Plataforma de Drones Aéreos Multi-tecnologia para Suporte a Comunicações Críticas

Project in Computer and Informatics Engineering

University of Aveiro

Beatriz Rodrigues, Bernardo Falé, Diogo Correia, Francisco Martinho, João Simões, Lara Rodrigues

Group 3

2021/2022

Plataforma de Drones Aéreos Multi-tecnologia para Suporte a Comunicações Críticas

Department of Electronics, Telecommunications and Informatics
University of Aveiro

Beatriz Rodrigues (93023) bea.rodrigues@ua.pt,
Bernardo Falé (93331) mbfale@ua.pt,
Diogo Correia (90327) diogo.correia99@ua.pt,
Francisco Martinho (85088) martinho.francisco@ua.pt,
João Simões (88930) jtsimoes@ua.pt,
Lara Rodrigues (93427) laravieirarodrigues@ua.pt

June 2022

Abstract

With the normalisation of the use of drones in a professional context we are reaching highland of what is possible to achieve with a human operator, being the next step the introduction of a tool that allows the operator to plan, visualize, analyse and change a mission in real time. This functionalities are displayed to the operator via a dashboard, which uses an Influx database to allow the dashboard to have real-time access to the information of a mission and a way to keep that same data persistent. The dashboard also has tools to create missions in a style similar to "Scratch" (drag-and-drop), so a user that doesn't have programming knowledge can create and deploy missions that will be translated to a Groovy file and loaded to an Unmanned Aerial Vehicle (UAV)'s. The UAV's communicate via WAVE (802.11p) between them as well as to the groundstation, this is justified with the virtually nonexistent setup time compared with WiFi that is commonly used and the need to establish the communication as fast as possible since the time window can be narrow, the UAV's are also geared with a variety of sensors, a LoRaWAN® board to collect data from another board that is geographically far as well as to serve as a gateway and a tracking device in case the UAV fails to return. All this data is then sent to the groundstation and resent to the Influx database, completing this way the cycle.

Resumo

Com a normalização da utilização de drones na área profissional estamos a atingir o planalto do que é possível fazer com um operador humano, sendo o próximo passo introduzir uma ferramenta que permita ao operador planear, visualizar, analisar e alterar missões em tempo real. Estas funcionalidades são apresentadas ao operador via dashboard que se apoia numa base de dados Influx, de maneira a permitir um acesso em tempo real aos dados de uma missão bem como providenciar persistência dos mesmos. A dashboard também contém ferramentas estilo "Scratch" (drag-and-drop) para permitir que qualquer operador crie e dê deploy a uma missão, com o mínimo de conhecimentos de programação. Missão esta que posteriormente será traduzida para um ficheiro Groovy e enviada para as Unmanned Aerial Vehicle (UAV)'s. As UAV's comunicam entre si via WAVE (802.11p) e com a groundstation, isto é justificado pelo tempo de setup ser virtualmente nulo em comparação com o WiFi convencional e a necessidade desta comunicação ser feita o mais rapidamente possível, visto que a janela temporal pode ser curta. As UAV's também se encontram equipadas com sensores variados e uma placa de LoRaWAN® para a recolha de informação a longas distâncias, bem como servir de gateway e fazer tracking do UAV em situação do retorno não ser possível. Todos estes dados recolhidos são enviados e tratados pela groundstation e enviados para a base de dados Influx, completando assim o ciclo.

Contents

List of Figures

List of Tables

Abbreviations

AES Advanced Encryption Standard

API Application Programming Interface

APU Accelerated Processing Unit

CSS Cascading Style Sheets

DB Database

ERD Entity Relationship Diagram

FANET Flying Ad-Hoc Network

HTML Hypertext Markup Language

ICT Information and Communication Technologies

IoT Internet of Things

ITS Intelligent Transportation Systems

JSON JavaScript Object Notation

LoRaWAN Long Range Wide Area Network

LPWA Low Power Wide Area

M2M Machine-to-Machine

OBU On-Board Unit

RDBMS Relational Database Management System

RPA Remotely-Piloted Aircraft

RSU Road-Side Unit

SPA Single-Page Applications

SQL Structured Query Language

TSDB Time-Series Database

UAS Unmanned Aircraft System

UAV Unmanned Aerial Vehicle

Chapter 1

Introduction

1.1 Motivation

An Unmanned Aerial Vehicle (UAV), commonly known as a drone, is an aircraft without any human pilot, crew or passengers on board. UAV's are a component of an Unmanned Aircraft System (UAS), which include additionally a ground-based controller and a system of communications with the UAV. The flight of UAV's may operate under remote control by a human operator, as Remotely-Piloted Aircraft (RPA), or with various degrees of autonomy, such as autopilot assistance, up to fully autonomous aircraft that have no provision for human intervention [1].

Every day new drones, with new and multiple functionalities, are launched in the world market. This incredible technology, extremely versatile, can be used for a thousand and one applications, and the possibilities for professional performance are very promising, both for well-established companies and entrepreneurs. They are an important part of the economy for their advanced mechanisms and impressive functionalities. The growing interest of users in drone technology has introduced new uses and purposes for these devices, which make them an excellent alternative for some specific objectives/missions.

Within several research projects that integrate drones, communication, and imaging, an Information and Communication Technologies (ICT) infrastructure has been developed that enables an innovative set of services and applications supported by aerial drones. Some of the real scenarios considered for the infrastructure include:

- monitoring and support in emergency situations and natural disasters, such as forest fires and earthquakes;
- patrolling urban areas and supporting law enforcement;
- tourism applications such as live-streaming of video from drones traversing places of interest.

The developed infrastructure supports the control, through a base of operations, of fleets of aerial drones, allowing the execution of missions autonomously, which already have some complexity. Multi-drone missions allow the control of multiple drones performing varied tasks as part of a central mission. Sensory data collection (telemetry), and video live-streaming by cameras integrated into the drones is also possible.

Nonetheless, this platform can now be further developed and improved by adding new features.

1.2 Goals

The goals of this work will focus mainly on two areas of work. First, it is intended to enrich the set of predefined missions that the current platform already supports. For this, it will be necessary to develop all the drone control logic needed to complete the mission, using whenever necessary the set of sensors that equip the various drones. In parallel, and in a complementary logic, the intention is to make the communication network formed by the drones, through new communication technologies, more efficient and more resilient. The various communication technologies available will be analyzed and routes for routing traffic (mission control and services developed with the drone platform) will be created, according to the existing communication conditions.

In summary, the specific objectives of this project are:

a. Addition and configuration of new communication technologies to the drone platform;

b. Development of traffic management mechanisms on the drone network;

c. Extension of the dashboard for monitoring the network and the performance of the traffic management mechanisms;

d. Designing of two to three missions to be executed on the platform and implementation of their respective logic.

1.3 Document structure

In addition to the introduction, this document has two more chapters. On Chapter 2 it is described the state of the art. The tools and technologies that are being used for the development of this project are presented. The information needed to support the use of these technologies is also provided. Chapter 3 presents the architecture proposed to achieve all the goals of this project and is also where the system requirements (functional and non-functional) are described, as well as the actors and use cases.

Chapter 2

State of the Art

2.1 Unmanned Aerial Vehicles

An UAV is an aircraft that is guided autonomously, by remote control, or both; it carries several sensors and technologies that allow the drone (in this case) to communicate with other entities. These autonomous flying vehicles are equipped with software and hardware that allows the collecting of the data; These blocks of data are essential to the stability of the UAV, especially when the autonomous mode is enabled. The UAV's are able to make processes faster and more flexible, while also improving the precision.They have also become more affordable. Also, most systems in which UAV's are integrated include a groundstation control module and a communication channel between both. UAV's can intervene in numerous scenarios ranging from agriculture, tourism, surveillance and sensing missions, logistics transportation, weather monitoring, fire detection, communication relaying, and emergency search and rescue. However, such new applications have a critical requirement in common, which is the need for a communications system that allows UAV's to directly connect to each other for data exchange.

2.1.1 Autonomous Control

The available UAVs are capable of executing tasks without human input, namely automated tasks. Autonomy can be considered at different levels. Remotely sending an action to be performed by the drone can be considered an autonomous task, since after receiving the command, the drone moves by itself without human interference. A higher level of autonomy would include having these commands being sent automatically, without having the user to provide each one individually. In order to achieve this, research has been done to equip the drone with an On-Board Unit (OBU) that we will address later.

