School of Computer Science and Information Technology
Lucerne University of Applied Sciences and Arts (Switzerland)

DRONE MONITORING WEB APPLICATION

Extending the Drone Monitoring Platform

BACHELOR THESIS

presented to School of Computer Science and Information Technology of Lucerne University of Applied Sciences and Arts (Switzerland) in consideration for the award of the academic grade of *Bachelor in Computer Science.*

by

Sven Fahrni

from

Bern, Switzerland

Declaration

Bachelor Thesis at Lucerne University of Applied Sciences and Arts
School of Computer Science and Information Technology

Title of Bachelor Thesis:	Drone Monitoring Web Application
Name of Student:	Sven Fahrni
Degree Program:	BSc AIML
Year of Graduation:	2025
Main Advisor:	Thomas Letsch
External Expert:	Jérôme Zürcher
Industry partner/provider:	Cyber Defence Campus

Code/Thesis Classification

- ☐ Public (Standard)
- ☐ Private

Declaration

I hereby declare that I have completed this thesis alone and without any unauthorized or external help. I further declare that all the sources, references, literature and any other associated resources have been correctly and appropriately cited and referenced. The confidentiality of the project provider (industry partner) as well as the intellectual property rights of the Lucerne University of Applied Sciences and Arts have been fully and entirely respected in completion of this thesis.

Place/Date, Signature _____

Submission of the Thesis to the Portfolio Database

Confirmation by the student

I hereby confirm that this bachelor thesis has been correctly uploaded to the Portfolio Database in line with the code of practice of the University. I rescind all responsibility

and authorization after upload so that no changes or amendments to the document may be undertaken.

Place/Date, Signature _____

Expression of thanks and gratitude

I would like to express my sincere gratitude to the Cyber Defence Campus for making this project possible. Special thanks to Dr. Bernhard Tellenbach for enabling the opportunity. I am particularly grateful to Llorenç Roma for his dedicated guidance during our weekly meetings, and to my supervisor, Thomas Letsch, and expert Jérôme Zürcher, whose valuable insights enriched my work. Finally, heartfelt appreciation goes to my friends and family for their constant support, encouragement, and feedback throughout this process.

Sven Fahrni, 2025

Summary

As the number of civilian drones continues to grow, monitoring the airspace has become increasingly important. In a previous project, the Cyber Defence Campus developed an initial version of a web application to help monitor drones. This Bachelor's thesis builds on that system by adding new features to make it more useful in real-world situations. The new version works without an internet connection, can handle more detailed drone data, and remains stable even when many signals are received at once or during spoofing attacks.

The project was developed using the Scrum framework, with regular meetings, clearly defined goals, and feedback loops to guide progress. To make sure everything worked as expected, the system was tested using automated tests, and realistic test scenarios.

Key improvements include full support for all message types defined in the ASD-STAN prEN 4709-002 standard, offline maps using MapLibre GL JS, and a clean API that allows external tools to access drone data without using the web interface. The performance of the system was also improved through better handling of large amounts of data using multithreading and optimized data processing. Overall, the new version of the application is more flexible, stable, and ready for future use.

Contents

List of Figures

List of Tables

Acronyms

API application programming interface. 2, 16, 31

ASD-Stan Aerospace and Defence Industries Association of Europe – Standardization. vii, 4, 5, 6, 15, 16, 20, 22, 23, 24, 25, 33, 39, 41, 48

CDN Content Delivery Network. 10, 11

CYD Campus Cyber Defence Campus. 1, 9, 12, 43

DTO Data Transfer Object. 21

ORM Object-Relational Mapper. 19

OSM OpenStreetMap. 8, 12, 35

OUI Organizationally Unique Identifier. 24

Remote ID Remote Identification. vii, 4, 5, 6, 10, 11, 18, 20, 37, 38, 43, 47

Chapter 1

Introduction

1.1 Background and Problem Statement

The use of civilian drones has increased dramatically in recent years, with over 90,000 drone pilots currently registered in Switzerland [5]. Civilian drones serve a wide range of applications, including wildlife rescue, building inspection, crop protection in vineyards, medical delivery, and even human search-and-rescue operations. However, this widespread adoption also introduces new challenges. An increasing number of incidents have been reported involving drones violating privacy boundaries and entering restricted airspaces [6, 7].

To address these concerns, authorities in the EU, US, and Switzerland have introduced regulations requiring drones to broadcast flight data in real time, including their location, altitude, and speed [8, 9, 10]. These regulations enable the identification of drones and the detection of potential airspace violations.

Two students at the Cyber Defence Campus (CYD Campus) have successfully developed a functional drone monitoring system that receives and interprets those broadcasts in real time [11]. Their software displays drone positions on a live map, enabling real-time tracking of drone activity. While the system shows strong potential, the current version captures only a subset of the available data, depends on internet connectivity and lacks resilience against spoofing attacks. These aspects present opportunities for further enhancement.

1.2 Objectives of the Thesis

Building upon the existing groundwork, the primary objective of this thesis is to develop a drone monitoring system capable of operating reliably in potentially adversarial environments. The following goals will be pursued

- **Offline Operation** Improve the system so it can operate without needing an internet connection.

- **Comprehensive Message Support** Expand the existing prototype to support a wider range of messages, enabling comprehensive monitoring.
- **Intuitive API** Develop a headless operation mode and provide a well-documented, user-friendly application programming interface (API) to support integration with external systems.
- **Security and Robustness** Strengthen the system's resilience by designing it to withstand adversarial conditions, such as the deliberate spoofing.

To meet these goals, the project will follow an iterative development process based on the Scrum framework. The four main objectives will be broken down into user stories, with a strong focus on testability to ensure that the final deliverable meets all requirements. The results of this work will provide a solid foundation for future improvements in drone monitoring technology.

1.3 Structure of the Thesis

This thesis is structured as follows:

- **Chapter 2: Theoretical Fundamentals**
  Introduces the foundational concepts relevant to this project, including Remote Identification and offline web mapping technologies.
- **Chapter 3: Ideas and Concepts**
  Describes the architecture of the existing system and presents solutions to improve its offline functionality.
- **Chapter 4: Methodology**
  Explains the project management approach, particularly the use of Scrum. It also covers the requirements engineering process and how the resulting system was tested.
- **Chapter 5: Implementation**
  Details the implementation, including backend architecture, message parsing, API design, frontend development, and offline map integration.
- **Chapter 6: Evaluation and Validation**
  Presents the evaluation of the system in terms of code quality, performance, and functionality, including results from unit and acceptance testing.
- **Chapter 7: Conclusion**
  Summarizes the outcomes, reflects on limitations, and suggests potential directions for future improvement.

The source code for the project can be found in the following GitHub repository: cyber-defence-campus/DroneIDReceiver. Some artifacts were produced by previous students. Only the code committed by the user `svenfahrni` should be considered for this thesis.

Chapter 2

Theoretical Fundamentals

The purpose of this chapter is to provide insights into the technical foundations necessary for implementing the additional functionality. The first section describes how drones can be identified followed by an in-depth examination of the ASD-Stan standard. In addition, the chapter introduces key technologies used for offline maps.

2.1 Remote Identification

Remote Identification (Remote ID) is a system that enables drones to transmit their identity and location information during flight. This information must be transmitted in an unencrypted format and can therefore be received by anyone, including the general public. The primary purpose of Remote ID is to enhance the safety, security, and accountability of drone operations, particularly in areas where drones may be operating near other aircraft or in sensitive locations [9].

In both Europe and Switzerland, drones are regulated under three main categories based on the level of risk involved in their operation: Open, Specific, and Certified. These categories also describe which drones are required to comply with Remote ID regulations. The vast majority of drones in Switzerland operate under the Open category. Drones in this category must weigh less than 25 kg, operate within visual line of sight (VLOS), and are not allowed to be flown over crowds. The Specific category applies to medium-risk operations, which require an operational authorization from the national aviation authority. The Certified category is reserved for high-risk operations, such as transporting people or hazardous goods, and requires both the drone and the pilot to be certified [12, 13].

For both the Open and the Specific categories, Remote ID has become mandatory starting January 1, 2024. There are very few exceptions; for example, drones that weigh under 250 g and do not possess a camera are exempt from this requirement.

2.1.1 ASD-Stan prEN 4709-002

To comply with the requirements laid out in the regulation, the *ASD-Stan* developed the standard prEN 4709-002. This standard covers all aspects of the regulation, including the transmission protocol and the structure of individual messages [14]. This section is based entirely on this standard, which is not publicly available and must be purchased through an authorized provider. As a result, a copy is not included in the appendix.

Broadcast Transport Protocols

Messages can be broadcast using Bluetooth 4.x, Bluetooth 5.x, Wi-Fi 2.4 GHz, or Wi-Fi 5 GHz. Drones are required to transmit their Remote ID messages via at least one of these protocols. While Bluetooth 4.x is considered a legacy option, it remains permitted under the current standard.

Direct Remote ID Messages

The standard defines seven different types of messages. Each message has a unique identifier called the message type, ranging from 0x0 to 0xF. The following list provides an overview of these messages. Detailed information about their structure and content can be found in Section 5.1.1.

- **Basic ID Message (0x0)** This mandatory message provides a unique identifier for the drone. The identifier may be a serial number or an identifier issued by American or British authorities. Therefore, for drones registered in Switzerland, a serial number is generally expected.
- **Location/Vector Message (0x1)** This message contains information about the drone's flight parameters, such as longitude, latitude, heading, and ground speed. It is also mandatory and must be transmitted once per second.
- **Reserved Message (0x2)** This message type is reserved for future use and is currently not in use.
- **Self ID Message (0x3)** This optional message can be used to convey additional information about the pilot or the purpose of the flight. It includes a free-text field that can be populated at the operator's discretion.
- **System Message (0x4)** This mandatory message provides information about the pilot, including their position, the number of drones operating nearby, and the classification of the drone (Open, Specific, or Certified).
- **Operator ID Message (0x5)** This mandatory message includes the UAS operator ID number, which is linked to the pilot. In Switzerland, obtaining this ID requires registration and the submission of contact details, an identification photo, and proof of insurance [15].

- **Message Pack (0xF)** This message type allows for the transmission of multiple individual messages within a single Wi-Fi or Bluetooth frame.

WiFi Message Structure

Messages transmitted over the Wi-Fi transport protocol share several important characteristics. Each broadcast Remote ID message is guaranteed to fit within a single Wi-Fi frame, which simplifies the decoding process significantly. Furthermore, the transmission protocol complies with the IEEE 802.11-2016 standard. This standard clearly defines the location of vendor-specific data, also referred to as the payload. The following table illustrates the structure of this payload for the various message types.

Direct Remote ID Message		
Msg Type (4 bits)	**Version** (4 bits)	**Message** (24 Bytes)
0x0 – 0xE	0x0 – 0xF	<Direct Remote ID message>

Table 2.1: Structure of a Direct Remote ID Message 0x0 - 0xE

The first four bits of each message indicate the message type, corresponding to the types defined in the previous section. For example, `0x3` indicates that a Self ID Message follows. The next four bits specify the message version. According to the ASD-Stan standard, each message type could support multiple versions. However, at the time of writing, only one version per message type is defined. For all messages except the Message Pack, the following 24 bytes contain the detailed message payload. The example below shows how an Operator ID Message is structured.

Operator ID (0x5) Message Format			
Offset (byte)	**Length** (bytes)	**Data field**	**Details**
1	1	Operator ID type	0: Operator ID 1–200: Reserved 201–255: Available for private use
2	20	Operator ID	ASCII Text. ASCII. Padded with nulls.
22	3	Reserved	Reserved

Table 2.2: Structure of the Operator ID Message

The only message that does not follow the structure defined in Table 2.1 is the Message Pack. This message type specifies how multiple individual messages can be grouped together. As with other messages, the first byte defines the message type and version. However, the following two bytes indicate the size of each individual

message and the number of messages contained within. Each subsequent message in the pack adheres to the structure outlined in Table 2.1.

Direct Remote ID Message Pack				
Msg Type (4 bits)	**Version** (4 bits)	**Single Msg Size** (1 byte)	**No of Msgs**	**Messages**
0xF	0x0–0xF	0x19 (25)	<1 Byte>	...

Table 2.3: Structure of a Direct Remote ID Message Pack (0xF)

Although not formally defined in the standard, recorded network traffic from previous students indicates that at least one drone transmitted multiple messages in a single frame without encapsulating them in a Message Pack. Fortunately, all message types except the Message Pack itself are limited to a maximum size of 25 bytes. Therefore, if a single message exceeds 25 bytes, it can be assumed to represent a Message Pack.

2.2 Offline Web Map Solutions

The current implementation uses the Google Maps API to display the geographical location of drones. Since both map data and satellite imagery are streamed from Google's servers, an active internet connection is required to retrieve and render this information. Without internet access, the visual mapping component becomes unavailable. To address this limitation, this section introduces the key concepts behind modern web-based mapping solutions and explores approaches for enabling offline functionality.

2.2.1 Tiled Web Maps

Modern map services divide the world into a grid of tiles. Each tile is identified by three attributes: the *x* coordinate, the *y* coordinate, and the zoom level. Whenever a user accesses a mapping service, the client requests the tiles corresponding to the area surrounding the desired location. The main advantage of this approach is that tiles can be dynamically loaded at runtime, offering an improved user experience and reduced data transfer compared to loading the entire map [16].

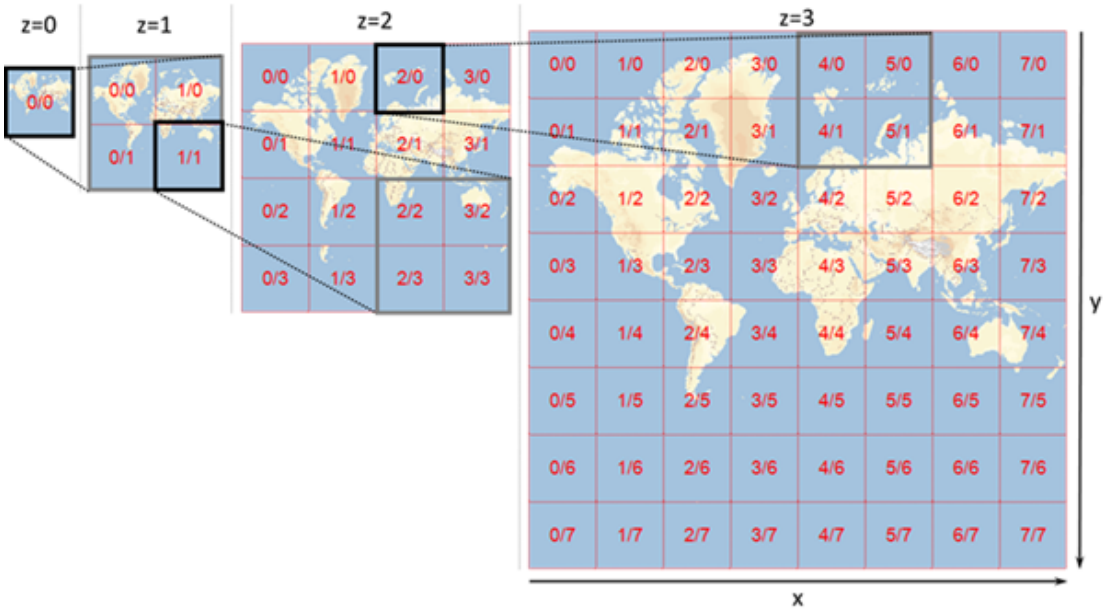


Figure 2.1: Visual example of a tiled web map [1]. The world map is divided into individual tiles. Different zoom levels provide tiles in varying resolutions.

A tiled web map may consist of either vector tiles or raster tiles. Raster tiles are pre-rendered image tiles, typically stored in formats such as PNG or JPEG. Traditionally, metadata such as city names is rendered directly into the images, making runtime modifications impossible [17]. Due to their large storage requirements, raster tiles are generally used only when necessary, such as for satellite imagery.

As the name suggests, vector tiles are stored in vector formats and rendered on the client side. This significantly reduces storage requirements. Geographic information such as forests, roads, or buildings is often provided in separate layers. Thanks to this structure, the client can determine which map elements should be rendered [18]. Vector tiles are typically used when fotos are not needed, for example, when displaying a street map.

2.2.2 OpenStreetMap

OpenStreetMap (OSM) [19] is a collaborative project that provides a free, editable map created and maintained by volunteers. Anyone can contribute by adding or updating geographical data, including roads, buildings, rivers, and points of interest. The map data is openly available, making it a valuable resource for this project.

Chapter 3

Ideas and Concepts

This chapter highlights different approaches to provide the new functionality. In order to choose the best-fitting approach, this chapter begins by describing the architecture of the existing system.

3.1 The Existing System

The current project was designed and implemented by two students at the CYD Campus. Their Bachelor Thesis “Building an Accessible and Affordable Drone Monitoring System Based on Remote ID” focused on building a cost-friendly and open-source drone-monitoring system.

3.1.1 System Context

The system is designed to be accessible to a large community, including people without much technical know-how. A **Technician** is responsible for installing and configuring the system. This process should stay as simple as possible through this work in order to continue guaranteeing the accessibility.

Additionally, a **Drone Observer** is able to monitor drones through a web application. The system supports both viewing drones in **real-time** and in a **replay mode**, which allows observers to review previous drone flights.

Drones continuously broadcast identification data, including their real-time position. The system captures and processes these messages and relays the most up-to-date information back to the observer.

