PROYECTO FIN DE GRADO

TITLE: Cloud-based Communication and Management System for Unmanned Autonomous Vehicles for Precision Agriculture in the EDGE

AUTOR/A: Malena Junguito Soria

TITULACIÓN: Grado en Ingeniería Telemática

TUTOR/A: Vicente Hernández Díaz

DEPARTAMENTO: DTE

VºBº TUTOR/A

Miembros del Tribunal Calificador:

PRESIDENTE/A: Waldo Saúl Pérez Aguiar

TUTOR/A: Vicente Hernández Díaz

SECRETARIO/A: Néstor Lucas Martínez

Fecha de lectura:

Calificación:

El Secretario/La Secretaria,

Resumen

El objetivo de este proyecto titulado "Sistema de Comunicación y Gestión Basado en la Nube para Vehículos Autónomos No Tripulados en la Agricultura de Precisión en el EDGE" es el diseño e implementación de un sistema de comunicación y gestión basado en la nube para vehículos autónomos no tripulados (UAV) utilizados en agricultura de precisión dentro de un marco de computación en el borde. El proyecto aborda diversos desafíos tecnológicos, económicos y ambientales para crear un sistema eficiente y escalable para la gestión de UAV.

El contexto tecnológico del proyecto implica la utilización de Robot Operating System 2 (ROS2) debido a su arquitectura modular, rendimiento en tiempo real y su extenso ecosistema de controladores y bibliotecas. Este sistema está diseñado para operar dentro de una arquitectura de computación en el borde (edge computing), que mejora las capacidades de procesamiento de datos en tiempo real y reduce la latencia al procesar datos más cerca de la fuente. La integración de la infraestructura en la nube permite la gestión centralizada de las operaciones de los UAV, proporcionando mecanismos robustos de comunicación y control.

Económicamente, el proyecto tiene como objetivo desarrollar una solución rentable mediante el uso de tecnologías de código abierto y la optimización del uso de recursos a través de la computación en el borde. Este enfoque minimiza la necesidad de hardware costoso y reduce los costes operativos al aprovechar los recursos existentes en la nube.

En términos ambientales, el proyecto promueve prácticas agrícolas sostenibles al permitir la aplicación precisa de recursos como agua, fertilizantes y pesticidas. El uso de UAV para la supervisión y gestión reduce la huella ambiental de las actividades agrícolas al optimizar el uso de recursos y minimizar los desperdicios.

La metodología seguida en este proyecto incluye un proceso de diseño sistemático, comenzando con el desarrollo de un modelo de información y diagramas de clases utilizando Unified Modeling Language (UML). El proyecto continúa a la fase de implementación, donde se desarrollan varios módulos como nodos ROS2, una Application Programming Interface (API) de servidor Django y protocolos de comunicación con la nube. Se llevan a cabo pruebas exhaustivas a través de simulaciones utilizando Webots para validar el rendimiento del sistema.

Los resultados obtenidos del proyecto demuestran la efectividad del sistema propuesto en la gestión de operaciones de UAV para la agricultura de precisión. Los resultados clave incluyen la correcta implementación de protocolos de comunicación entre la nube, los dispositivos en el borde y los drones, asegurando el procesamiento adecuado de los datos. El uso de contenedores Docker para la implementación mejora la escalabilidad y flexibilidad del sistema, permitiéndole adaptarse a diferentes dispositivos y aplicaciones agrícolas.

En conclusión, este proyecto de fin de grado proporciona una contribución significativa al campo de la agricultura de precisión mediante el desarrollo de un sistema robusto y escalable de comunicación y gestión para UAV. La capacidad del sistema para procesar datos en tiempo

real mejora la eficiencia operativa y apoya prácticas agrícolas sostenibles. El trabajo futuro se centrará en extender las capacidades del sistema para incluir modelos adicionales de UAV, así como nuevos protocolos de comunicación y uso de vehículos de tierra, y en optimizar aún más los protocolos de comunicación para mejorar el rendimiento.

Abstract

The aim of this project titled "Cloud-based Communication and Management System for Unmanned Autonomous Vehicles for Precision Agriculture in the EDGE" is the design and implementation of a cloud-based communication and management system for unmanned autonomous vehicles (UAVs) used in precision agriculture within an EDGE computing framework. The project addresses various technological, economic, and environmental challenges to create an efficient and scalable system for UAV management.

The technological context of the project involves the utilization of the Robot Operating System 2 (ROS2) due to its modular architecture, real-time performance, and extensive ecosystem of drivers and libraries. This system is designed to operate within an edge computing architecture, which enhances real-time data processing capabilities and reduces latency by processing data closer to the source. The integration of cloud infrastructure allows for the centralized management of UAV operations, providing robust communication and control mechanisms.

Economically, the project aims to develop a cost-effective solution by utilizing open-source technologies and optimizing resource usage through edge computing. This approach minimizes the need for expensive hardware and reduces operational costs by leveraging existing cloud resources.

Environmentally, the project promotes sustainable agricultural practices by enabling precise application of resources such as water, fertilizers, and pesticides. The use of UAVs for monitoring and management reduces the environmental footprint of agricultural activities by optimizing resource usage and minimizing waste.

The methodology followed in this project includes a systematic design process, beginning with the development of an information model and class diagrams using Unified Modeling Language (UML). The project then progresses to the implementation phase, where various modules such as ROS2 nodes, a Django server una Application Programming Interface (API), and cloud communication protocols are developed. Extensive testing is conducted through simulations using Webots to validate the system's performance.

The results obtained from the project demonstrate the effectiveness of the proposed system in managing UAV operations for precision agriculture. Key achievements include the successful implementation of communication protocols between the cloud, edge, and UAVs, ensuring timely data processing. The use of Docker containers for deployment enhances the system's scalability and flexibility, allowing it to adapt to different UAV models and agricultural applications.

In conclusion, this final degree project provides a significant contribution to the field of precision agriculture by developing a robust and scalable communication and management system for UAVs. The system's ability to process data in real-time enhances operational efficiency and supports sustainable agricultural practices. Future work will focus on extending

the system's capabilities to include additional UAV models, as well as new communication protocols and the use of ground vehicles, and further optimizing communication protocols to improve performance.

Acronyms list

Table 1. Acronyms in the document

Acronym	Meaning
API	Application Programming Interface
CAL	Component Abstract Layer
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DB	Data Base
EU	European Union
FDP	Final Degree Project
FMEC	Conference on Fog and Mobile Edge Computing
GDPR	General Data Protection Regulation
GNSS	Global Navigation Satellite Systems
GRSA	Guía para trabajar la Responsabilidad Social y Ambiental
GRyS	Next Generation Networks and Services Group
QoS	Quality of Service
REST	Representational State Transfer
ROS	Robot Operating System
ROS2	Robot Operating System 2
SAR	Synthetic Aperture Radar
UAVs	Unmanned aerial vehicles
UGV	Unmanned Ground Vehicles
UML	Unified Modeling Language
WSAN	Networks Wireless Sensors and Actuator

Content index

Figure index

1. Introduction

1.1. Project motivation and framework

The Next Generation Networks and Services Group (GRyS)[1] is a consolidated group of experienced and expert researchers, together with promising young academics, with extensive experience in Cyber-Physical Systems (CPS), Internet of Things, cooperative robotics, networks wireless sensors and actuators (WSAN), advanced software architectures, distributed applications and middleware systems, high-performance and fault-tolerant resilient systems, semantic middleware, adaptive systems, decision support systems, cloud and edge computing .

They have a solid track record of participation in national and international research projects, some of which are led by themselves as project coordinators.

One of the projects they have led is AFarCloud [2], [3], a complex project composed of different subprojects, specifications and varied applications related to precision agriculture, from field monitoring, tracking of livestock and their constants or autonomous robotics. Entities and organizations from all over the European Union (EU) participated in it.

1.2. Technical and academic objectives

The technical objectives of this final degree project are:

- Develop a more structured communication and control system based on the data and information models designed for AFarCloud [2], [3], specifically those related to the management of autonomous aerial vehicles, by incorporating the use of ROS2.
- Design and implement different modules and methods for information exchange in the edge computing architecture with the vehicle, managed from the cloud. This allows for its integration into this section of the project for future implementations and improvements, driven by the interest of potential clients.
- Create a suitable cloud architecture for communicating with the edge computing architecture to be designed in this project, as well as an appropriate autonomous vehicle (drone) for its integration, beyond simulation.
- Simplify cloud resources and use an available drone simulator for this project, in order to facilitate the development and testing of the system.

In an academic way, this project gets the following competences and abilities:

- Gain experience in the design and implementation of communication and control systems for autonomous aerial vehicles, using ROS2 and edge computing.
- Develop skills in the design and integration of modules and methods for information exchange in complex systems, managed from the cloud.
- Learn about the challenges and opportunities of developing and implementing systems for autonomous vehicles, including safety and efficiency considerations.

- Gain experience in the use of simulation tools for the development and testing of complex systems, including the integration of cloud-based architectures and autonomous vehicles.
- Develop problem-solving skills by addressing the technical challenges associated with the development and implementation of a communication and control system for autonomous aerial vehicles.
- Improve communication and collaboration skills by working in a team to design and implement a complex system, managed from the cloud and incorporating autonomous vehicles.
- Gain experience in the documentation and presentation of technical projects, including the design, implementation, and evaluation of a complex system for autonomous aerial vehicles.

1.3. Report structure

The structure of the rest of the report follows a detailed index that includes various sections. The report begins with an overview of the state of the art in precision agriculture, covering topics such as real-world examples, Cloud-based management and processing, precision agriculture applications, security, scalability and edge computing integration. This section delves into the core technologies used, specifically focusing on the Robot Operating System 2 (ROS2) for programming unmanned aerial vehicles (UAVs), Webots for simulation and Docker and containerization. It explains how these tools are integral to precision agriculture, enabling efficient management and real-time operation of UAVs.

Moving on, the report transitions to design specifications and restrictions, followed by a description of the proposed solution. This section includes details on the design, information model, class diagrams, UML system design with component diagrams and sequence diagrams, design details encompassing ROS2 nodes and topics, DJANGO Server Application Programming Interface (API), Cloud API, and Implementation details like Drone simulation, UAV Driver, Component Abstract Layer, UAVController, Django server, ROS2 Broker, Substituting the UAV, and Dockerization.

The report then presents the Results section, focusing on time delays and memory consumption, followed by a Budget analysis. It proceeds to discuss the Project impact, Conclusions, Future works, and includes a References section. Finally, the report contains Annexes with information on Main ROS2 messages, Auxiliar messages, a user manual detailing the installations required, Compilation, Postman configuration, and execution steps for the EDGE, UAV simulation, Testing module, and Postman.

2. State of art

The agricultural sector is undergoing a significant transformation driven by technological advancements, particularly in the realm of automation and robotics. Unmanned aerial vehicles (UAVs), commonly known as drones, have emerged as promising tools for precision agriculture, offering capabilities such as aerial imaging, crop monitoring, and pesticide application. To effectively manage and utilize UAVs in agricultural operations, robust and adaptable architectures are essential.

This section is structured to provide a comprehensive overview of the technologies and scientific advancements relevant to the programming and simulation of UAVs for precision agriculture. Subsection 2.1 delves into the core technologies used, specifically focusing on the Robot Operating System 2 (ROS2) for programming UAVs and Webots for simulation. It explains how these tools are integral to precision agriculture, enabling efficient management and real-time operation of UAVs. The subsequent subsections offer a summary of scientific papers that address the challenges of using UAVs in precision agriculture. These summaries cover issues such as sensor accuracy, data processing, and the environmental impact of UAV deployment. Additionally, they highlight the various sensors used in UAVs, like multispectral and thermal cameras, and discuss new approaches and innovations designed to overcome existing challenges and enhance the effectiveness of UAVs in agricultural applications.

2.1. ROS2 (Robot Operating System 2) and Webots

ROS2 (Robot Operating System 2) and Webots are powerful tools that have significant applications in precision agriculture. Here is a detailed state of the art:

2.1.1. ROS2 in Precision Agriculture and technology definition

ROS2 [4] is a flexible open-source framework for writing robot software that has seen increasing adoption in precision agriculture applications. Some key advantages of ROS2 for agriculture include:

- Modular architecture that allows easily building complex systems from reusable components.
- Extensive ecosystem of drivers and libraries for sensors, actuators and algorithms
- Real-time performance and determinism needed for closed-loop control.
- Abstraction of hardware details, enabling code reuse across different robot platforms.
- Powerful tools for visualization, simulation, monitoring and debugging.

ROS2 is being used in a wide range of agricultural robotics applications such as autonomous tractors, harvesters, sprayers, and specialty crop robots for tasks like weeding, pruning, and picking. The modular nature of ROS2 allows rapidly integrating new sensors and algorithms to adapt to the specific needs of each crop and farming operation.

ROS2 provides several key features that enable robust, real-time control of agricultural robots:

Quality of Service (QoS)

ROS2 introduces a flexible Quality of Service (QoS) system that allows specifying policies for reliability, durability, history, etc. on a per-topic basis. This enables:

- Guaranteed delivery of critical sensor and control messages.
- Configurable tradeoffs between latency, bandwidth and reliability.
- Seamless integration of real-time and non-real-time components.

The QoS system is built on top of the DDS middleware, which provides a publish-subscribe architecture with advanced QoS features.

Real-Time Performance

ROS2 supports real-time execution using the Real-Time Executor and other real-time-safe components. This allows:

- Deterministic, low-latency response to sensor inputs.
- Precise timing and synchronization of control outputs.
- Prioritization of critical tasks to meet deadlines.

Real-time performance is essential for closed-loop control of agricultural robots to ensure stable operation and safety.

Lifecycle Management

ROS2 provides a standard Lifecycle interface for managing the state of nodes, allowing:

- Coordinated initialization, configuration and shutdown.
- Consistent error handling and recovery.
- Introspection of node state for monitoring and diagnostics.

This makes it easier to build reliable, production-ready robot systems that can handle faults and unexpected situations.

Security

ROS2 includes built-in support for authentication, encryption and access control using DDS Security. This enables:

- Secure communication between nodes and across networks
- Restricted access to sensitive data and commands
- Auditing and logging of security-relevant events

Security is critical for agricultural robots that may be operating in public spaces or connected to the internet.

Scalability

ROS2 is designed to scale to large systems with many nodes and devices. It supports:

- Efficient discovery and communication between nodes
- Partitioning of the system into logical domains
- Bridging between different ROS2 networks

This allows building complex, distributed systems that integrate robots, sensors, cloud services and other components.

2.1.2. Webots State of the Art

Webots [5] is a powerful robot simulator that provides a realistic simulation environment for testing and developing ROS2 applications, and others. Some key features include:

Physics-Based Simulation

Webots uses the ODE physics engine to provide accurate simulation of robot dynamics, sensor noise, and environmental interactions. This allows:

- Realistic testing of control algorithms and behaviors
- Evaluation of robot performance and safety
- Generation of synthetic training data for machine learning

The physics simulation is tightly coupled with the robot models and sensor models to ensure fidelity.

Sensor Simulation

Webots provides detailed simulation of a wide range of sensors including cameras, lidars, radars, GPS, IMUs, force/torque sensors, etc. This enables:

- Realistic sensor data for algorithm development and testing
- Simulation of sensor failures and environmental effects
- Automated testing of perception and localization algorithms

The sensor models are highly configurable and can be easily extended to support new sensor types.

Robot Modeling

Webots supports importing and simulating robots described in the URDF and SDF formats. This allows:

- Reuse of robot models across simulation and real hardware
- Rapid prototyping of new robot designs
- Simulation of multi-robot systems and swarms

The robot models can be easily customized with new geometry, sensors, and control interfaces.

Scripting and Plugins

Webots provides a Python API for controlling the simulation and robots. This enables:

- Scripting of complex scenarios and environments
- Customization of robot controllers and behaviors
- Integration with external tools and libraries

The `webots_ros2` package [6] provides a ROS2 interface that allows running ROS2 controllers directly in the Webots simulation.

Visualization and Debugging

Webots includes a powerful 3D visualization interface for inspecting the simulation. This allows:

- Real-time monitoring of robot state and sensor data.
- Visualization of planned paths, object detections, etc.
- Debugging of robot controllers and algorithms.

The visualization can be customized with overlays, annotations, and custom renderings.

2.1.3. Precision Agriculture Applications

- Autonomous Tractors and Harvesters[7]: ROS2 and Webots are being used to develop self-driving tractors and harvesters that optimize paths and speeds based on real-time sensor data and field conditions. Webots allows testing these systems in realistic simulations before deployment.
- Specialty Crop Robots[7]: ROS2 and Webots enable building modular robotic systems for tasks like weeding, pruning, thinning and selective harvesting of specialty crops. Webots provides a simulation environment to develop and test the perception, planning and control algorithms for these robots.
- Precision Sprayers[7]: ROS2 and Webots are used to control sprayers that only apply chemicals where needed based on sensor data from cameras, lidars and other instruments. Webots can simulate the spraying process and sensor feedback to optimize the algorithms.
- Monitoring Systems[7]: ROS2 and Webots integrates data from various sensors and drones to monitor crop health, soil moisture, weather conditions, etc. Webots allows simulating the sensor data streams to test the monitoring algorithms.
- Decision Support Tools[7]: ROS2 enables building AI-powered decision support systems that analyze data from multiple sources to optimize farming practices. Webots can simulate the data collection process to train and validate these machine learning models.

2.1.4. AgROS Example

AgROS [8] is a precision agriculture platform built on ROS2 that uses machine learning to optimize crop production. It includes:

- ROS2 drivers for integrating various sensors, drones and robots.
- Webots simulation of the farm environment and equipment
- Machine learning models for predicting yields, detecting pests, etc.
- Decision support algorithms for optimizing irrigation, spraying, etc.

AgROS allows farmers to virtually test different scenarios and configurations in Webots before deploying them in the real world. The ROS2 architecture enables easily integrating new sensors and algorithms as the technology evolves.

2.1.5. Other Examples

- FarmWise [9], [10] uses ROS2 to control autonomous weeding robots that precisely remove weeds while preserving crops.
- Fieldwork Robotics [11], [12] develops ROS2-based harvesting robots for soft fruits and vegetables. Their robots use advanced perception and manipulation to selectively pick ripe produce.
- John Deere [12], [13] incorporates ROS2 into their precision agriculture products like self-driving tractors and sprayers.

In summary, ROS2 and Webots are enabling a new generation of smart, connected agricultural robots and systems that can optimize crop production while reducing waste and environmental impact. By providing a flexible, modular architecture and realistic simulation environment, they are accelerating the development and adoption of precision agriculture technologies.

2.2. Edge Computing in Precision agriculture

Edge computing [14] is a promising approach precision agriculture that brings computation and data storage closer to the devices where it is needed, rather than relying on a central location that can be far away. This reduces the need to send data to a distant data center, which can improve response times and save bandwidth. Edge computing enables real-time data processing and decision making at the edge of the network, allowing for faster reaction times and more efficient use of resources in precision agriculture applications. One key advantage of edge computing in precision agriculture is its ability to handle large amounts of data generated by various sensors and devices deployed in the field.

By processing this data locally at the edge, it reduces the need to transmit raw data to the cloud or a central server, which can be costly and time-consuming. Edge devices can perform data aggregation, filtering, and analysis, sending only the most relevant information to the cloud for further processing or storage, as it is done in the project.

Another benefit of edge computing is its potential to improve the reliability and resilience of precision agriculture systems. If the connection to the cloud is interrupted or slow, edge devices can continue to operate autonomously, making decisions based on local data. This ensures that critical tasks, such as irrigation control or pest management, can still be carried out even in the face of network disruptions. Edge computing also enables the development of more advanced applications in precision agriculture, such as real-time monitoring and control of agricultural processes. For example, edge devices can be used to monitor soil moisture levels and automatically adjust irrigation schedules based on local weather conditions and crop needs. This can lead to significant water savings and improved crop yields.

Furthermore, edge computing can help address privacy and security concerns in precision agriculture. By processing and storing sensitive data locally at the edge, it reduces the risk of data breaches or unauthorized access that can occur when data is transmitted to the cloud. Edge devices can also implement security measures, such as encryption and access control, to protect the integrity of the data and the system. In conclusion, edge computing is a promising approach for precision agriculture that can help address the challenges of data management, reliability, and security. By bringing computation and data storage closer to the devices where it is needed, edge computing enables real-time data processing, improved decision making, and more efficient use of resources in precision agriculture applications.

As the adoption of precision agriculture continues to grow, edge computing will play an increasingly important role in enabling more advanced and sustainable farming practices.