2.2 Flying Ad-Hoc Networks

Flying Ad-Hoc Networks (FANET's) are a subclass of Mobile Ad-Hoc Networks (MANET's) that provides wireless communications between moving UAV's, and is the concept of having a network composed by different UAV's connected in an ad-hoc. Some of the most distinct characteristics of FANET's is that the communicating nodes are highly mobile and that most of these nodes use wireless communication protocols, such as IEEE802.11(p,a,n). However, a considerable amount of problems arise with these "characteristics", such as low band-width, (relative) high connectivity time, and low transmission rates. In a FANET, drones can communicate with each other through OBUs, in an environment commonly described as V2V communication, and can communicate with Roadside Units (RSU's) in a V2I paradigm.

2.2.1 Entities

The FANET is composed by various elements:

- **On-Board Unit (OBU)**: It is equipped on the UAV, and is responsible for the communication between drones and road-side units. The available OBU's are the PC Engines APU 2, and they are equipped with sufficient processing power to conclude any given task. These OBU's are able to connect to other interfaces through the wireless protocols previously referenced.
- **Road-Side Unit (RSU)**: The static node in a FANET. It is usually installed near high traffic roads and is normally connected by cable or fiber to a fixed network. The three main objectives of the RSU's are to extend the communication range of the network and to provide internet connectivity to the OBU.

2.2.2 Domains and Characteristics

The FANET is divided in 3 communication domains :

- In-Drone domain: Connected OBU's
- Ad-hoc domain: Where V2V and V2I communications are formed
- Infrastructure domain: Connections between the RSU's and the internet

After carefully analysing FANET's, we can say that the UAV's movement is somewhat limited, although mobile, namely because of the connection constraints and several external factors, such as the weather conditions. On the other hand, the drones are equipped sensors that make possible the prediction of movements through the retrieval of critical data i.e speed, direction, and position. If there is a high number of vehicles equipped with OBU's, the network can have a large number of connected elements, and therefore, can be considered a large scale network, which is good. After the collecting of information by the UAV's, the data is ready to be transmitted, thus the ease of developing useful applications and services.

2.3 Mission

A mission consists in interacting with the drone by sending high-level commands.

2.4 Mission Planning

The mission planning framework aims at improving the process of building a specific mission. Using Groovy, it allows the user to write a mission script with commands close to natural language, a command may be created and integrated in Groovy like a plugin. In a mission script, the user can assign any number of available drones and instruct them to perform commands according to the restrictions they see fit. Multiple drones may be controlled in the same mission, either working independently or collaborating towards a goal. During said missions a drone will also send back telemetry of itself and its sensors, a senor can also be easily created and import as a plugin in the framework [2].

2.5 Communication

2.5.1 Wave

IEEE 802.11p(WAVE) is an approved amendment to the IEEE 802.11 standard to address the specific problems in vehicular communications, which reduces the connection setup from values of 1-2 seconds to around 10-20 milliseconds (which is very critical in vehicular environments with very

low connectivity times).
It defines enhancements to 802.11 (the basis of products marketed as WiFi) required to support Intelligent Transportation Systems (ITS) applications. This includes data exchange between high-speed vehicles and between the vehicles and the roadside infrastructure, so called V2X communication (V2V and V2I communications are performed using wave). This protocol can establish rapid connections (10- 20ms) and can provide up to 1000 meters of communication range, but can not reach high bandwidth values (higher than 27 Mbps).
The WAVE communication is performed with a mini-PCIe wireless card allowing communication on the 5.9 GHz band [3].

2.5.2 LoRaWAN

Overview

The Long Range Wide Area Network (LoRaWAN)® is a Low Power Wide Area (LPWA) end-to-end system architecture designed to wirelessly connect battery operated 'things' to the internet in regional, national or global networks with ease; It is the most popular technology used among low power networks, due not only to its low-cost nature, but also for its simplistic implementation. Concerning safety and authentication, LoRa makes use of Advanced Encryption Standard (AES) encryption and two layers of security; One layer for network security, that is used to secure devices, and the application layer, which is used to protect application data.

Features

LoRaWAN® has a variety of features that allows it to be a great solution of a wide variety of scenarios like Internet of Things (IoT), Machine-to-Machine (M2M), smart city & industrial applications, in those features we can include the following:

- Low-cost, mobile and secure bi-directional communication
- Low power consumption
- Easily scalable
- Supports redundant operation
- Supports geolocation
- Allows devices to be updated remotely
- Open-source

Network Architecture

LoRaWAN® is the key of the network architecture, it configures the end device's and connects them to an IP connected network. Figure 2.1 encapsulates how LoRaWAN® is generally used, starting from a device connected via a gateway to a much larger network where the data is then used.

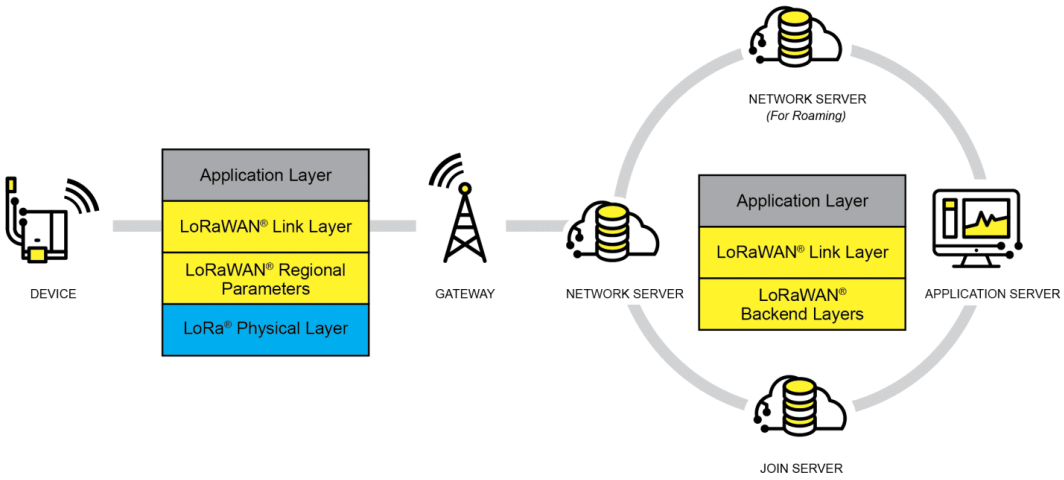

Figure 2.1: Network Architecture [4]

Components of an ordinary network:

Devices: Regular devices such like sensors, detectors, and actuators. They send the gathered data to the gateway. Unlike other networking tech, there are no unicast links between devices and gateways; Any LoRaWAN® module can receive transmission, as long as they are within the same parameters.

Gateways: This is where the messages are forwarded from the Devices to the Servers, and vice versa.

Servers: It monitors the previous components, and is responsible for removing the duplicates caused by the forwarding of the same information; It also routes the messages to the corresponding application layers.

2.6 Data Persistence

2.6.1 Database

The telemetry that the Drones feed the groundstation is not kept forever. It is in fact just saved momentarily with MongoDB, and then lost whenever new information comes along. There is no data persistence.

This was good enough for the expectations set, so far, for the management of the information created on a mission. It had to be possible to watch the mission's development in real time on the Dashboard, but it didn't need to be stored for future reference.

Upon looking at the new objectives set for us, it is clear that the project is going towards saving the information from every mission and drone, in order to be referred to any time in the future.

Drone Telemetry

A Relational Database Management System (RDBMS), like MySQL, was thought first as an alternative to MongoDB (because of its popularity, and our knowledge on databases), but then, after some research, we came to the conclusion that a Time-Series Database (TSDB) would be the way to go to save the telemetry from the drones.

One reason is its ability to handle time-related data, where the data is stored in "collections" that are aggregated over time. In this type of system, every point that is stored comes with a

timestamp. Having in consideration our need to browse huge data sets of drone measurements, with each measurement having a unique timestamp, a RDBMS would work, but a TSDB makes the work easier and more efficient.

TSDB are optimized for a fast ingestion rate. RDBMS on the other hand, because it has to re-index the data for it to be accessed faster, the performance tend to decrease with the size of the collection, as seen in Figure 2.2 [5].