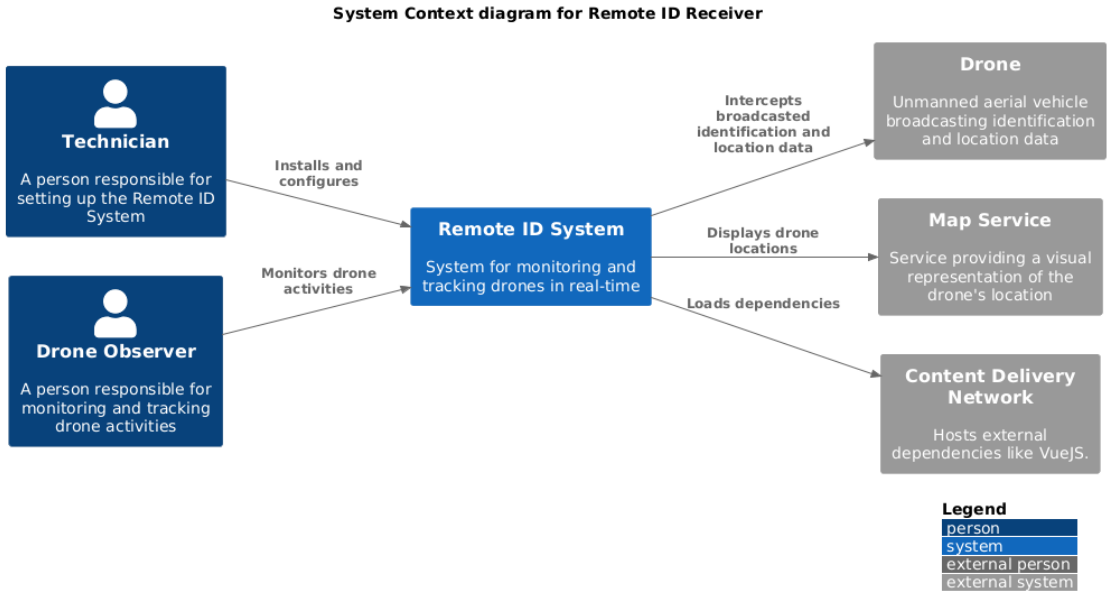


Figure 3.1: System Context - A high-level overview of the Remote ID System, its users, and external dependencies, following the C4 model.

The visualization works by relying on an external **Map Service**. This service provides geographical information as well as satellite imagery to the observer. In the current implementation, Google Maps is used as a map provider.

Lastly, a **Content Delivery Network (CDN)** is used to host dependencies for the frontend. This includes the VueJS framework as well as other internal tools.

3.1.2 Drone Identification

In order to detect drones, the system captures Remote ID messages by intercepting Wi-Fi frames that are broadcast by drones. Most Wi-Fi adapters are designed to connect to a wireless access point. For this project, the system should be able to capture any traffic without connecting to a specific access point. This requires a Wi-Fi adapter capable of operating in **monitor mode**. Monitor mode enables the device to listen to all nearby wireless traffic, including unassociated packets. DevWiki [20] has compiled a list of Wi-Fi USB adapters that support monitor mode.

Intercepting on Windows Subsystem for Linux

Windows Subsystem for Linux (WSL) is a very lightweight virtual machine that can be used as a Docker backend. Wi-Fi network cards are not supported using the default distribution. Therefore, a few additional steps are needed:

- **Forward the USB Network Adapter to WSL** — USBIPD-WIN is an official Microsoft tool that can be used to forward the USB port to WSL 2 [21].

- **Recompile the WSL 2 Kernel with Wireless Support** — Sean Hung provides a guide to recompile the official WSL 2 kernel together with the firmware of the USB adapter [22].

3.1.3 Drone Spoofing

In order to test the system, the previous contributors created a spoofer capable of generating fake Remote ID messages [23]. This spoofer then broadcasts the generated packets over a Wi-Fi network interface. Additionally, captured traffic from real drones has been provided. Replaying this traffic over a network interface and checking whether it is received by the system installed on another device can serve as the foundation for a thorough acceptance test. In order to transmit these Wi-Fi messages, a Wi-Fi adapter that supports **packet injection** must be used.

3.1.4 Compatible Wi-Fi Adapter

During this project, the *Archer T2U Plus – AC600 High Gain Dual Band USB Adapter* was used as the network adapter. It supports both monitor mode and packet injection. Its high-gain antenna should help in detecting network traffic over long distances. Drivers are available for all major operating systems, including Windows, macOS, and Linux [2].

Figure 3.2: TP-Link AC600 [2] - The USB WiFi adapter used for this project.

3.2 Offline Functionality

In the current implementation, the system is reliant on an active internet connection. This connection is required to access two external systems: the CDN and the map service. This work aims to eliminate these two dependencies.

The CDN is used by the frontend to load essential JavaScript libraries and frameworks. These assets are not bundled with the application itself but are fetched dynamically from online sources during runtime. The dependence on the CDN can be resolved by integrating the required JavaScript assets during the installation phase. This process, called bundling, is natively supported by the VueJS framework.

In order to support offline functionality for the visualization, a tiled web map must be available to the frontend. There are two possible approaches to achieve this.

3.2.1 Offline Map Approach #1: Caching Map Tiles

One way to offer offline maps is by caching tiles from an online tile server. Using this approach, the user can select the required area while online. The application then downloads all the necessary tiles and stores them on the user's device. This method is commonly used in offline GPS applications, where only a specific region needs to be available.

The main advantage is that services like Google Maps can still be used, including satellite imagery. The user can also update the map version whenever an internet connection is available. However, there are also downsides to this approach. An active internet connection is still needed to prepare the offline map. Local storage can also be a limiting factor, since individual tiles must be stored in the web browser itself. Furthermore, the official JavaScript library of Google Maps does not support this approach as of June 2025 [24]. This approach would require a significant rewrite of the frontend.

3.2.2 Offline Map Approach #2: Providing a Tile Server

A more advanced option is setting up a custom tile server. The tiles must be downloaded in advance by the technician. OSM provides a large collection of tiles covering the entire globe. This server can then run on the same device as the backend, allowing full offline access to map data. The main advantage is that it offers detailed offline maps for the entire world. Furthermore, online map servers can still be used. The user can therefore continue using Google Maps as a provider and is not limited to the offline map. However, the setup is more complex, and satellite imagery may not be freely available. The current implementation does not support custom tile servers, so a significant rewrite of the frontend is also required.

3.2.3 Decision Process

Both approaches require a similar technical effort to implement, as they both necessitate a substantial rewrite of the frontend. The first approach requires implementing caching logic for map tiles in the frontend, whereas the second approach involves setting up a custom tile server. Therefore, the decision is based entirely on business requirements.

One key requirement of the application is that it should be able to serve a detailed map of Switzerland. A detailed tiled web map of Switzerland is approximately 8 GB in size [25]. This requirement conflicts with the first approach, as web browsers cannot store such large amounts of data locally [26]. Consequently, the decision was made in coordination with the CYD Campus to set up a custom tile server.

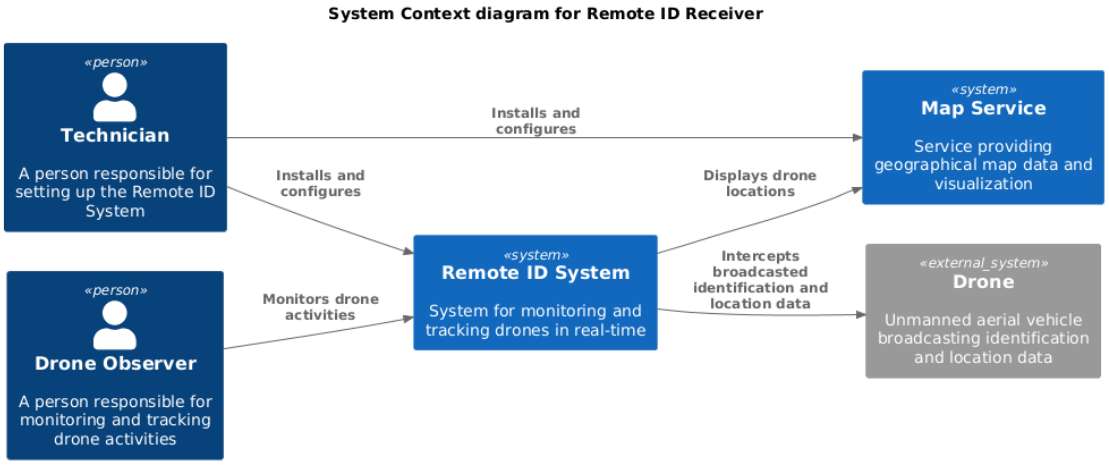


Figure 3.3: System Context - A high-level overview of the Remote ID System with offline capability, its users, and external dependencies, following the C4 model.

The adjusted c4 context diagram shows the new system with offline capabilities. The CDN has vanished, as it is no longer needed. The map service is now part of the internal system and is getting configured by the technician.

Chapter 4

Methodology

4.1 Project Management

Scrum has been chosen as the project management method because it allows for a flexible and adaptive approach to managing work. All parties involved preferred to hold regular meetings rather than trying to plan every detail of the project at the very beginning.

4.1.1 Product Vision

The vision is to create a high-performing drone monitoring platform that can be scaled to a nation-wide level.

With the input of Llorenç Roma, which took the role of the product owner, a vision has been established that gives a clear purpose for this project. The vision is not designed be achieved in this work alone. However it influences design decisions and motivates to design software components that can be reused in future projects.

4.1.2 Product Goal

The goal is to create a high-performing drone monitoring platform that can be used to monitor drones in a specific location without depending on the internet.

This project clearly defines the first step toward realizing the broader vision. This goal guides the development process by emphasizing modular architecture, robust data handling, and offline-capable user interfaces, setting the stage for future expansion and integration at a national level.

4.1.3 Epics

To organize the project more clearly, five main epics were created. These epics help group related tasks and make it easier to manage the overall work. They are not all the same size, but together they cover all the important requirements of the project. Using epics also helps to focus on specific goals during development and track progress more effectively.

- **Offline Functionality**: Operation does not require Internet connectivity.
- **Message Support**: Support all messages of the ASD-Stan standard.
- **API Design**: Design an intuitive REST API that can be used by other front-ends.
- **Performance**: Provide stability in an adversarial environment.
- **Optional Tasks**: Propose an architecture that can be scaled to a nation-wide level.

The project was structured around five development sprints, each lasting two weeks. This structure was chosen to ensure steady progress during development while also reserving time for thesis writing and testing. Each of those sprints was designed to roughly represent one epic. During these ten weeks, the student held weekly meetings with Llorenç Roma, who took on the role of Product Owner. A complete time plan, including all meetings, can be found in Appendix A.

4.1.4 Requirements Engineering

During the project, each epic was broken down into specific user stories to better manage and organize the work. These user stories were designed to follow the INVEST principle: they are Independent, Negotiable, Valuable, Estimable, Small, and Testable. To ensure both functional and non-functional requirements are captured, each story includes clearly defined acceptance criteria and technical requirements. An example of an individual user story is shown below.

User Story 1.2: Provide an Offline Replacement for the Map

As a drone observer,
I want to view and interact with maps even when I'm offline,
so that I can use location-based features without relying on internet access.

Acceptance Criteria

- ☐ The map contains the following geographic features: forests, lakes, rivers, streets, and buildings.
- ☐ The map contains labels for streets and city names.

- ☐ The user is able to zoom.
- ☐ The map client must support all previously implemented requirements for visualizing drones.

Technical Requirements

- ☐ Map: Only the initial setup is dependent on the internet.
- ☐ The map must be able to display at least 1,000 individual drones at once.

4.1.5 Story Book

Over the course of the project, a story book consisting of eight user stories was created (Appendix B). This section provides an overview of those stories. The main stakeholder is the drone observer, already introduced in Section 3.1.1. One requirement for the application is to `provide an API to be utilized by other applications/frontends`. To accommodate external systems, which do not yet exist, the stakeholder `external developer` was introduced. A full version of the story book can be found in the appendix. It includes acceptance criteria and technical requirements for each user story.

1.1 Frontend Libraries Available Offline As a drone observer, I want the application to load its frontend libraries without needing internet access, so that I can use the app even when I have no connectivity.

1.2 Provide an Offline Replacement for the Map As a drone observer, I want to view and interact with maps even when I'm offline, so that I can use location-based features without relying on internet access.

2.1 Message Support As a drone observer, I want to have as much information about the drone as possible, so that I can have an in-depth overview of the drones.

3.1 Vendor-Agnostic API As an external developer, I want to have vendor-agnostic information about drones, so that I can display drones from different data sources on my map.

3.2 ASD-Stan API As an external developer, I want to have full ASD-Stan support, so that I know exact details about the drones in the air.

3.3 DJI API As an external developer, I want to have full DJI support, so that I know exact details about the drones in the air.

4.1 Performance As a drone observer, I want to know how many drones are supported, so that I understand the limits of the receiver.

4.1.6 Deviations from Scrum

Scrum defines four standard types of meetings: sprint planning, sprint review, sprint retrospective, and daily scrum. Given the relatively small size of this project, we opted for weekly meetings between the developer and the Product Owner, without any Scrum Master.

4.2 Testing Strategy

The testing strategy was structured around a clearly defined Definition of Done, which ensured a consistent and high-quality development process. A user story was only considered “Done” when all of the following criteria were fulfilled:

- ☐ Acceptance criteria are met.
- ☐ Technical requirements are met.
- ☐ Automated tests have been successfully executed.
- ☐ Feature has been manually tested.
- ☐ Code is commented and contains no TODOs.
- ☐ Feature has been reviewed with the Product Owner.

This approach ensured that both functional correctness and code quality were addressed for each feature. To ensure the reliability of critical logic, automated tests were implemented and integrated into the development workflow using GitHub Actions. These actions automatically generate public coverage reports based on both unit and integration tests. In addition, manual testing was carried out at the end of each sprint to validate user-facing components, such as the monitor-view.

At the end of the project, two more sophisticated acceptance tests were performed. These were designed to test the entire system, including the drone sniffing logic, which was not directly worked on in this project. Additionally, a performance test verified the system’s behavior under load. These tests are further described in Section 6.2.

Chapter 5

Realization

This section highlights both the architecture and the implementation of the drone monitoring system. Figure 5.1 illustrates the internal structure of the Remote ID system. The system is built on four fundamental components: the frontend, backend, map service, and database.

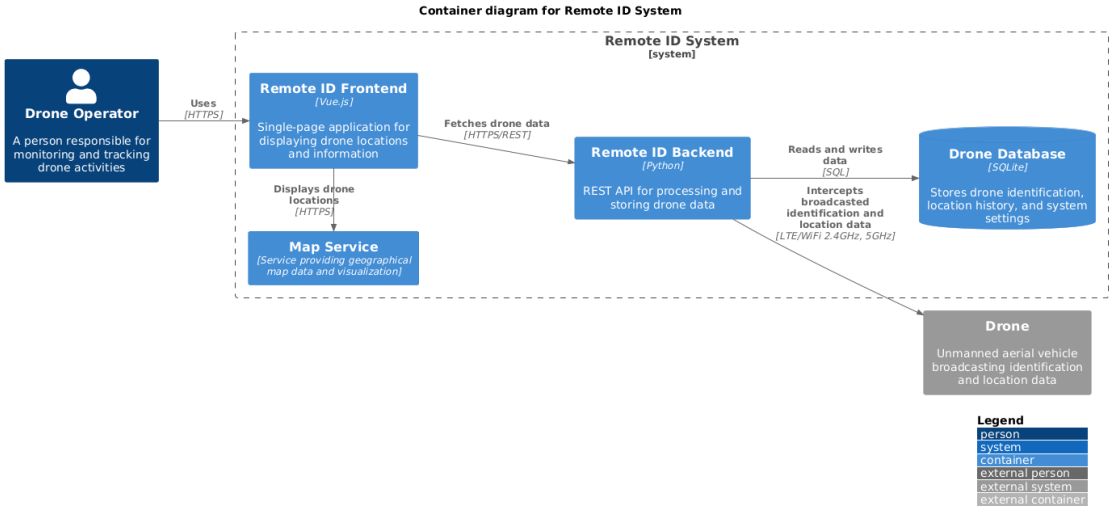


Figure 5.1: Container Level - A more detailed look at the building blocks of the Remote ID system.

The **frontend** serves as the main interface for the drone observer. It communicates with the backend to retrieve real-time drone locations, as well as detailed information about individual drones via a REST API.

The **backend** is responsible for intercepting drone broadcasts, which are transmitted over LTE, Wi-Fi 2.4 GHz, and Wi-Fi 5 GHz. It processes these messages and stores them in a database. Additionally, the backend exposes both REST and WebSocket APIs, which the frontend uses to access the data.

The **map service** provides map tiles. In the previous implementation, this service was external. With recent adjustments to the frontend library, tiles can now

be loaded from any source, including a custom offline tile server hosted on nearly any hardware.

For the **database**, SQLite was chosen due to its simplicity, low overhead, and ease of integration. Since the system is designed to run on low-cost hardware and may not require the complexity of a full-scale database server, SQLite offers a lightweight and efficient solution for storing and querying drone data.

5.1 Backend

This C4 component diagram illustrates the internal architecture of the Remote ID system backend. The backend has two main responsibilities: processing packets from drones and making the data accessible to the frontend. To provide a clear separation of concerns, the architecture is organized into three layers: the Interface Layer, Business Layer, and Persistence Layer.

- **Interface Layer** This layer is responsible for communicating with external components. It consists of a `SniffManager`, which listens to Wi-Fi network traffic and captures Wi-Fi frames broadcast by drones. Additionally, a `DroneAPI` provides a well-defined interface for the frontend.
- **Business Logic Layer** This layer handles the core logic of the system. A `PacketProcessor` defines how drone messages are parsed and mapped to database models. A `DroneService` describes how the API can access these persisted messages.
- **Persistence Layer** This layer is responsible for interacting with the database. It uses SQLAlchemy as an Object-Relational Mapper (ORM) to interface with the SQLite database.

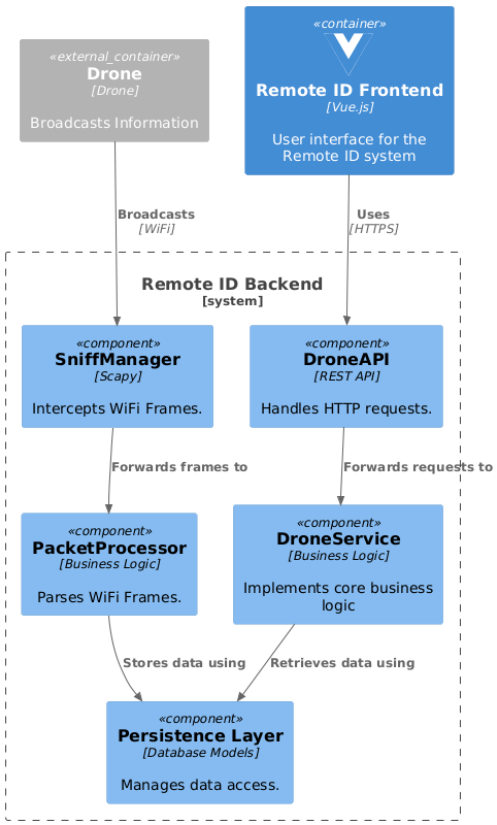


Figure 5.2: Backend Architecture - A C4 Component Diagram of the Remote ID Backend Container.