2.3. Docker and containerization

Docker[15], [16] has emerged as the leading platform for building, deploying and running applications using containers. Containers provide a standardized way to package an application's code, configurations and dependencies into a single object that can run consistently across different computing environments.

Key features of Docker include:

- Lightweight: Containers share the host operating system kernel, making them more efficient than virtual machines
- Portable: Applications packaged as containers can run on any machine that has Docker installed, regardless of the underlying infrastructure
- Scalable: Containers can be easily scaled up or down to meet changing demands
- Secure: Containers provide an additional layer of security by isolating applications from each other and the underlying host

Docker uses a client-server architecture with the following main components:

- Docker daemon: Runs on the host machine and manages containers.
- Docker client: Command-line interface for interacting with the daemon.

- Docker images: Read-only templates used to create containers, built from a series of layers.
- Docker containers: Runnable instances of Docker images.
- Docker registries: Stores and distributes Docker images, e.g. Docker Hub.

Docker has become the de facto standard for containerization, with widespread adoption across the software development lifecycle. It enables building, testing and deploying applications more efficiently and consistently. The portability and scalability of Docker containers make them ideal for modern cloud-native architectures and microservices, and that is the reason why it is chosen for this system, to allow anyone and anywhere to execute and test it.

Looking ahead, the future of containerization will likely involve further advancements in areas like security, networking, storage and orchestration. But Docker has firmly established itself as the leading platform for building and running containerized applications.

2.4. Application of Unmanned Aerial Vehicles in precision Agriculture[17]

Unmanned aerial vehicles (UAVs) have the potential to revolutionize precision agriculture by providing farmers with a new tool for collecting data and applying the right treatment at the right area. UAVs can be used to collect high-resolution images and videos of crops, which can then be used to assess crop health, soil conditions, and water usage. This data can then be used to make informed decisions about irrigation, fertilization, and pest control. UAVs can also be used to apply pesticides and herbicides more precisely, which can reduce the amount of chemicals used and improve environmental safety.

Despite the many potential benefits of UAVs in agriculture, there are also some challenges that need to be addressed. One challenge is the need for better sensors and data processing algorithms. UAVs are currently limited by the resolution of their sensors and the ability of algorithms to process the large amounts of data that they collect. Another challenge is the need for better regulations for the use of UAVs in agriculture. In many countries, there are strict regulations on the use of UAVs, which can make it difficult for farmers to use them.

Another significant challenge that most UAV-based solutions are proprietary and closed systems. This means that access to the underlying software and hardware configurations is restricted, limiting the flexibility to customize and integrate these solutions with other systems. Furthermore, only a limited number of UAVs offer platforms that support or implement widely accepted standards. This lack of standardization makes it difficult to ensure compatibility and interoperability between different UAV systems, posing additional hurdles for developing a cohesive and adaptable solution.

Despite these challenges, the potential benefits of UAVs in agriculture are so great that it is likely that they will continue to be adopted by farmers in the coming years. As the technology continues to develop and the regulations become more flexible, UAVs are likely to play an increasingly important role in precision agriculture.

2.5. A Robot Operating System Framework for Secure UAV Communications[18]

This paper presents a robot operating system (ROS) framework for secure unmanned aerial vehicle (UAV) communications. The authors discuss the definition of UAVs, related work, their proposed method, tests, and conclusions. They propose a method for secure UAV communication using a ROS framework. They implemented the method and conducted tests that showed the method is effective.

The proposed method consists of three layers: the perception layer, the decision layer, and the action layer. The perception layer collects data from sensors, the decision layer processes the data and makes decisions, and the action layer executes the decisions. The ROS framework provides a communication infrastructure for the three layers.

The authors conducted tests to evaluate the performance of their method. The tests showed that the method is effective in improving the security of UAV communications. The method is also efficient and scalable.

The proposed method is a promising approach to secure UAV communications. The method is effective, efficient, and scalable. The ROS framework provides a flexible and extensible platform for implementing the method.

2.6. Agroview: Cloud-based application to process, analyze and visualize UAV-collected data for precision agriculture applications utilizing artificial intelligence [19]

The article presents a comprehensive overview of a cloud-based application designed for processing, analyzing, and visualizing data collected by Unmanned Aerial Vehicles (UAVs) in precision agriculture applications, similar to the final purpose of the project.

This innovative tool integrates artificial intelligence to enhance the efficiency and accuracy of data processing. The authors emphasize the significance of utilizing hyperspectral data for detecting specific agricultural issues like Laurel Wilt Disease and nutritional deficiencies in crops, particularly avocados. The application of cloud computing and the acquisition of NIR-Green-Blue digital photographs from UAVs are highlighted as crucial components for effective crop monitoring. The article references various studies on remote sensing techniques, machine learning applications, and high-throughput phenotyping in agriculture, underlining the importance of advanced technologies in modern agricultural practices.

Additionally, it discusses the extraction of detailed information from high-spatial-resolution UAV-acquired images, emphasizing the potential for precise tree-level analysis in orchards. Overall, the article underscores the pivotal role of technology, particularly UAVs and artificial intelligence, in revolutionizing precision agriculture practices for improved crop management and disease detection.

2.7. A compilation of UAV applications for precision agriculture [20]

The use of Unmanned Aerial Vehicles (UAVs) has gained significant attention in precision agriculture due to their ability to provide high-resolution spatial and temporal data for various applications.

UAVs equipped with multispectral and hyperspectral sensors can be used for crop monitoring, disease detection, and yield estimation. Machine learning algorithms have been employed to analyze the vast amounts of data collected by UAVs, enabling more accurate and efficient decision-making in agricultural practices.

Additionally, UAVs can be integrated with other technologies such as Global Navigation Satellite Systems (GNSS) and Synthetic Aperture Radar (SAR) to enhance their capabilities in precision agriculture. Multiple UAV systems have been developed and evaluated for agricultural applications, demonstrating the potential for improved crop management and increased productivity.

However, the design of multi-UAV systems in cyber-physical applications poses several challenges, such as coordination, communication, and privacy preservation. Despite these challenges, the integration of UAVs with other technologies, for example the potential contribution of this system, and the development of advanced data analysis techniques have significantly contributed to the advancement of precision agriculture, enabling more efficient and sustainable farming practices.

2.8. Boosting Precision Agriculture through Unmanned Aerial Vehicle Remote Sensing and Edge Intelligence: A Survey [21]

Precision agriculture has been revolutionized by the integration of unmanned aerial vehicles (UAVs), remote sensing, and edge intelligence. The use of UAVs equipped with multispectral and hyperspectral sensors enables the acquisition of high-resolution images, which can be analyzed to monitor crop health, detect pests and diseases, and optimize irrigation systems. Edge intelligence, which involves processing data locally on the UAV or in a nearby edge computing device, allows for real-time decision-making and reduces the need for data transmission to the cloud. This survey aims to provide an overview of the current state of the art in precision agriculture using UAV remote sensing and edge intelligence. It will discuss the various applications of UAVs in precision agriculture, including crop monitoring, precision farming, and autonomous farming. The survey will also explore the challenges and limitations of implementing UAV remote sensing and edge intelligence in precision agriculture, such as data processing and storage, and the need for standardized data formats and protocols. Additionally, it will highlight the potential of UAV remote sensing and edge intelligence in enhancing the efficiency and sustainability of agricultural practices, reducing the environmental impact of farming, and improving crop yields.

3. Design specifications and restrictions

In the realm of software development, the project entails the establishment of a sophisticated system with specific specifications and design constraints. The core elements of this endeavor encompass various facets crucial to its successful implementation.

The foundation of version control will be laid using GitHub, enabling meticulous tracking of code changes, collaboration among team members, and maintaining a comprehensive history of project alterations. This ensures a structured approach to code management and project evolution.

The system architecture will be designed to be dockerized, allowing for seamless deployment, scalability, and portability. This dockerization approach enhances the system's flexibility and ease of deployment across different environments.

The system developed in this Final Degree Project (FDP) provides its functionalities and services to other external components through a Representational State Transfer (REST) API. The REST server that implements this API has been developed using the Django Framework. Additionally, to ensure that the functionalities are provided correctly according to the specifications outlined in subsequent chapters, Postman has been used as the REST client for testing.

Data storage will rely on a single SQLite3 database, ensuring data integrity and consistency across the system. Python will be the language of choice for developing all modules, ensuring uniformity and compatibility throughout the system components.

To simulate vehicle behavior, a Mavic 2 Pro drone will be emulated in Webots environment, replicating real-world drone functionalities and interactions within the system.

A tester module will be crafted to validate communication with the EDGE, acting as a surrogate for the simulated or physical drone in real-world scenarios. This module serves as a critical component in verifying communication protocols and system functionality.

Communication between the UAV (simulator/tester) and the Broker will be facilitated through ROS2, enabling efficient data exchange and messaging capabilities within the system architecture.

In subsequent chapters, it is detailed the communication between certain internal elements of the developed system that has been conducted using two sockets, ensuring seamless communication channels and effective data transfer mechanisms.

System management at the edge involve making Create, Read, Update, Delete (CRUD) requests to various URLs on a server, enabling streamlined operations and efficient system control at the edge interface.

These specifications and design restrictions collectively form a structured framework for developing a robust system that integrates version control practices, cloud management

strategies, drone simulation technologies, communication protocols, and database handling mechanisms within a cohesive architectural framework.

4. Description of proposed solution

The base idea of the project is shown in the Figure 1.

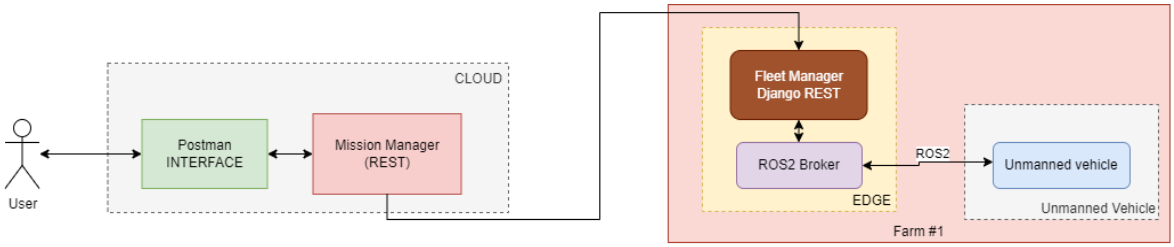


Figure 1. Initial idea

With the focus in getting a modular, actual, and innovative approach in the solution the architecture became the one illustrated in Figure 2. Architecture of the system.

The architecture of the system is split in two main blocks: the cloud and the farm, physically separated but also with different structure, technologies, and planning. The objective of this design is to get the most modular and adaptative system possible.

The first block is hosted in the cloud, and it is not developed in this project. A simplification of the cloud elements with a simple mock server has been developed, and requests from Postman are sent to carry out the whole system and functionality. It is the direct interface with the final user, the farmer, or the plan provider, with a graphical interface to set plans, see results and track the execution. Understanding a plan as a collection of missions to one or more agents, in this architecture they could be Unmanned Aerial Vehicle (UAV) with Robot Operating System 2 (ROS2) communications technology, a drone for example, or Unmanned Ground Vehicles (UGV) with ISOBUS or ISO 11783 standard communication technology, for instance a tractor. In this development, the part of ground vehicles and ISOBUS is left for future works, and it is focused on UAVs and the use of ROS2.

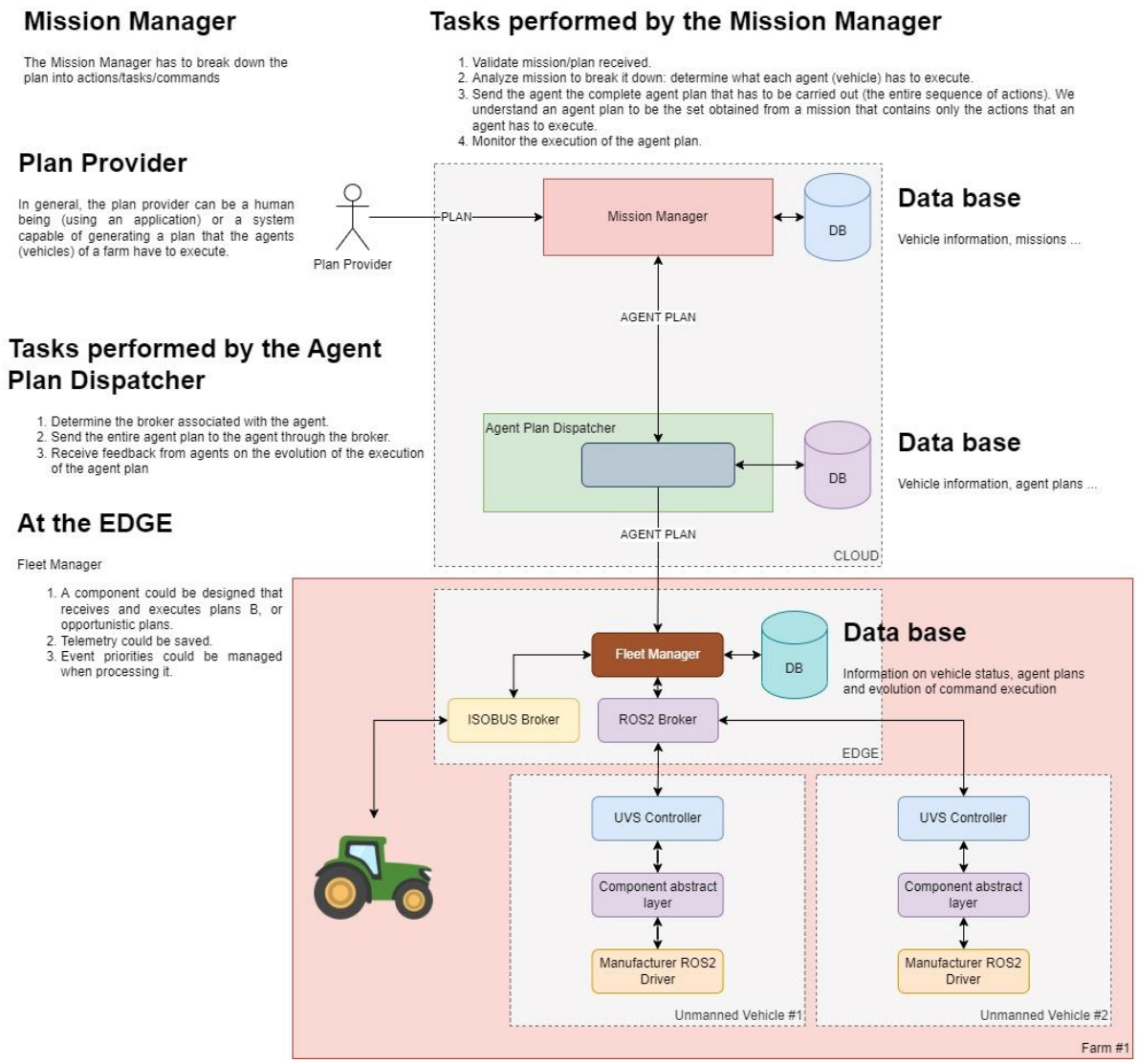


Figure 2. Architecture of the system

The two modules included in the cloud architecture are the Mission Manager and the Agent Plan Dispatcher. The first oversees the interaction with the user, the reception and collection of the information in the specific data base, for example information of the plans, the execution state, results, events or problems to be shown to the farmer, to get everything updated and make decisions as soon as possible from any part of the world with only a simple internet connection. In the other direction, when the user decides to start a plan, change one that is in execution or make decisions when an event happens or an alarm is thrown by a vehicle, this module transforms all the graphical information, converts it into a normalized structure to be saved in the data base and sends it to the Agent Plan Dispatcher. This second module has another data base with the administration of the vehicles: ready, in charge, in execution of a mission, with an alarm or stopped; and the agent missions that is being executed, in queue, failed or finished and which vehicle is associated with it. It is also the direct interface with the edge, carrying out the communication with and from there, sending the plans to the vehicles that are free and receiving the information of the execution results and so on to be sent to the Mission Manager, and in the end to be shown to the administrator.

The second block is the main objective of the development of this project. The physical location of the elements of the block is in the edge, in the farm, with two principal modules. The EDGE section is allocated in the farm in a fixed place with a reduced computational power and a small data base, as typically seen in the edge actual developments. The Fleet Manager module gets the information from the user through its online interface with cloud, it saves temporal data of it in the data base, and then passes it to the corresponding broker. And in the other direction, it receives the results, data, or alarms from the vehicles through the brokers, temporarily saves it in the data base and sends it to the cloud; this data will be deleted from the data base when the cloud confirms its reception, in order to avoid losing information. The other two modules included are brokers for communicating with diverse vehicles using diverse protocols. The development done in this project is focus on the ROS2 Broker, leaving the ISOBUS one for future works as mentioned before. The broker module receives the mission from the Fleet Manager and publishes it to the UAVs, and also receives the results, data and alarms from them to be passed up in the architecture.

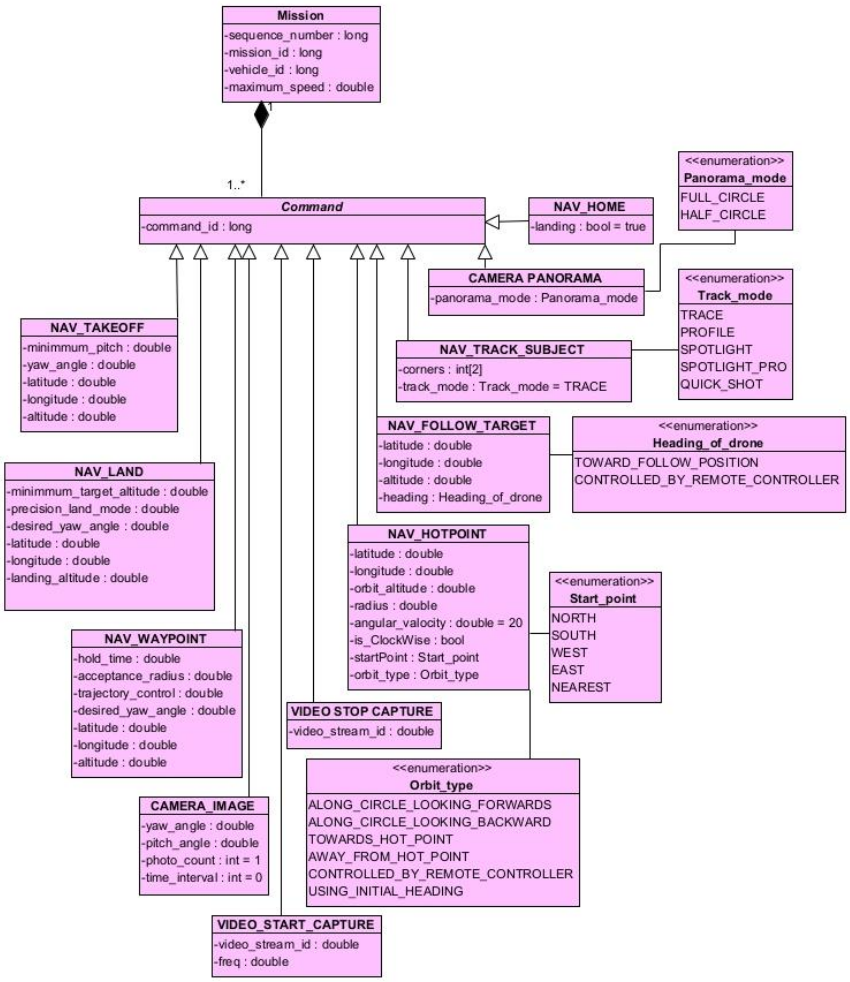
The UAV modules are the only ones which are not in a fixed place, as they are inside the vehicle. The different aerial vehicles have one common block: the controller and two different ones depending on the model and brand of the drone: the driver and Component Abstract Layer (CAL). This design is to get an easy way to change the vehicle, the brand, the protocol to control the motors, or sensors.

The controller manages the communication with the edge, the sending or reception of the information (missions, events, results…) and the transformation of the commands into actions with the use of the CAL, which provides an interface and actuates in the parameters of the driver to obtain the movement and the expected behavior. Finally, the driver, specific for each drone, actuates in the parts of the vehicle, for example accelerates or decelerates the motors to achieve a movement, or reads the position sensors to determinate if the goal position is arrived.