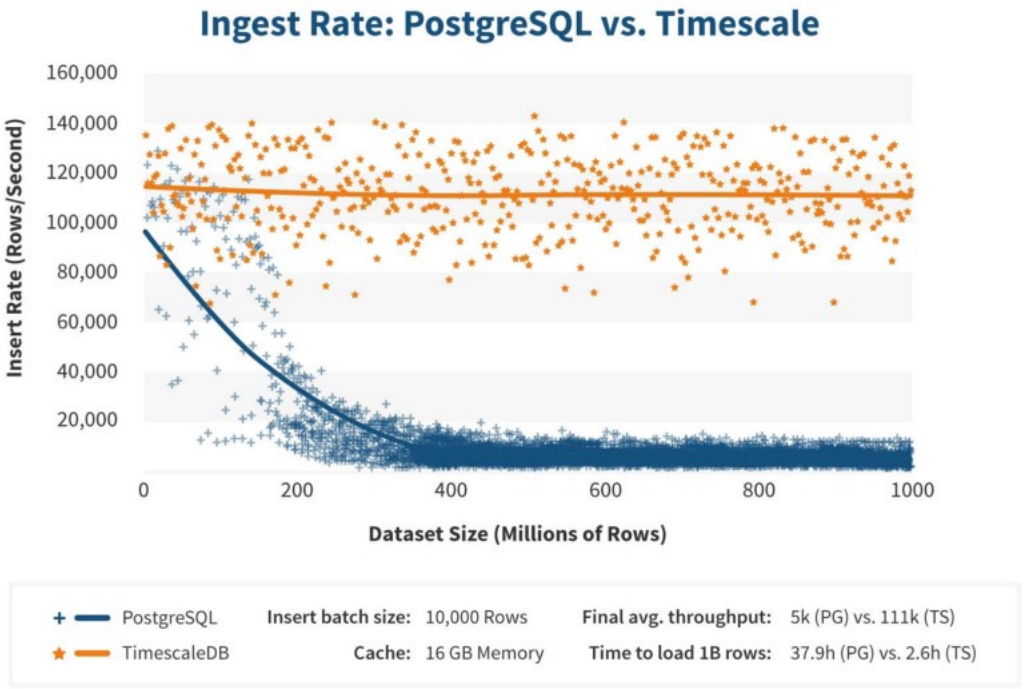

Figure 2.2: Different behaviour observed between RDBMS and TSDB [5]

InfluxDB

We chose InfluxDB because of its popularity among time-series databases, and also because of its easy integration with Grafana. This is important because Grafana is a tool that creates time-series data visualization to Dashboards, which is exactly what we need.

InfluxDB works with retention policy, that defines how long it keeps the data stored. This might be useful when dealing, for example, with test missions that are not meant to be archived. You could either want to check some values once and forget about it (like what is happening now with MongoDB), or save it only until next week, when you know they won't be needed anymore and would just become accumulated waste in the future, if not discarded.

Mission Data

The data from each mission is very basic. It simply shows which drones were used, the status, and a few more attributes. Having this into consideration, it didn't seem to make sense to use a TSDB for this type of data. Then, contrary to the drones data, we decided to use MySQL to save the missions information.

2.6.2 API

We opted to use the already existing Application Programming Interface (API) (drone-backend), implemented in Django. This API already provides a few endpoints to communicate with the

groundstation and the dashboard. Our goal is to preserve that communication (to prevent any errors with the already existing dashboard) and add other endpoints that will provide and edit the information from the new databases.

2.7 Dashboard

The dashboard is the module that has the objective to create an abstraction layer between the code behind the functionalities and the final user. It is the interface the user will use to interact with the mission. It will also give him a lot of information: values of the sensor, location, commands to control the drone, battery information, live video, and much more.

It allows the upload and tracking of missions, connect to multiple drones and groundstations, displaying a map for easy interpretation. This dashboard also contains a section dedicated to the display and control of the drone camera quality settings, and multiple charts that illustrate the continuous status of the attached drone sensors.

The front-end is built with Nuxt.js, which is a JavaScript framework mainly based on Vue.js, made to create interactive Single-Page Applications (SPA), using also HTML and CSS for the structure and visuals of the website [6].

The back-end uses MongoDB as a database and Django to connect to the groundstation for control and management, like described on 2.6.2.

This module is very well developed, so the plan here is to continue using it and just introduce the following new features:

- Get and display as much relevant data to the user as possible from API;
- Implement the possibility to further modify the missions already running;
- Allow creating missions without needing to have programming skills;
- Others that may be considered useful throughout the project's development.

Chapter 3

System Requirements and Architecture

3.1 System Requirements

This section will present the system requirements specification. The following subsections begin with a description on the requirements elicitation process, followed by a context description, actors classification, use-case diagrams, non-functional requirements, functional requirements, and the system's assumptions and dependencies.

3.1.1 Requirements Elicitation

In order to collect the system requirements the team gathered firstly with its supervisors, professor Susana Sargento, professor Miguel Luís, professor Pedro Rito, and investigator Margarida Silva; And identified the already documented and undocumented problems. The following phase consisted in analyzing the State of the Art, particularly, looking for projects and researching for studies that might be similar to the one being developed. Thus, we were able to recognize some innovative ideas. Finally, a brainstorm was performed to assemble as much concepts as possible; After the triage and filtering process we proposed some solutions that helped us define the most important functional and non-functional requirements, that will be addressed later; In this process we sought to select the most clear, concise and testable solutions.

3.1.2 Actors

The targeted users of this project are those that may need to preform tasks such as disaster prevention, patrolling zones of difficult access, etc.. To use this system, the users should present a level of expertise that allows them to create missions, access history and view live missions. The main actors are described as follows:

- Security Forces: Security Forces can create missions, access history and view live missions. They use drones equipped with cameras and sensors that allow them to preform tasks such as surveillance and monitoring public gathering restrictions.
- Civil Protection: They are capable of creating missions, access history and view live missions. They use drones equipped with cameras and sensors that allow them to preform tasks such as patrolling difficult access zones and disaster prevention.
- Drone Admin: The Drone administrator is the user with the most authority within the system. He is capable of creating missions, access history, access internal and SOS logs and view live missions.

3.1.3 Use Cases

Figure 3.1 presents the use case model of whole system, the package represent the web application that will provide the exchange of information between the users and the drones.

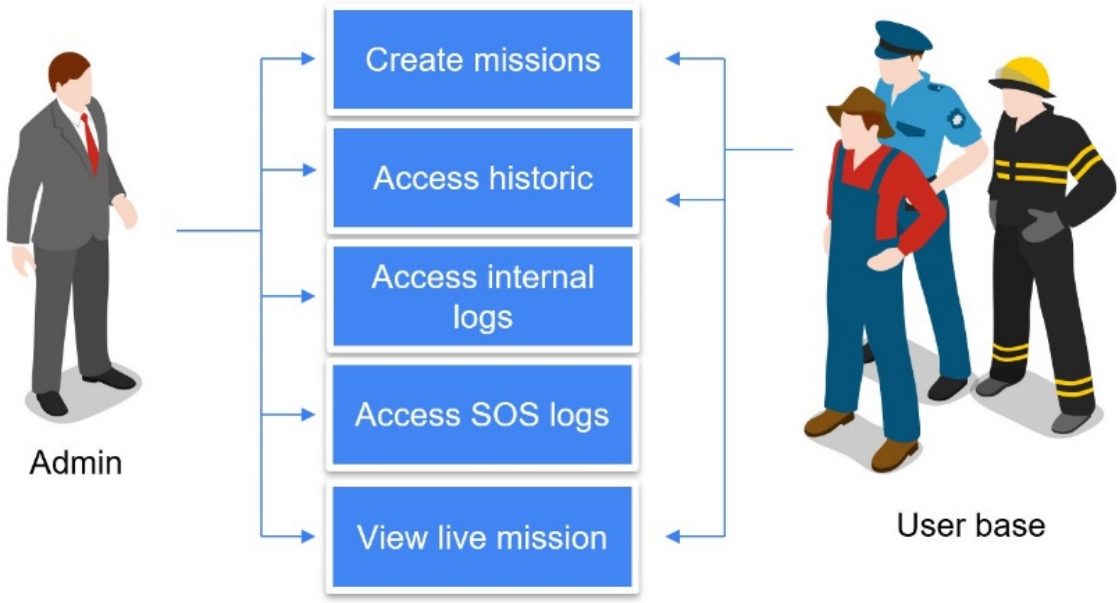

Figure 3.1: Use cases

3.1.4 Non-functional Requirements

- **Usability:** Creating and executing a mission must be intuitive as well as analysing the data. **Priority:** High
- **Efficiency:** The system must be capable of showing live data with low latency as well as update persistence date in windows inferior to 10 seconds. **Priority:** High
- **Capacity:** The system must capable of recording relevant historic of all deployed missions. **Priority:** High
- **Availability:** Since the system is used in a critical context its up time must be as close as possible to 100%. **Priority:** High
- **Security:** The dashboard as well as the OBU's must be secure and robust against attackers. **Priority:** Medium
- **Recoverability:** In the event of crashing (software or hardware) the UAV's must be easily recoverable and deployable. **Priority:** High
- **Maintainability:** The system must be constructed with future proofing in mind. **Priority:** Medium