5.1.1 Persistence Layer

One goal of the application is to capture as much data from drones as possible. An external frontend should later be able to retrieve detailed information about each individual message. To satisfy this requirement, every message defined in the ASD-Stan standard is persisted individually. Figure 5.3 shows the models representing these individual messages in the persistence layer.

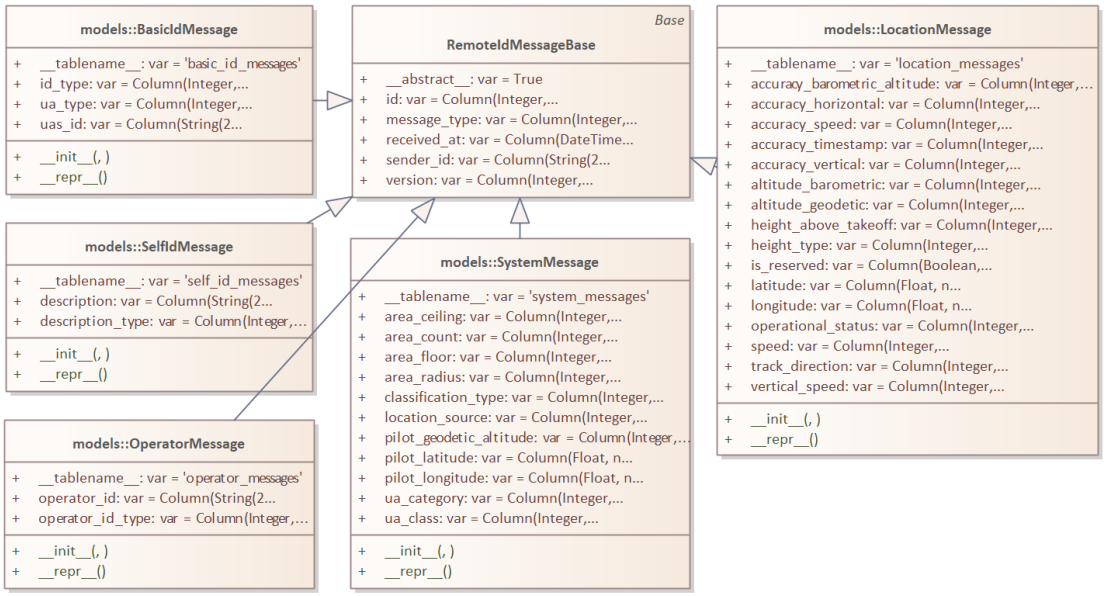


Figure 5.3: Domain Models - UML class diagram showing the message models in the persistence layer.

`The RemoteIdMessageBase` class defines the common attributes of a Remote ID message. By inheriting from the SQLAlchemy `Base` class, these models define the schema of the database. This design offers several advantages. First, the schema can be automatically generated and validated at application startup, eliminating the need to write migration scripts to define the database structure. Second, the `Base` class provides methods such as `BasicIdMessage.store(obj)`, which internally generates and executes a `CREATE` statement. This removes the need to write raw SQL statements in the application code.

5.1.2 Drone Service

Each of the domain models is tightly coupled to the specific standard they implement. In contrast, the frontend requires information about drones regardless of the underlying standard. Figure 5.4 shows several abstractions that have been developed to represent drones independently of the message format.

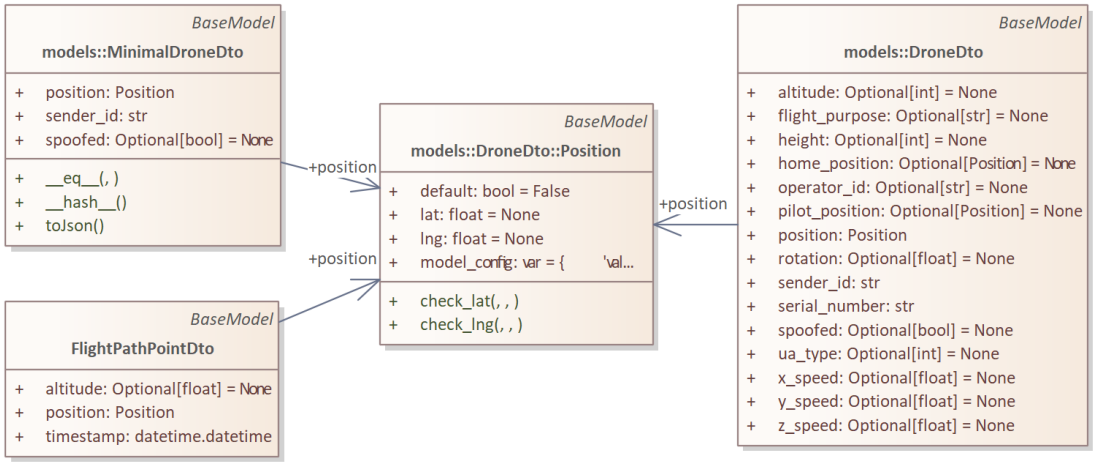


Figure 5.4: Drone Abstractions – UML class diagram showing the DTOs and abstractions for drone representation.

The `MinimalDroneDto` is a Data Transfer Object (DTO) that represents the minimal set of attributes required to display a drone on a map. It includes a position, a flag indicating whether the drone has been spoofed, and a `sender_id` used to uniquely identify the drone. If the underlying messages were broadcast using Wi-Fi, this will be the drone's MAC address. This DTO also includes functionality for serialization and equality checks to help detect duplicates.

The `DronePathPoint` contains the minimal information needed to show a drone's location at a specific time step. This DTO is used in a list to display the drone's flight path.

The `Position` class consists of latitude and longitude attributes. When these values are set, Pydantic automatically validates whether they fall within the correct geographic range.

The `DroneDto` provides a more detailed representation of the drone. It includes the drone's position, the pilot's position, and the first spoofed location. It also contains telemetry data such as speed and altitude. The height represents the relative elevation between the pilot and the drone. This DTO is used in the frontend's detail view.

The information required for a `DroneDto` is typically spread across multiple packets. Therefore, business logic is needed to combine the relevant attributes. The abstract class `DroneService` defines the behavior that must be implemented for each standard.

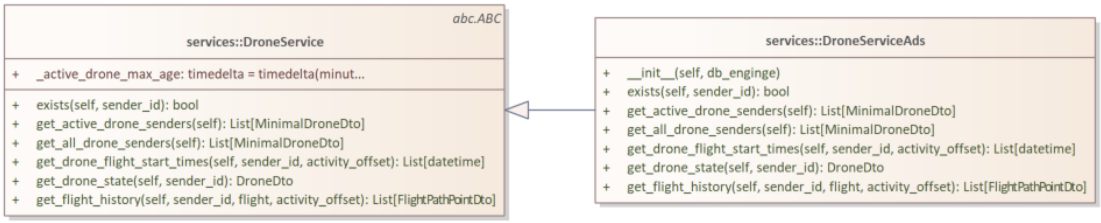


Figure 5.5: Drone Service - UML class diagram showing the abstract DroneService and one of its implementations.

The abstract class `DroneService` is located between the `DroneAPI` and the `Persistence Layer`. Each implementation is designed to support a specific protocol.

- **get_active_drone_senders** This method identifies drones that are currently active in the air. It returns a list of `MinimalDroneDto` instances for all drones that have broadcast a signal within a configurable time frame, defined by `_active_drone_max_age`. In the ASD-Stan implementation, this method queries location/vector messages (0x1).
- **get_all_drone_senders** This method retrieves all captured drones, including both active and inactive ones. It returns a list of `MinimalDroneDto` instances. Like the previous method, it queries location/vector messages (0x1) in the ASD-Stan implementation.
- **get_drone_flight_start_times** Flights are identified by their start time. This method queries all messages for a given drone and orders them by timestamp. A timestamp is considered the start of a flight if no message was received for a fixed prior time span.
- **get_flight_history** This method retrieves the flight path of a given drone. The user specifies a start time, and the method returns the subsequent positions of the drone. The flight is considered complete when no further packets are available, or a time gap greater than a fixed duration is encountered.
- **get_drone_state** This method provides detailed information about a drone. For the ASD-Stan implementation, it constructs a `MinimalDroneDto` from multiple underlying message types.

5.1.3 Drone API

The functions defined by the `DroneService` are exposed through the `DroneAPI`, which resides in the `Interface Layer`. To provide the best possible developer experience, all available API calls are documented using the `OpenAPI` standard. A dashboard that displays this functionality is available at `http://localhost:port/docs`.

Figure 5.6: API Documentation - Screenshot of the OpenAPI documentation dashboard for the standard agnositc endpoints.

Figure 5.6 highlights the APIs available to the frontend. Each of these endpoints corresponds to a function in the `DroneService`. For example, when querying all active drones via `/api/drones/active`, the `get_active_drone_senders` method is invoked on all underlying `DroneService` implementations.

Additionally, endpoints have been added to allow querying of the specific underlying messages for each standard. These are intended to support future projects by providing detailed access to raw message data. Figure 5.7 shows the endpoints for ASD-Stan specific messages.

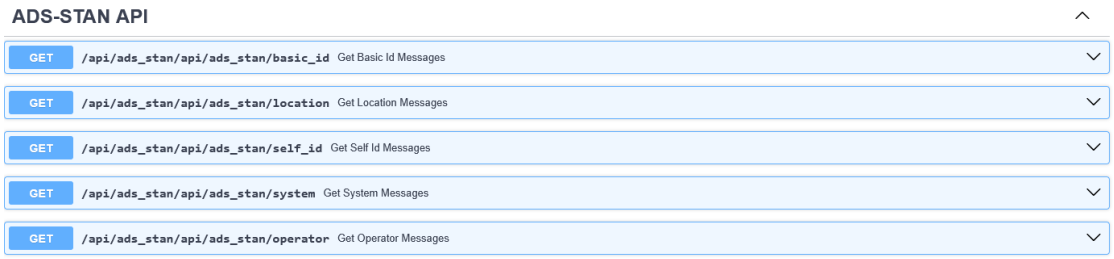


Figure 5.7: API Documentation - Screenshot of the OpenAPI documentation dashboard for the ASD-Stan endpoints.

Finally, Figure 5.8 shows the endpoint for the dji message.

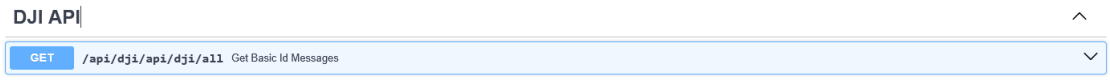


Figure 5.8: API Documentation - Screenshot of the OpenAPI documentation dashboard for the DJI-Standard endpoints.

5.1.4 Message Support

The previous sections described the persistence and retrieval of individual messages. However, the `DroneSniffer` component only provides raw Wi-Fi frames. To support the full ASD-Stan specification, most changes were made in the `PacketProcessor` component, which is responsible for transforming the binary payload of Wi-Fi frames into database models that can be persisted.

The `PacketProcessor` uses two subcomponents: the `Parser` and the `Mapper`. The parser converts the binary payload from a Wi-Fi frame into structured Python classes. The mapper then maps these classes into the database models defined in Section 5.1.1.

While it is technically possible to parse Wi-Fi frames directly into database models, doing so would couple the parsing logic to the persistence layer. This would reduce the flexibility and reusability of the parser in other contexts. By returning only the minimal necessary structured data, the parser remains decoupled and modular.

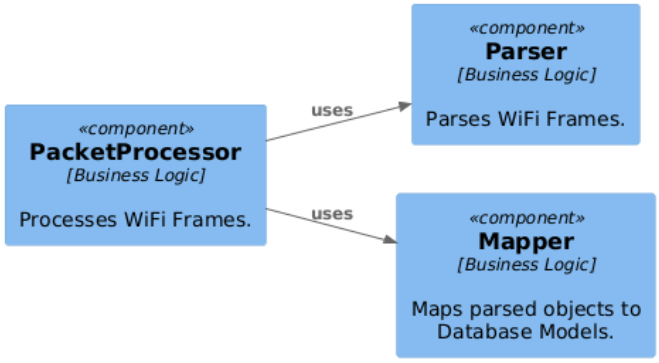


Figure 5.9: PacketProcessor Architecture - A C4 Component Diagram of the Remote ID PacketProcessor Component.

Abstract Parser Class

Central to the parsing logic is the abstract `Parser` class, which was originally defined in the previous project. Each implementation of this class supports one specific protocol. To determine which parser should be applied to a given Wi-Fi frame, each implementation defines a list of supported Organizationally Unique Identifiers (OUIs) (Organizationally Unique Identifiers). These identifiers are present in every Wi-Fi frame and indicate the organization that manufactured the drone.

The methods `extract_header` and `dec2hex` are helper functions inherited from the previous implementation.

Each concrete implementation of the `Parser` is expected to return instances of the abstract class `ParsedMessage`. This class also records which parser created the message—typically set to either `DJI` or `ASD-Stan` by the concrete subclass.

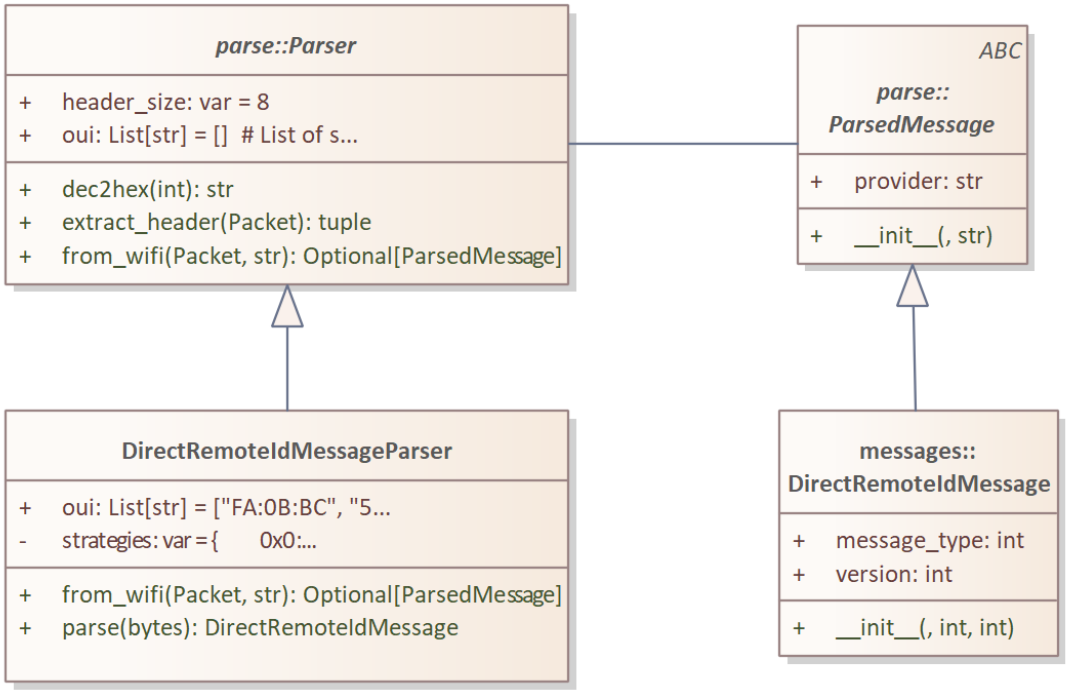


Figure 5.10: Parser Architecture - The abstract Parser class with its ASD-Stan implementation.

AST-Stan Implementation

The implementation `DirectRemoteIdMessageParser` lies in the `parse::asd_stan` package and is designed to support every ASD-Stan message. It overrides the method `from_wifi` which returns a `DirectRemoteIdMessage` from a WiFi-Frame. This subclass defines the common attributes of all possilbe ASD-Stan messages which are defined in Table 2.1.

In order to organize the parsing of different messages, a strategy pattern has been utilized. The `parse` function from the `DirectRemoteIdMessageParser` simply detects the message type and then delegates the parsing to the appropriate strategy. The following Figure illustrates how a BasicIdMessage can be parsed using its own `BasicIdParsingStrategy`.

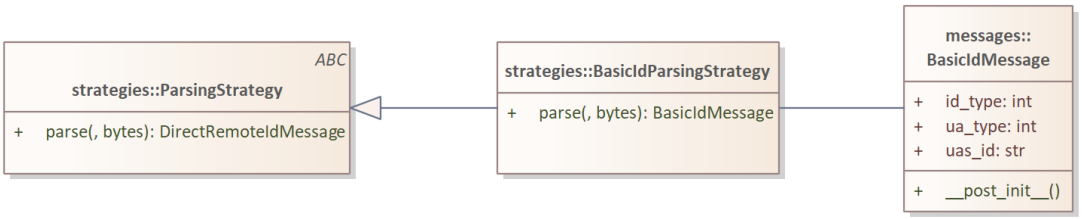


Figure 5.11: Parsing Strategies - An example implementation of the Parsing Strategy.

5.2 Performance

This section analyzes potential bottlenecks in the system and proposes solutions to improve performance. Previous work has identified three major issues that arise when the application is under load: First, the application occasionally drops packets if the `SniffManager` is busy. Second, the persistence layer throws errors when the database system becomes overloaded. Third, if too many drones are spoofed, the system crashes.

To further investigate these issues and identify bottlenecks, the following simplified sequence diagram illustrates the flow of a captured drone signal through the system's three layers.