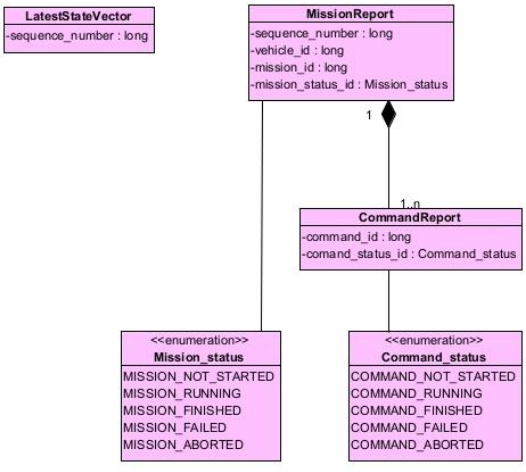
4.1. Design

4.1.1. Information model

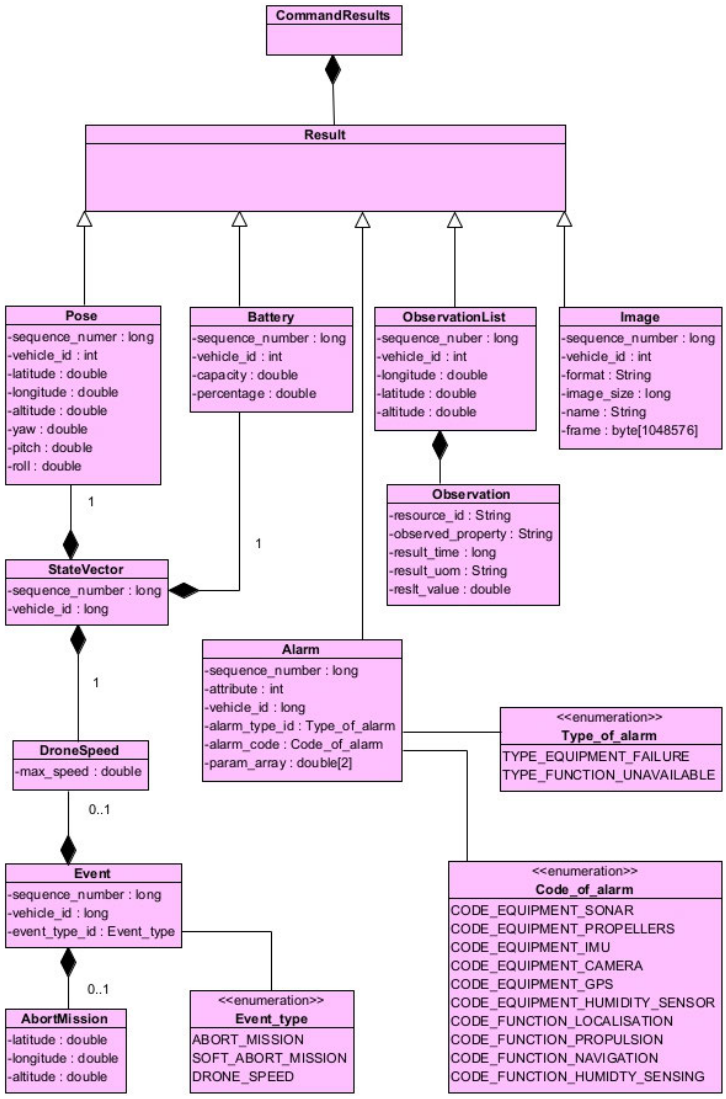
The information model in Figure 3 is based on the data model of AFarCloud, with the names and attributes of the messages and information exchanged.



a)



b)



c)

Figure 3. Information model: a) Mission, b) Mission Report and Latest State Vector and c) Command Resul

4.1.2. Class diagrams

The Figure 4 specifies the modules which compose and would be located in the edge, a simple device with limited resources and computational power. The main two are the FleetManager, a Django RESTFull Server, in the diagram is specified the API; the Broker, which can be, in one hand, ROS2Broker, to communicate with devices with this kind of communication protocol, mainly drones as the one chosen in this development and in the other hand, ISOBUSBroker, not implemented but taken into account in the design to manage autonomous vehicles with the ISOBUS protocol, tractors for example; and finally the data base, in this case a SQLite data base which saves the information contained in the requests to the FleetManager server, some of it permanently and other temporarily.

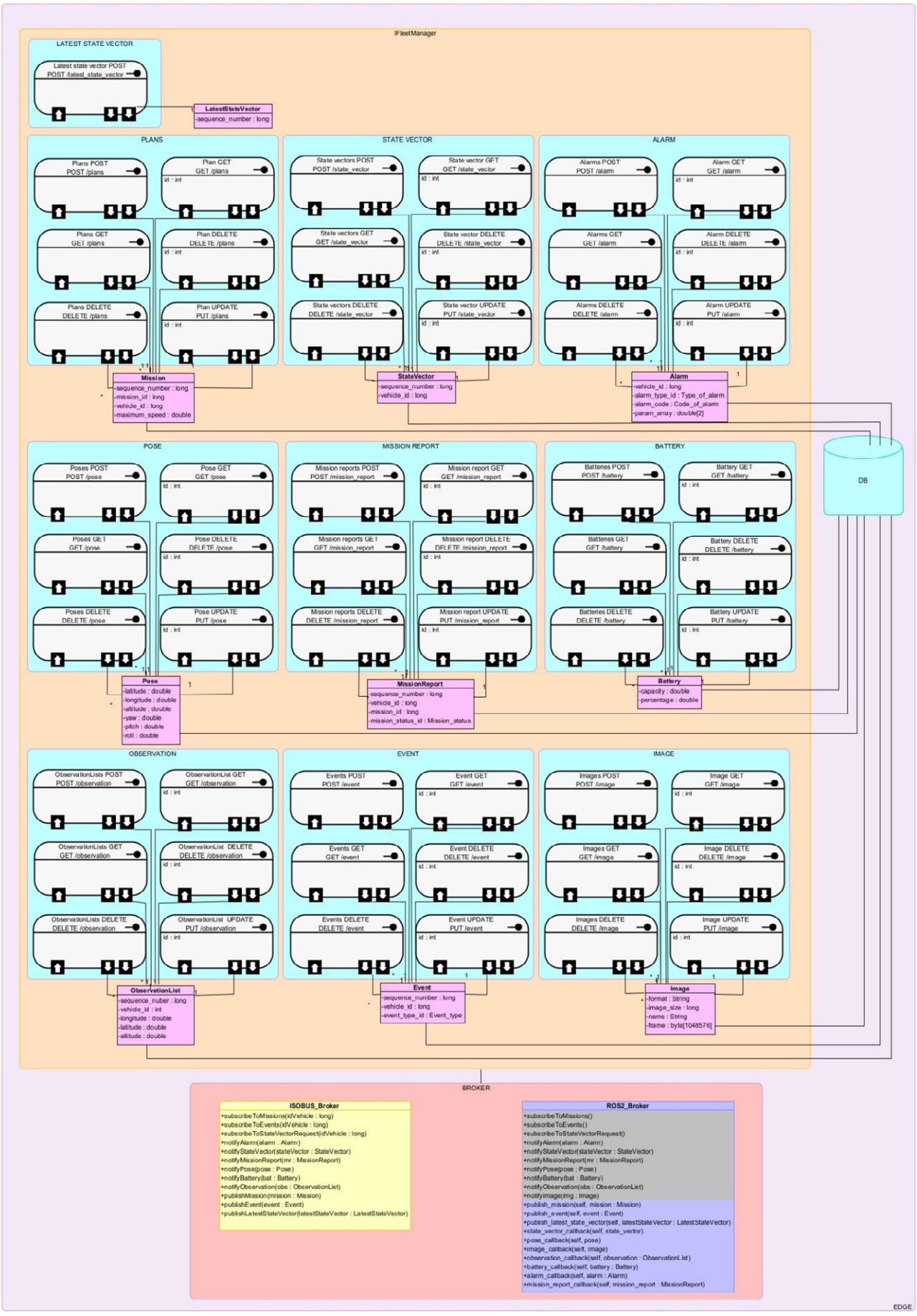


Figure 4. EDGE class diagram

The definition or implementation of the modules in the cloud are not a feature of this work but a simplification of it has been done to get a full experience of the system developed. The Figure 5 describes the minimum API required in the cloud to send the information obtained from the UAV. This implementation allows to present the normal function of the system with the temporal storage of the data.

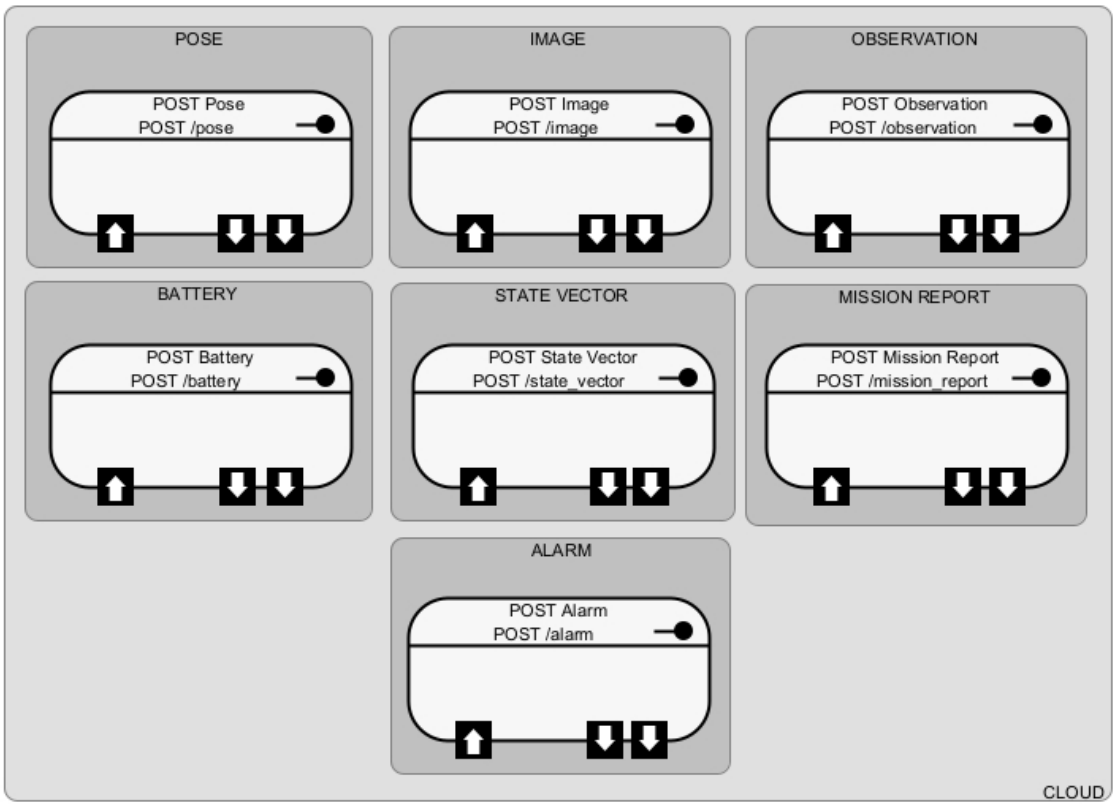


Figure 5. Minimum CLOUD class diagram

The Figure 6 illustrates the goal of the design: to have the most modularization as possible. With the definition of the interfaces the idea is to be able to change minimum parts of the design and get a full working environment without any change; for example, you can change the UAV, only changing the module of the driver, following this definition, or if the device is ISOBUS controlled, only changing the broker and, following this definition, changing the Unmanned Vehicle modules.

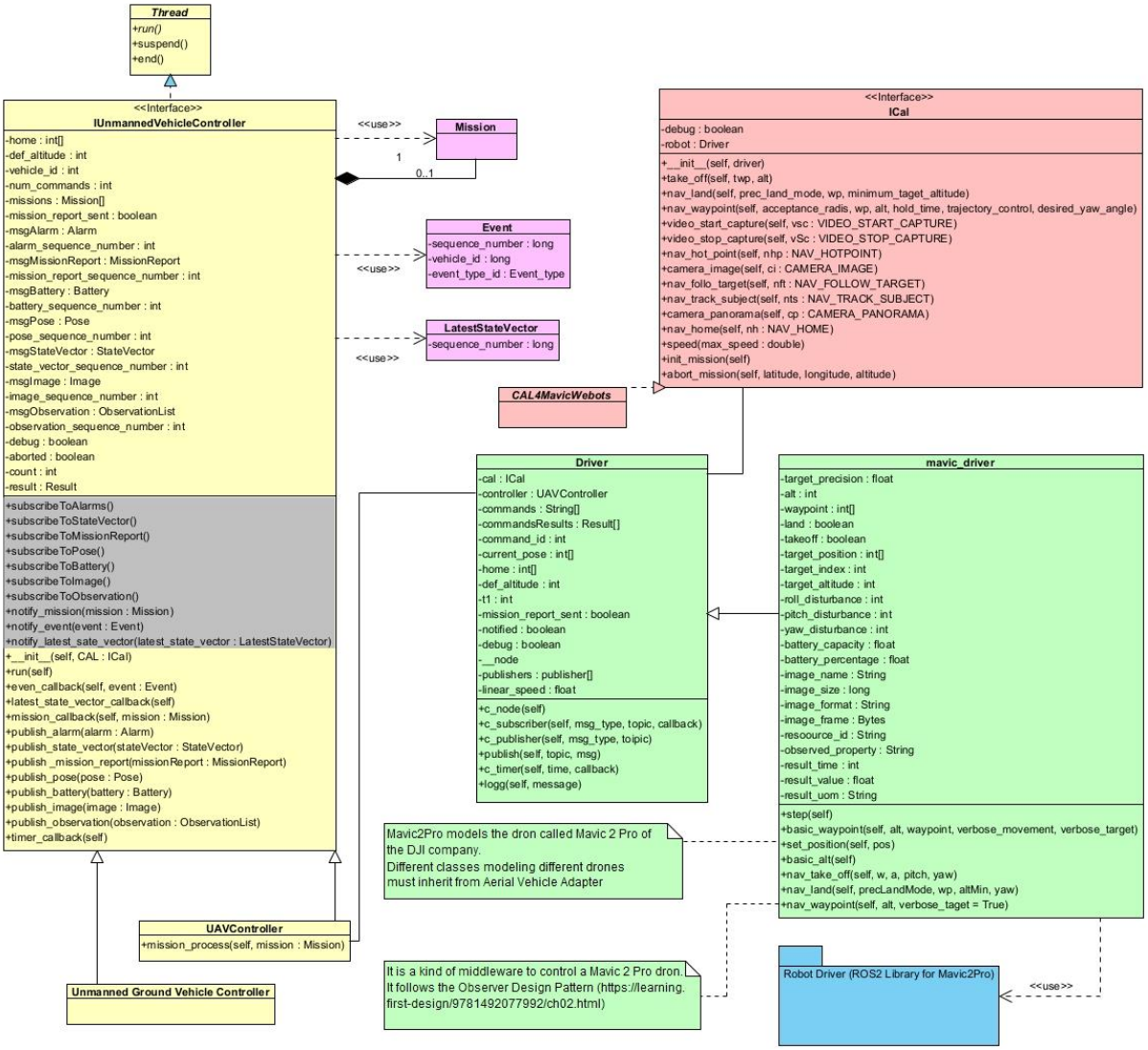


Figure 6. Unmanned vehicle class diagram

4.2. UML system design

4.2.1. Component diagram

The Figure 7 shows the modules design, the communication between them and the libraries or protocols used by each. In the EDGE, as mentioned above, the Fleet Manager with the Django server, communicated with the ROS2Broker, in this development case, through two sockets, one to send information from the server to the broker when a request is got, and the other one to send the data of the messages received from the UAV.

Both, the ROS2Broker and UAV use ROS2 python libraries to exchange information by publishing or subscribing to messages topics, and also, the driver, in the case of Webots drone, to communicate, manage and control the motors, sensors and parts of the drone.

The information exchanged between the modules in the UAV is with objects, arrays or queues filled by the driver and extracted by the UAV Controller or filled by the CAL and extracted by the driver.

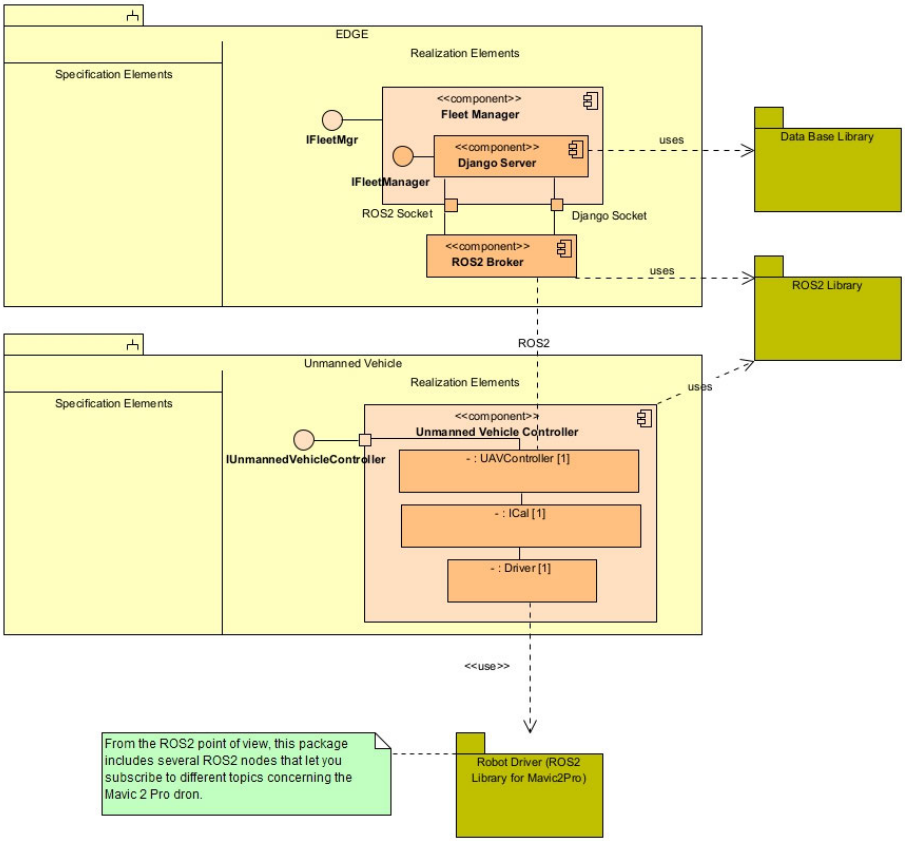


Figure 7. Component diagram

4.2.2. Sequence diagrams from EDGE to Vehicle

The collection of figures: Figure 8, Figure 9, Figure 10 and Figure 11 compose a main sequence diagram in a normal execution.

The common deployment is ordered: first the FleetManager and second the ROS2 Broker, to stablish the socket communication in both directions, then the UAV execution, which initializes the modules inside it, timers, publishers and subscribers to ROS2 topics, as shown in Figure 8.

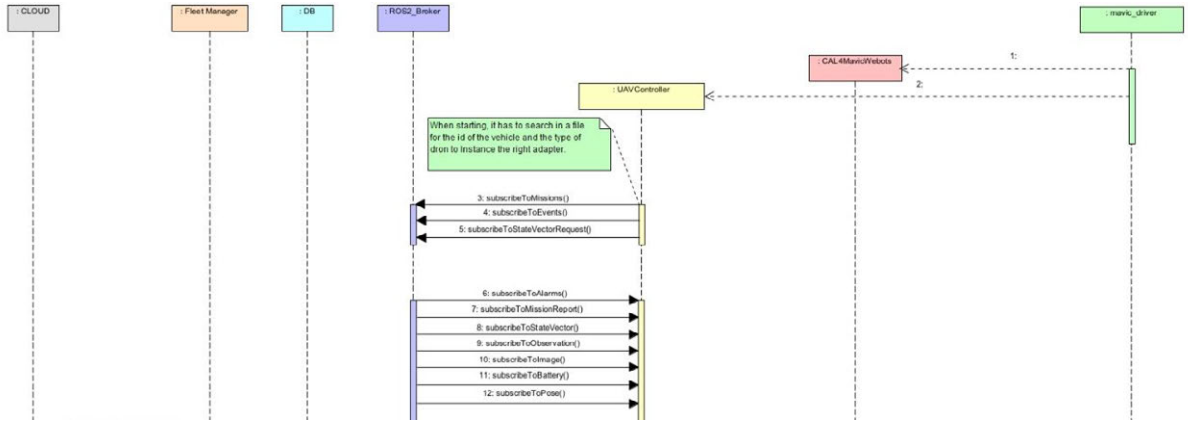


Figure 8. Normal initiation

In a normal situation, illustrated in Figure 9, the user from the cloud sends a plan to the EDGE through a request to the FleetManager server, in the body of the request there is a JSON plan which is saved in data base, and then sent to the ROS2Broker by the socket. When the ROS2Broker receives it, it parses the information in the JSON to a ROS2 message and publishes, in this case, to the mission topic. The UAV is subscribed to this topic, so it receives a notification through its callback with the message published by the broker, the controller manages it, it takes the information in it, first to detect if the message is for it or for other UAV, and then to split the commands and information and take the respective information and call the CAL functions that acts in the driver and gets the behavior desired. Also, it sets the information of the mission execution and report to be sent later.