3.1.5 Functional Requirements

The following list presents requirements specifying functional requirements:

- **Server running InfluxDB and Django API:** InfluxDB is key since it will allow for a quick and efficient access to all the available data, allowing the user to read as close to real time data as possible in a dashboard built in Django. Django has been chosen for being a powerful and robust web framework, allowing us to create the most complete dashboard possible, as well as giving ease to maintain it and/or modify it. **Priority:** Critical

- **APU running voyage in a docker image:** Voyage is a light weight distro based on Debian, giving us a familiar interface to work giving all the essential tools for a low cost, it was also was chosen for the fact of the playbooks we were given access are based on Debian, other light weight distros could have been chosen for the same effect. **Priority:** High

- **APU driving WAVE board:** One of the main points of this project is to ensure that the communications are in a reliable and fast manner, for that reason WAVE was chosen for it's high frequency, low setup time, for this reason all the modules that communicate (OBUs & groundstation) must be equipped and able to drive a WiFi WAVE enabled board. **Priority:** Critical

- **APU driving LoRa WAN board:** This board has three purposes, the main objective is to enable the APU to collect data that would be impossible for it reach for being to far away, the second objective is to relay information APU to APU functioning as a gateway until it reaches a groundstation, the third is to add a security measure, since in previous works it has been proven that a APU can go rogue and not answer even to it's owner remote controller, giving the user a way to track and collect the APU since it will function as a beacon broadcasting it's location. **Priority:** High

- **Ground station must have access to the internet:** The ground station is the man in the middle of all the missions, it's job is to collect the data from the APU's via WAVE,WiFi,LoRa and send it to the dashboard so it can be viewed and subsequently stored.

- **Easy to program missions interface:** A dashboard with the ability to create missions with the minimum knowledge of how to program needs to be a focus so the maximum amount of users can use it, this is possible by creating a drag and drop logic system like scratch, to create said missions, that will then be translated to a Groovy file and seamlessly loaded. **Priority:** High

3.1.6 Assumptions and Dependencies

Unfortunately regardless of how many counter measures and precautions we take there are always assumptions that have to be made and are out of our control, the following list illustrates all the assumptions that are made:

- The ground station has a constant connection to the internet, to mitigate this problem the groundstation will also have cellular internet, however we will always be dependent of a internet provider to connect the groundstation to the dashboard if it isn't being hosted locally.

- If UAV's crash or fail to return will be structurally stable enough and have enough power to start the SOS LoRa beacon.

- The server will have enough capacity to handle any peak of information sent by the groundstations and maintain the service stable.

3.2 System Architecture

3.2.1 Domain Model

The domain model on Figure 3.2 describes the relations between the entities within our system.

When looking at those entities, we begin with two that preform the manual interactions. One is the Admin, that has access to the groundstation and can directly setup and deploy drones. The other is a User that represents anyone that wants to interact with a Drone through the Dashboard. The former is not mandatory for the system to properly function, as long as there is a drone connected to the groundstation, making it a zero or more to many relationship, whereas the latter is a many to one relationship (at least one User using a single platform).

The groundstation is the heart of the system because it talks to both Drones and Dashboard. It basically fetches the telemetry from the Drones and makes it available to the User through the interface in the Dashboard.

The data that the Drone transmits consists of information from the Drone itself, and also information from the RSUs that it talks to, if that's the case. Drones can also talk to each other to preform "Relay", which works as a sort of connection boost. Drones don't need the RSUs to neither function nor send data so their relationship is of zero or more to many.

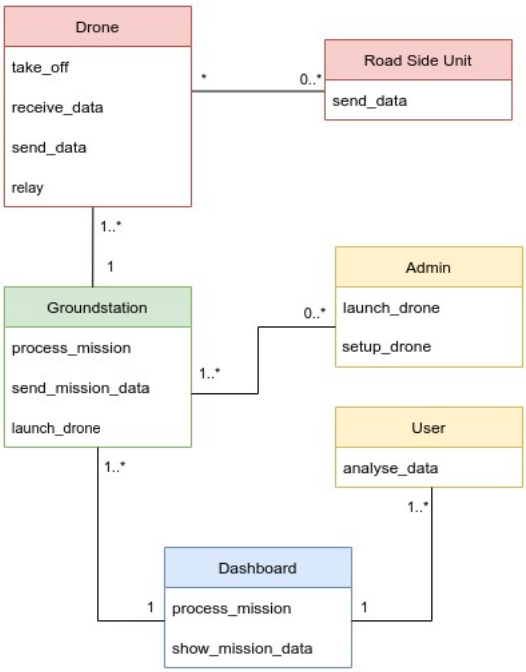

Figure 3.2: Domain Model

3.2.2 Deployment Diagram

The diagram on Figure 3.3 is intended to explore the hardware and software of our system.

The Drone module is represented by two nodes in the diagram, the Endpoint and the Gateway. This happens because a drone can both send its own information directly to the groundstation, or work as a bridge for other drones to send their information. This module is equipped with an APU running Voyage Linux, communicates using WAVE, and uses LoRaWAN to talk to the Internet. It can also have multiple sensors to collect various types of additional data or assist navigation during missions.

The groundstation also uses LoRaWAN and WAVE to communicate with the drones and retrieve their data.

The data from drones reaches the user through an API built in Django. This application is meant to facilitate the flow of information from the groundstation to both the database and the dashboard.

The data persistence is done with InfluxDB, because of its TSDB characteristics, which makes sense for this project.

The user is capable of keeping track of the drone behaviour and telemetry through a dashboard built as an SPA with Nuxt.js, CSS and HTML.

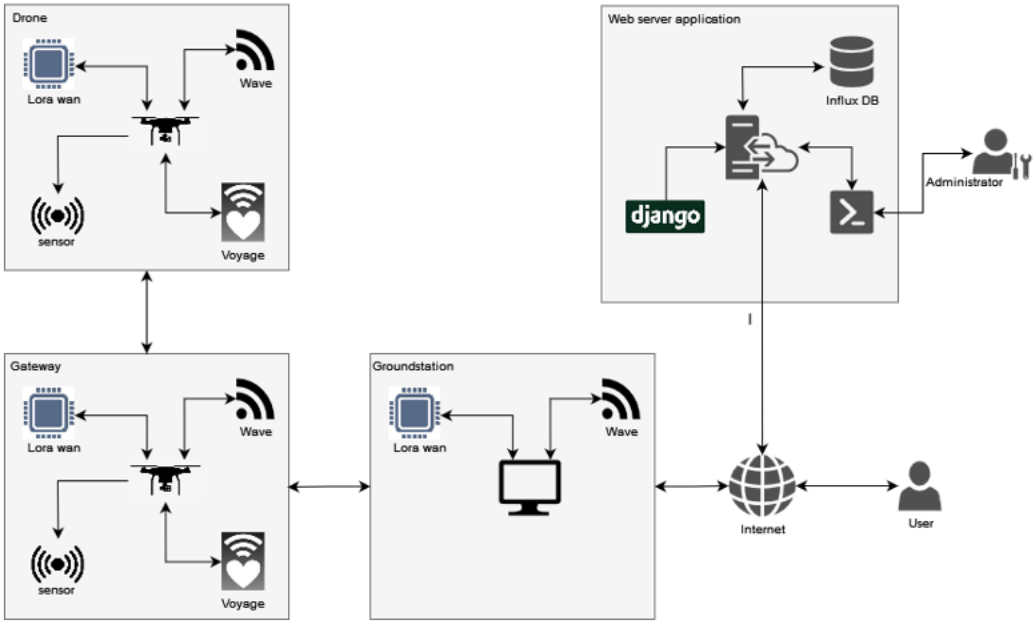

Figure 3.3: Deployment Diagram

Chapter 4

Implementation

4.1 Communication

4.1.1 WAVE

To implement this new way of communication we had to abandon the old pc companion (Jetson) since there wasn't any USB WAVE interfaces available. With that we opted to implement in a APU a WAVE enabled Wifi card via PCIexpress. However changing to another PC companion was followed with a series of problems, most of them derivated of the light distro that was ran in the APU (Voyage) and the migration of an architecture ARM to AMD64.