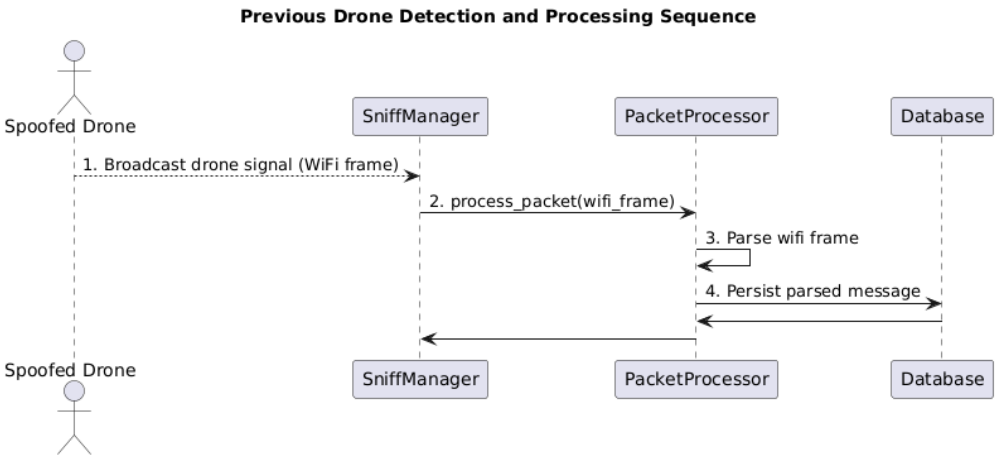


Figure 5.12: Overall information flow in the prototype: from sniffing a spoofed drone frame to saving the decoded message in the database.

Four main components interact with each other. The internal communication consists of the following steps:

1. **Signal Broadcast:** A spoofed drone broadcasts a signal in the form of a Wi-Fi frame.
2. **Signal Capture:** The `SniffManager` detects the frame and invokes a blocking processing function.
3. **Signal Parsing:** The `PacketProcessor` parses the frame into a structured message.
4. **Message Persistence:** The parsed message is stored in the database (blocking).

A major issue arises from this architecture: drone detection is tightly coupled with message processing and persistence. This is particularly problematic because the `AsyncSniffer` does not invoke the processing function in a separate thread. As a

result, while a message is being parsed or stored, the sniffer is unable to capture new packets. To prevent this issue, the sniffing and processing operations must be decoupled.

In addition, parsed messages are currently inserted into the database individually, with each insert executed as a separate transaction. This introduces significant transaction overhead, which can be reduced by using batch inserts (i.e., insert-many operations). This phenomenon has been demonstrated by experiments [27].

5.2.1 Decoupling Sniffing and Processing

Whenever a Drone Packet is captured, there will be processing power needed to parse and persist the message. In order to decouple the capturing from the processing, the **producer-consumer pattern** has been utilized. This concurrency pattern breaks the system into three distinct elements: a producer who is responsible for generating work, a blocking queue which holds the work that needs to be processed, and a consumer who is responsible for doing the work.

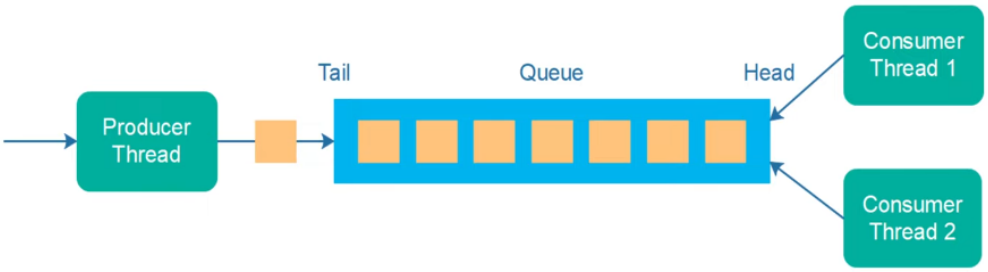


Figure 5.13: Producer–consumer design: the SniffManager fills a queue while multiple worker threads pick up packets for parsing. Visualized by Jenkov [3]

The `SniffManager` acts as a single producer. Whenever a drone signal is captured, the signal is placed in the queue but not immediately processed. The `PacketProcessor` acts as the consumer. Multiple worker threads can check the queue for tasks and process these frames independently.

This pattern offers multiple advantages:

1. **Decoupling** Packet capturing and processing are now decoupled. The `SniffManager` does not have to wait for the packets to be processed.
2. **Load Balancing** The packets can be processed by multiple consumers. This automatically distributes the workload across multiple threads.
3. **Backpressure Management** The application can be overwhelmed when the `SniffManager` captures more packets than the `PacketProcessor` can handle. By limiting the queue size, a maximum number of pending packets can be set. When the system captures more packets than it can handle, some

packets will be dropped. This is preferable to letting the system crash. However, determining the appropriate queue size is challenging, as it depends on the hardware.

By applying the producer-consumer pattern, two key problems are addressed. First, by limiting the queue size, the system avoids memory exhaustion and therefore prevents crashes. Second, the `SniffManager` gains significantly more time to sniff packets, as its callback function no longer waits for the processing and persistence steps to complete.

Python's concurrency library natively provides a `ThreadPoolExecutor` class that implements the producer-consumer pattern. However, this implementation does not allow the queue size to be limited, which means backpressure management is not guaranteed. To address this, a custom `LimitedThreadPoolExecutor` was designed. The `LimitedThreadPoolExecutor` is a specialization of `ThreadPoolExecutor` that replaces the internal queue with a bounded one.

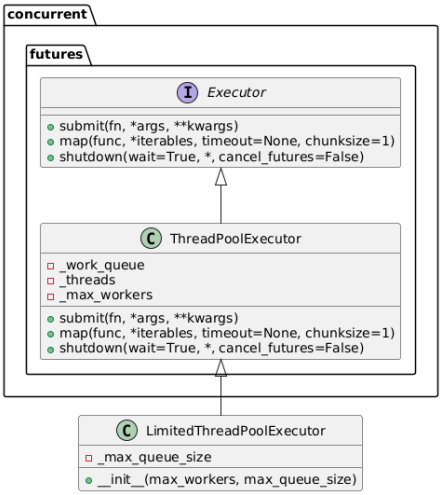


Figure 5.14: UML view of the LimitedThreadPoolExecutor highlighting the bounded queue that prevents overload.

The `LimitedThreadPoolExecutor` can be used just like the `ThreadPoolExecutor`. The only difference is that the maximum queue size can be defined via the constructor parameter `max_queue_size`.

Figure 5.15 further highlights that there is no direct communication between the `SniffManager` and the `PacketProcessor`. The `SniffManager` only submits the packets to the executor, which immediately returns. Processing is then performed by separate threads. The database component has been omitted from this diagram for simplicity.

By applying the producer-consumer pattern, two problems are addressed. First, by limiting the queue size, the system should avoid memory exhaustion and thus prevent crashes. Second, the `SniffManager` should have significantly more time to

sniff packets, as its callback function no longer waits for processing and persistence to complete.

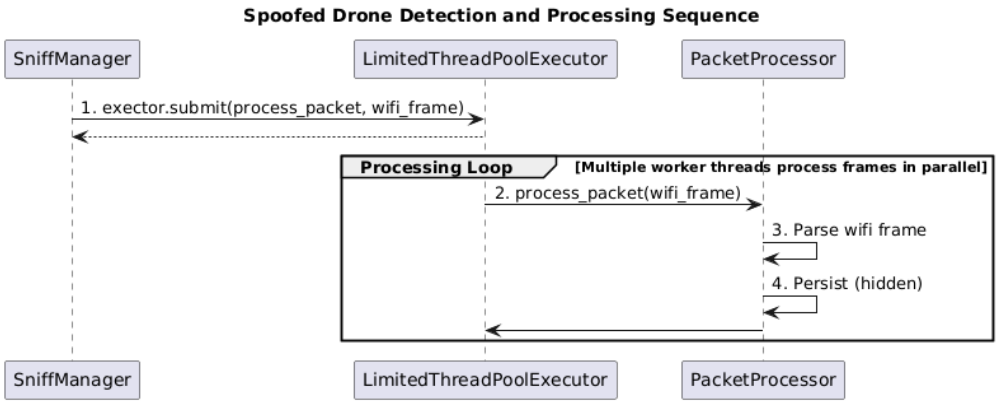


Figure 5.15: End-to-end handling of a single drone packet once the new threading setup is in place.

The trade-off of backpressure management is that messages will be ignored under heavy load. Unfortunately, there can always be a point where more messages are being sent than can be handled by the system. In such cases, it is more beneficial to ignore a certain portion of messages than to have a system that crashes.

5.2.2 Reducing Database Calls

To address the issue where the persistence layer crashes under pressure, a `TimeBuffer` has been implemented. The `TimeBuffer` component provides a mechanism for collecting and batching data over time intervals. By accumulating drone messages before persisting them, the system can leverage bulk insertion operations (e.g., `insertMany`) rather than individual inserts. This offers several key advantages:

1. **Decoupling** Packet processing and persistence are now decoupled. The `PacketProcessor` does not have to wait for the packets to be persisted.
2. **Reduced Transaction Overhead** Previously, each insert was committed individually. Now, a single transaction contains multiple inserted drone messages.

Assuming that bulk inserts are faster than individual inserts, this should result in fewer failures in the persistence layer.

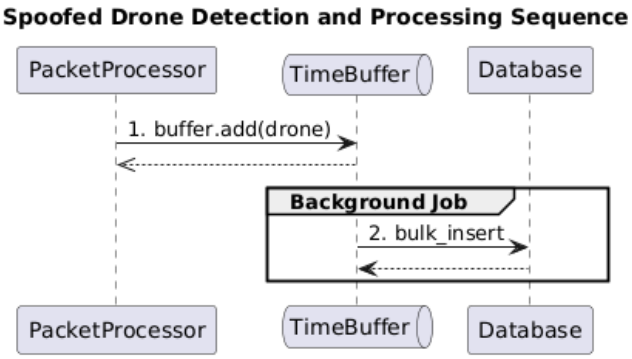


Figure 5.16: TimeBuffer sequence: packets are batched and flushed to the database at fixed intervals to enable bulk inserts.

The `PacketProcessor` can simply add the parsed drone messages to the `TimeBuffer`, without waiting for persistence. A separate background thread ensures that the added messages are written to the database at a fixed time interval.

The UML diagram in Figure 5.17 illustrates the internal structure of the `TimeBuffer` class. The `interval_s` parameter of the constructor defines the flush interval in seconds. The `on_flush` callback indicates what happens with the flushed data—in this case, a bulk-insert function.

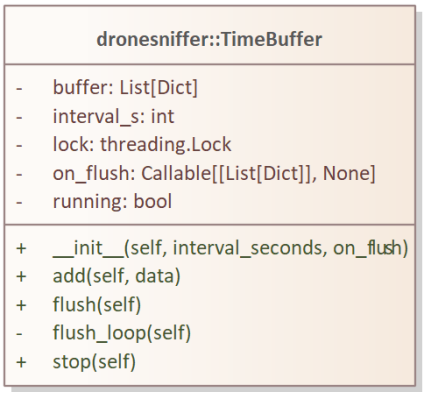


Figure 5.17: UML class diagram of the TimeBuffer showing its internal structure.

Because of its modular design, this component has also been used to reduce WebSocket operations. Instead of sending updates whenever a drone is parsed, it sends updates only at a fixed interval.

5.3 Frontend

To accommodate the increased performance requirements as well as offline capability, the frontend had to be almost entirely rewritten. As a positive side effect, all components are now designed to be mobile-friendly.

5.3.1 Frontend Architecture

Figure 5.18 illustrates the internal structure of the Remote ID Receiver frontend. It highlights the major software components and how they interact both internally and with external systems.

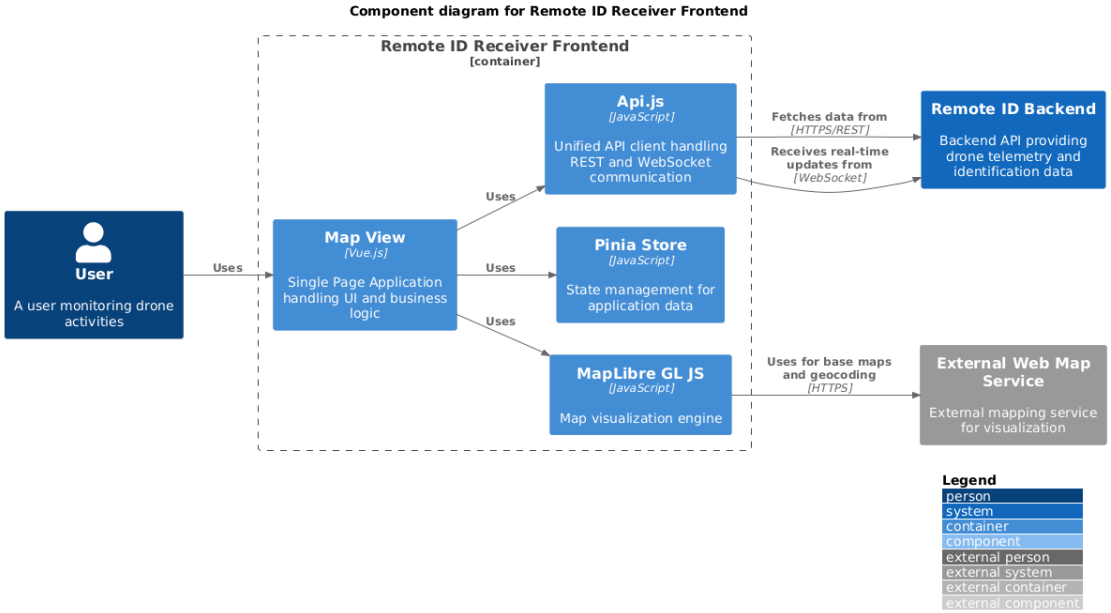


Figure 5.18: Frontend Architecture - A C4 Component Diagram of the Remote ID Frontend Container.

At the core of the frontend is a Single Page Application (SPA) developed using Vue.js. This architecture enables a responsive user interface without requiring full page reloads. The SPA serves as the main entry point for the drone operator, providing a real-time view of drone activity.

All communication with the backend is abstracted through a central module, `api.js`. This module handles both API requests and WebSocket connections. By encapsulating communication logic in a single file, the application achieves a clear separation of concerns. This setup also simplifies frontend development and testing by allowing easy mocking of backend responses.

To manage the application state, the project uses Pinia, a state management library designed specifically for Vue.js. Pinia acts as a centralized store, holding shared

data such as drone information and user preferences. This approach promotes a reactive and maintainable architecture by decoupling state from individual components.

For map visualization, the system uses MapLibre GL JS, an open-source mapping library that renders interactive maps using WebGL. This component is responsible for displaying drone positions, flight paths, and related geographical information. The rationale behind selecting MapLibre is discussed in the next Section.

5.3.2 Selection of the Map Library

Before this project, the application used the `vue3-google-map` library to provide map functionality. At the time, this was a reasonable decision because it was easy to integrate and worked well with Google Maps. However, major limitations became clear during development: Google Maps does not support offline caching for its web library [24]. Furthermore, the library is completely restricted to Google Maps and does not allow the use of other map sources. Since offline functionality is now considered an important requirement, it became necessary to look for an alternative that would allow for caching or support for different map sources, including self-hosted ones.

Three open-source libraries were considered as alternatives: Leaflet, OpenLayers, and MapLibre GL JS. These were compared based on key criteria, including rendering engine, support for vector and raster tiles, and their suitability for offline and real-time use cases.

One important technical requirement was support for **WebGL**, which allows efficient rendering of large amounts of data and provides better performance for animations and real-time updates. Since the map component is expected to display many moving drones simultaneously, using a WebGL-based renderer helps reduce CPU usage and ensures smoother performance in the browser.

Another important aspect was **vector tile support**. Compared to raster tiles, vector tiles are more lightweight and flexible. They allow styling on the client side and reduce the load on the backend server, which is particularly important because the map service is hosted locally and needs to serve tiles efficiently.

	Leaflet	**Maplibre GL JS**	**OpenLayers**
Raster Tile Support	yes	yes	yes
Vector Tile Support	supported via external plugins	yes	yes
Rendering Engine	DOM + Canvas	WebGL	Canvas / WebGL

Table 5.1: Comparison of Map Libraries

While Leaflet is very lightweight and arguably the most popular option, it lacks support for vector tiles and does not use WebGL for rendering. This makes it less suitable for high-performance, real-time applications like drone monitoring. OpenLayers, on the other hand, is very powerful and flexible, but it comes with a steeper

learning curve and more complex API. In the end, **MapLibre GL JS** was chosen because it offers a good balance between performance, modern features, and flexibility.

5.3.3 Benchmarking of the Map Library

Before starting the implementation, a benchmark was conducted to evaluate the performance of MapLibre GL JS under realistic conditions.

Test Scenario:

- Rendering static icons for drones, including their flight paths.
- Simulating 10,000 drones, each updating its position every 3 seconds, based on the ASD-Stan protocol.
- Let the drones update their position for 10 minutes.

Environment: IPhone 15 Pro.

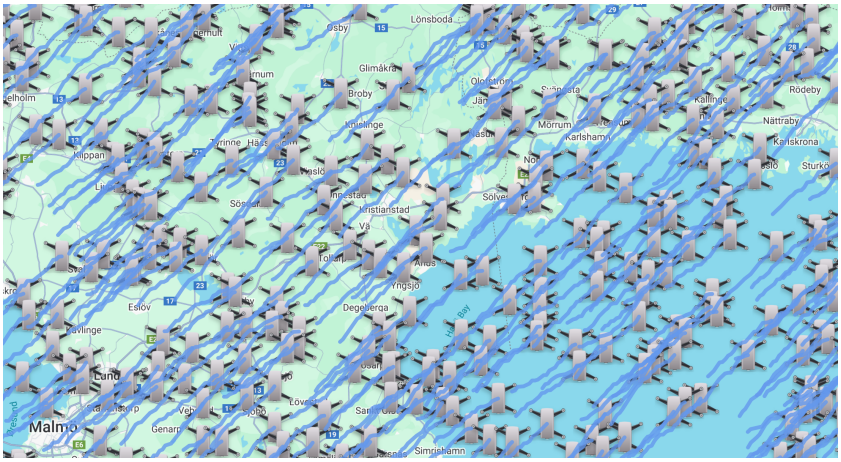


Figure 5.19: Benchmark of MapLibre GL JS – Displaying 10 000 live drones updating their position every 3 seconds.

Results: The system performed fluently and displayed no signs of lag while interacting with the map, such as during drag operations. This demonstrates that MapLibre GL JS is capable of handling high-volume, real-time rendering scenarios efficiently, making it well-suited for the needs of this project.

5.3.4 Visualization Strategies

The frontend is required to support two distinct monitoring modes: live monitoring and replay. While the replay mode focuses on displaying a single past flight, the live mode must visualize all currently available drones in real time. Additional complexity

arises from the ability to individually toggle pilot markers, home positions, and drone paths on or off.

