to develop a UWB-based RTLS in a two-person team. This project not only serves the purpose of a framework for positioning systems, but also provides a good knowledge base on how RTLSs work and what challenges the implementation of such systems poses. Some portions of the project were exceptionally hard to implement, as there are no other open-source examples of systems as complete as ours. After regular meetings with our supervisor, we can say with confidence that he feels the same way as us.

Overall, we are proud of what we have achieved so far and look forward to the next phase of this project. We are happy that we could bring our ideas to life, especially after many setbacks we have encountered. The project is certainly not over yet, as there are many features left to implement and issues to resolve, which we intend to work on in the future. We believe that our open-sourced solution has the potential to make a significant impact in various industries and applications, and we are determined to make it a success.

5.4. Acknowledgements

We are grateful for our supervisor's support and guidance throughout the project. His expertise and knowledge have been invaluable in helping us navigate the challenges and obstacles that came our way.

We also thank Karol Szustakowski, our university colleague, for his valuable advice and guidance on topics related to algorithmics.

Figures

Listings

References

[1] Chen Yongguang and H. Kobayashi. “Signal strength based indoor geolocation”. In: *Proceedings of the IEEE International Conference on Communications* (2002), pp. 436–439. DOI: 10.1109/ICC.2002.996891.

[2] Bluetooth SIG Inc. *Enhancing Bluetooth Location Services with Direction Finding*. 2019. URL: https://www.bluetooth.com/wp-content/uploads/2019/03/1901_Enhancing-Bluetooth-Location-Service_FINAL.pdf (visited on 04/15/2022).

[3] Wikipedia contributors. *Ultra-wideband — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Ultra-wideband (visited on 12/09/2022).

[4] David Barnwell. *What’s the deal with Ultra-Wideband?* 2021. URL: https://www.bmw.com/en/innovation/bmw-digital-key-plus-ultra-wideband.html (visited on 04/16/2022).

[5] Wikipedia contributors. *AirTag — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/AirTag (visited on 04/16/2022).

[6] Decawave (now Qorvo). *DWM1001-DEV product page*. URL: https://www.qorvo.com/products/p/DWM1001-DEV (visited on 12/09/2022).

[7] Zephyr Project. *Zephyr Project*. URL: https://www.zephyrproject.org/ (visited on 12/09/2022).

[8] Pozyx. *Pozyx’s UWB RTLS system*. URL: https://www.pozyx.io/products/pozyx-uwb-rtls (visited on 12/09/2022).

[9] Eliko. *Eliko UWB RTLS*. URL: https://eliko.tech/uwb-rtls-ultra-wideband-real-time-location-system/ (visited on 12/09/2022).

[10] Ubisense. *Ubisense SmartSpace*. URL: https://ubisense.com/smartspace/ (visited on 12/09/2022).

[11] Ubisense. *Ubisense Dimension4*. URL: https://ubisense.com/dimension4/ (visited on 12/09/2022).

[12] Wikipedia contributors. *User story — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/User_story (visited on 12/09/2022).

[13] Sarah Hatton. “Choosing the Right Prioritisation Method”. In: *19th Australian Conference on Software Engineering (aswec 2008)* (2008). DOI: 10.1109/ASWEC.2008.4483241.

[14] Decawave (now Qorvo). *DW1000 APS007 Application Note*. 2014. URL: https://www.qorvo.com/products/d/da008443 (visited on 01/02/2023).

[15] Sewio. *UWB Technology for Precise Indoor Tracking - Sewio RTLS*. URL: https://www.sewio.net/uwb-technology/ (visited on 12/10/2022).

[16] Decawave (now Qorvo). *DW1000 APS006 Application Note*. 2014. URL: https://www.qorvo.com/products/d/da008440 (visited on 12/10/2022).

[17] Mickael Viot et al. *Ultra-Wideband For Dummies®, Qorvo Special Edition*. John Wiley & Sons, Inc., 2021. ISBN: 978-1-119-80960-9.

[18] Decawave (now Qorvo). *DW1000 User Manual*. 2017. URL: https://www.qorvo.com/products/d/da007967 (visited on 12/10/2022).

[19] Decawave (now Qorvo). *DW1000 APS013 Application Note*. 2015. URL: https://www.qorvo.com/products/d/da008448 (visited on 12/10/2022).

[20] Dries Neirynck, Eric Luk, and Michael McLaughlin. "An Alternative Double-Sided Two-Way Ranging Method". In: *2016 13th Workshop on Positioning, Navigation and Communications (WPNC)* (2016). DOI: 10.1109/WPNC.2016.7822844.

[21] Zephyr Project. *Bluetooth tools: Running on QEMU and Native POSIX*. URL: https://docs.zephyrproject.org/3.1.0/connectivity/bluetooth/bluetooth-tools.html#running-on-qemu-and-native-posix (visited on 01/07/2023).

[22] Abdelmoumen Norddine. "An Algebraic Solution to the Multilateration Problem". In: *2012 International Conference on Indoor Positioning and Indoor Navigation* (2012). DOI: 10.13140/RG.2.1.1681.3602.

[23] Junlin Yan et al. "Feasibility of Gauss-Newton method for indoor positioning". In: *2008 IEEE/ION Position, Location and Navigation Symposium* (2008). DOI: 10.1109/PLANS.2008.4569986.

[24] Chuanyang Wang et al. "Bias Analysis of Parameter Estimator Based on Gauss-Newton Method Applied to Ultra-wideband Positioning". In: *Applied Sciences* (2019). DOI: 10.3390/app10010273.

[25] Zephyr Project. *Zephyr Scientific Library (zscilib)*. URL: https://github.com/zephyrproject-rtos/zscilib (visited on 12/11/2022).

[26] Zerui Fang et al. "An Implementation and Optimization Method of RTLS Based on UWB for Underground Mine". In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (2021). DOI: 10.1109/TrustCom53373.2021.00171.

[27] Oskar Sundin. "UWB-TWR performance comparison in a hybrid node network". MA thesis. Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering, 2021.