The first problem was to configure the APU to accept a WAVE card, that was fixed with a reflash of it with the correct drivers and by using a detailed tutorial that was provided on NAP wiki. With that a simple connection via WAVE could be established between two APUs using WAVE.

Simple tests were made to check if the improvement of setup times that were promised in the literature were indeed correct, which they were.

4.1.2 APU

Implementing the new pc companion brought about some problems which this section will now document.

At this stage simple networking tests were no longer relevant to be performed and it was time to dive in the docker environment, this is where the first problem was met during the building process. Here we discovered that the ROS2 is extremely active and gets regular staged updates, however recklessly updating ROS2 modules can bring incompatibilities that impedes docker from building the images with success. That was solved by freezing ROS2 modules on Galactic version.

After building with success we tried to run the docker drone image that failed since we were not able to register the APU as PC companion to the flight controller. This happened since the PIX4 (flightcontroller) wouldn't announce a SYMLINK, to counter act this behaviour and avoid the script confusing the PIX4 for a general USB device a file was added that linked the vendor to a SYMLINK. Still this was not enough to establish the connection, since previous solution was proven to not be enough, after some more investigation and heavy debugging using lsusb, usb-devices we realised that the problem was a missing driver from the distro we were using since it was light weight. Finnaly the communication PC <-> flight controller was established after finding the driver, installing it and recompiling the kernel used to enable its flag.

4.1.3 LoraWan

Applying this new set of communications required a new module for installation in the APU. A LoraWAN module concentrator card was installed in the free slot of the board. After some research we managed to install the drivers, and modify the kernel to best suit the necessities of the concentrator.

Configuration of the gateway packet forwarder was then needed for it to work correctly; At first, these configurations allowed the gateway card to forward the packets to the Instituto de Telecomunicações network server.

```
"gateway_conf": {
    "gateway_ID": "0016c001ff10600c",
    /* change with default server address/ports */
    "server_address": "127.0.0.1",
    "serv_port_up": 1700,
    "serv_port_down": 1700,
    /* adjust the following parameters for your network */
```

Figure 4.1: Simple configuration of the packet forwarder

Nonetheless, we sought to make this setup fully local and mobile. We made this possible by installing the network server on the APU itself. This means that the packets received are now actually forwarded to the local host; We can see the configurations in the image above.

Thus, we can access the network server through the DNS of the APU board.

A PyCom end-node was used to send LoRaWAN packets to test the drone gateway functionality. We can see a graph demonstrating that several packets were received and forwarded with success in the image below. (print do gráfico de pacotes recebidos, e um exemplo de pacote)

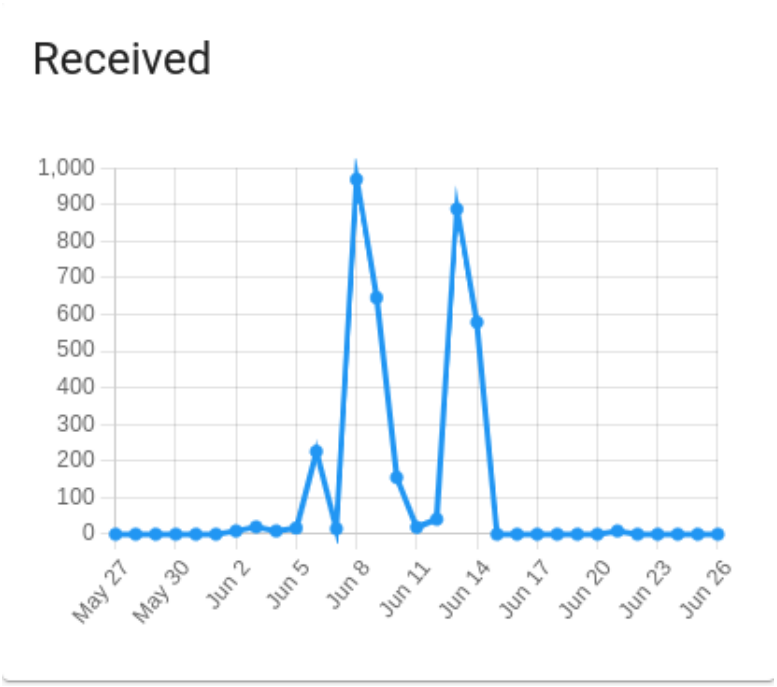

Figure 4.2: Packets received when testing

We also display a dummy packet received by the gateway for illustrative purposes.

The gateway service is constantly running, so the deployment can be instant when needed. It also boots up with machine.

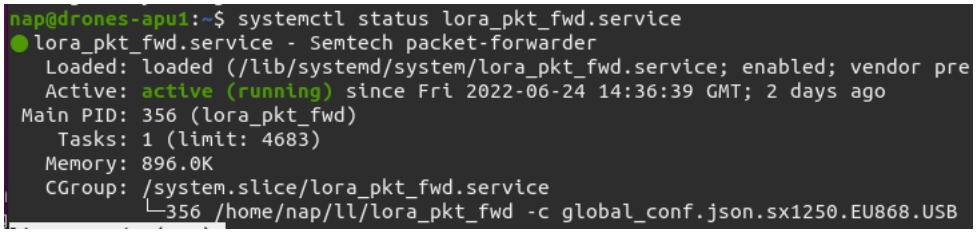

Figure 4.3: Service running instance lora_pkt_fwd

Figure 4.4: Example of a received packet

4.2 Data Persistence

After some research and understanding of the already existing modules (mainly the ground-station **fleet-manager** and the API **drone-backend**, that connected the groundstation to the dashboard) we came up with the diagram on Figure 4.5, that represents the whole process from when the data from missions and drones is available in the groundstation, up until when it is stored in the new databases and available to be fetched through a GET HTTP request to the drone-backend API.

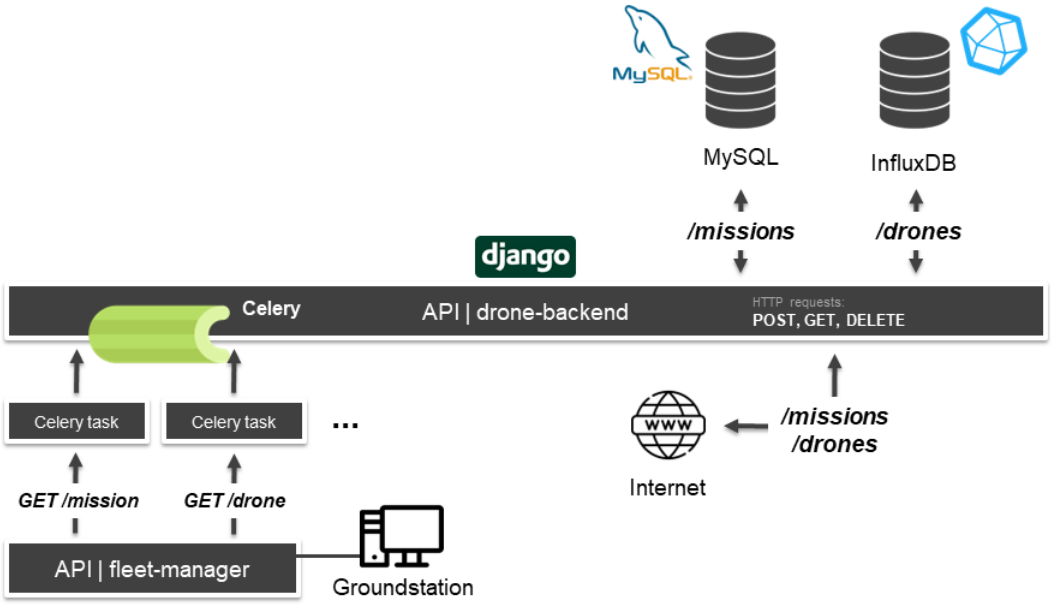

Figure 4.5: Data Persistance Schema

4.2.1 Database

To understand what was supposed to be preserved, we began to interact with the groundstation and its own API (fleet-manager). From the many endpoints it provided, we found that /**mission** and /**drone** were the ones of interest for us.