To manage this complexity effectively, a strategy pattern has been implemented. This design allows the application to return only the relevant data needed for visualization based on the current mode and user settings. Although JavaScript does not support interfaces in the traditional sense, an interface is included in the UML diagrams to clarify the design intent.

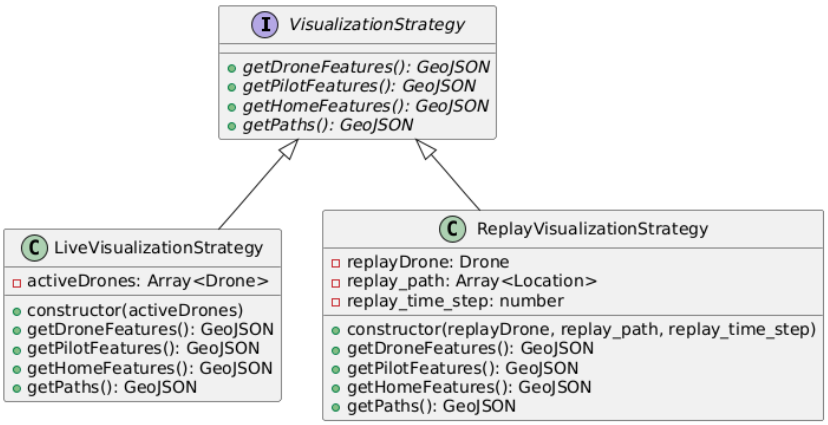


Figure 5.20: Visualization Strategy – An UML diagram of the visualization strategy.

5.3.5 Render Loop

Building on the strategy pattern, a continuous render loop is used to regularly fetch the most up-to-date data and pass it to the mapping library. Although re-rendering the map in response to individual drone events might be more efficient for a small number of drones, this loop-based approach offers better scalability. By decoupling rendering from individual events, the system ensures smooth updates even when tracking a large number of drones simultaneously.

```
1  const render = () => {
2
3      const strategy = mapStore.getVisualizationStrategy()
4
5      map.value.getSource('path-source').setData(strategy.getPaths()
       )
6      map.value.getSource('drones-source').setData(strategy.
       getDroneFeatures())
7      map.value.getSource('pilots-source').setData(strategy.
       getPilotFeatures())
8      map.value.getSource('homes-source').setData(strategy.
       getHomeFeatures())
9
10     requestAnimationFrame(render)
11   }
12
```

To implement the render loop efficiently, the requestAnimationFrame function is used. This method synchronizes the rendering process with the browser's refresh rate, resulting in smoother visual updates and better performance.

5.4 Offline Web Map

This section higlights how an offline web map server has been integrated. The server is used to provide the map to the fronted and is hosted on the same device as the backend. This process is divided into three different steps:

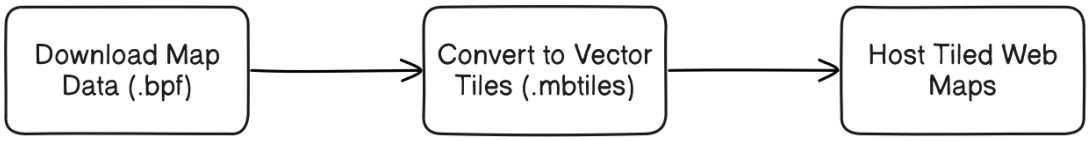


Figure 5.21: The process of creating a Tiled Web Map from OpenStreetMap data.

5.4.1 Step 1: Download Map

The first step is to download map data. This data originates from the OSM project, which provides detailed geographic information in the form of .pbf (Protocolbuffer Binary Format) files. This format is intended as a faster alternative to a gzipped XML format [28]. A map of the whole planet can be stored in a single file of approximately 80 GB. These files are region-specific and can be downloaded based on the required area of coverage (e.g., a country or city) [29].

For this project, PlanetOSM has been chosen as the map provider. PlanetOSM is funded by the OpenStreetMap Foundation and provides a copy of all the street map data. The data itself is updated weekly and therefore reflects the latest community changes. The specific file used to serve the whole globe can be found under `Files > pbf > planet-250519.osm.pbf` [29].

However, it is important to note that this file is not yet a Tiled Web Map. In order to create a tiled web map, the file must be converted into an appropriate format.

5.4.2 Step 2: Convert to Vector Tiles

Once the raw data is downloaded, it needs to be processed into vector tiles. The MbTiles format describes how a Tiled Web Map, also called a Tile Set, can be stored in a single file. It is based on SQLite. The standard mainly defines how data must be stored and accessed [30].

Tilemaker [31] is an open-source tool that can convert from the PBF format to the MbTiles format. It transforms the large binary file into multiple smaller vector tiles

that can then be served by a Tile Server. The main advantage of Tilemaker is its ease of use, particularly through its Docker image.

The command used to generate the vector tiles was:

```
docker run -it -rm -v $(pwd):/data ghcr.io/systemed/tilemaker:master
/data/planet-250519.osm.pbf -output /data/globe.mbtiles
```

5.4.3 Step 3: Host Tiled Web Map

The final step is to serve the generated vector tiles via an HTTP server. Tileserver GL [32] is a map server that can be used to host and serve these tiles to the frontend. The server runs locally and exposes endpoints that the frontend map library can consume.

The command used to host the server was:

```
docker run -rm -it -v $(pwd):/data -p 8080:8080 maptiler/tileserver-gl:latest
-file globe.mbtiles
```

5.4.4 Results

MapTile generates an endpoint to inspect the map data at `localhost:8080`. The following screenshot provides a close inspection of the map. The map displays all the necessary details for drone monitoring: forests, lakes, rivers, streets, and buildings, as well as street and city names.

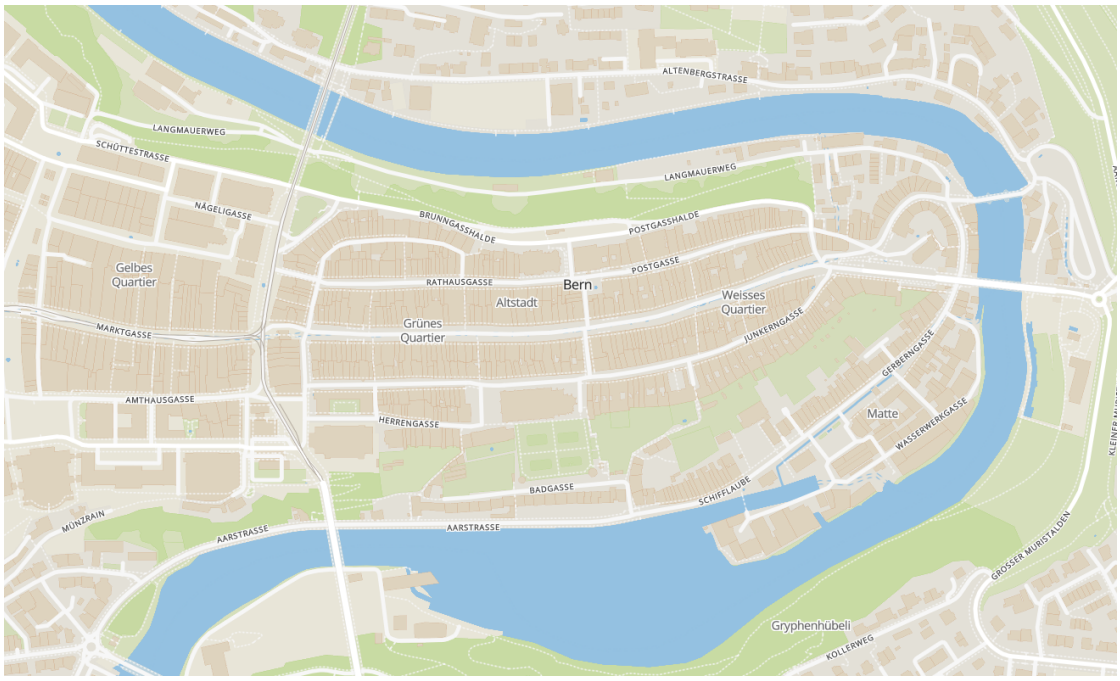


Figure 5.22: The resulting map displayed hosted by TileServer GL.

5.5 Proposing a scalable architecture

One crucial idea for future work is to implement a system that provides support for multiple receivers. Those receivers could then be placed around Switzerland, near critical infrastructure. Such a system would allow for large-scale monitoring of compliant drones. However, implementing such a system introduces new challenges. Therefore, this section addresses those challenges and proposes a scalable architecture that can be distributed over multiple devices.

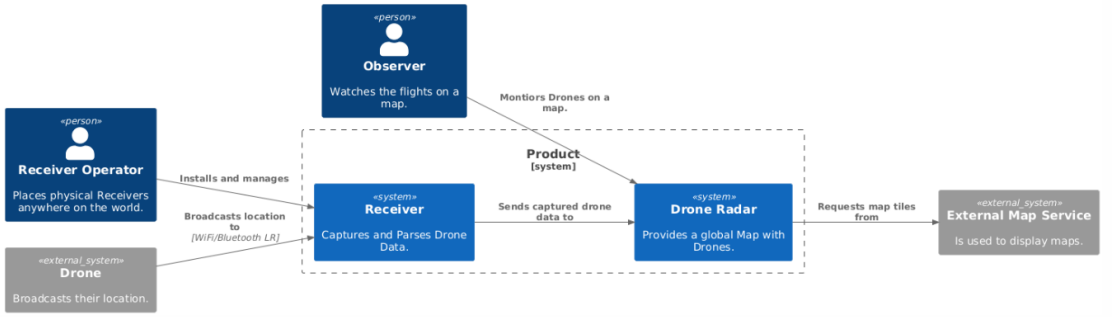


Figure 5.23: System Overview – A high-level view of the proposed architecture.

The new system consists of two distinct parts: the receiver and the drone-radar. The receiver can be placed anywhere and is designed to capture Remote ID messages. Those messages are then relayed to the drone-radar. The radar then provides an interface for drone observers to monitor the active drones.

One challenge is the scalability of the system. Estimating the number of drones that have to be supported is a difficult task. Just last year, the Chinese company DAMODA set a world record by flying 10 179 drones simultaneously [33]. Given the rapid adaptation of drones it makes sense to be able to distribute the workload of the drone-radar over multiple machines in order to support as many drones as needed. This can be done by employing an event-based architecture. The domain model in Figure 5.24 visualizes the key events needed to visualize drones on a map.

Whenever a receiver sends information of a drone to the drone-radar the event `drone.detected` gets fired. The event contains the minimal amount of data required to display a drone: the ID of the drone and its operator, the ID of the receiver and the position of the drone. These events are emitted at a high rate and may contain conflicting information: what happens if multiple receivers report different locations for one single drone?

To resolve such conflicts and ensure consensus, a new event `mapstate.updated` is introduced. This map state contains all data necessary to display drones on the map. The key concept is that each drone appears only once, with a single position. This state must be computed in near real-time and updated at regular intervals. In order to find consensus, several algorithms can be employed. For example, majority voting.

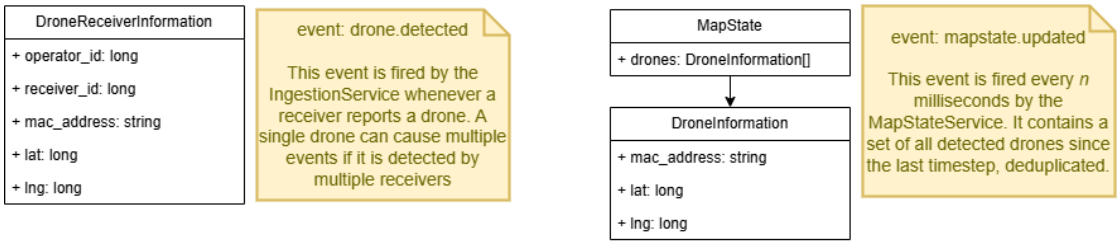


Figure 5.24: Domain Model – Minimal events required to monitor drones.

From this domain model, a microservice architecture for the `drone-radar` can be established. The existing `Remote ID receiver` can be reused to capture drone messages. Those messages can then be sent to an **ingestion service** which provides an interface for receivers to report drone messages. These messages are then transformed into the `drone.detected` event and placed in a `message-bus`. Several instances of this service can be deployed.

The **map state service** listens to this event stream and calculates the `map-state` at a specific interval. There are multiple frameworks to perform stateful computations over event streams. Apache Flink [34] is an open-source processing engine that allows the distribution of this process over multiple machines. Whenever a `map-state` is calculated, the `mapstate.updated` event gets emitted.

The **radar service** serves the frontend to the end user. Whenever the `map-state` gets updated, the frontend gets notified over a web-socket. The current implementation of the frontend can be served for this implementation, with little adjustments for more features like drone-details.

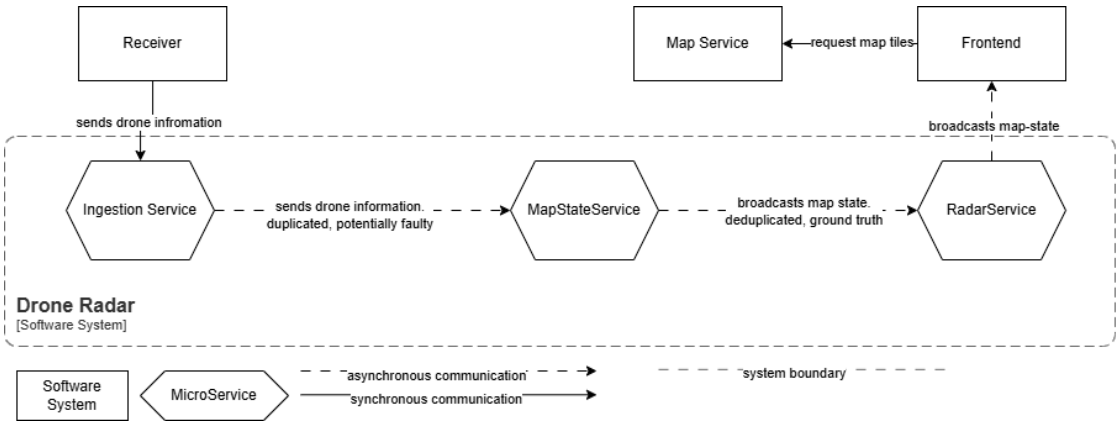


Figure 5.25: Microservice Architecture – A diagram showing the microservices and the communication between them.

This architecture is designed to be a first step towards a nationwide drone monitoring platform and supports multiple receivers and standards by design. A first prototype of the map state service can be implemented and evaluated to estimate the feasibility and performance of the system. At the same time, the architecture allows for extensions for future projects, like more advanced spoofing detection.

Chapter 6

Evaluation and Validation

6.1 Code Quality

In order to gain objective feedback for the code quality, various tools have been utilized. SonarQube was used to detect various issues regarding security, reliability or maintainability. It is important to note that these scans have only been performed on the backend of the application. This decision has been made because bugs in the frontend are often visual in nature and can be better detected by manual system tests.

The code coverage has been measured using pytest in combination with the pytest-cov plugin. To automate this process and ensure consistent quality checks, a continuous integration pipeline was established using GitHub Actions. This setup enables automated testing and analysis every time new code is pushed to the repository or a pull request is created. The workflow includes steps for installing dependencies, running unit tests and checking code coverage. However, SonarQube runs have to be run locally on the users machine. This decision has been made because of the monetary nature of SonarQube cloud.

A detailed coverage report is provided in the appendix. The newly developed `ASD-Stan parsing` component achieves a line coverage of 92.5%, while the `DroneService` component reaches 94.6%. The primary testing focus was placed on the Business Logic Layer, which is reflected in the high coverage values. In contrast, the Sniffing Layer which was not worked on in this project shows a coverage of 0%, as it consists of boilerplate code used to integrate the scapy Library. Furthermore this component will automatically be tested in the acceptance test in Section 6.2. Therefore unit testing was deemed unnecessary for the current scope.

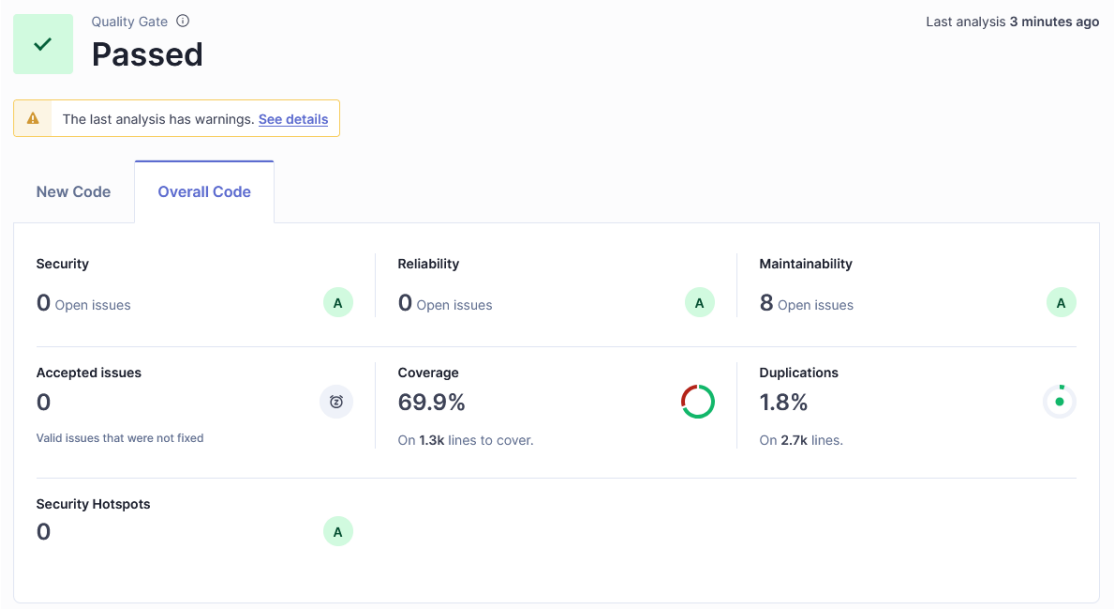


Figure 6.1: Code Quality - A report for the code quality issued by a local Sonar Qube instance.