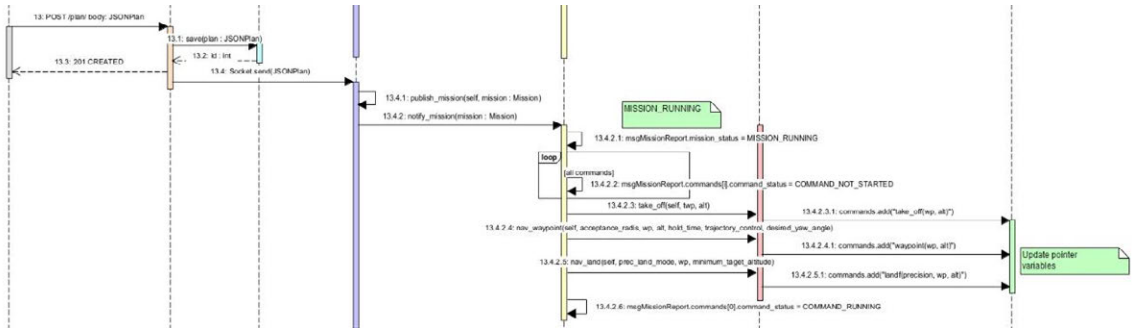


Figure 9. Normal Plan POST

A command result is made up of:

- Pose: contains the location and position of the UAV, latitude, longitude, altitude, yaw, pitch and roll. The last three parameters are commonly used to specify the drone's inclination in the three axes.
- Battery: composed of capacity and percentage.
- Image: made up of image format, size, name and the byte value.
- Observations: contains the location in which the observation is done and a list of detailed observations. This observation details are resource id, property observed, the time when was measured, the value measured and the units.

All of them optional, if the goal of the command finished is to get an image or to take an observation, a temperature measure for example or to take data from a remote sensor it can be included, if not, only a Pose and Battery result can be introduced. The driver introduced the result information, and the controller extracts it, it sets the last command in execution as finished and the next one as in execution and it publishes the information to the broker to be sent then to the cloud. When the Fleet Manager gets this information from the broker socket saves it in the data base, publishes it to the cloud and when a confirmation is got it is deleted. This process is illustrated in Figure 10.

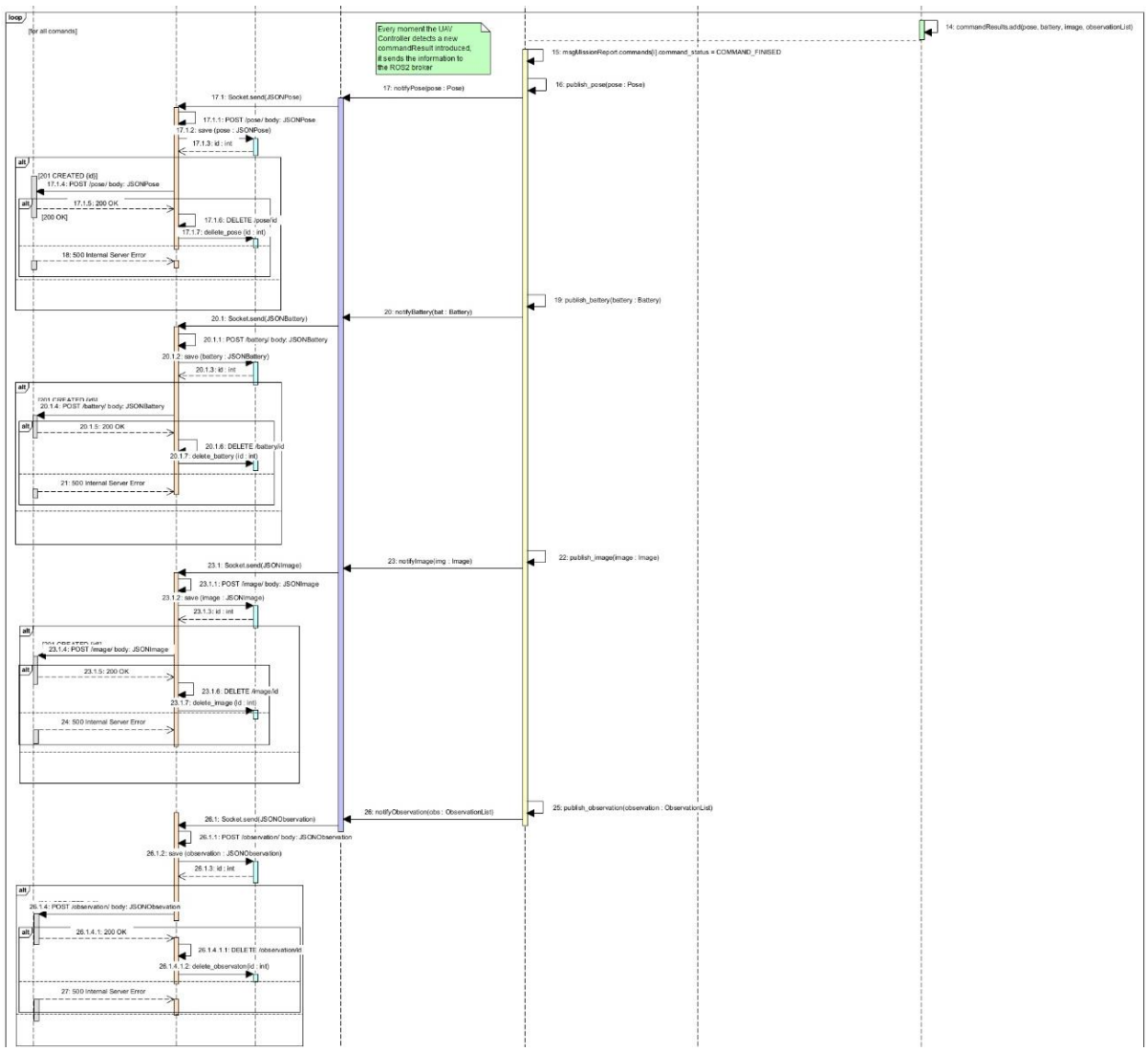


Figure 10. Normal command result set

Also, periodically the UAV publishes a mission report and a state vector message to its topics, and the broker gets the information because of its subscription to the topics and, in the same way as the command result information, it saves it in the data base, publishes it to the cloud and when a confirmation is got it is deleted, as shown in Figure 11.

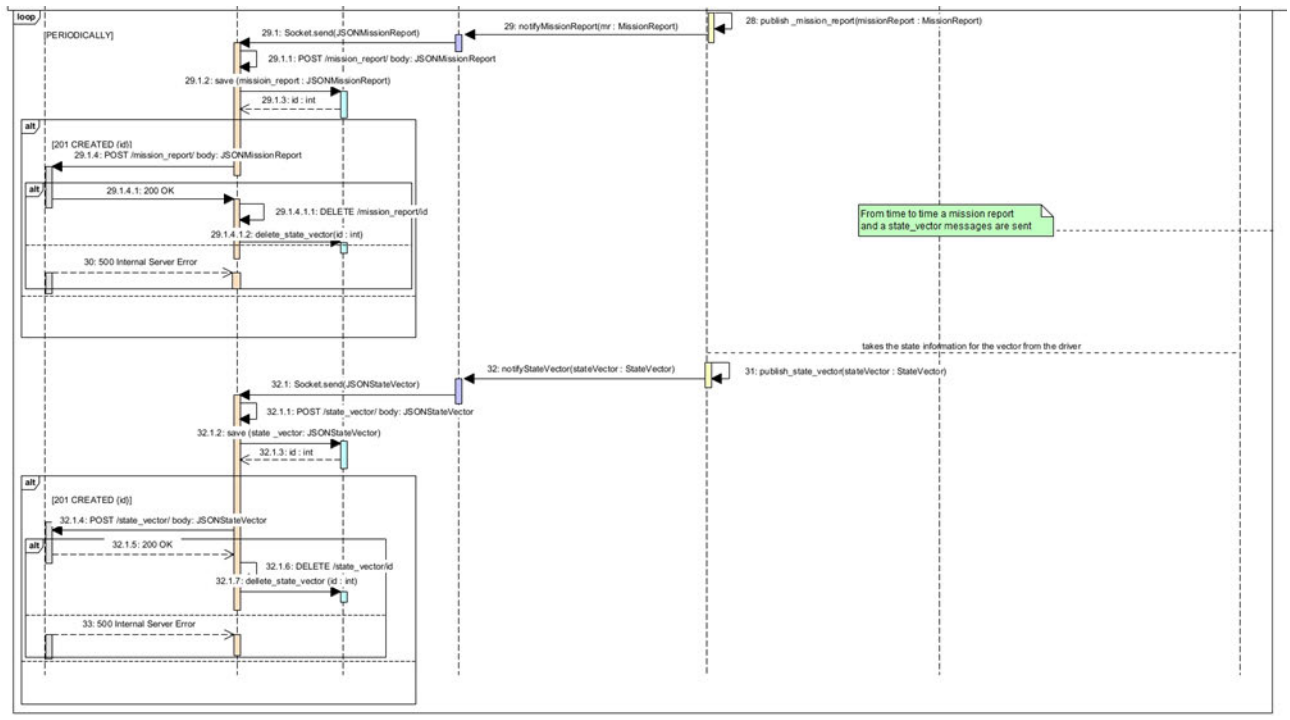


Figure 11. Periodic updates

When some kind of failure occurs in the UAV the driver introduces the specifications of it in the Results array and when the controller extracts it, it publishes the information in the alarm topic and it sets the command in execution as failed as well as the mission, and the rest commands as aborted, and publishes a mission report with this last update. Exposed in Figure 12.

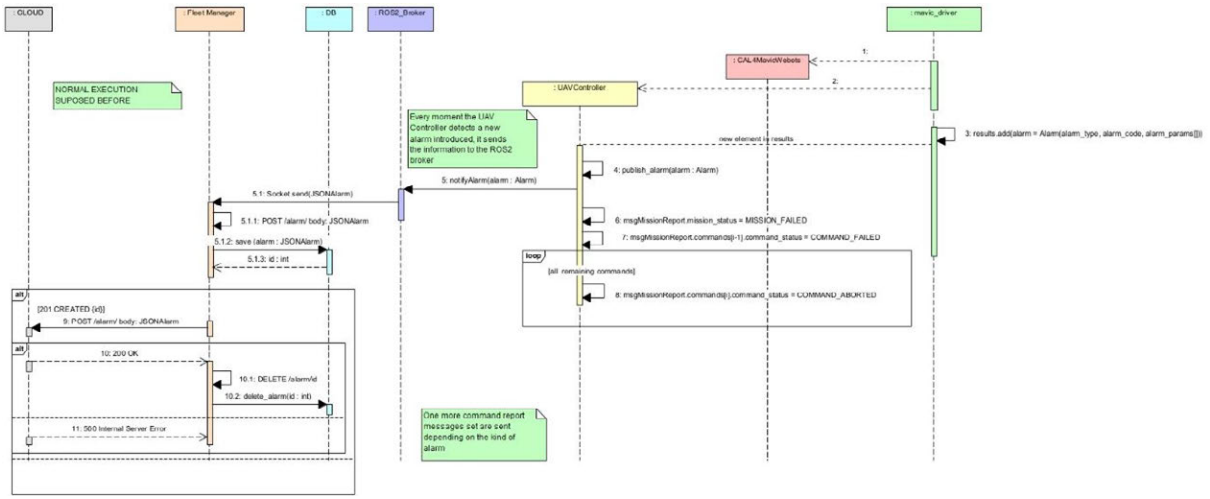


Figure 12. Alarm sequence diagram

In the other hand, when an event occurs, seen in Figure 13, and it's posted by the cloud to the Fleet Manager, it passes the information through the socket to the Broker, which parses the JSON into the even ROS2 message and publishes it to the UAV. In the Controller, when the call-back run, because of the message published into the event topic, depending on the kind of event the speed param is changed to the driver; or if the event is some kind of abort, it sets the remaining commands to aborted, and the mission too, it calls the CAL function to empty

the commands queue and depending on the kind and the parameters adds or not a final command to be executed.

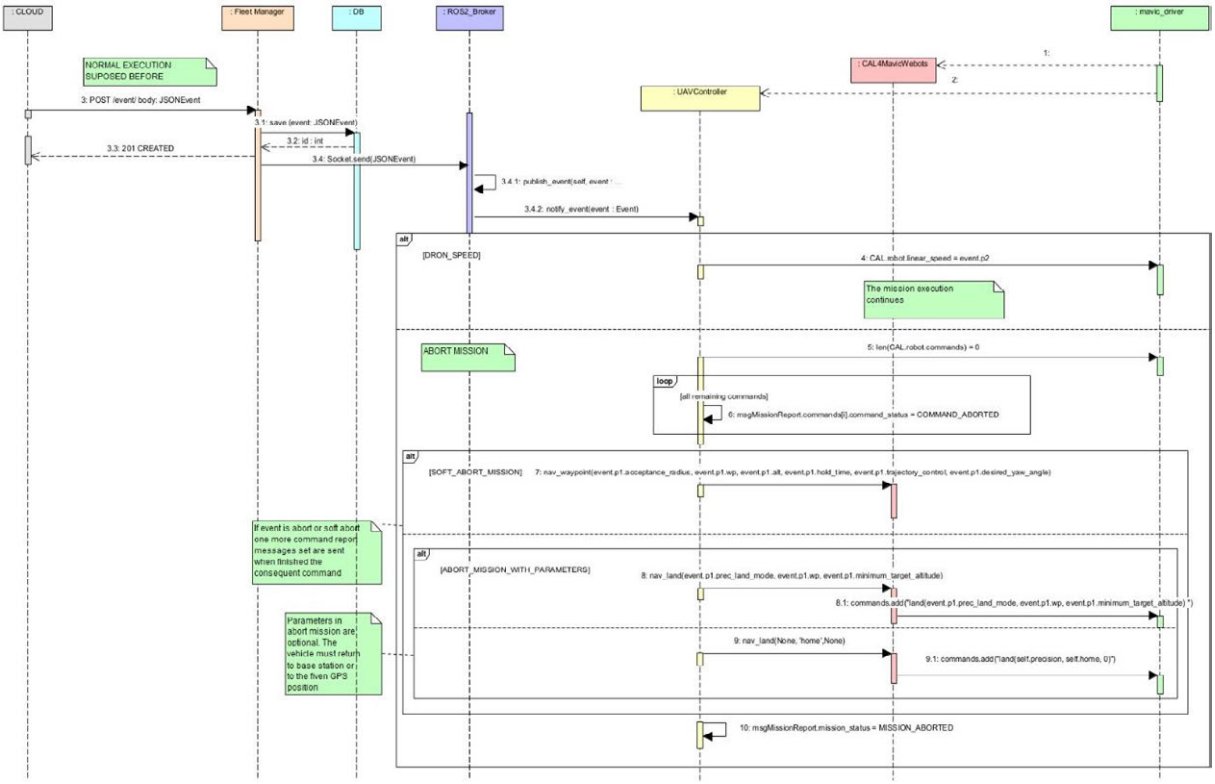


Figure 13. Event sequence diagram

At any moment of the execution the administrator from the cloud can send a latest state vector request to the EDGE server, which passes the request to the ROS2Broker, it publishes a latest_state_vector message to the corresponding topic and in response the Controller of the UAV takes the information from the driver and publishes a state vector, out of the periodicity, that follows the path back to the Cloud. Sequence shown in Figure 14.

The latest state vector request body is the only one which is not saved nor temporarily or permanently in the EDGE database, because it only contains a sequence number, so no important information needs to be saved. The state vector response from the drone is actually saved.

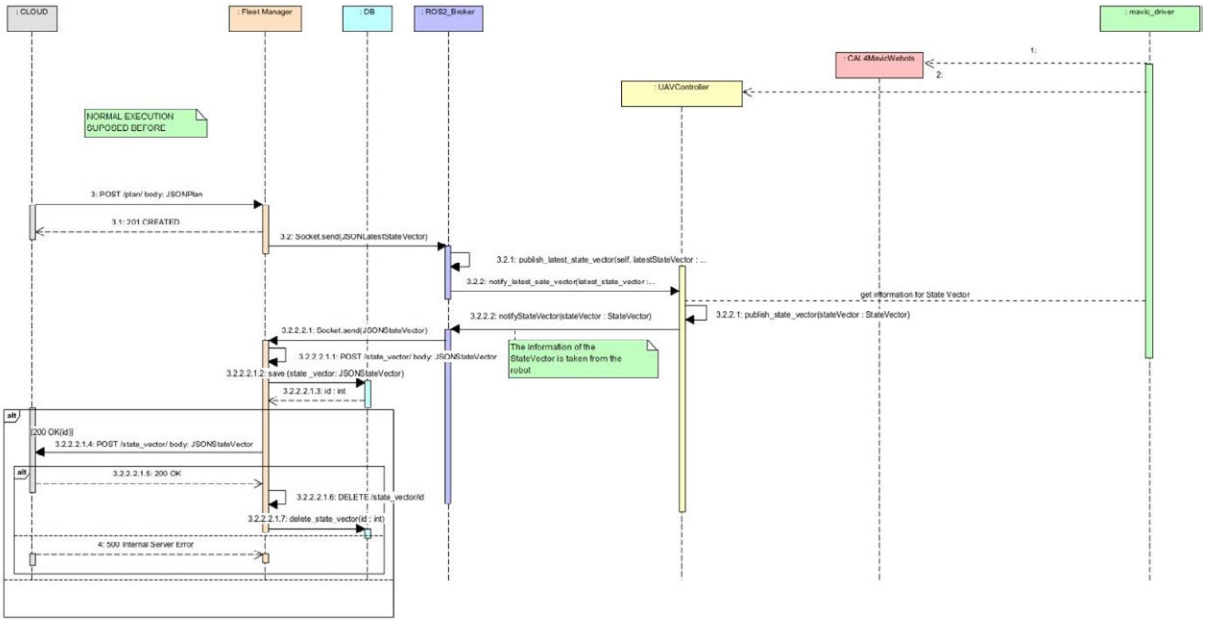


Figure 14. Latest State vector sequence diagram

4.3. Design details

4.3.1. ROS2 Nodes and topics

The ROS2 nodes and topics published and subscribed to are explained in Table 2.

Table 2. ROS2 nodes and topics

ROS2 nodes	**ROS2Broker**	**mavic_driver**
TOPICS SUBSCRIBED TO	• alarm <u>Periodically reported</u> • state_vetor • mission_report <u>Command execution result</u> • pose • battery • image • observation	• mission • event • latest_state_vector
TOPICS PUBLISHED BY	• mission • event • latest_state_vector	• alarm <u>Periodically reported</u> • state_vetor • mission_report <u>Command execution result</u> • pose • battery • image • observation

There are also another two ROS2 nodes, which appears in the execution of Webots: Mavic_2_PRO, and ROS2Supervisor, used to control the robot in the simulation with the external controller.

The ROS2 messages defined, using the instructions of in the ROS2 humble tutorial [22], and described in Table 12 and Table 11 of the A.1. They are based on the D2.6 Semantic Middleware of the investigation project AFarCloud [23].

4.3.2. DJANGO Server API

Based on the different information and messages exchanged between the ROS2Broker and the UAV, the respective models and serializers have been designed to save, temporarily or permanently, this information exchange. In order to manage the data base in a simple way, all the body requests are JSON kind, which is the deserialized by the server and saved in the database.

The Django server API contains all kinds of possible request from the cloud, relative to the plans and events, with all the CRUD operations, but for the latest state vector, and just the create (POST) option is provided because the information in the body is not relevant to be saved, deleted, updated, or requested, it is simply transmitted.

Also, in this API there are all CRUD requests related to the topics from the UAV. All these requests can be done from cloud, but they are expected to be used from the own server, to create and delete the temporary data.

In one hand the information and request from the cloud, as plans or events, the information is kept permanently in the data base. The information from the UAV is saved temporarily in the data base until it is sent to the cloud and an ok is get, then the information saved before is deleted. In the other hand the requests about latest state vector are simply transmitted, but not saved.