With already some idea of the structure of the data for both cases, and with each type of database in mind, we began organizing the information.

As expected, from the /**drone** endpoint we got a lot of data, which we separated into two groups, **tags** and **fields**. This is a common denomination within InfluxDB, to make a clear distinction between the data that is indexable (the tags), and the data that changes overtime and that is meant to be analysed (the fields).

Tags	Fields
droneId	pos (Lat,Lon,Alt)
state	pos (North,East,Down)
mission	vel (North, East, Down)
armed	heading
flightMode	speed
landState	battery
home (Lat,Lon,Alt)	

Table 4.1: Drone data separated into tags and fields.

With this separation, it became trivial to setup the TSDB on InfluxDB, since all it takes is the identifier of the measurement, the timestamp, the list of fields and the list of tags.

The /**mission** endpoint offered way less data, and was clearly suited for a RDBMS. Uppon a GET HTTP request, this endpoint responded with the fields present on Table 4.2.

Fields	Description
missionId	Mission identifier
status	Status of the mission (failed, running or finished)
cause	Reason for the failure
start	Date of the beginning of the mission
end	Date of the end of the mission
activeDrones	List of drones active within the mission
usedDrones	List of all drones used in the mission

Table 4.2: Mission data fields.

We were presented with a situation of one to many, where a single mission could have more than one drone. This led us to create two tables, one for the basic information on the mission, and the other for the association between mission and drone. Even though there are two arrays of drones, we decided to represent the active drones as a state of the used drones (because usedDrones contains activeDrones).

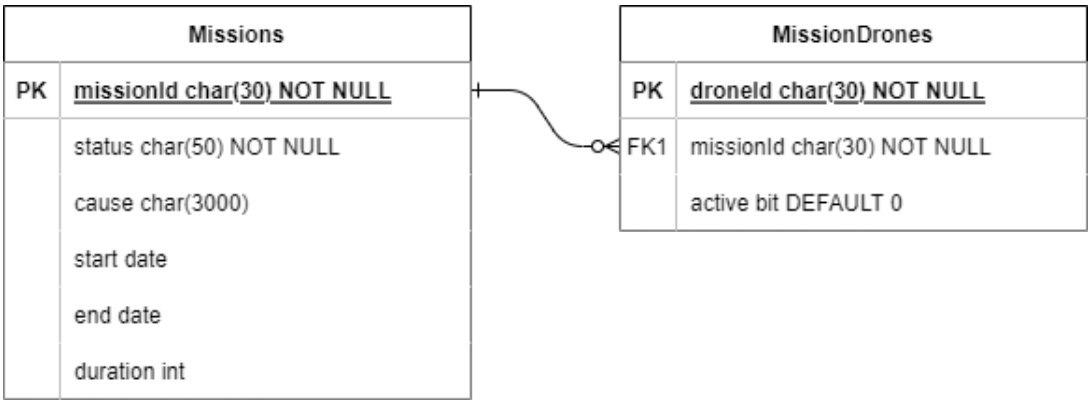

Figure 4.6: Missions Entity Relationship Diagram (ERD)

The insertion, deletion and fetching of data from these databases is entirely handled in the drone-backend API.

4.2.2 API

Because of the stage of the development of the drone-backend API, we began to develop the endpoints that handle the data to be saved on an independent, made from scratch, application, also in Django, to avoid annoying collisions with the already existing material. It is debatable whether it was worth it or not. When the endpoints were, to a certain extent, working as we expected, we merged the code from this dummy application into drone-backend.

The API has a single Django application, called *app*, where all the views to the several endpoints are mapped. For our project, we reused the already existing endpoint **/drones** and its method in the views file, and created a new application, called *missions*, to handle a new endpoint **/missions** (not to be confused with the endpoint /mission, that is external to the purposes of our project).

Missions

To save and fetch missions information, we added an endpoint that allows the methods GET, POST and DELETE, and that can also filter missions either through its properties, such as status, duration, number of drones, etc or even through the dates of when the mission started and/or ended.

To communicate with MySQL, Django Models were used to setup the tables needed, and handle the missions' database on an Object Oriented way.

POST /missions

This method is meant to take a JSON with the same format as the mission data provided by the fleet-manager API, Figure 4.7. Through Django's Models, the missions table will be updated with a new row, or an already existing row will change, depending on the mission id provided. The MissionDrones table, that makes the one to many association between missions and drones, will also be updated, depending on the *usedDrones* and *activeDrones* fields.

It is important to notice that this endpoint won't trigger a mission in the groundstation. Instead, it is just meant to save the data from an existing, already running, mission.


```
{
    "missionId": "NFXxgkod",
    "status": "failed",
    "cause": "not enough drones to fulfill mission requirements",
    "start": "2022-06-23 23:39:53,547",
    "end": "2022-06-23 23:39:54,369",
    "plugins": [],
    "remoteTasks": [],
    "activeDrones": [],
    "usedDrones": [
        "drone01"
    ]
}
```


Figure 4.7: Mission Data JSON

GET /missions

The fetching of mission data saved in the database can be done as a whole, or through some filters. Similarly to the approach in the POST method mentioned above, Django Models are used to retreive the data from MySQL.

If the intention is to filter missions that happened within an interval of time, url parameters as *start* and *end* can be passed to the endpoint, to specify those dates, encoded on ISO 8601 format.

```
GET /missions?start=2022-06-01&end=2022-06-07
GET /missions?end=2022-05-25
```

To filter by specific attributes, those attributes should be encoded into the url route. The type of attribute should be encoded first, and then its value. This can be combined with the url parameters mentioned above, to filter by both an attribute and a time interval.

```
GET /missions/status/finished?start=2022-06-01
GET /missions/missionId/NFXxgkod
```

The *duration* attribute can either have a value specified in the url route, or a range of values in the url parameter, where the min and max values for the range are separated by a comma (always use a comma, even if there is only a min or a max in the range). The value is in seconds.

```
GET /missions/duration/20
GET /missions/duration?range=,2
GET /missions/duration?range=4,27
```

The *begin* and *finish* attributes are meant to be followed by the *start* and/or *end* url parameters, and those time intervals act uppon the attribute. More specifically, if the attribute is, for example, /begin, then the url parameters represent the time interval of when the missions started.

```
GET /missions/begin?start=2022-06-20&end=2022-07-02
GET /missions/finish?start=2022-04-03
```

The ***latest*** attribute is, in a way, a special attribute. It always responds with the last mission created and, apart from the usual structure seen in Figure 4.7, it also returns the telemetry from all drones used in that mission. It is possible to either just get the drones telemetry or the mission data by alternating the url route with *drones* or *mission*, respectively.

```
GET /missions/latest
GET /missions/latest/drones
GET /missions/latest/mission
```

DELETE /missions

The delete method can be used to delete a mission from the database. This will also remove the associated drone entries from MissionDrones table and can use the attributes *missionId* and *status* to filter missions to remove.

```
DELETE /missions/status/failed
DELETE /missions/missionId/NFXxgkod
```

Drones

To handle the data from drones, we used an already existing endpoint, but changed some of the methods, mainly GET and POST. This data can be grouped or filtered by tags and time intervals. As mentioned before, in Section 4.2.1, InfluxDB divides its data into two types of values, **tags** and **fields**.

Django doesn't have Models for InfluxDB, so we used the **InfluxDB Python Client Library** instead. This library allows for easy communication with the database from Python.