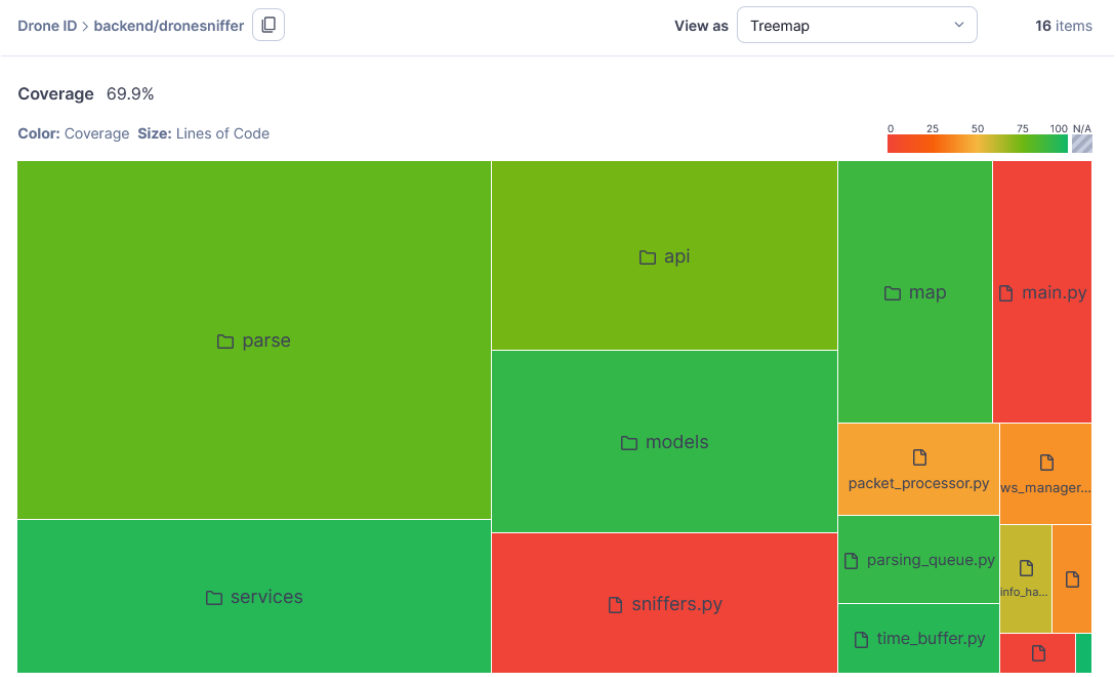


Figure 6.2: Code Coverage - A report for the code coverage issued by a local Sonar Qube instance.

In total 139 individual unit- and integration tests were executed.

6.2 Functionality Tests

All user stories had passed the Definition of Done by the end of the project. To fully verify the functionality of the final application, acceptance tests were conducted. The aim of these tests was to replicate the system's behavior under realistic conditions without internet access.

6.2.1 Realistic Scenario

The first test was designed to replicate a realistic usage scenario as closely as possible, without relying on an actual drone. It was carried out using a fake drone running on a separate system that was completely isolated and shared no network or connection with any other systems.

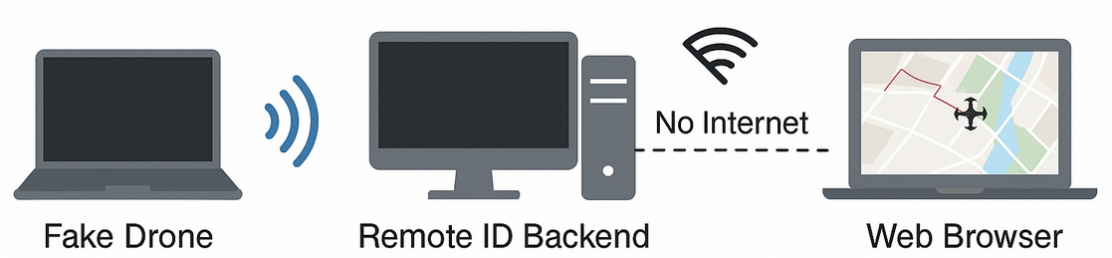


Figure 6.3: Realistic Scenario - A Fake Drone broadcasting real drone traffic over a USB WiFi Network-Adapter. There was no physical or virtual connection between Drone and backend. The backend and Web Browser shared the same network without internet access in order to communicate. AI-Generated Image by ChatGPT [4].

The following systems were installed:

- **Fake Drone** – A laptop was set up to replay Wi-Fi traffic that had been captured from a real drone compliant with the ASD-Stan standard. The recording files were provided by previous students. This system was equipped with a network card capable of **packet injection** in order to broadcast the traffic. However, it had no internet connection.
- **Remote ID Backend** – The Remote ID backend was installed on a separate physical computer with the goal of capturing the network traffic broadcast by the fake drone. This system had a network card capable of operating in **monitor mode**. It also included an offline map instance.
- **Web Browser** – A third system was connected to the backend via the web interface, with the goal of viewing the captured drone on the map.

Result

The realistic scenario was successfully tested and demonstrated its value. Initially, the test failed due to incompatible hardware. However, after switching to the network adapter described in Section 3.1.4, the issue was resolved and the test completed successfully. Given the strong dependency on this specific network card, a recommendation for a compatible adapter and driver has been included in the testbook. Figure 6.4 illustrates the drone's path, which closely aligns with the actual recorded flight trajectory.

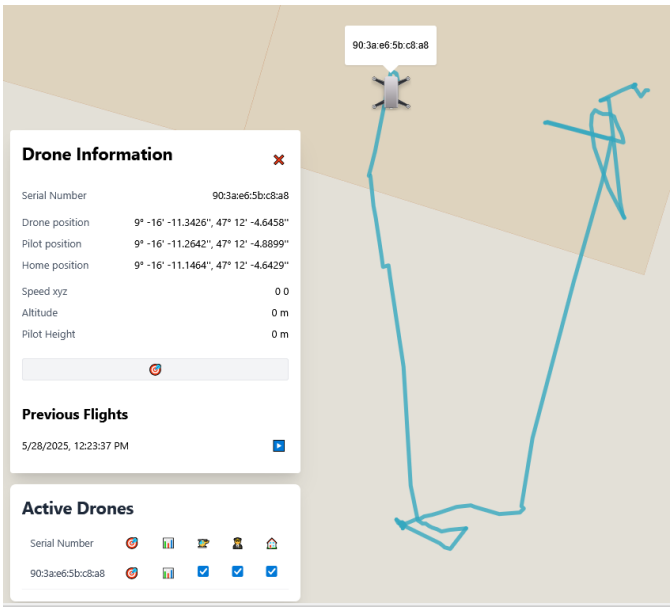


Figure 6.4: Acceptance Test - A screenshot demonstrating the successful outcome of the first acceptance test.

The path displayed on the map shows where the real drone had reported its position. The flight path is located in Zürich, where the previous thesis had been written. Therefore, the location of the track seems plausible. A key detail is that the drone reports zero altitude and height at the start and end of the flight. This could be due to two reasons: either the drone was carried through the building by hand, or the drone reported incorrect data. The parsing logic appears to be correct because the altitude did change in the middle of the flight when the drone was in the open.

```
Starting to read and send packets from parrot_anafi4_real.pcapng on interface wlxec750c53433
b...
Found 5091 packets to send...
Sending packets:  13%|                                  | 666/5091 [01:04<06:44, 10.95it/s]
```

Figure 6.5: Replay Script - Output of the Replay Script which is used to reproduce the WiFi frames broadcast by a real drone.

In order to make testing easier, a replay script which replays real drone traffic has been implemented.

6.2.2 Lab Scenario

The second test scenario aims to assess the application's ability to handle multiple simultaneous drone signals, simulating conditions of high drone traffic. A drone spoofer provided by the CYD Campus was used to generate multiple drone signals to evaluate system responsiveness. It consisted of four systems:

- **Drone Spoofer 50** This spoofer is utilized to simulate traffic of 50 individual drones. Location/vector messages are sent once per second for every drone.
- **Drone Spoofer 1** This spoofer is used to test the responsiveness of the system. A single drone is simulated and fully controllable via a keyboard.
- **Remote ID Backend** The Remote ID backend was installed on a separate physical computer.
- **Web Browser** A Web Interface with the goal of viewing the captured drones on the map.

Result

The lab scenario for spoofing multiple drones was executed successfully. Controlling the spoofed drone felt smooth and the position updated immediately. Logs in the backend indicate that around 155 individual messages get persisted every second. The spoofer sends updates for each drone every second. Each update consists of three individual Remote ID messages. The persisted 155 messages per second on the backend sound plausible.

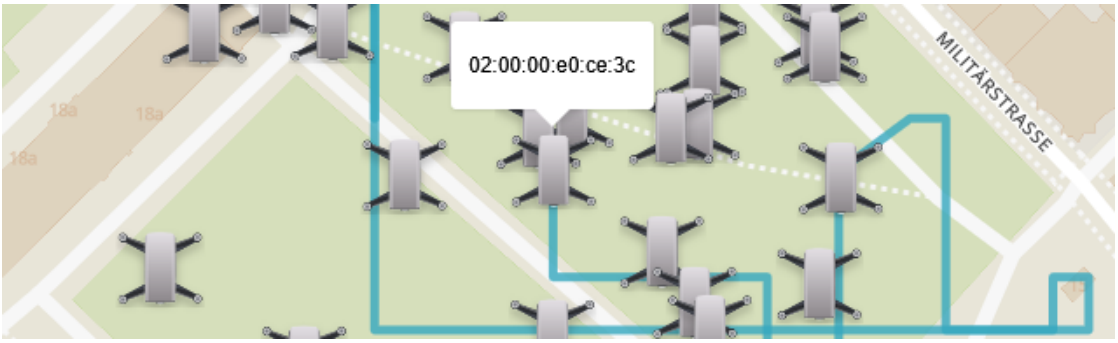


Figure 6.6: Acceptance Test - A screenshot demonstrating the successful outcome of the second acceptance test. The frontend display 51 simulated drones. One of them is controlled manually and can be used to verify the systems responsiveness.

6.2.3 Resilience

To evaluate system resilience, the Lab Scenario was adapted. A manually controlled drone continued to test system responsiveness, while a single drone spoofer flooded the backend with an unmanageable volume of messages. This test specifically assessed the effectiveness of the `LimitedThreadPoolExecutor` introduced in Section 5.2.1. The spoofer initially simulated 100 drones, with the load gradually increased manually.

At first, the system handled the load well. Signs of lag began to appear around 800 spoofed drones. The manually controlled drone's movements became less fluid, with occasional stuttering. Despite this, the system remained usable, responding to commands with a maximum delay of approximately 200ms. At 1000 spoofed drones, performance noticeably degraded, with delays of about one second. This amounts to around 3000 persisted messages per second. When the spoofed load reached 5000 drones per second, the system became unusable. Although the frontend still displayed drones, the controlled drone only responded once every 30 seconds.

By increasing the drones to 100 000 drones, the system did not crash. It showed resilience by simply ignoring the messages as expected. While the system did not crash and still persisted messages, the frontend became unusable at such volume. This represents a major issue. In order to recover from such an event, the user has to wait a certain amount of time until the drones are not marked active anymore. This might be addressed by implementing clustering techniques.

6.3 Results

The next two figures show the new frontend. A few elements have been altered to ensure mobile-friendliness. The list of all drones has been moved to a sidebar, which can be opened and closed via a button. The settings menu can also be toggled using the button in the top-left corner. The list of active drones and the drone information panel now share the same position in the bottom-left corner. This layout provides more space for drone monitoring.

The drone information panel now displays additional data: a serial number, an operator ID, a UAV type, and the flight purpose. Arguably, the most valuable attribute is the operator ID. This identifier enables authorities to verify the pilot's identity without requiring direct communication.

The background map is fully interchangeable, supporting both vector and raster tiles. This demonstrates that using services like Google Maps remains possible in the new version. Drone technicians can easily switch map providers by modifying an environment variable in a dedicated configuration file. This approach offers ease of use for straightforward applications, while also enabling the integration of offline maps for more technically advanced users.

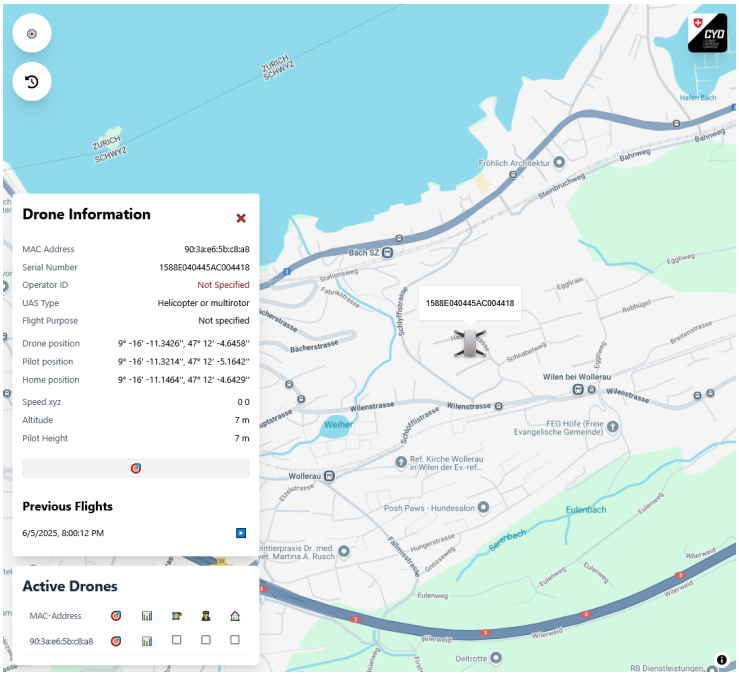


Figure 6.7: Live View - A screenshot demonstrating the live monitoring feature.

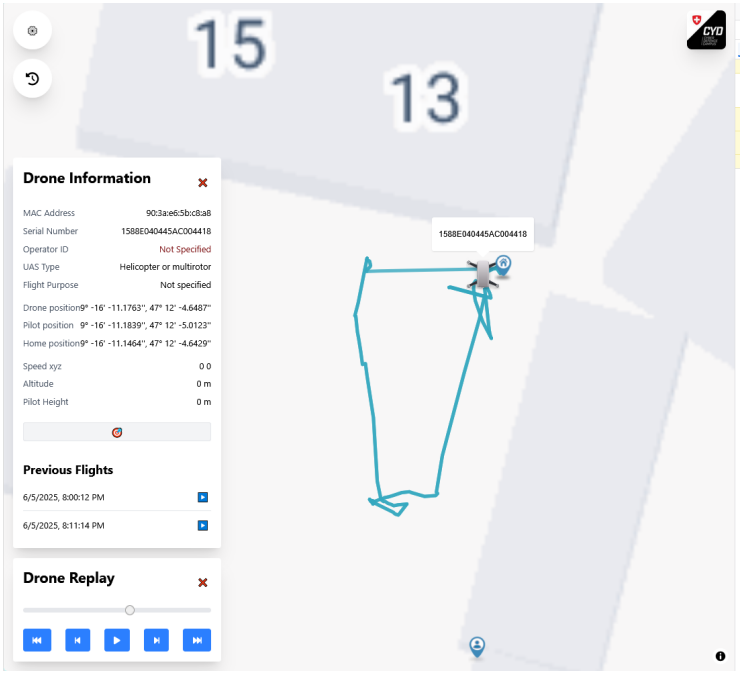


Figure 6.8: Replay View - A screenshot demonstrating the replay monitoring of past flights.

Chapter 7

Conclusion

This concluding chapter reflects the development process, summarizes the key limitations and lays out potential future improvements.

7.1 Reflection

The goal of this work was to enhance the Drone Monitoring Application by introducing additional functionalities. This thesis successfully extended the system's capabilities while also improving its robustness and reducing its reliance on online services. The objective of enabling offline operation has been fully achieved. The backend now captures a broader range of information, which is effectively displayed on the frontend and can be valuable for authorities. An intuitive API further supports the integration of external applications. While the system is now more resilient and performant, there remains potential for future improvements.

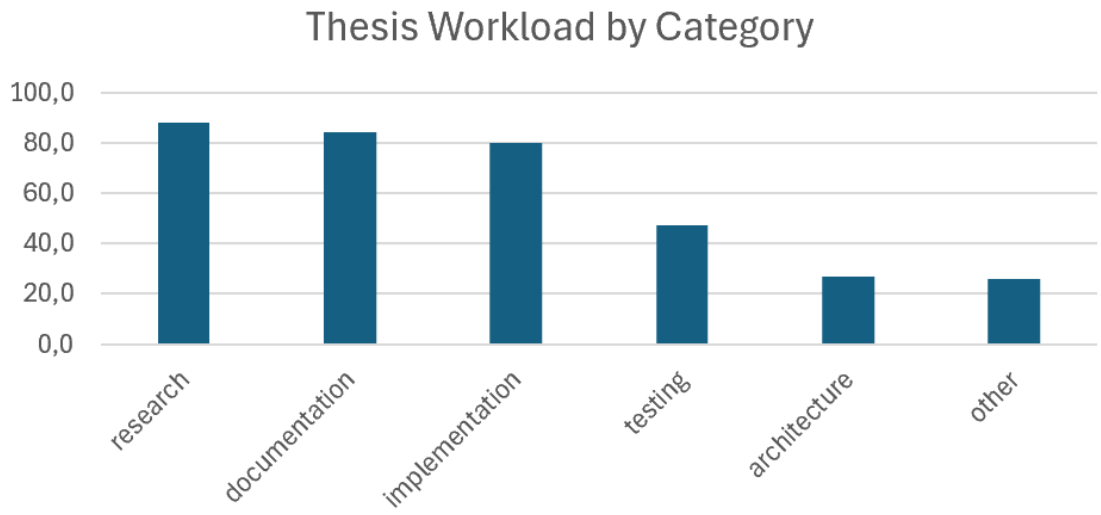


Figure 7.1: Thesis Workload by Category - Time effort broken down into distinct categories to reflect on the final outcome.

Using Scrum with well-defined user stories and epics was very effective. Organizing the work into sprints supported regular testing and provided a helpful structure for the weekly meetings with the product owner. These meetings kept the project focused and ensured that clear goals were followed throughout. In total, I spent 352 hours on the project, which is close to the expected 360 hours.