In this API documentation there are all of these requests mentioned, and some examples (request and body) to test out the communication, the module and the full system.

https://documenter.getpostman.com/view/30533683/2sA2r8145F

4.3.3. Cloud API

A future implementation or deployment should have a Cloud module, with all the processing, data bases and the graphic environment for the farmer to use the system, but for the moment it is simplified using a mock server in Postman with the POST requests required to test the system and complete the process of saving and deleting the temporary information.

The following documentation shows the needed API, which can be used to create a mock server using Postman, which is needed to execute this proof of concept.

https://documenter.getpostman.com/view/30533683/2sA2r8145N

4.4. Implementation

The most relevant modules deployed are briefly described below.

4.4.1. Drone simulation

The UAV Module is split in three:

UAV Driver, in charge to execute the actions in the drone, simulated or not, as activate, increase or decrease the speed of the motors, or taking the information of the position sensor for example, used and sent in the different messages; it is specific for the drone used.

The Component Access Layer, also specific for the UAV used, it implements the Control Access Layer Interface with the corresponding commands expected in this design. This is one of the points for future works, because only the nav_land, nav_home, nav_take_off and nav_waypoint commands are implemented, as well as the alarms which are only for abort a mission, in the two modalities, but not to change the drone's speed.

Finally, the UAV Controller, in charge of the communications, publish and extract the messages from the respective topics, as well as extracting the command result from the UAV to be published, or command the other submodules to produce the actions from the information of the messages received.

This definition of module and submodules is used to get the most code reutilization as possible, and making easier the change of the UAV, from the one simulated, tested or any physical or simulated one.

The code which is in the folder webots_ros2_mavic_afarcloud was developed based on the ROS2 Webots generic project [24].

4.4.2. UAV Driver

The main file of this module is mavic_driver, the specific UAV Driver for the Mavic 2 Pro simulated drone in Webots. It was developed with the example [25] as initial point.

It is composed of some variables and parameters used during the execution to get the correct movement, all of them taken from the example. The initialization, with the collection and instantiation of all the devices of the robot such as motors or sensor for their subsequent control. The description of the ROS2 node: mavic_driver, and the instantiation of the CALMavicWebots element, and with it, the instantiation of the UAVController, used to manage the ROS2 communication.

It contains different functions used by the UAVController to create subscriptions, publishers and timers and to publish messages into topics, as well as to show logging information on the prompt. This is because in the way that Webots initialize the drone is giving it the ROS2 Node to manage the different devices of the drone, and the tools to keep it alive and create more nodes and communications needed.

After that you can find some functions defined to control the Webots simulated drone: the step function is executed in every step of the simulation; in it, based on the example one, all the parameters needed to control the motors are read and updated from the sensors or calculated from them and the target altitude and position. It also put the final command result when the drone is detected to be landed.

The other functions can do the calculations previously commented or set the target position and altitude from the information extracted of the commands, to get a correct behavior. Also, it detects when the actual position is enough similar to the target position, when it sets the command as ended, introducing the command result set to the array consulted by the UAVController and passing to the next command execution.

4.4.3. Component Abstract Layer

The second submodule of the UAV component is the CAL4MavicWebots, whose functions are called from the UAV Controller during the message processing to manage the actuation to the parameters of the UAV to achieve the movement wanted and the behavior expected.

This element fills an array with the consequent ordered commands of the drone, which actually act on the paraments, and that is extracted by the mavic_driver as commented before.

It is in charge of introducing, deleting or changing the actions in that array, for example introducing the corresponding ones when a Plan is received, or deleting all of them and introducing a last one when an abort event is received.

4.4.4. UAV Controller

The third one is the UAV Controller, run as a thread and is in charge of the communication and coordination of the rest of the submodules. It has some parameters such the UAV identification, the default altitude or the home coordinates. Also, it has some flags, to avoid sending messages when the mission is ended, all the messages and sequence numbers for them.

In its initialization it creates the node, the subscriptions, the publishers, and the timer to manage the ROS2 communication, sending and receiving messages, and periodic information published.

The thread condition of this module is used to check in a loop if a new result is introduced by the drone when an alarm occurs, or a command is finished. In that case it extracts it from the array make up the message corresponding to the topic, and it publishes it.

It contains the subscribed topics call-back functions, executed when those messages are received, a timer call-back function used to publish periodically the state vector and mission report messages, which parameters are updated during the execution of the system; and the functions to make up the messages to the different topics, used to publish them.

The mission call-back acts when a mission(plan) is received. First it checks if a plan is running, in that case it saves that mission to be executed when the actual one has finished, if not it extracts the information of each command and its parameters to call the CAL functions for each and include them in the commands array of the robot as told before

The event call-back is only implemented to take action when the event type is aborting mission, for the moment. It defines all the unreached commands and the mission status as aborted in the mission report, and it calls the abort mission function to the CAL to update the commands for the drone. In addition, it publishes a mission report message out of the periodic ones.

The latest state vector call-back simply publishes the state vector message with the actual parameters.

4.4.5. Django server

The development of the Django server has been done following the steps in [26] to create a basic RESTful Web Service with Django REST Framework. During the execution some libraries were out of date, so I needed to check error by error looking for the mistakes and looking for the actual versions of the libraries and how those actions are done with them. After looking for a lot of documentation and people with the same errors as me in similar, or not, projects I substitute all the error functions, modules and lines with the recommendations and documentation [27], [28], [29], [30], [31], [32] I found everything started working right.

With this simple Django server, in which you can make requests with a JSON in the body, and it is saved in the corresponding table of the data base, if it is correctly formatted. I started introducing all the models, serializers and views of the data model, those which are needed to be saved, or requested but not saved, as the latest state vector. The full API is documented and commented in the 4.3.2 section.

This server begins opening a socket and waiting for de ROS2Broker connection, which next opens its socket which this module connects to then. In this point the server is open for requests, those with the correct format in its body are saved in the data base.

When a request a question of type plan, event or latest state vector is received it is saved in the data base and the JSON of the body is sent to the ROS2 Broker by the socket to be then parsed and sent to the UAV which is the real destination.

When information is received from the socket connected to the ROS2 Broker, that is in JSON format, it makes an intern request the server to save the information temporarily. Then it makes a similar request to the cloud, and in the moment a 200 Ok is received another internal request is done to delete the data saved before.

This way, following the typical concepts of the EDGE computing, as few information is saved in the EDGE, only until it is sure is really saved in the cloud, in which there are space enough to save and display all the information exchanged.

In any case, to manage or consult the information of the data base (DB), the full API can be used from anywhere.

4.4.6. ROS2 Broker

The main function of this module is to connect the server, interface to the cloud the UAV, serving as interface between them. It was developed based on the simplest example of a ROS2 topic subscriber, as explained in [33].

It first looks for the Server socket connection to continuo opening a second socked and waiting to the server connection, as mentioned before. In its initialization it has these connections, and the subscriptions and publishers to the ROS2 topics mentioned in Table 2.

In one hand, when a JSON is received through the socket it takes all the information in it to fill the corresponding ROS2 message and publish it to the drone. In the other hand, when a ROS2 message is published to any of the topics which it is subscribed to, it takes the information to make up a JSON that then sends to the server through the socket.

4.4.7. Substituting the UAV: tester

Another module has been deployed in order to test the communication at the moment of the deployment on the fields is called *tester,* based also in [33]. In it the user can find a substitution of the UAV, all the actions that the UAV used to do, as introducing the information when a command is finished, or updating the position, the battery information or setting an alarm, in this module is selected by the user using the keyboard.

When a mission is received and while the mission is not finished, failed, or aborted a set of options are presented to the user to be selected, this is the main menu:

```
Mission running
      1. Set command as finished
      2. Update position
      3. Set alarm
      4. Update battery level
      5. Update observation info
------ Enter the action number:
```

As long as the user types a number other than 1, 2, 3, 4 or 5, this message is shown again.

1. When the user types ‘1’:
   The PoseInfo, BatteryInfo and ObservaionListInfo, and the ImageInfo (in future works) elements are added to the array with the current latitude, longitude, altitude, yaw, pitch and roll info, the battery capacity and percentage and one observation information: resource_id, observed_property, result time, value and units. Then, as in normal execution with the simulator, the UAVController extracts and publishes them.
2. When a ‘2’ is typed, this set of messages are shown and it waits for the user to enter the new information by the keyboard, line by line:

```
- Enter the new latitude (if not an int it will be generated
randomly):
- Enter the new longitude (if not an int it will be generated
randomly):
- Enter the new altitude (if not an int it will be generated
randomly):
- Enter the new yaw (if not an int it will be generated
randomly):
- Enter the new pitch (if not an int it will be generated
randomly):
- Enter the new roll (if not an int it will be generated
randomly):
```

All these parameters must be floats, if some of the information introduced by the user to update a parameter is not a float or is empty, that parameter is generated randomly between 0 and 100.

3. When the '3' is selected, you can choose between:

```
Select the alarm type
      1. Equipment failure
      2. Function unavailable
      3. Back
------ Enter the selected type number:
```

If the text written is not an integer, the program requests for a correct answer, if that answer is different from 1, 2, or 3, the message is shown again, waiting for a correct one.

1. If the Equipment failure is selected the options are:

```
Select the alarm code
      1. Equipment sonar
      2. Equipment propellers
      3. Equipment IMU
      4. Equipment camera
      5. Equipment GPS
      6. Equipment humidity sensor
      7. Back
```

If the text written is not an integer, the program requests for a correct answer, if that answer is different from a number between 1 and 7, the message is shown again, waiting for a correct one.

2. Otherwise, if number 2 is chosen, Function unavailable the options are:

```
Select the alarm code
      1. Function localization
      2. Function propulsion
      3. Function navigation
      4. Function humidity sensing
      5. Back
------ Enter the selected code number:
```

Again, if the text written is not an integer, the program requests for a correct answer, if that answer is different from a number between 1 and 5, the message is shown again, waiting for a correct one.

After any of the choice of an alarm code different from back, two float parameters are requested to the user and then the AlarmInfo element is added to the command result array to be extracted by the UAVController and the corresponding actions are taken. After this the main options can't be chosen, only typing a '0' the drone is set as landed and it sends the final command result and mission report information.

If a back option is selected, it breaks the wait for a correct alarm code or alarm type, and it returns to the previous menu option.

4. When '4':

```
Select the battery param to change:
      1. Capacity
      2. Percentage
      3. Back
```

If '1':

```
      ------ Enter the new battery capacity (mAh):
```

With the value introduced it updates the battery capacity that is introduced in the Battery message next time a command is set as completed.

If '2':

```
      ------ Enter the new battery percentage (%):
```

It waits for an integer value between 0 and 100, and different from the previous one, if after two opportunities to enter a correct value, if it is not obtained, it returns to the previous menu.

5. In the case of '5':

```
Select the observation param to change:
Resource id
Observed property
Result value
Result uom
Back
```

When selecting the options from 1 to 4:

```
------ Enter the new resource id(string):
------ Enter the new observed property(string):
------ Enter the new result value(float):
------ Enter the new result uom(string):
```

If the value is not valid the change of the parameter is aborted and it goes back to the main options menu, as well as if the 5th options is selected.

In the moment a mission is set as aborted or failed this message is shown:

```
Enter '0' to set the drone as landed after the mission
abort/alarm:
```

Waiting in a loop for the user to write ‘0’, to set the drone as landed after the event, and when it happens, the current positions is added to the queue too.

In the moment a mission is finished, the main menu is no shown again until another mission is received.

The complete split of the code into different python modules has been achieved thanks to the forum question and answer: [34]

4.4.8. Dockerization

At the beginning of this development all the modules where coded, tested and executed directly on a Linux computer. When all the features and modules were tested and considered as finished, the Dockerization began.

To get an easier execution environment and an easy way to execute and test the design, the modules in the edge were converted into docker modules.

As a good enough image of ROS2 humble was not available, it was decided to make one. Thus, a Dockerfile based on an image of Ubuntu 22.04 was created, in which all the tools needed to execute the ROS2Broker module were installed. With this Dockerfile the image is built, and then with the run command of the Docker the module is executed the same way as in the host computer due to network configuration and open ports in the container.

In the case of the Fleet Manager, the Django server, the Dockerfile base is an image of python 3.10.12, in which it installs the requirements, but as mentioned in 4.4.5, some files and lines had errors because of the different versions so those modified files are substituted in the container. Then with the run command of the Docker the server is woken up and the connection with the Broker socket and so on happen normally due to network configuration and open ports in the container.

To allow testing the full system from any device the Tester module is also Dockerized, in a similar way that the ROS2Broker. In the case of using Webots to simulate the drone, the Fleet Manager module and the ROS2Broker module are executed in Docker and the UAV in the host computer.

The execution can be done in a block, using the docker composer, and then accessing to the logs in different terminals to ensure the normal or correct execution, or module by module with the execution of each container separately, seeing directly the logs. All the options and better instructions are specified in a README file in the folder or in the Execution section.

5. Results

5.1. Communication tests

To ensure the proper functionality and behavior of the system all the communications were tested separately.

5.1.1. Cloud – EDGE communication

The interface between the cloud and the edge is through a REST API, so a bench of test was deployed and executed using Postman, it has all the possible operations with their respective URLs, bodies, and headers. It was executed with correct parameters and information and with wrong one: JSON bodies with incorrect format, entries, or types. The result was some issues in API definition, for example the return information, finally defined as the id in create operations and a simple OK or not OK in the other operations, and also, a refining process of the API URLs and parameters.

Due to the association between the Interface of the edge with the Django server models and serializers definition with these tests also some problems were detected in that definition. In the execution some wrong model types and parameter widths needed to be adapted for a correct execution with nearer to real data.

This helped to debug the models, endpoints, and operations, and ensure the correct behavior.

In the other direction, in the communication from the edge to the cloud the tests were similar but simpler, because the Cloud module is not a feature of this project and it is only a simplification of it, as mentioned before in this document, so only the needed operations were tested in order to have a way to do a proof of concept without a real Cloud model and module.

After all the testing and the debug, the correct behavior was achieved, and the errors detected and responded properly.

5.1.2. Fleet manager – ROS2 Broker communications

Having the correct way to get the JSON information from the cloud in the Fleet Manager, the Fleet Manager to ROS2 Broker evaluation began.

This test consisted in sending the JSON objects through the socket to the broker, check if it is received correctly in the destination and no collisions occurred. The same in the other direction, that allowed to also check the continuity of this information to the cloud when it is received.

The result was the need to have a perfect order execution of the modules involved to stablish the two sockets between them avoiding block situations. First the Fleet Manager opens the socket for connections and waits for a new one, and the broker first looks for a connection to the socket of the Fleet Manager. Then the broker opens a new socket and waits for a

connection, completed by the Fleet Manager at that moment. After this ordered connection establishment the information exchange can begin.

The exchange hadn't any complications, so the validation of the model continued.

Once the JSON arrived at the broker, it is parsed to a custom ROS2 message, so in the test some errors in the compatibility of the types or widths appeared and the proper adaptations were made.

5.1.3. ROS2 Broker – Unmanned Aerial Vehicle

In the moment of having correct parsed ROS2 messages in the broker, the ROS2 exchange was ready to happen, and due to the use of a simulator it was impossible to test this interface so the test module explained in 4.4.7 section was used to substitute the drone and test this interface. It can only be used, not only for this communication check, but also for a real communication check in a real deployment, even this was its original objective.

These tests consist in publishing example messages to the topics to check the subscriptions in both modules. This was achieved with some ROS2 native tools to create temporary subscribers and publishers and make some metrics related to the topics, as number of messages published, number of nodes associated with a topic as either subscribers or publishers…

5.1.4. Unmanned Aerial Vehicle modules information exchange

Because the UAV Controller is a thread, to keep checking for new results (Observation, Image, Pose, Battery or Alarm) produced by the controller, some concurrence tools were evaluated to be used, like queues, and locks instead of using a simple array to exchange command results.

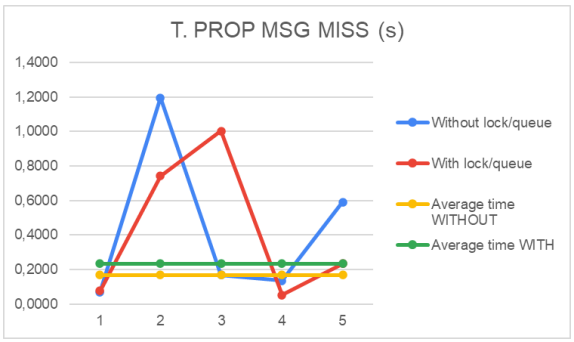
In this test the result array was converted to a queue, and a lock was integrated for locking and unlocking the enqueue and dequeue procedures.

In Table 3 are the results of taking times and memory consumption in 5 executions of the tester module, first without and then with the concurrence tools for different message types. Rounded in dark blue the measures without locks and queues and rounded in light blue the ones with them. The first column contains the propagation times of each message, result of differencing the timestamps of its send and reception. Next columns contain the processing time, result of differencing time between the messages are received and then the execution begins, in the case of mission message, or the JSON object is ready and sent through the socket, in the rest of the cases where the information is produced by the UAV. Finally, in last column is the memory consumption or release during processing.

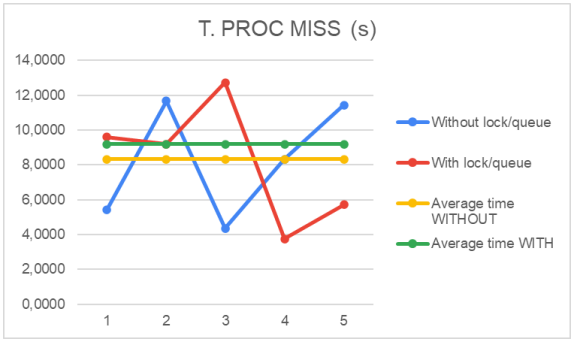
Table 3. Times and memory consumption

T. PROP MSG **MISSION** (s)	T. PROC MISS (s)	MEMO. PROC MISS (kB)	CONCURRENCE
0,0701	5,4378	12,5000	without lock/queue
1,1936	11,6616	0,2510	without lock/queue
0,1683	4,3562	2,5120	without lock/queue
0,1348	8,3149	2,5760	without lock/queue
0,5915	11,4249	2,5760	without lock/queue
0,0761	9,5867	2,5760	with lock/queue
0,7424	9,1889	40,0000	with lock/queue
1,0021	12,7282	14,6000	with lock/queue
0,0522	3,7504	14,7000	with lock/queue
0,2326	5,7156	14,0000	with lock/queue
0,1683	8,3149	2,5760	without lock/queue
0,2326	9,1889	14,6000	with lock/queue
T. PROP MSG **MISSION REPORT** (s)	T. PROC MISS REP (s)	MEMO. PROC MISS REP	CONCURRENCE
0,0307	2,4335	23,6000	without lock/queue
0,2827	1,9078	19,0000	without lock/queue
0,0941	0,1508	18,1000	without lock/queue
0,0386	0,0019	18,6000	without lock/queue
0,2583	0,3267	17,5000	without lock/queue
0,0017	0,0025	0,3930	with lock/queue
0,0017	3,1328	22,4000	with lock/queue
0,0021	0,0028	3,2160	with lock/queue
0,3138	0,5641	-24,7000	with lock/queue
0,0014	1,1931	21,5000	with lock/queue
0,0941	0,3267	18,6000	without lock/queue
0,0017	0,5641	3,2160	with lock/queue
T. PROP **POSE** (s)	T. PROC POSE (s)	MEMO. PROC POSE (kB)	CONCURRENCE
0,3288	0,0023	0,4400	without lock/queue
0,0528	0,0080	0,4400	without lock/queue
0,0888	0,0010	0,4400	without lock/queue
0,0175	0,0011	0,4400	without lock/queue
0,2364	0,0010	0,4400	without lock/queue
0,2823	0,0102	-30,0000	with lock/queue
0,2823	0,0032	-0,4940	with lock/queue
0,2261	0,0025	0,4400	with lock/queue
0,2822	0,0018	0,4400	with lock/queue
0,2821	0,0024	0,4400	with lock/queue
0,0888	0,0011	0,4400	without lock/queue
0,2822	0,0025	0,4400	with lock/queue
T. PROP **EVENT** (s)	T. PROC EVENT (s)	MEMO. PROC EVENT (kB)	CONCURRENCE
0,7256	2,1675	16,5000	without lock/queue
0,0753	5,2577	24,3000	without lock/queue
0,0068	2,6905	-37,2000	without lock/queue
0,2187	4,1161	15,1000	without lock/queue
0,4804	2,5370	-4,8880	without lock/queue
0,0004	4,3267	15,9000	with lock/queue
0,0822	3,9072	23,7000	with lock/queue
0,0200	5,1074	26,8000	with lock/queue
0,0285	3,9305	26,3000	with lock/queue
0,1756	4,0327	26,5000	with lock/queue
0,2187	2,6905	15,1000	without lock/queue
0,0285	4,0327	26,3000	with lock/queue
T. PROP **ALARM** (s)	T. PROC ALARM (s)	MEMO. PROC ALARM (kB)	CONCURRENCE
0,0276	0,8021	13,1000	without lock/queue
0,0000	1,0185	16,5000	without lock/queue
0,1930	1,1472	15,4000	without lock/queue
0,0244	0,1542	16,2000	without lock/queue
0,0232	0,1433	16,8000	without lock/queue
0,0980	2,5423	20,8000	with lock/queue
0,0380	4,9652	24,8000	with lock/queue
0,8511	6,2975	6,8860	with lock/queue
2,2047	8,1485	-9,2640	with lock/queue
0,8086	6,2925	5,8230	with lock/queue
0,0244	0,8021	16,2000	without lock/queue
0,8086	6,2925	6,8860	with lock/queue

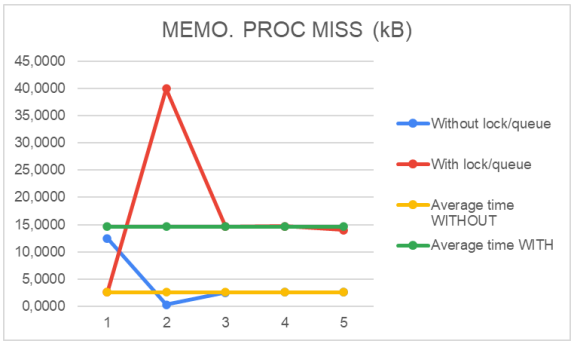
In Figure 15 are disposed the previous table results in graphs to appreciate a better comparison of the values. For all the cases the blue line is used for measurements without locks and queues, and the red one for the measures including concurrence tools. Lines yellow and green are used to mark the average value with and without the use of those tools, respectively.