POST /drones

This method takes a JSON formatted like the drone data provided by the fleet-manager API, Figure 4.8. This JSON will then be divided into tags and fields and, together with the timestamp, it is sent to the TSDB through the influx library for python.

```
{
    "info": {
        "droneId": "drone02",
        "state": "ready",
        "currentCommand": null,
        "sensors": [],
        "mission": null,
        "errors": [],
        "warns": []
    },
    "telem": {
        "droneId": "drone02",
        "armed": false,
        "position": {
            "lat": 40.6333467,
            "lon": -8.660358,
            "alt": 8.207001
        },
        "positionNED": {
            "north": -0.018270543,
            "east": -0.019297345,
            "down": -0.17144233
        },
        "velocityNED": {
            "north": 0.0,
            "east": 0.0,
            "down": 0.0
        },
        "heading": -4.612808704,
        "speed": 0.0,
        "home": {
            "lat": 40.6333468,
            "lon": -8.6603578,
            "alt": 8.22
        },
        "flightMode": "hold",
        "landState": "on_ground",
        "battery": {
            "voltage": 12.150001,
            "percentage": 1.0
        },
        "gpsInfo": {
            "satellites": 10,
            "fixType": "fix_3d"
        },
        "healthFailures": [],
        "timestamp": "2022-06-25 01:51
    }
},
```


Figure 4.8: Drone Data JSON

Then, this will represent a *drone* measurement that was taken at that specific *time*.

GET /drones

The data saved in the TSDB can be fetched through this method. If no additional route or parameters are provided to the endpoint, then the response will be a JSON with all fields and tags of all measurements of all drones.

```
{
    "time": "2022-06-24T07:47:54.250000Z",
    "armed": "False",
    "battery": 1.0,
    "batteryVolt": 12.149,
    "droneId": "drone01",
    "flightMode": "land",
    "heading": -20.66571426,
    "homeAlt": "8.003",
    "homeLat": "40.6334848",
    "homeLon": "-8.6608399",
    "landState": "on_ground",
    "missionId": "pTYeZDT6",
    "posAlt": 8.203,
    "posDown": 0.19549368,
    "posEast": -2.09521,
    "posLat": 40.6334843,
    "posLon": -8.6608397,
    "posNorth": -0.63981044,
    "speed": 0.060827624,
    "state": "ready",
    "velDown": 0.0,
    "velEast": 0.06,
    "velNorth": -0.01
},
```


Figure 4.9: Drone Data from drone-backend

The measurements can be filtered by a time interval or tags.

To filter by a time interval, the interval should be encoded into the url parameters *start* and/or *end*, with dates on ISO 8601 format.

To filter by tags, these should be encoded in the url route, first the tag, then the value. If only the tag is provided, the endpoint will, instead, group the measurements by that tag's values (i.e.: if armed is provided, not followed by any value, then the response json will have two keys, *"True"* and *"False"*, each key with an array with the measurements corresponding to drones that are armed, or not.

```
GET /drones/flightMode/land
GET /drones/armed?start=2022-05-08
GET /drones?start=2021-07-20&end=2022-01-02
```

4.2.3 Celery

To get the data from the groundstation automatically, we needed a system that would perform predoic HTTP requests to the fleet-manager API and update our databases. Because we were working on an already working API, we analysed the services it was using to make a real time communication between the groundstation and the dashboard, and we found Celery, which is a system that performs tasks asynchronously.

Having this in mind, we added two new tasks to fetch the data on drones and missions from the groundstation and save it to the databases.

The task to save drones first checks if the drone is on any mission, and only then, if that is the case, the measurement will be saved. This is to prevent saving useless data and overcrowding the database.

The task to save missions will save all new missions, and will only update existing ones if they aren't in a state of *"finished"*.

4.3 Dashboard

Regarding the work done on the dashboard, the choice was to focus first on the mission-generating web platform and then on the real-time data display in the Grafana dashboard.

4.3.1 Mission Generator

As defined in the initial objectives, a single-page application (SPA) was developed to facilitate the process of creating missions. Through this page, any user (with or without knowledge in the area of programming and UAV's) should be able to create missions and send them to the drone without any difficulty.

From the user's perspective, all that needs to be done, is just drag and drop the command that the user wants the drone to execute onto the mission timeline. On the back-end, what is happening is an interpretation of the mission timeline that the user is creating and the conversion of the respective commands into the Groovy language, the language in which the missions are designed.

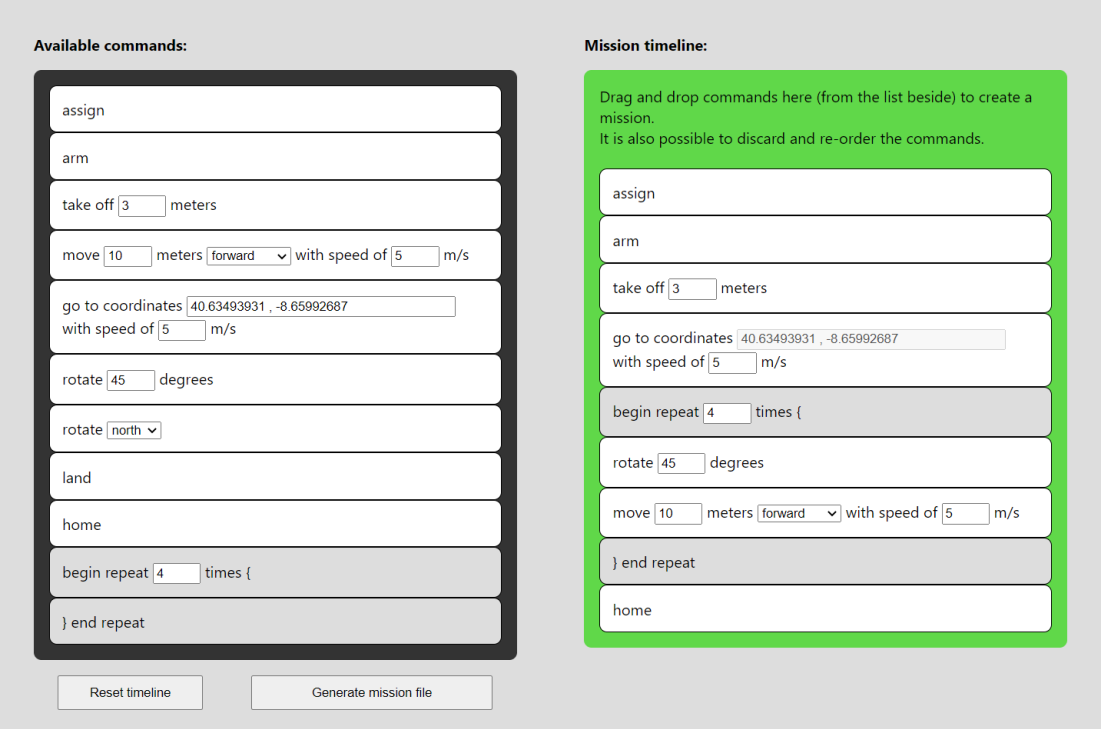

Figure 4.10: Overview of the mission generator interface

At the moment there are 11 commands available, but it is possible, quite simply, to add more new commands to the JavaScript interpreter. The default commands are the following:

- `"assign"` (assigns a drone to this mission)
- `"arm"` (gets the drone ready to fly)
- `"take-off"` (lifts off the drone x meters)
- `"move"` (moves in a certain direction with a speed of x meters per second)
- `"go-to"` (goes to a specific set of coordinates with a speed of x meters per second)
- `"rotate-card"` (rotates through cardinal points)

- `"rotate-deg"` (rotates through degrees)
- `"land"` (lands at the current position)
- `"home"` (lands at the position where it initially took off)
- `"repeat"` (repeats x times the commands from a begin repeat to an end repeat)
- `"end"` (closes the repeat cycle)

If there are parameters in that command, the user can and should change them for the purpose he wants. It is possible to reorder the commands on the mission timeline by dragging the desired command above or below the remaining commands there. It is also possible to discard a command on the mission timeline by dragging and dropping it back into the list of available commands.

To develop this page, HTML, CSS and vanilla JavaScript were used. We also used the Leaflet plugin, an open-source JavaScript library for mobile-friendly interactive maps. This plugin is responsible for the appearance of the map when the user chooses the go-to command, to give the user the possibility to select an exact coordinate on that map, to where he intends to send the drone. For the map tiles, we use the Mapbox API and opted for the preset `"streets-v11"` .

The platform makes use of the browser's Geolocation API (to locate the user's position and display it on the map) and Web Storage API (more specifically, `localStorage`) to leave the last mission saved for the next time the user visits the platform. It is also protected with some rules in order for the mission to be valid. These are:

- The first command of the timeline must be `"assign"`;
- The second command of the timeline must be `"arm"`;
- The third command of the timeline must be `"take off"`;
- The last command of the timeline must be `"land"` or `"home"`.