Figure 7.1 shows that most of the time was spent on research. This was expected, as a solid understanding of Remote ID and mapping libraries was essential. Surprisingly, writing the report took the second most time. Compared to earlier projects, which were more technical but narrower in scope, this thesis required broader thinking and more effort to clearly explain the system. Due to the page limit, some early documentation had to be left out. For a future thesis, I would focus on fewer features with more theoretical detail. Testing and architecture took the least time, which was also expected. Time spent on architecture is hard to measure because it often involved refactoring, which is difficult to track.

To manage the project effectively, the total effort was estimated for each user story. In most cases, the actual effort slightly exceeded the initial estimates. Nevertheless, the time estimations proved valuable for time-boxing research and development activities.

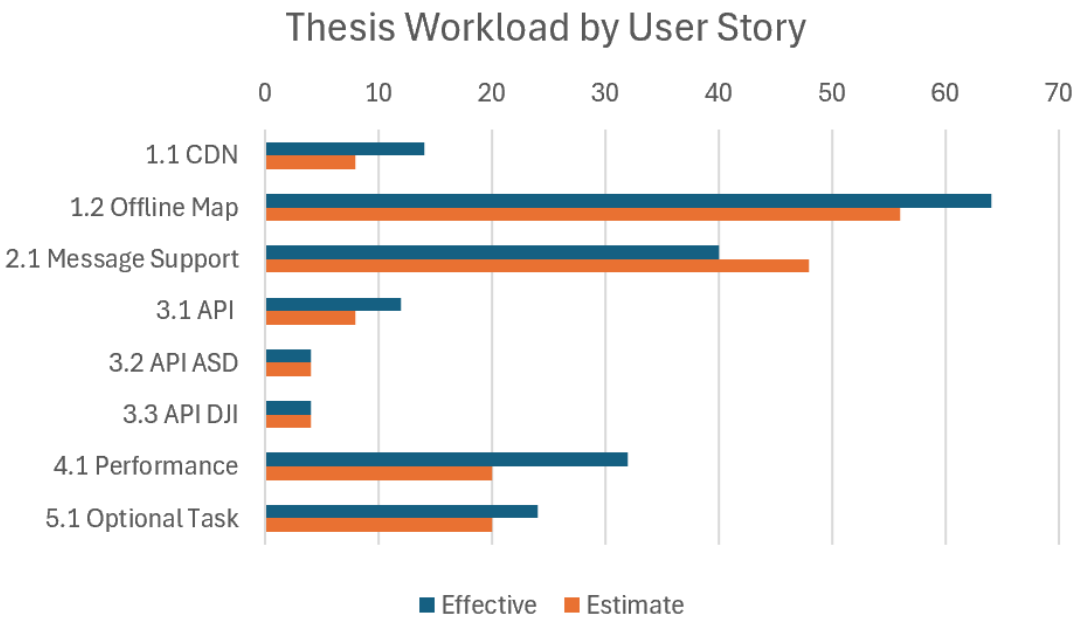


Figure 7.2: Thesis Workload by User Story, excluding time spent for documentation.

7.2 Limitations

This section outlines the key limitations of the current system. The following points list constraints that users and developers should consider when working with the

system:

- **Compliance Required** The current system only detects drones that are fully compliant with either the DJI or ASD-Stan standard. Although support for other standards can still be implemented, the system will not recognize drones that have been manipulated to avoid transmitting Remote ID data. This limitation is very hard to overcome and would require more sophisticated approaches.
- **Monitor Mode Required** In order to sniff traffic, a network card that supports monitor mode is required. Not all WiFi adapters support this functionality.
- **Packet Injection Required** In order to test the application, a network card that supports packet injection is required. Not all WiFi adapters support this functionality.
- **Developed for Linux** The application was originally developed to run on a Raspberry Pi and is therefore centered around Linux. When developing in Windows Subsystem for Linux (WSL), additional work is required to access the network interface.

7.3 Further Improvements

There are additional approaches that could extend the current system. This section explores potential enhancements that could broaden the system's capabilities and improve its practical applications:

- **Additional Transport Protocol Support** The ASD-Stan standard supports four transmission methods: WiFi 2.4 GHz, WiFi 5 GHz, Bluetooth Low Energy, and LTE. However, the current implementation of the drone sniffer does not support Bluetooth. According to the specification, drones are only required to broadcast Remote ID over one transmission method. This means that a compliant drone might still go undetected if it only broadcasts over Bluetooth. To overcome this issue, an additional Bluetooth sniffer could be implemented.
- **Additional Standard Support** The ASD-Stan standard is not the only one that defines how drones can broadcast information. Future contributors could extend the functionality to include broader standard support.
- **Multi-Receiver Support** Currently, the system can only be used with one receiver. A distributed system could be developed to allow monitoring over multiple physical locations.
- **Limit Drones for Frontend** The frontend becomes cluttered and unresponsive when a large amount of drones are detected at a single location. Implementing clustering techniques to group large amounts of drones together would reduce visual clutter and increase performance.

7.4 Declaration of Use: AI Tools

This document has been created with the assistance of ChatGPT, an artificial intelligence language model developed by OpenAI. The model helped to refine the language used in this text, to fix spelling errors and grammar mistakes and to generate the declared images. Additionally, GitHub Copilot was used to generate some code artifacts, although its main purpose was to ensure that the code is adequately commented. The ultimate responsibility for the content of the code and report remains with the author.

Bibliography

[1] P. Group, "Ptv xserver map tile api documentation." https://xserver2-dashboard.cloud.ptvgroup.com/dashboard/Content/TechnicalConcepts/Rendering/DSC_Map_Tile_API.htm, 2025. Accessed: 2025-05-22.

[2] TP-Link Technologies Co., Ltd., "Archer t2u plus – ac600 high gain dual band usb adapter." https://www.tp-link.com/ch/home-networking/adapter/archer-t2u-plus/, 2025. Accessed: 2025-05-31.

[3] J. Jenkov, "The producer consumer pattern." https://jenkov.com/tutorials/java-concurrency/producer-consumer.html, Apr. 2021. Accessed: 2025-06-01.

[4] OpenAI, "ChatGPT." https://chat.openai.com/, 2025. Accessed: 2025-06-01.

[5] Der Bundesrat, "Drohnen in der schweiz: Bundesrat verabschiedet postulatsbericht." https://www.news.admin.ch/de/nsb?id=103652, Dec. 2024. Zugriff am 30. Mai 2025.

[6] Swiss Life, "Drohnen über dem grundstück: Das sind ihre rechte." https://www.swisslife.ch/de/private/blog/immo/drohnen-eingriff-in-die-privatsphaere-verboten.html, Mar. 2025. Zugriff am 30. Mai 2025.

[7] SWI swissinfo.ch, "Drone incidents on the rise." https://www.swissinfo.ch/eng/society/airspace-risk_drone-incidents-on-the-rise/44255130, July 2018. Accessed on May 30, 2025.

[8] Federal Aviation Administration, "Remote identification of drones." https://www.faa.gov/uas/getting_started/remote_id, 2025. Accessed on May 30, 2025.

[9] European Union Aviation Safety Agency, "Remote identification will become mandatory for drones across europe." https://www.easa.europa.eu/en/document-library/general-publications/remote-identification-will-become-mandatory-drones-across, Nov. 2023. Accessed on May 30, 2025.

[10] Bundesamt für Zivilluftfahrt (BAZL), "Offene kategorie." https://www.bazl.admin.ch/bazl/de/home/drohnen/open1.html, 2025. Zugriff am 30. Mai 2025.

[11] Cyber-Defence Campus, "Remoteidreceiver: Web application to monitor drones based on remote id technology." https://github.com/cyber-defence-campus/RemoteIDReceiver, 2025. Accessed on May 30, 2025.

[12] European Union Aviation Safety Agency, "Operating a drone." https://www.easa.europa.eu/en/domains/drones-air-mobility/operating-drone, 2025. Accessed: 2025-05-31.

[13] Bundesamt für Zivilluftfahrt, "Die drei kategorien." https://www.bazl.admin.ch/bazl/de/home/drohnen/anfaenger2/categories.html, 2025. Accessed: 2025-05-31.

[14] ASD-STAN, "Aerospace series – unmanned aircraft systems – part 002: Direct remote identification requirements." https://www.dinmedia.de/en/technical-rule/asd-stan-pren-4709-002-corr/373597143, Feb. 2023. Accessed: 2025-05-31.

[15] Federal Office of Civil Aviation (FOCA), "Registration." https://www.bazl.admin.ch/bazl/en/home/drohnen/anfaenger2/uasgate/registration.html, Mar. 2025. Last modified: 25 March 2025.

[16] geo.admin.ch, "Web map tiling services wmts: Verfügbare dienste und daten." https://www.geo.admin.ch/de/wmts-verfuegbare-dienste-und-daten, Dec. 2021. Veröffentlicht am 6. Dezember 2021.

[17] OpenStreetMap contributors, "Raster tiles." https://wiki.openstreetmap.org/wiki/Raster_tiles, September 2022. Accessed: 2025-05-31.

[18] Mapbox contributors, "Mapbox vector tile specification." https://github.com/mapbox/vector-tile-spec, Jan. 2016. Version 2.1, released January 19, 2016. Accessed: 2025-05-31.

[19] OpenStreetMap Foundation, "Openstreetmap foundation." https://osmfoundation.org/, 2025. Accessed: 2025-05-31.

[20] DeviWiki contributors, "List of wireless adapters that support monitor mode and packet injection." https://deviwiki.com/wiki/List_of_Wireless_Adapters_That_Support_Monitor_Mode_and_Packet_Injection, 2021. Accessed: 2025-05-31.

[21] Microsoft Learn Contributors, "Connect usb devices." https://learn.microsoft.com/en-us/windows/wsl/connect-usb, July 2024. Accessed: 2025-05-31.

[22] S. Hung, "usb-wifi_monitor-mode_on_wsl2." https://github.com/seanhungtw/usb-wifi_monitor-mode_on_WSL2, 2024. Accessed: 2025-05-31.

[23] F. Müller, S. Brunner, and L. Romá, "droneremoteidspoofer." https://github.com/cyber-defence-campus/droneRemoteIDSpoofer, 2023. Accessed: 2025-05-31.

[24] "Google issue tracker: Issue 35827808." https://issuetracker.google.com/issues/35827808, 2025.

[25] OpenStreetMap contributors, "Switzerland extract from openstreetmap." https://planet.osm.ch/, 2025. Data licensed under the Open Database License (ODbL) 1.0. Accessed on 2025-05-31.

[26] MDN contributors, "Storage quotas and eviction criteria." https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria, 2025. Accessed: 2025-05-31.

[27] M. Oliveira, "Individual inserts vs. bulk inserts." https://medium.com/nazar-io/sql-performance-killers-individual-inserts-vs-bulk-inserts-14176c201673, Nov. 2023. Published in NAZAR on Medium.

[28] OpenStreetMap contributors, "Pbf format." https://wiki.openstreetmap.org/wiki/PBF_Format, 2025. Accessed: 2025-06-01.

[29] OpenStreetMap contributors, "Planet osm: Weekly openstreetmap data dumps." https://planet.openstreetmap.org, 2025. Accessed: 2025-06-01.

[30] Mapbox, "Mbtiles specification." https://github.com/mapbox/mbtiles-spec, 2025. Accessed: 2025-06-01.

[31] Tilemaker Contributors, "Tilemaker container image." https://github.com/systemed/tilemaker/pkgs/container/tilemaker, 2025. Accessed: 2025-06-01.

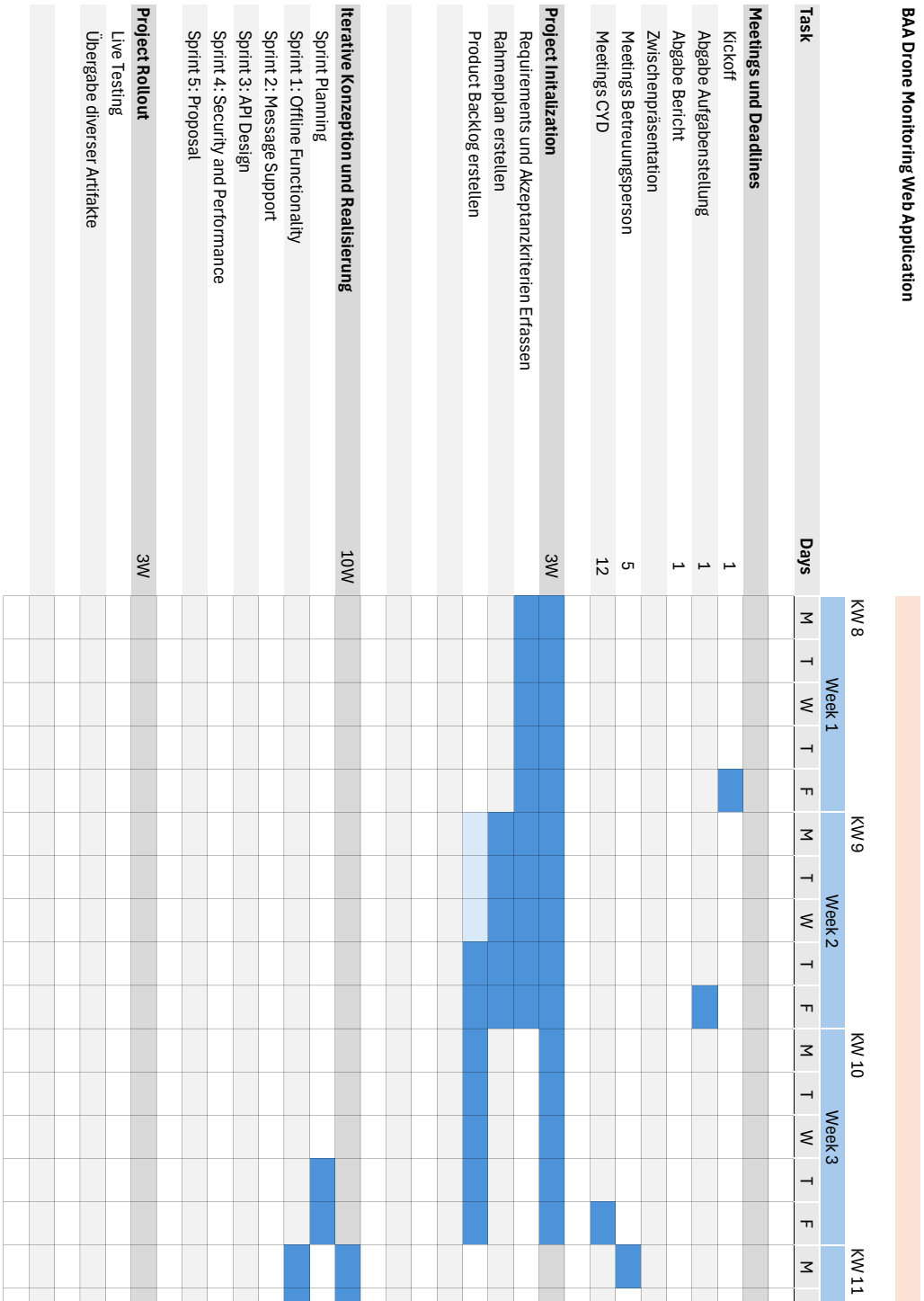
[32] MapTiler Team, "Tileserver gl." http://tileserver.org/, 2025. Accessed: 2025-06-01.

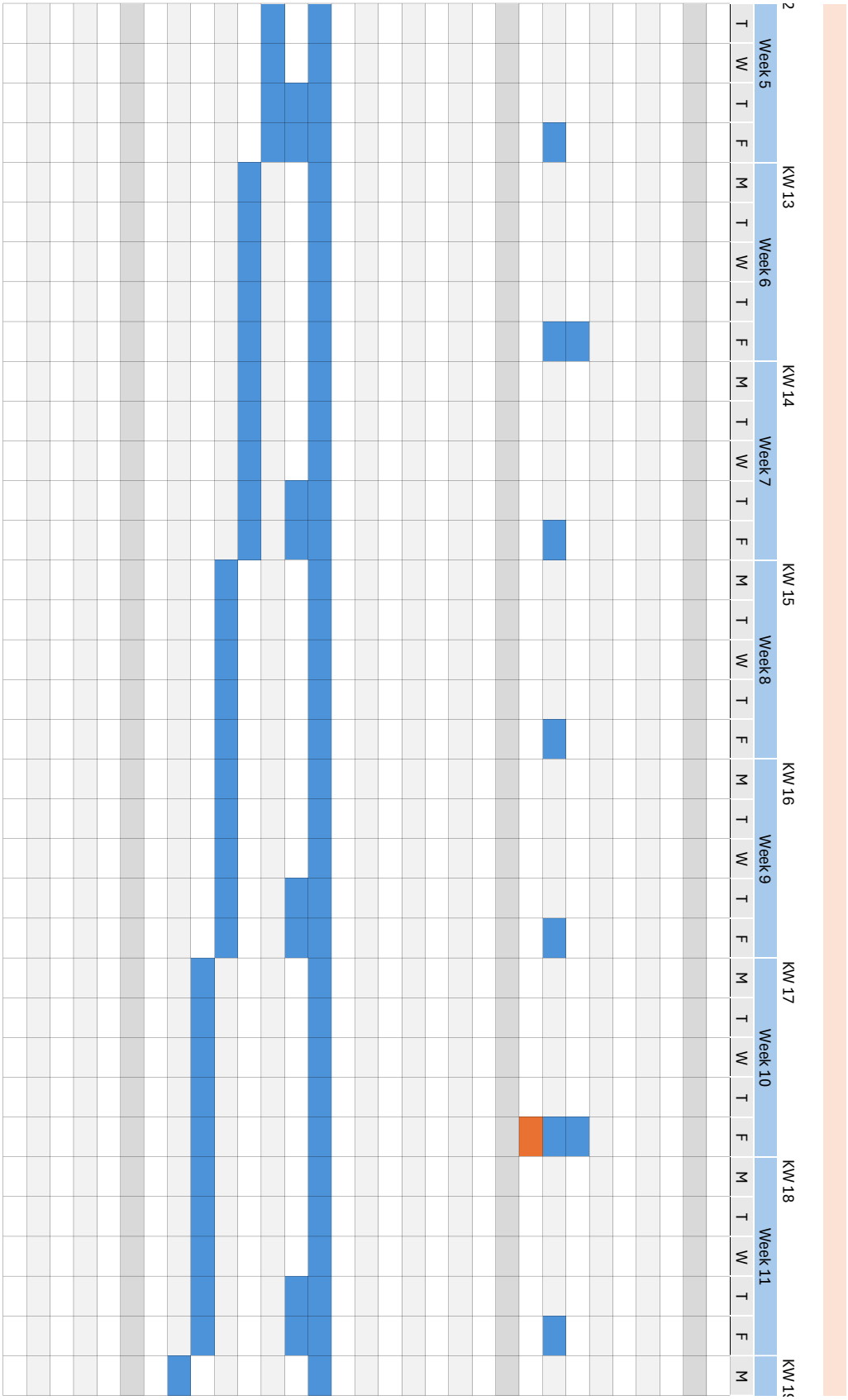
[33] Guinness World Records, "Largest aerial image formed by multirotors (drones)." https://www.guinnessworldrecords.com/world-records/758271-largest-aerial-image-formed-by-multirotors-drones, 2024. Accessed: 2025-06-04.