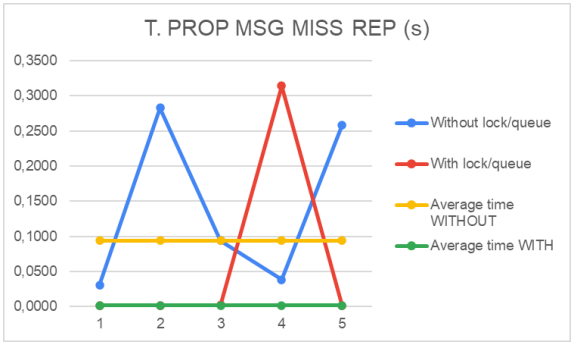
a)



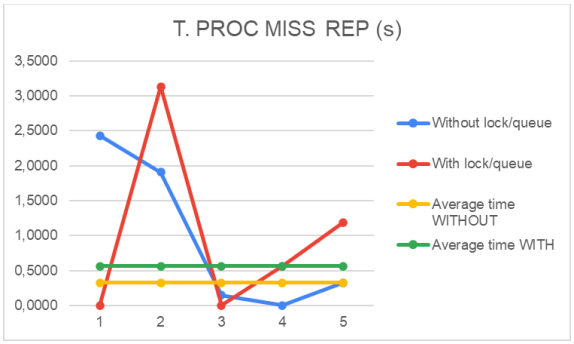
b)



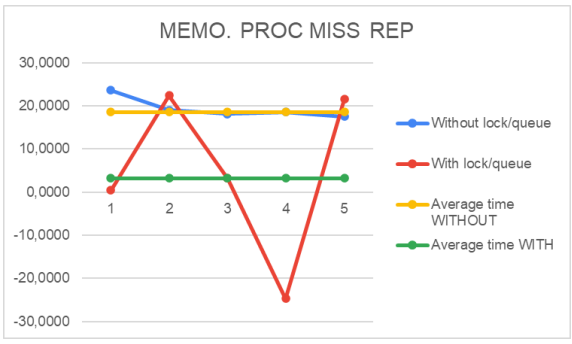
c)



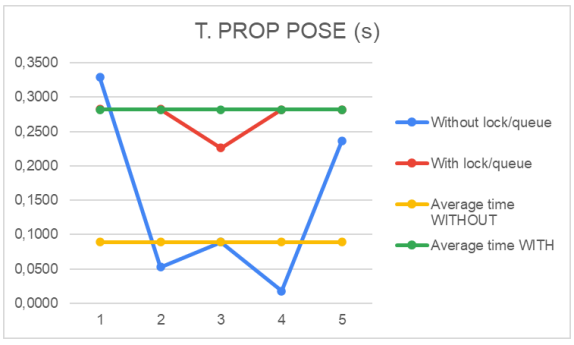
d)



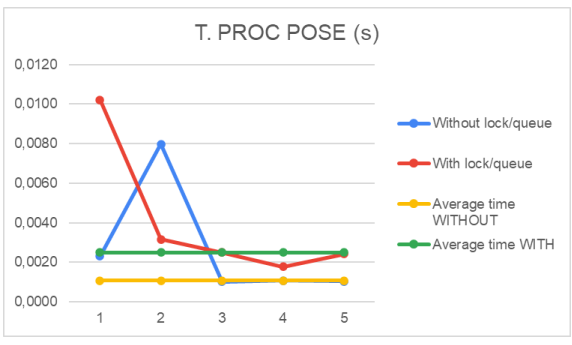
e)



f)



g)



h)

Figure 15. Propagation time, process time and memory used during process for the different messages: a), b) and c) Mission, d), e) and f) Mission report, g), h) and i) Pose, j), k) and l) Event and m), n) and o) Alarm.

This experiment shows whether the communication times are feasible or not to the application. Seeing this table and figures we can appreciate that in most of the cases the time increases significantly.

In the case of memory consumption, the measurements are more different, because the use of memory depends not only on the data exchanged but also in the information

saved/exchanged before, so that, not a clear conclusion can be extracted. But it is easy to see that the data used is more dissimilar in with concurrence tools.

The propagation time depends on the network state and should be independent of the use of concurrence tools, but in this experiment, it can be appreciated that the propagation time is much higher with queues and locks. A bigger use of the Central Processing Unit (CPU) may be the reason. However, the average total value is 0.5961 seconds, acceptable in any case.

The disparity in the test executions results is due to the use of random generated params values in it, same range but different values in each case and the relation of the results with previous executions, CPU state, network state, etc. So not each point can be compared with the same number execution of the opposite condition, only the general disposition of points from the other.

Concluding, using just one UAV Controller to extract the results and one UAV to produce them highlights that the concurrence potential problems are too few to explain the delay observed in time process, and no problems of this kind have never been seen in many executions performed all over the development and testing process. So, the normal array was finally used without concurrence tools.

5.2. Dockerization and adaptations

With all the project working, to allow everyone try the proof of concept it has been containerized in Docker, and this needs some adaptations to keep working. Even with the configuration of a network for the containers to allow the communication, due to ROS2 protocol needs, the connection to the sockets is to specific IP addresses so and ports, so the containers are assigned to static IPs and the ports open in them to allow the connections.

With this minimal changes the full system was executed in the different ways explained: together with the docker composer or separately docker by docker, with the simulator or with the tester also in a container. Several times with different plans, events, and latest state vector requests, as well as normal complete execution and different alarms and events.

5.3. Global verification

Once all the communications and modules were tested a global execution and validation process was done, trying to be as real as possible. With the examples in the Postman API mentioned in 4.3.2 section, a plan was published using the simulator UAV, and then the test, and the execution was followed and checked by reviewing all the logs thrown by the modules.

During this test I realized that when a command is finished and the result information is published and goes its way to the cloud, the JSON objects of the different results: Image, Observations, Battery and Pose get together and the Fleet Manager couldn’t differentiate them and put each in the corresponding body of the request to its own DB or to the cloud, so the code that extracts the JSON information from the broker was adapted to differentiate them when they come in a row and make the consequent requests.

The final complete simulation consisted of following the steps in README file included in the project, or the ones in:

1. A normal run of the entities: EDGE, ROS2 Broker and UAV / test module, in Figure 16: up left cornet the EDGE module, up right the ROS2 Broker module and down the UAV module, simulation execution in Figure 16.

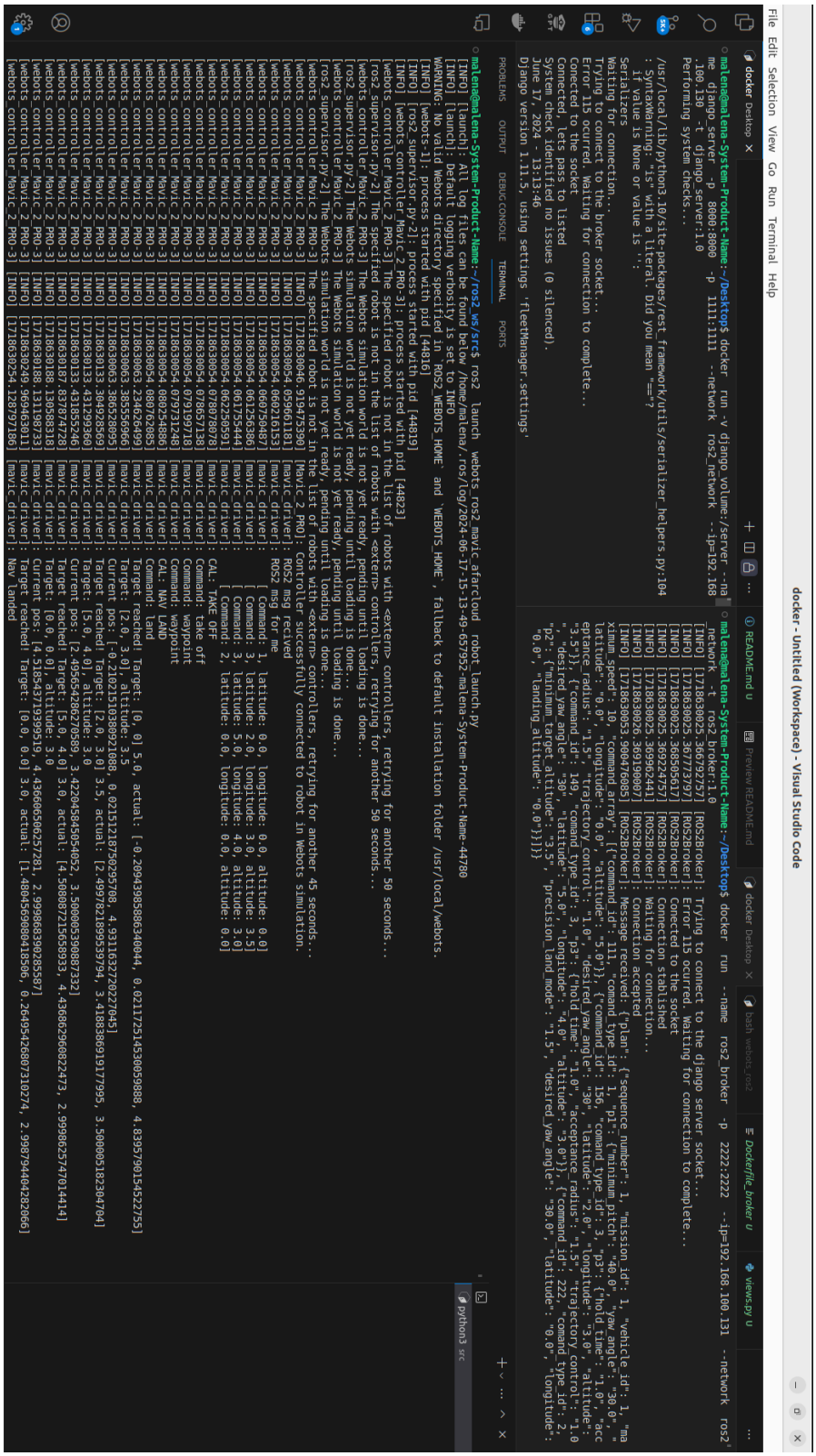


Figure 16. Normal initial run logs

2. Use Postman's example to publish a plan and see the correct reception, in Figure 17 the example plan used, and in the down part of the screenshot the request response received.

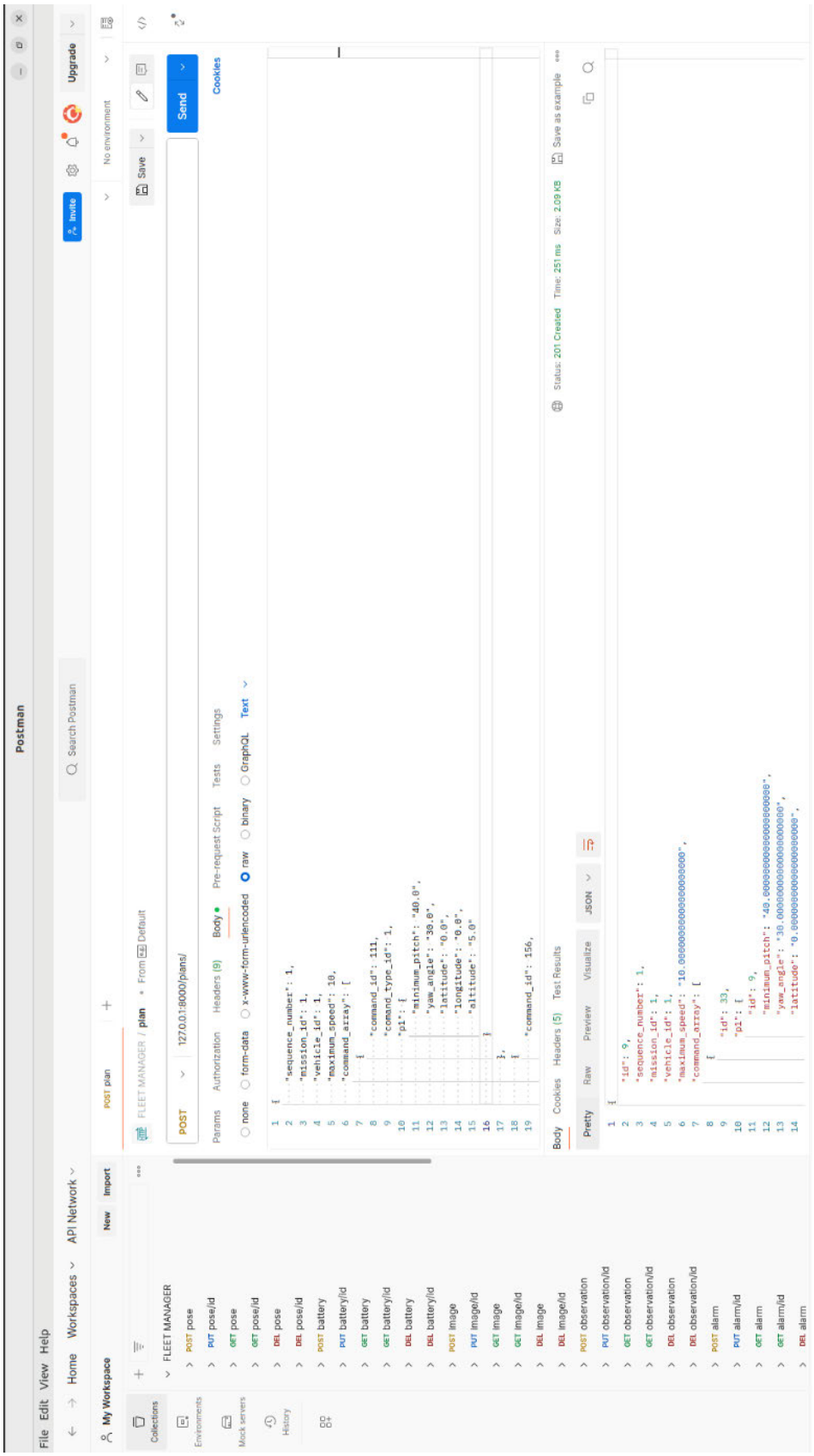


Figure 17. Postman execution example

3. Review Webots execution, some pictures taken during it in Figure 18, Figure 19, Figure 20 and Figure 21.

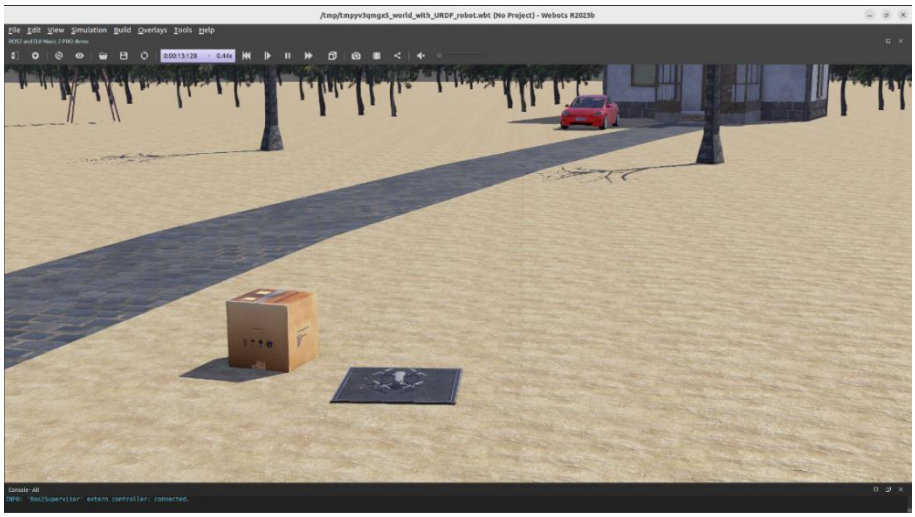


Figure 18. Webots drone before plan reception

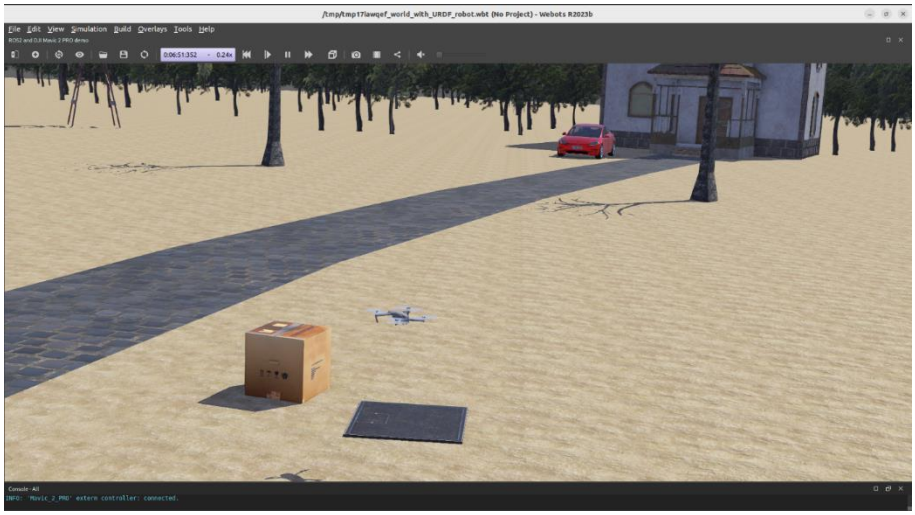


Figure 19. Webots drone beginning execution

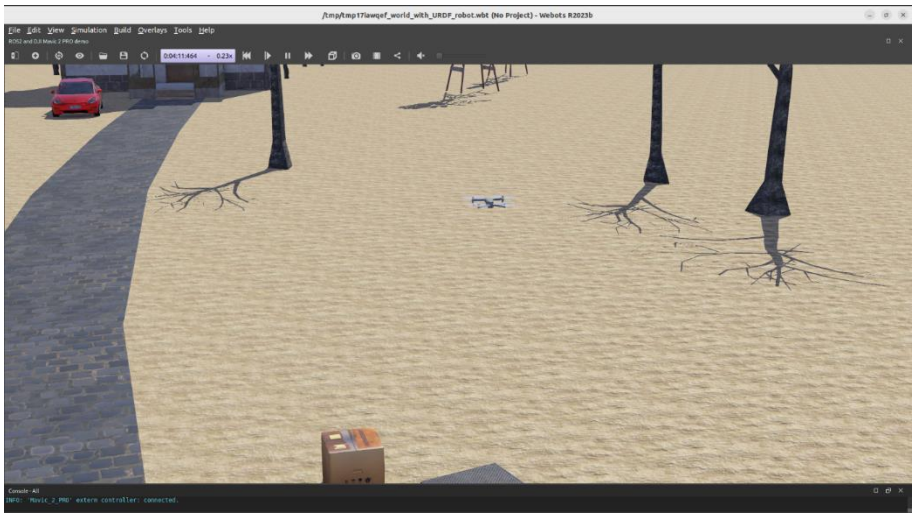


Figure 20. Webots done during execution

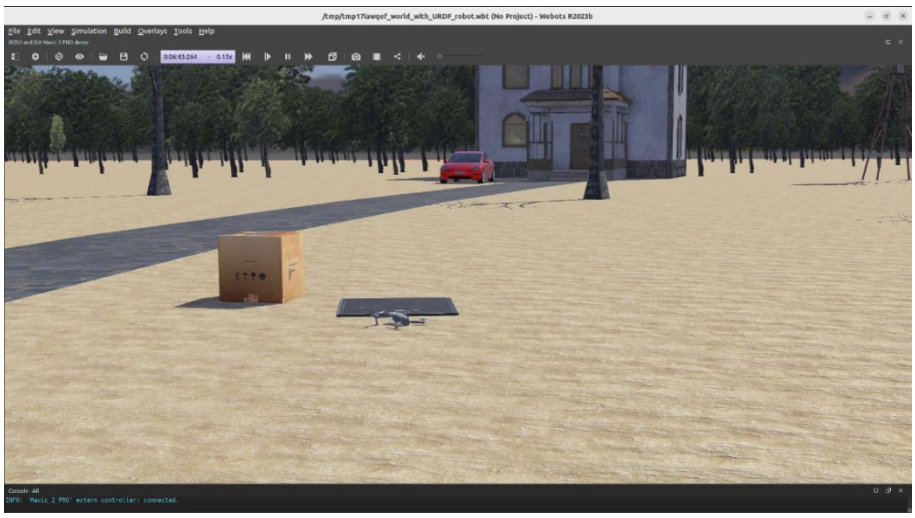


Figure 21. Webots drone landed after execution

4. Review terminals logs to check:
    a. New mission reception and process, Figure 22 up left corner the EDGE module, up right the ROS2 Broker module and down the UAV module.
    b. Periodically sent information: mission report and state vector, Figure 23 left side EDGE module, right side ROS2 Broker.
    c. Command finalization information: Pose, Battery and Observations, Figure 24 left side EDGE module, right side ROS2 Broker.
    d. Whether an alarm occurs or not, Figure 25 up left cornet the EDGE module, up right the ROS2 Broker module and down the test module, figure from test execution.