This web platform is available online to try out at `https://peci-g3.github.io/mission-generator/` and its source code is public and can be found here.

4.3.2 Grafana Dashboard

In the matter of viewing, collecting and analyzing data about the drones and the missions, an instance of Grafana was created. This open source analytics and monitoring solution will get the data through the API presented in the previous section (4.2.2). It would have been possible to interconnect directly with both databases (MySQL and InfluxDB), but since the API was already implemented, everything was concentrated in a single dataset. In this way, we are also able to better filter the amount of information that comes in.

However, since the API returned a JSON as a response to GET requests, and since Grafana has no native support for handling JSON data, it was necessary to install an additional plugin to handle this task. This plugin was Infinity, a Grafana data source to visualize data from any JSON APIs or URLs (and many more formats, but not relevant to this project in particular).

After that, two individual dashboards were created: one for the mission data, the other for the drone data. This way it is easier to categorize the huge amount of information coming from the API that can be shown with Grafana's charts. It is also possible to add new panels in a very intuitive way and without the need for advanced programming knowledge, in case more metrics need to be analyzed.

Unfortunately, it was not possible to integrate the Grafana instance within the dashboard that already existed initially. Nevertheless, if manually installed along with the `drone-backend` and the `fleet-manager`, it is possible to see the charts changing as the real-time data comes in.

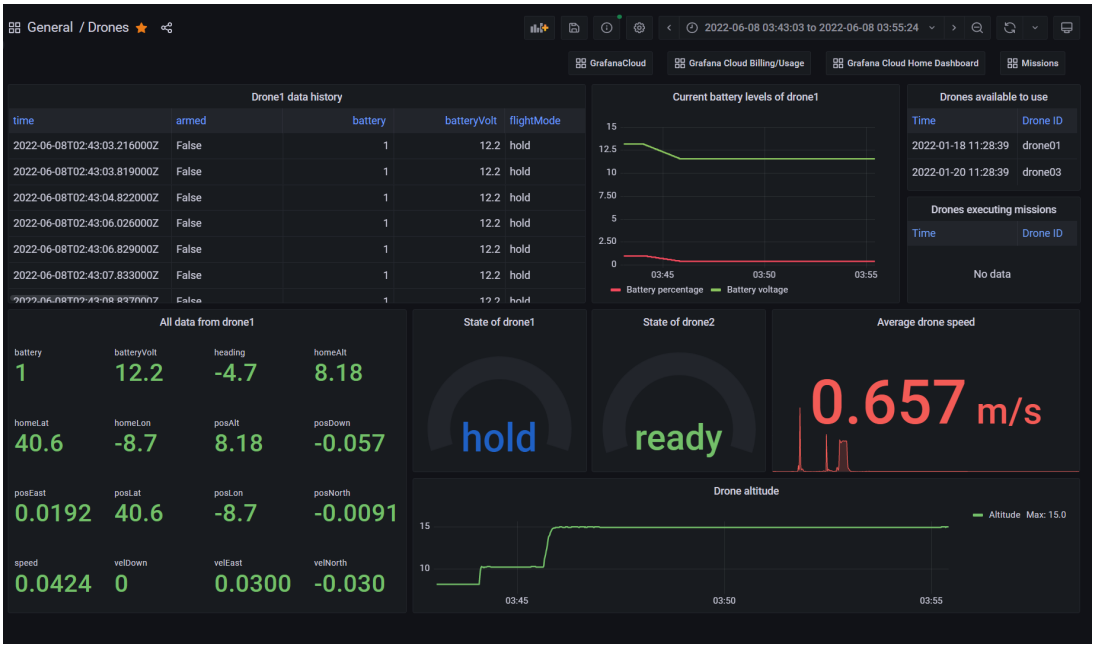

Figure 4.11: Overview of the drones dashboard on Grafana

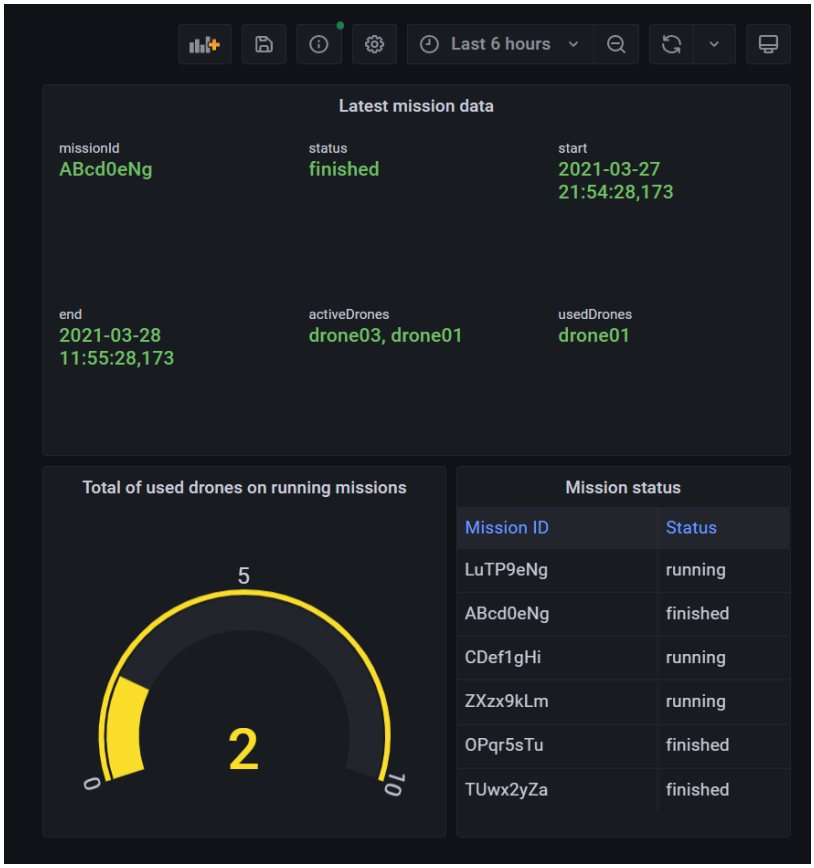

Figure 4.12: Overview of the missions dashboard on Grafana

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The main goals of this project were achieved with the exception of Grafana's integration with the already existing dashboard.

It is now possible to create a mission through the drag-and-drop system we developed and send it to the drones.

We can also view the telemetry sent in real time on the dashboard and later access information about ongoing missions.

We have migrated the communications system from the PC companion to the APU v3 and can now establish communications between drones and groundstation via WAVE. We can also deploy drones to capture lost packets, and thus having a more robust network of long range networks.

5.2 Future Work

The future work on this project should start by migrating the entire communication system from the current APU to an APU with more memory, in order to avoid future problems.

Another goal is the integration of Grafana in the already existing dashboard. This point was an initial requirement of the project that we could not fullfil.

Nonetheless, efforts are being made right now to have both of these goals implemented so that we can complete our work.

References

[1] Wikipedia contributors. "Unmanned aerial vehicle — Wikipedia, the free encyclopedia". (2021), [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Unmanned_aerial_vehicle&oldid=1062928707`. (accessed: 09.01.2022).

[2] M. Silva, *A mission planning framework for fleets of connected drones*. 2021, (accessed: 01.12.2021).

[3] A. Figueiredo, *Mobility sensing and V2X communication for Emergency Services*. 2021, (accessed: 01.12.2021).

[4] LoRa Alliance. "What is LoRaWAN® Specification". (2021), [Online]. Available: `https://lora-alliance.org/about-lorawan/`. (accessed: 01.12.2021).

[5] R. Kiefer. "TimescaleDB vs. PostgreSQL for time-series". (2021), [Online]. Available: `https://blog.timescale.com/blog/timescaledb-vs-6a696248104e/`. (accessed: 02.12.2021).

[6] Nuxt Team. "Nuxt - The Intuitive Vue Framework". (2021), [Online]. Available: `https://nuxtjs.org/`. (accessed: 02.12.2021).

[7] D. Amaral, G. Pereira, J. L. Costa, and J. T. Rainho, *Drone Relay - Technical Report*. 2020/2021, (accessed: 03.12.2021).