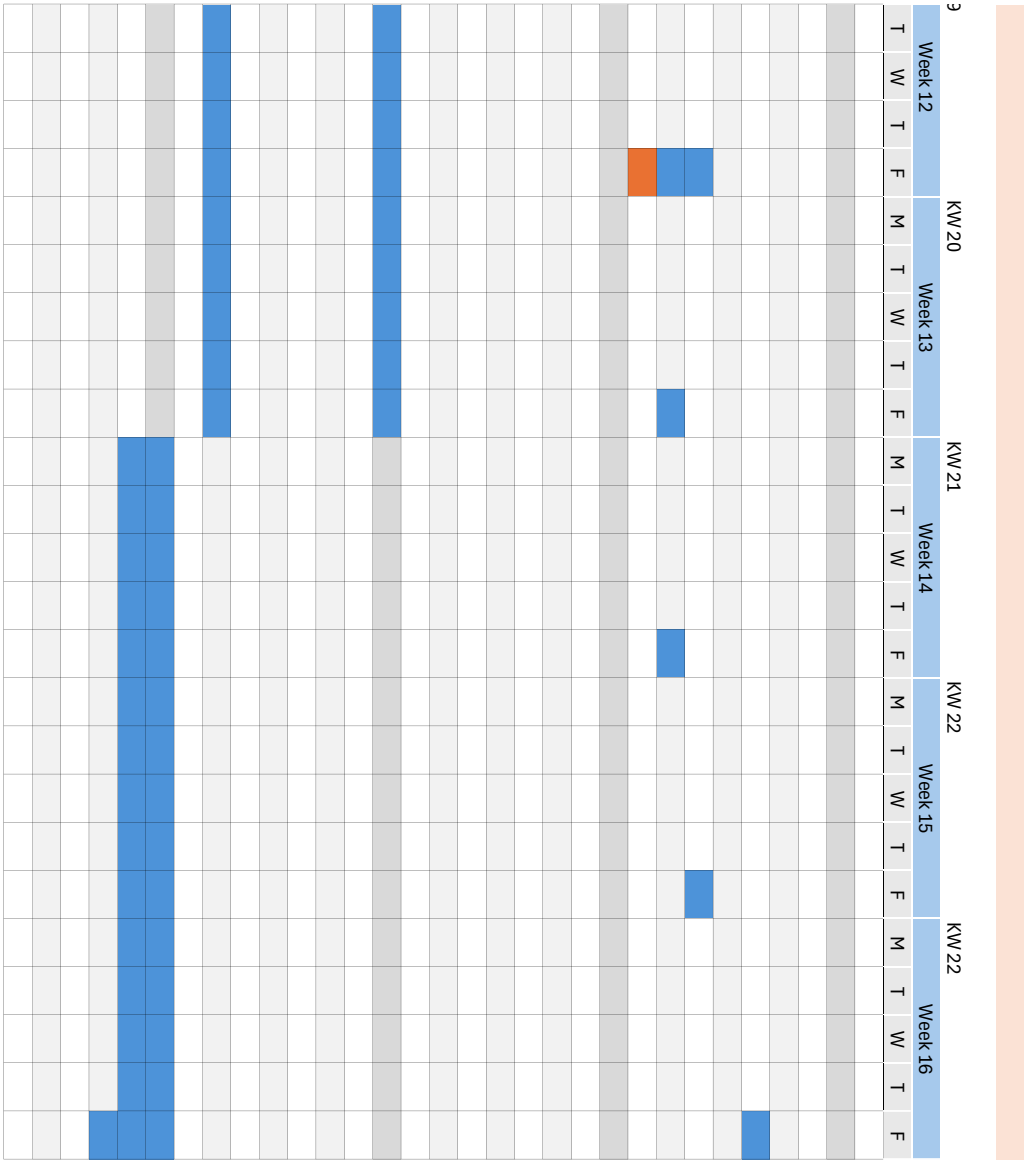
[34] Apache Flink, "Apache flink: Stateful computations over data streams." https://flink.apache.org/, 2025. Accessed: 2025-06-04.

Appendix A

Timeplan




2
KW 13
KW 14
KW 15
KW 16
KW 17
KW 18
KW 19
Week 5
Week 6
Week 7
Week 8
Week 9
Week 10
Week 11
T W T F M T W T F M T W T F M T W T F M T W T F M T W T F M T W T F M


KW 20
KW 21
KW 22
KW 22
Week 12
Week 13
Week 14
Week 15
Week 16
T W T F M T W T F M T W T F M T W T F M T W T F

Appendix B

Story Book

Story Book

User Story 1.1: Frontend Libraries Available Offline

As a user,
I want the application to load its front-end libraries without needing internet access,
so that I can use the app even when I have no connectivity.

Acceptance Criteria

- App loads and functions without an internet connection (except the map).
- User interface displays correctly and is fully usable offline.

Technical Requirements

- CDN dependencies must be removed or replaced with local bundles.
- Internet is only required during the bundling process.

Est 8h

User Story 1.2: Provide an Offline Replacement for the Map

As a user,
I want to view and interact with maps even when I'm offline,
so that I can use location-based features without relying on internet access

Acceptance Criteria

- The map contains the following geographic features: forests, lakes, rivers, streets, buildings.
- The map contains labels for streets and city names.
- The user is able to zoom.
- The map client must support all previously implemented requirements to visualize drones.

Technical Requirements

- Map: Only the initial setup is dependent on the internet.
- The map must be served from a Raspberry PI.
- The map must be able to display at least 500 individual drones at once.

Est 56h

User Stories

User Story 2.1: Message Support

As a user,
I want to have as much information about the drone as possible,
so that I can have an in-depth overview of the drones.

Acceptance Criteria

- Basic ID: Display the Serial Number as well as the UA Type.
- Location: Continuously display the location.
- Self ID: Display the mission purpose, if given.
- System Message: Display the location of the pilot.
- Operator ID: Show the registration number of the operator.

Technical Requirements

- All messages can be parsed.
- All messages will be persisted without information loss.
- Support Bundled Messages (arriving every 3 seconds in a message-pack).
- Support Individual Messages (arriving every 1 second in a message-pack).

Est 48h

User Story 3.1: Vendor Agnostic API

As an external developer,
I want to have vendor agnostic information about drones,
so that I can display drones from different data-sources on my map.

Acceptance Criteria

- Ability to list all received drones.
- Ability to list all currently flying drones.
- Ability to get metadata for an individual drone (location, pilot location, serial number, ...)
- Ability to view past flights.

Technical Requirements

- Design an abstraction that can be built from ASD-STAN as well as DJI.
- Deploy an Open API dashboard to visualize the endpoints.

Est 8h

User Story 3.2: ASD-STAN API

As an external developer,
I want to have full ASD-Stan support,
so that I know exact details about the drones in the air.

Acceptance Criteria

- An endpoint exists for every individual type of message.
- Every endpoint contains all intercepted data.

Technical Requirements

- Deploy an Open API dashboard to visualize the endpoints.

Est 4h

User Story 3.3: DJI API

As an external developer,
I want to have full dji support,
so that I know exact details about the drones in the air.

Acceptance Criteria

- An endpoint exists for every individual type of message.
- Every endpoint contains all intercepted data.

Technical Requirements

- Deploy an Open API dashboard to visualize the endpoints.

Est 4h

User Story 4.1: Performance

As a user,
I want to know how many drones are supported,
so that I know the limits of the recevicer.

Acceptance Criteria

- Bottlenecks are identified.
- Optimizations made were possible.

Est 20h

Appendix C

Journal

Arbeitsjournal BAA

Date	Hours	Work	Category	Feature
17. Feb. 25	2,0	Erstellung Rahmenplan	research	
18. Feb. 25	2,0	CYD Meeting	other	
21. Feb. 25	9,0	Kickoff, Ausarbeiten der Epics	research	
24. Feb. 25	8,0	Research Offline Map	research	1.2 Offline Map
26. Feb. 25	4,0	Setup Prototype TileServer GL	research	1.2 Offline Map
28. Feb. 25	10,0	Migration Vite	implementation	1.1 CDN
5. Mär. 25	4,0	Setup Mocking for Frontend	implementation	1.1 CDN
6. Mär. 25	10,0	research offline map	research	1.2 Offline Map
7. Mär. 25	4,0	research map library	research	1.2 Offline Map
8. Mär. 25	10,0	benchmarking 10k	research	1.2 Offline Map
10. Mär. 25	4,0	architecture frontend	architecture	1.2 Offline Map
13. Mär. 25	4,0	architecture frontend	architecture	1.2 Offline Map
13. Mär. 25	6,0	implementation frontend	implementation	1.2 Offline Map
14. Mär. 25	10,0	implementation frontend	implementation	1.2 Offline Map
17. Mär. 25	2,0	documentation frontend	documentation	1.2 Offline Map
18. Mär. 25	2,0	documentation frontend	documentation	1.2 Offline Map
20. Mär. 25	6,0	research performance	research	4.1 Performance
21. Mär. 25	8,0	performance improvements	implementation	4.1 Performance
22. Mär. 25	6,0	research remote id	research	2.1 Message Support
23. Mär. 25	6,0	research asd-stan	research	2.1 Message Support
24. Mär. 25	2,0	research asd-stan	research	2.1 Message Support
25. Mär. 25	2,0	research parser	research	2.1 Message Support
27. Mär. 25	8,0	architecture parser	architecture	2.1 Message Support
28. Mär. 25	8,0	implementation parser	implementation	2.1 Message Support
3. Apr. 25	6,0	implementation parsingqueue	implementation	2.1 Message Support
4. Apr. 25	2,0	testing parser	testing	2.1 Message Support
7. Apr. 25	2,0	architecture api	architecture	3.1 API
9. Apr. 25	2,0	architecture api	architecture	3.1 API
10. Apr. 25	8,0	implementation api	implementation	3.1 API
11. Apr. 25	4,0	implementation asd	implementation	3.2 API ASD
11. Apr. 25	4,0	implementation dji, mapper	implementation	3.3 API DJI
17. Apr. 25	8,0	generat refactors	implementation	
18. Apr. 25	8,0	preparation midterm	other	
24. Apr. 25	8,0	preparation midterm	other	
25. Apr. 25	4,0	midterm	other	
1. Mai. 25	9,0	research performance	research	4.1 Performance
2. Mai. 25	5,0	architecture parser	architecture	4.1 Performance
2. Mai. 25	4,0	implement performance	implementation	4.1 Performance
9. Mai. 25	8,0	research apache flink	research	5.1 Optional Task
15. Mai. 25	8,0	architecture optional task	architecture	5.1 Optional Task
16. Mai. 25	8,0	documentation optional task	documentation	5.1 Optional Task
19. Mai. 25	9,0	testing	testing	
20. Mai. 25	9,0	testing	testing	
21. Mai. 25	9,0	testing	testing	
22. Mai. 25	9,0	testing	testing	
23. Mai. 25	9,0	testing	testing	
26. Mai. 25	9,0	report	documentation	
27. Mai. 25	9,0	report	documentation	
28. Mai. 25	9,0	report	documentation	
29. Mai. 25	9,0	report	documentation	
2. Jun. 25	9,0	report	documentation	
3. Jun. 25	9,0	report	documentation	
4. Jun. 25	9,0	report	documentation	
5. Jun. 25	9,0	report	documentation	
6. Jun. 25	4,0	abgabe	other	
	356,0	Hours Total		

Appendix D

Test Book

Test Book

Acceptance Test: Real Time Scenario

System 1: Remote ID Backend

Prerequisites

- Linux Environment.
- Remote ID Receiver installed according to README.md.
- Network Interface Card capable of monitoring-mode with a valid driver installed. Example: TP-Link AC600 with the rtl8812au driver at commit 63cf0b4
- In a network with system 2 without internet connection.

Test Steps

'''bash
sudo python3 ./backend/dronesniffer/main.py -p 3000
'''

Expected Results

- Log contains no error
- Log contains "Starting API on port 80"

System 2: Remote ID Frontend

Prerequisites

- GUI Required
- Browser Required
- In a network with system 1 without internet connection

Test Steps

- Navigate the Browser to 'localhost:3000'
- Click the settings icon and check if the network interface from System 1 shows up
- If the network interface shows up, select it and press save

Expected Results

- Network Interface shows up

- System 1 contains log message 'sniffers - Setting sniffing interfaces to'...
- System 1 contains no error logs

System 3: Fake Drone

Prerequisites

- Linux Environment
- Not in a network
- Network Interface Card capable of injection-mode with a valid driver installed. Example: TP-Link AC600 with the rtl8812au driver at commit 63cf0b4
- Remote ID Receiver installed according to README.md

Test Steps

- Identify the interface name by running 'iw dev'
- Run drone script via 'sudo python3 RemoteIDReceiver/Receiver/resources/wifi/send.py -i <interface name>'

Expected Results

- System 3 shows no error logs
- System 3 shows a progress bar that indicates how many packets are beeing sent
- System 1 logs messages in the following format: 'info_handler - Saving 1 messages to the database'. Those messages are expected to show up every second until the script finishes.
- System 2 shows live drones on the map.

Acceptance Test: Spoofing Multiple Drones

Prerequisites

- Linux Environment
- Remote ID Receiver installed according to README.md
- (Drone ID Spoofer)[https://github.com/cyber-defence-campus/droneRemoteIDSpoofer] installed.
- Network Card in Monitor Mode

Test Steps

- Start the Remote ID Receiver using
  '"bash
  sudo python3 ./backend/dronesniffer/main.py -p 3000
  '"

- Identify the interface name by running 'iw dev'
- Start the Spoofer in a separate terminal using
  '"bash
  sudo python3 spoof_drones.py -i <interface name> -r 50
  '"
- Start the Spoofer in a separate terminal using
  '"bash
  sudo python3 spoof_drones.py -i <interface name> -m
  '"
- Connect to the Remote ID Receiver at 'localhost:3000'
- Use WASD in the second spoofing terminal to move the drone

Expected Results

- The spoofer logs "Packets sent: n", where n increases by 50 every second.
- The map shows 51 drones.
- When controlling the drone via WASD, a drone on the map moves

Appendix E

Coverage Report

Coverage report: 81%

Files | Functions | Classes

filter...

hide covered

coverage.py v7.8.2, created at 2025-06-03 19:07 +0000

File ▲	*statements*	*missing*	*excluded*	*coverage*
backend/dronesniffer/api/ads_stan_api.py	80	5	0	94%
backend/dronesniffer/api/dji_api.py	32	0	0	100%
backend/dronesniffer/api/drone_api.py	41	7	0	83%
backend/dronesniffer/exceptions.py	2	0	0	100%
backend/dronesniffer/info_handler.py	17	7	0	59%
backend/dronesniffer/map/__init__.py	0	0	0	100%
backend/dronesniffer/map/mapping_service.py	49	7	0	86%
backend/dronesniffer/models/__init__.py	2	0	0	100%
backend/dronesniffer/models/direct_remote_id.py	102	6	0	94%
backend/dronesniffer/models/dtomodels.py	60	12	0	80%
backend/dronesniffer/models/settings.py	27	0	0	100%
backend/dronesniffer/packet_processor.py	47	27	0	43%
backend/dronesniffer/parse/__init__.py	0	0	0	100%
backend/dronesniffer/parse/ads_stan/__init__.py	0	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/__init__.py	0	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/basic_id.py	9	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/direct_remote_id.py	8	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/location_vector.py	23	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/message_pack.py	7	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/operator_id.py	8	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/self_id.py	8	0	0	100%
backend/dronesniffer/parse/ads_stan/messages/system_message.py	17	0	0	100%
backend/dronesniffer/parse/ads_stan/parser.py	39	7	0	82%
backend/dronesniffer/parse/ads_stan/strategies/__init__.py	0	0	0	100%
backend/dronesniffer/parse/ads_stan/strategies/base.py	6	1	0	83%
backend/dronesniffer/parse/ads_stan/strategies/basic_id.py	8	0	0	100%
backend/dronesniffer/parse/ads_stan/strategies/location_vector.py	37	3	0	92%

File ▲	statements	missing	excluded	coverage
backend/dronesniffer/parse/ads_stan/strategies/message_pack.py	22	1	0	95%
backend/dronesniffer/parse/ads_stan/strategies/operator_id.py	7	0	0	100%
backend/dronesniffer/parse/ads_stan/strategies/reserved.py	9	5	0	44%
backend/dronesniffer/parse/ads_stan/strategies/self_id.py	7	0	0	100%
backend/dronesniffer/parse/ads_stan/strategies/system_message.py	18	0	0	100%
backend/dronesniffer/parse/dji/__init__.py	3	0	0	100%
backend/dronesniffer/parse/dji/messages/__init__.py	0	0	0	100%
backend/dronesniffer/parse/dji/messages/dji_message.py	22	0	0	100%
backend/dronesniffer/parse/dji/parser.py	128	67	0	48%
backend/dronesniffer/parse/parser_service.py	27	4	0	85%
backend/dronesniffer/parse/parser.py	34	10	0	71%
backend/dronesniffer/services/__init__.py	0	0	0	100%
backend/dronesniffer/services/drone_service_ads.py	54	0	0	100%
backend/dronesniffer/services/drone_service_dji.py	52	1	0	98%
backend/dronesniffer/services/drone_service.py	24	6	0	75%
backend/dronesniffer/settings.py	13	8	0	38%
backend/dronesniffer/time_buffer.py	39	10	0	74%
backend/dronesniffer/ws_manager.py	31	18	0	42%
Total	**1119**	**212**	**0**	**81%**

coverage.py v7.8.2, created at 2025-06-03 19:07 +0000