In Figure 22, Figure 23, Figure 24 and Figure 25 can be seen typical log outputs from:

- EDGE module: prints JSON received through the socket and CRUD request to the server.
- ROS2 Broker module: displays information received from the UAV (Mission Report, State Vector, Alarm, Pose, Image and Observation) and from the EDGE (Mission / Plan, Event, Latest State Vector) including timestamps used for concurrence tools check done and explained in section 5.1.4.
- UAV module: writes reduced messages produced by the different modules contained in the UAV.
- Docker test module: presents to the user the graphical interface detailed in point 4.4.7 as well as information published with timestamps used to validate the use of locks and queues as mentioned before.

Furthermore, the last verification videos, with simulation and testing modules, are included with the code. Concluding in a correct execution and behaviour, the validation of the model and coherent results as expected.

Figure 22. New mission logs

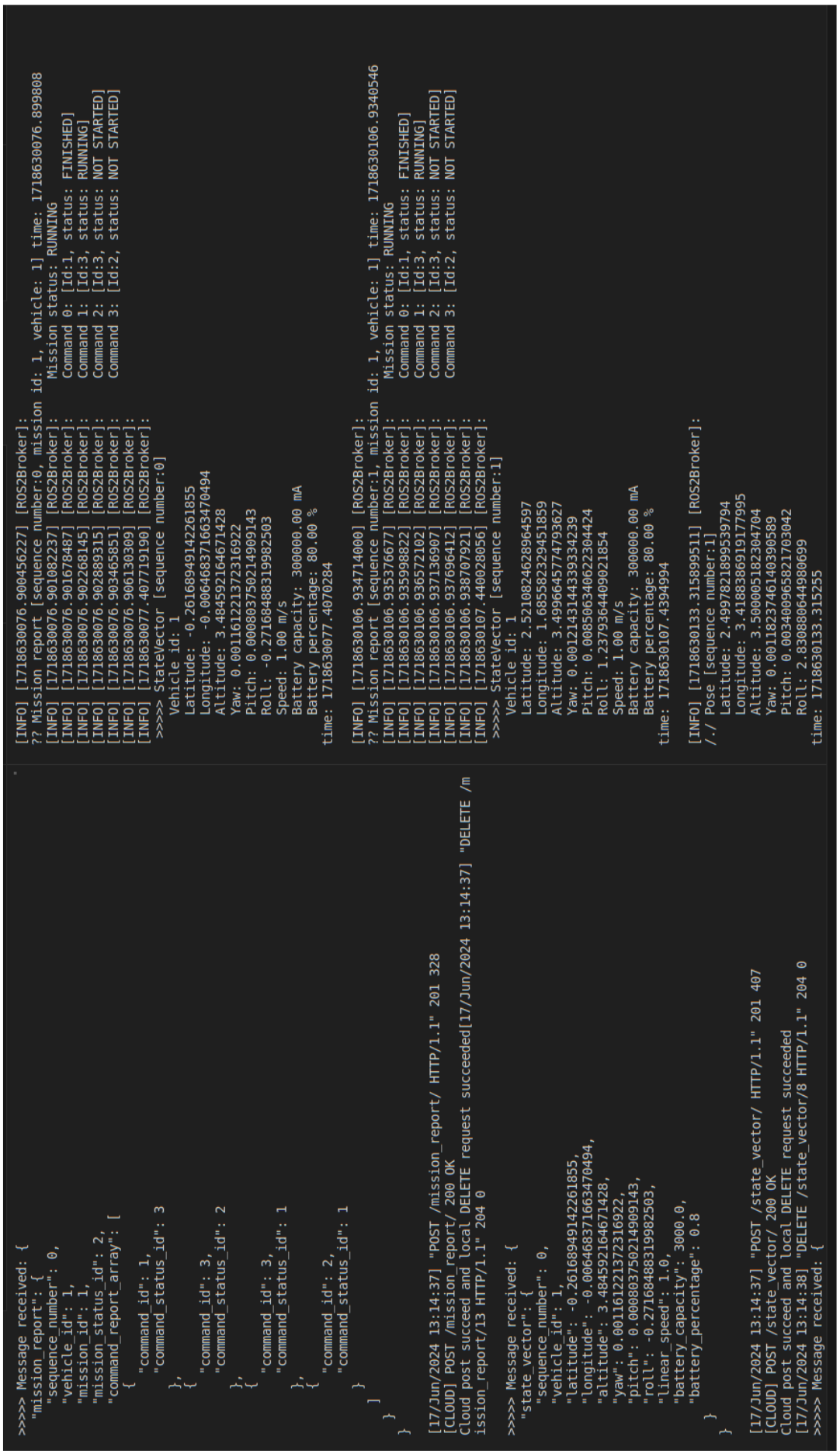


Figure 23. Mission report and state vector logs

Figure 24. Command result logs: Pose, Battery, Observation

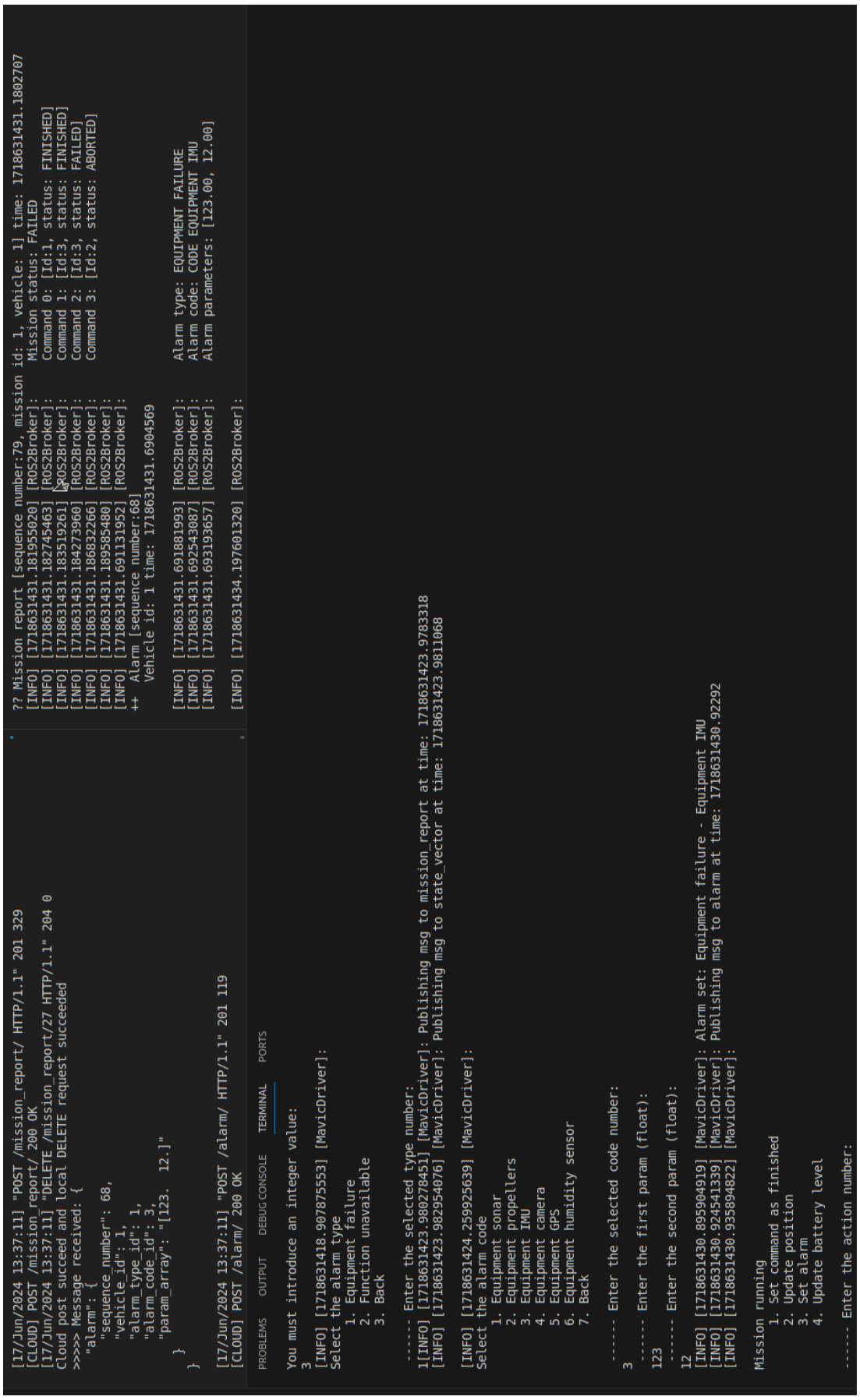


Figure 25. Test alarm set, send and reception

6. Budget

6.1. Partial budget

Thanks to the use of open-source technologies as ROS2 and Webots and free distribution technologies and tools: Docker, Postman, Python and Visual Studio Code the development, implementation, containerization, and validation steps cost is free. The main cost to expend is in labor and indirect costs.

6.1.1. Design step

Table 4. Design step budget

Units (number)	Concept (description)	Partial (unit cost)	Total (cost)
1	Visual Paradigm License	Free	Free
1	Postman license	Free	Free
TOTAL			**Free**

The design process involved the use of Visual Paradigm for data model design and Postman for APIs design and documentation.

6.1.2. Implementation step

Table 5. Implementation step budget

Units (number)	Concept (description)	Partial (unit cost)	Total (cost)
1	Visual Studio Code License	Free	Free
1	ROS2 License	Free	Free
1	Webots License	Free	Free
TOTAL			**Free**

6.1.3. Containerization and validation step

Table 6. Containerization and validation step budget

Units (number)	Concept (description)	Partial (unit cost)	Total (cost)
1	Docker License	Free	Free
1	Visual Studio Code License	Free	Free
1	ROS2 License	Free	Free
1	Webots License	Free	Free
1	Postman License	Free	Free
TOTAL			**Free**

6.1.4. Common utilities

Table 7. Common utilities budget

Units (number)	Concept (description)	Partial (unit cost)	Total (cost)
4	Computer	27.78 €/month	80.00 €
4	Telematics engineer labor	2,125.00 € / month	8500.00 €
1	Indirect costs: light, electricity, Wi-Fi…	15% of total	1287.00 €
TOTAL			**9867.00 €**

This section includes common costs through all the steps of the project.

To calculate the depreciation cost of a computer with a total cost of 1200€ for use over 4 months, the following steps are taken:

1. Determine the expected total useful life of the computer: It is assumed that the computer has a useful life of 5 years (60 months), which is a reasonable estimate for this type of equipment.

2. Calculate the monthly depreciation cost: The total cost of the computer is divided by its useful life in months.

   Monthly Depreciation Cost = Total Cost / Useful Life in Months

   Monthly Depreciation Cost = €1200 / 60 months ≈ 20.00€

3. Calculate the depreciation cost for 4 months: The monthly depreciation cost is multiplied by the number of months of use.

   Depreciation Cost for 4 Months = 20.00€ × 4 = 80.00€

Thus, the depreciation cost of a computer with a total cost of 1200€ for use over 4 months is approximately 80.00€.

Considering that we employ a Telematics Engineer (graduated) with a salary of 30,000.00€ a year, considering taxes, the cost for 4 months should be around 8,500.00€.

Labor = Year salary / 14 payments * months = 30,000.00 / 14 * 4 ≈ 8,500.00€

Following typical calculation procedures for research projects in the European Union, indirect cost of light, electricity, Wi-Fi… is calculated as the 15% of the total cost of personal and equipment.

Indirect cost = Total * 15% = 8580.00 * 15 /100 = 1287.00 €

6.2. General budget

Table 8. Budget summary

Total budget of design	Free
Total budget of implementation	Free
Total budget of validation	Free
Total common budget	**9867.00 €**
TOTAL	**9867.00 €**

7. Project impact

Following instructions and recommendations from [35] the following impacts of this project have been detected:

7.1. Ethic impact

7.1.1. Harms and risks: Health and bodily harm

1. **Physical injuries from drone malfunctions or crashes**: Drones can malfunction or crash due to technical failures, pilot error, or adverse weather conditions. These incidents can cause serious injuries to farm workers or bystanders if they are struck by falling drones or drone components.

2. **Injury from propeller blades**: The propellers of drones spin at high speeds and can cause severe cuts or lacerations if someone comes into contact with them. This risk is heightened during takeoff, landing, and maintenance activities.

3. **Exposure to chemicals**: Drones are often used to spray pesticides, herbicides, and fertilizers. Incorrect handling or drone malfunction during these operations can lead to unintended exposure to these chemicals, posing risks of poisoning, skin irritation, respiratory issues, and other health problems.

4. **Hearing damage**: Drones can generate significant noise, especially larger agricultural models. Prolonged exposure to high noise levels can lead to hearing damage or loss for nearby workers.

5. **Eye injuries**: workers can suffer eye injuries from direct contact with drones, debris thrown up by propellers, or chemicals sprayed by drones. Wearing proper eye protection is essential to mitigate this risk.

6. **Electromagnetic interference**: Drones emit electromagnetic fields, which could potentially interfere with medical devices such as pacemakers. This risk is generally low but could be significant for individuals with such medical devices working near operational drones.

7. **Burns and fire hazards**: Drones powered by lithium-ion batteries pose a fire risk if the batteries are damaged or improperly handled. Overheating, short-circuiting, or physical damage to the batteries can cause burns or fires.

8. **Unauthorized management**: If an unauthorized person connects to the same network and gains access to the ROS2 topics where drone information is published, they could publish to them, potentially sending malicious commands to the drones. Similarly, if the server's API URLs for getting or publishing information are compromised, an attacker could manipulate the drones' operations. This could result in physical harm,

such as drones crashing, spraying chemicals incorrectly, or operating in unsafe manners, leading to injuries, crop damage, or environmental harm.

Mitigation Measures

To mitigate these risks, several safety measures can be implemented:

- **Protective gear**: Use of protective equipment such as gloves, eye protection, and hearing protection.
- **Maintenance and checks**: Regular maintenance and checks to ensure drones are in safe working condition.
- **Chemical handling protocols**: Implement strict protocols for handling and applying chemicals, including protective clothing and equipment.
- **Safe operating procedures**: Develop and enforce safe operating procedures, including clear zones around drone operations and adherence to safety guidelines.
- **Emergency preparedness**: Have emergency procedures in place for dealing with drone crashes, chemical spills, and other incidents.
- **Authentication and authorization**: in the case of receive a publication of plans, first authenticate, and check the authorization for the manager who publishes it.

7.1.2. Rights: information privacy

Data Security:

- **Unauthorized access**: Drones collect and transmit a significant amount of data, including images, videos, and sensor readings. If this data is not properly treated, it could be intercepted by unauthorized parties, leading to potential data breaches.
- **Data integrity**: Ensuring the integrity of the data transmitted between drones and control systems is crucial. If the data is tampered with during transmission, it could result in incorrect actions being taken based on falsified information.

Privacy of farmers and landowners:

- **Surveillance concerns**: Drones equipped with cameras and sensors can inadvertently capture images or data from neighboring properties, raising concerns about unauthorized surveillance and privacy infringement.
- **Personal data collection**: Inadvertent collection of personal data, such as images of individuals or personal activities on the farm, can lead to privacy violations if not properly managed and anonymized.

This could occur if anyone else connects to the same network and gets the topics of ROS2 to which the information is published, and they subscribe to them. Or in the case of the server taking the URLs of the API to get or publish information.

Data Ownership:

- **Clarifying ownership rights**: There could be disputes over who owns the data collected by drones. Is it the farmer or the service provider? Clear agreements must be established to define data ownership and usage rights.
- **Data sharing and usage**: How the data is used, shared, and who has access to it can be contentious. Farmers need assurance that their data will not be misused or sold to third parties without their explicit consent.

Compliance with regulations:

- **Adhering to privacy Laws**: Different regions have varying privacy laws and regulations, such as General Data Protection Regulation (GDPR) [36] in Europe. Ensuring compliance with these laws when collecting, transmitting, and storing data is essential to avoid legal repercussions.
- **Liability issues**: In case of data breaches or misuse, determining liability can be complex. Who is responsible if sensitive data is leaked—the farmer or the data processing service.

Secure communication:

- **Authentication and Authorization**: Implementing robust authentication and authorization mechanisms to ensure that only authorized personnel can access and control the drones and the data they collect.

Data storage and retention:

- **Secure storage solutions**: Ensuring that data is stored in secure, protected environments to prevent unauthorized access and data loss.
- **Data retention policies**: Establishing clear policies on how long data is retained and ensuring it is securely deleted when no longer needed to minimize the risk of outdated or irrelevant data being compromised.

Mitigation Measures

To address these issues, several measures can be implemented:

- **Data encryption**: Use strong encryption for data both in transit and at rest to protect against unauthorized access and interception.
- **Clear data ownership agreements**: Establish clear contracts and agreements defining data ownership, rights, and responsibilities between all parties involved.

- **Compliance with regulations**: Ensure compliance with relevant data protection laws and regulations, and regularly review practices to stay updated with legal requirements.
- **Access control**: Implement strict access control measures, including authentication and authorization, to ensure that only authorized users can access the data.
- **Privacy policies**: Develop and enforce privacy policies that outline how data is collected, used, shared, and protected.
- **Secure storage solutions**: Use secure, reputable storage solutions and services to store the data collected by drones.
- **Regular audits and monitoring**: Conduct regular audits and monitoring of data handling practices to ensure compliance and security standards are maintained.
- **Training and awareness**: Educate farmers and all stakeholders about the importance of data privacy, security and provide training on best practices.

7.2. Social impact

7.2.1. Attending Needs and Basic Services Access for Social Health: Food

Increase food production:

- **Enhanced crop monitoring**: Drones can monitor crop health with high precision, using various sensors to detect pest infestations, nutrient deficiencies, and water stress early. This allows farmers to take timely actions to address these issues, leading to healthier crops and higher yields.
- **Optimized resource use**: By providing detailed data on crop and soil conditions, drones help farmers apply water, fertilizers, and pesticides more efficiently, reducing waste and maximizing crop output.

Improved food security:

- **Higher yields and reduced losses**: With better crop management facilitated by drones, farmers can achieve higher yields and reduce crop losses due to pests, diseases, and environmental stressors. This leads to a more stable and abundant food supply.
- **Resilience to climate change**: Drones provide real-time data and predictive analytics, helping farmers adapt to changing climate conditions and mitigate the impacts of extreme weather events, thereby safeguarding food production.

Reduced chemical exposure:

- **Precision application**: Drones enable the targeted application of pesticides and fertilizers, minimizing the amount used and reducing the risk of chemical exposure for

farm workers and nearby communities. This leads to better health outcomes and a safer environment.

Related SDGs:

- **SDG 2: Zero hunger**: By increasing agricultural productivity and ensuring a stable food supply, drones help to eliminate hunger, achieve food security, and promote sustainable agriculture.
- **SDG 3: Good health and well-being**: Reducing chemical exposure through precision agriculture improves health and well-being for farmers and rural communities.

7.2.2. Respect to Work Rights and Internal Social Responsibility: Enhance Work Conditions

Employment and skill development:

- **New job opportunities**: The adoption of drone technology in agriculture creates new job roles, such as drone operators, maintenance technicians, and data analysts. This diversification of job opportunities can stimulate local economies and provide new career paths, especially for young people.

Improved work conditions:

- **Safety enhancements**: Drones can take over hazardous tasks such as spraying chemicals, reducing the risk of injuries and exposure for farm workers. This leads to safer working conditions and a reduction in occupational hazards.
- **Reduction in manual labor**: Automation of tasks through drones reduces the physical strain and labor-intensive work required in traditional farming, improving overall working conditions.

Related SDGs [37]:

- **SDG 8: Decent work and economic growth**: Promotes sustained, inclusive, and sustainable economic growth, full and productive employment, and decent work for all. The introduction of new technologies and job opportunities through the use of drones in agriculture aligns with this goal.

7.2.3. Technology Sustainability and Socioeconomic Aspects: Enhance Productivity, Affordability, and Social Integration and Adaptation of Innovation

Enhanced productivity:

- **Optimized farming practices**: Drones provide detailed data that helps farmers make informed decisions on planting, irrigation, and harvesting. This leads to more efficient farming practices, higher productivity, and better use of resources.

- **Precision agriculture**: By enabling precise monitoring and management of crops, drones help to maximize yields and minimize resource waste, contributing to more sustainable agricultural practices.

Affordability:

- **Cost savings**: Drones help reduce the overall cost of farming operations by optimizing the use of inputs like water, fertilizers, and pesticides, and by reducing crop losses. This makes advanced agricultural technology more affordable and accessible, especially for smallholder farmers.
- **Scalability**: The use of drones can be scaled to different farm sizes, making the technology adaptable and cost-effective for both large agribusinesses and small farms.

Social Integration and adaptation of innovation:

- **Access to advanced technology**: Making drone technology available to small-scale farmers and those in developing regions promotes social integration and reduces the technology gap between smallholders and large agribusinesses.
- **Community development**: The widespread adoption of drones fosters innovation and economic development within agricultural communities. It helps integrate modern technology into traditional farming practices, promoting overall community resilience and development.

Related SDGs [37]:

- **SDG 9: Industry, innovation, and infrastructure**: Builds resilient infrastructure, promotes inclusive and sustainable industrialization, and fosters innovation. Drones represent a significant technological advancement that supports these objectives.
- **SDG 10: Reduced inequalities**: Reduces inequality within and among countries by providing access to advanced agricultural technologies for smallholder farmers and marginalized communities.
- **SDG 12: Responsible consumption and production**: Ensures sustainable consumption and production patterns by optimizing resource use and reducing waste through precision agriculture practices.

7.3. Environmental impact

7.3.1. Materials

The project does not involve the use of non-renewable or non-recyclable materials. The choice of materials for the autonomous aerial vehicles (drones) will depend on the specific models selected, but the aim is to prioritize renewable and recyclable options whenever possible.

7.3.2. Energy

The project aims to develop a more efficient communication and control system for autonomous aerial vehicles, which can lead to reduced energy consumption. The energy source for the project will vary depending on the specific installation and farmer, but the consumption will be very low, only required for charging the drones and powering the small processing device at the edge.

7.3.3. Water

The use of more efficient tools to monitor the state of the fields can contribute to better water management and conservation. The system is designed to minimize water usage and promote sustainability.

7.3.4. Emissions

The project aims to minimize direct and indirect emissions. Since the drones are electric, there will be no emissions during operation.

7.3.5. Efluentes and residues

The project aims to minimize waste and promote recycling. The drones can potentially be reused for recreational purposes or other applications after their primary use in the project, further reducing waste.

The project has minimized risks and emphasized ethical considerations throughout its design and implementation. The project's focus on sustainability, efficiency, and minimizing waste has reduced the environmental impact, ensuring compliance with regulations and laws related to data protection and privacy. The use of renewable and recyclable materials, efficient energy consumption, and minimized water usage further contribute to a reduced ecological footprint. The project’s emphasis on safety measures need, such as protective gear, maintenance checks, and chemical handling protocols, mitigates physical risks associated with drone malfunctions, crashes, and chemical exposure. Additionally, measures to prevent unauthorized management and ensure data security have been implemented to prevent potential misuse. While physical risks depend on the actual deployment in the field, the project's comprehensive approach to safety and security ensures that these risks are minimized. The project's alignment with ethical professional standards, laws, and regulations related to data protection, privacy, and environmental sustainability demonstrates its commitment to responsible innovation and responsible use of technology in precision agriculture.

8. Conclusions

8.1. Conclusions

After the completion of this final degree project, significant progress has been made in addressing the challenge of developing an efficient and scalable communication and management system for unmanned autonomous vehicles (UAVs) in precision agriculture. Traditional methods often face limitations in real-time data processing, scalability, and integration with existing agricultural practices. To overcome these limitations, a cloud-based system within an EDGE computing framework was developed using the Robot Operating System 2 (ROS2).

The system leverages the modular architecture and real-time capabilities of ROS2, integrated with an edge computing architecture to enhance real-time data processing and reduce latency by processing data closer to the source. The cloud infrastructure provides centralized management of UAV operations, facilitating robust communication and control mechanisms.

Key findings from the project include:

- Effective Communication Protocols: The successful implementation of communication protocols between the cloud, edge, and UAVs ensures timely data processing and decision-making.
- Scalability and Flexibility: The deployment of Docker containers enhances the system's scalability and flexibility, allowing adaptation to various UAV models and agricultural applications.
- Enhanced Precision Agriculture: The system's capability to process and analyze data in real-time significantly improves decision-making and operational efficiency in precision agriculture.

Several new insights were identified:

- System Validity: The proposed system was validated as effective for managing UAV operations in precision agriculture. The integration of ROS2 and edge computing enabled real-time data processing and seamless UAV operation integration within a cloud infrastructure.
- New Problem Aspects: The project revealed new challenges, such as the necessity for enhanced security measures to protect data integrity and privacy in cloud-based systems and the need for sophisticated algorithms for data analysis and decision-making.
- Future Work Directions: Future work should focus on expanding the system's capabilities to include additional UAV models, new communication protocols, and ground vehicles. Further optimization of communication protocols to reduce latency and enhance data throughput is essential.

In conclusion, this project significantly advances the field of precision agriculture by delivering a robust and scalable UAV communication and management system. Leveraging technologies such as ROS2 and edge computing is crucial for enhancing agricultural practices. Future research will build on these findings, addressing identified challenges and expanding system capabilities to support sustainable and efficient farming practices.

8.2. Future works

The main objectives of the projects have been full solved, but some advanced features can be implemented as future works.

Implement actions in the UAV to perform the rest of the possible commands in the data model: camera and video related commands, hot point fly, follow a target and track a subject; for the moment the plans and missions can only be make up of nav take off, nav waypoint, nav home and nav land commands.

Also, the implementation of picture taking, or sensors' parameters recollecting to be included in the command result set. Now the implementation of this information sending for both is done, but in the case of observation parameters a simple example is included, which can be changed with the tester, but for image not an example or real image is even sent.

New communication protocols can be included to manage the unmanned vehicle, ISOBUS for instance, as commented in section 4, with an easy adaptation of the broker and the vehicle modules.

Furthermore, the use of different UAV can be included and new commands for new features needed.

9. References

[1] Group of Next-Generation Networks and Services, “Group of Next-Generation Networks and Services.” Accessed: Jun. 15, 2024. [Online]. Available: https://grys.etsist.upm.es/about

[2] AFarCloud, “afarcloud.” Accessed: Jun. 15, 2024. [Online]. Available: http://www.afarcloud.eu/

[3] E. Commission, “Aggregate Farming in the Cloud,” Nov. 2021, doi: 10.3030/783221.

[4] Open Robotics, “ROS 2 Documentation: Humble.” Accessed: Jun. 15, 2024. [Online]. Available: https://docs.ros.org/en/humble/index.html

[5] Cyberbotics, “Simulating your robots with Webots.” Accessed: Jun. 15, 2024. [Online]. Available: https://cyberbotics.com/

[6] Cyberbotics, “Webots ROS 2 packages.” Accessed: Jun. 15, 2024. [Online]. Available: https://github.com/cyberbotics/webots_ros2

[7] K. Svyatov, R. Zhitkov, V. Mikhailov, and I. Khayrullin, “Methods and Software Tools for the Safety Movement of Agricultural Highly Automated Vehicle Based on Deep Learning Methods,” 2023, Accessed: Jun. 12, 2024. [Online]. Available: https://www.preprints.org/manuscript/202311.1922/v1

[8] N. Tsolakis, D. Bechtsis, and D. Bochtis, “AgROS: A Robot Operating System Based Emulation Tool for Agricultural Robotics,” *Agronomy*, vol. 9, no. 7, p. 403, Jul. 2019, doi: 10.3390/agronomy9070403.

[9] Farm Wise Labs, “Farm Wise,” *2024*. Accessed: Jun. 12, 2024. [Online]. Available: https://farmwiselabs.com/

[10] Víctor Mayoral Vilches, “ROS Botics companies.” Accessed: Jun. 12, 2024. [Online]. Available: https://github.com/vmayoral/ros-robotics-companies

[11] Field Robotics, “Field Robotics.”

[12] Víctor Mayoral Vilches, “ROS Botics companies.” Accessed: Jun. 12, 2024. [Online]. Available: https://github.com/vmayoral/ros-robotics-companies

[13] Deere & Company, “John Deere,” 2024. Accessed: Jun. 12, 2024. [Online]. Available: https://www.deere.com/en/index.html

[14] J. Lloret, “Edge Computing in Precision agriculture,” in *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, Dec. 2022, pp. 1–1. doi: 10.1109/FMEC57183.2022.10062622.

[15] Docker Inc., “Docker.” Accessed: Jun. 15, 2024. [Online]. Available: https://www.docker.com/

[16] Docker Inc., “Docker docs: Manuals,” 2024, Accessed: Jun. 15, 2024. [Online]. Available: https://docs.docker.com/manuals/

[17] Z. Liu and J. Li, “Application of Unmanned Aerial Vehicles in Precision Agriculture,” *Agriculture*, vol. 13, no. 7, p. 1375, Jul. 2023, doi: 10.3390/agriculture13071375.

[18] H. Lee, J. Yoon, M.-S. Jang, and K.-J. Park, “A Robot Operating System Framework for Secure UAV Communications,” *Sensors*, vol. 21, no. 4, p. 1369, Feb. 2021, doi: 10.3390/s21041369.

[19] Y. Ampatzidis, V. Partel, and L. Costa, “Agroview: Cloud-based application to process, analyze and visualize UAV-collected data for precision agriculture applications utilizing artificial intelligence,” *Comput Electron Agric*, vol. 174, p. 105457, Jul. 2020, doi: 10.1016/j.compag.2020.105457.

[20] P. Radoglou-Grammatikis, P. Sarigiannidis, T. Lagkas, and I. Moscholios, “A compilation of UAV applications for precision agriculture,” *Computer Networks*, vol. 172, p. 107148, May 2020, doi: 10.1016/j.comnet.2020.107148.

[21] J. Liu, J. Xiang, Y. Jin, R. Liu, J. Yan, and L. Wang, “Boost Precision Agriculture with Unmanned Aerial Vehicle Remote Sensing and Edge Intelligence: A Survey,” *Remote Sens (Basel)*, vol. 13, no. 21, p. 4387, Oct. 2021, doi: 10.3390/rs13214387.

[22] Oper Robotics, “Creating custom msg and srv files.” Accessed: Jun. 12, 2024. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html

[23] AFarCloud Consortium, “D2.6 Semantic Middleware (V1),” 2020.

[24] Open Robotics, “Setting up a robot simulation (Webots)”, Accessed: Jun. 12, 2024. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Advanced/Simulators/Webots.html

[25] Cyberbotics, “Webots ROS2 Mavic example.” Accessed: Jun. 12, 2024. [Online]. Available: https://github.com/cyberbotics/webots_ros2/tree/master/webots_ros2_mavic

[26] G. C. Hillar, “Django RESTful Web Services,” O’Reilly, 2018.

[27] StackOverflow, “import error ‘force_text’ from ’django.utils.encoding.” Accessed: Jun. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/70382084/import-error-force-text-from-django-utils-encoding

[28] StackOverflow, “I’m facing this error "ImportError: cannot import name ‘smart_text’ from ‘django.utils.encoding.’” Accessed: Jun. 12, 2024. [Online]. Available:

https://stackoverflow.com/questions/75939945/im-facing-this-error-importerror-cannot-import-name-smart-text-from-django

[29] StackOverflow, “MakeMigration Error on Django - ImportError: cannot import name ‘FieldDoesNotExist’ from ‘django.db.models.’” Accessed: Jun. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/63300404/makemigration-error-on-django-importerror-cannot-import-name-fielddoesnotexi

[30] Django, “ImportError: cannot import name ‘ugettext_lazy’ from ‘django.utils.translation.’” Accessed: Jun. 12, 2024. [Online]. Available: https://forum.djangoproject.com/t/importerror-cannot-import-name-ugettext-lazy-from-django-utils-translation/10943

[31] StackOverflow, “ImportError : cannot import name ‘ugettext_lazy.’” Accessed: Jun. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/70656495/importerror-cannot-import-name-ugettext-lazy

[32] Django, “Django 4.2 release notes.” Accessed: Jun. 12, 2024. [Online]. Available: https://docs.djangoproject.com/en/4.2/releases/4.2/

[33] Open Robotics, “Writing a simple publisher and subscriber (Python).” Accessed: Jun. 12, 2024. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html

[34] ROS Answers, “Including a python module in a ros2 package.” Accessed: Jun. 12, 2024. [Online]. Available: https://answers.ros.org/question/367793/including-a-python-module-in-a-ros2-package/

[35] C. Fernández Aller and R. Miñano, “Guía para trabajar la Responsabilidad Social y Ambiental (GRSA).” Accessed: Jun. 08, 2024. [Online]. Available: https://oa.upm.es/35542/1/Guia_Responsabilidad_Social_y_Ambiental-V2-1.pdf

[36] T. e. p. a. t. c. o. t. e. union, “General Data Protection Regulation GDPR.” Accessed: Jun. 12, 2024. [Online]. Available: https://gdpr-info.eu/

[37] United Nations, “Sustainable Development Goals.” Accessed: Jun. 08, 2024. [Online]. Available: https://www.un.org/sustainabledevelopment/

[38] Cyberbotics, “Webots installation tutorial.” Accessed: Jun. 12, 2024. [Online]. Available: https://cyberbotics.com/doc/guide/installation-procedure#installation-on-linux

[39] Open Robotics, “ROS2 installation tutorial.” Accessed: Jun. 12, 2024. [Online]. Available: https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

[40] ROS Answer, “pip install setuptools==58.2.0.” Accessed: Jun. 12, 2024. [Online]. Available: https://answers.ros.org/question/396439/setuptoolsdeprecationwarning-setuppy-install-is-deprecated-use-build-and-pip-and-other-standards-based-tools/

Annexes

A.1 Main ROS2 messages

Table 9. ROS2 main messages

PUBLISHER	SUBSCRIBER	TOPIC	MSG TYPE	DESCRIPTION
ROS2 Broker	UVS Controller	mission	afarcloud_msgs /msg/Mission	int32 sequence_number int32 mission_id int32 vehicle_id float32 maximum_speed afarcloud_aux_2_msgs/Command[100] command_array
ROS2 Broker	UVS Controller	event	afarcloud_msgs /msg/Event	int32 ABORT_MISSION = 1 int32 SOFT_ABORT_MISSION = 2 int32 DRONE_SPEED = 3 int32 sequence_number #Identifier of the vehicle that should process the event int32 vehicle_id #Identifier of the type of event int32 event_type_id # Parameters that depend on the type of command afarcloud_aux_msgs/AbortMisionParam p1 float64 p2 #maximum linear speed [0, 5] m/s
UVS Controller	ROS2 Broker	alarm	afarcloud_msgs /msg/Alarm	#Type of alarm int32 TYPE_EQUIPMENT_FAILURE = 1 int32 TYPE_FUNCTION_UNAVAILABLE = 2 #Code of the alarm int32 CODE_EQUIPMENT_SONAR = 1 int32 CODE_EQUIPMENT_PROPELLERS = 2 int32 CODE_EQUIPMENT_IMU = 3 int32 CODE_EQUIPMENT_CAMERA = 4 int32 CODE_EQUIPMENT_GPS = 5 int32 CODE_EQUIPMENT_HUMIDITY_SENSOR = 6 #… int32 CODE_FUNCTION_LOCALISATION = 50

				int32 CODE_FUNCTION_PROPULSION = 51 int32 CODE_FUNCTION_NAVIGATION = 52 int32 CODE_FUNCTION_HUMIDTY_SENSING = 53 int32 sequence_number #Identifier of the vehicle that generates the alarm int32 vehicle_id #Identifier of the type of alarm int32 alarm_type_id #Identifier of the type of alarm int32 alarm_code_id #Parameters that depend on the type of alarm float64[2] param_array
UVS Controller	ROS2 Broker	state_vector	afarcloud_msgs/msg/StateVector	int32 sequence_number #Identifier of the vehicle int32 vehicle_id #Latitude and longitude in degrees float64 latitude float64 longitude #Altitude in meters float64 altitude #Orientation float64 yaw float64 pitch float64 roll #Battery capacity in Ah (last full capacity) float64 battery_capacity #Battery charge percentage on 0 to 1 range float64 battery_percentage #Linear speed vector of the vehicle float32 linear_speed
UVS Controller	ROS2 Broker	mission_report	afarcloud_msgs/msg/MissionReport	#Mission status int32 MISSION_NOT_STARTED = 1 int32 MISSION_RUNNING = 2 int32 MISSION_FINISHED = 3 int32 MISSION_FAILED = 4 int32 MISSION_ABORTED = 5 int32 sequence_number #Identifier of the vehicle int32 vehicle_id

				#Identifier of the mission int32 mission_id #Mission status int32 mission_status_id #Mission commands status afarcloud_aux_msgs/CommandReport[100] command_report_array
UVS Controller	ROS2 Broker	pose	afarcloud_msgs/msg/Pose	int32 sequence_number #Identifier of the vehicle int32 vehicle_id #Latitude and longitude in degrees float64 latitude float64 longitude #Altitude in meters float64 altitude #Orientation float64 yaw float64 pitch float64 roll
UVS Controller	ROS2 Broker	battery	afarcloud_msg/msg/Battery	int32 sequence_number #Identifier of the vehicle int32 vehicle_id #Capacity in Ah (last full capacity) float64 capacity #Charge percentage on 0 to 1 range float32 percentage
UVS Controller	ROS2 Broker	image	afarcloud_msg/msg/Image	int32 sequence_number #Identifier of the vehicle sending the image int32 vehicle_id # The format of the image data: jpeg, png, etc string format # Number of bytes of the image data. int32 image_size # The name with which you want to store the image. #If empty the timestamp will be used string name #The image buffers. MAX_BUFFER_SIZE = 1048576 # 1024 x 1024 is the maximum frame size. # Data will be allocated at this size, but only the # actual size of the frame will be sent. byte[1048576] frame

UVS Controller	ROS2 Broker	observation	afarcloud_msg/msg/ObservationList	int32 sequence_number #Identifier of the vehicle sending the image int32 vehicle_id #Latitude and longitude in degrees float64 latitude float64 longitude #Altitude in meters float64 altitude #List of observations afarcloud_aux_msgs/Observation[10] observation_array
ROS2 Broker	UVS Controller	latest_state_vector	afarcloud_msg/msg/LatestStateVector	int32 sequence_number

A.2 Auxiliar messages

Table 10. ROS2 auxiliary messages embedded in the main message.

MSG TYPE	DESCRIPTION
afarcloud_aux_2_msg/msg/Command	# Command_type uint8 NAV_TAKEOFF = 1 uint8 NAV_LAND = 2 uint8 NAV_WAYPOINT = 3 uint8 CAMERA_IMAGE = 4 uint8 VIDEO_START_CAPTURE = 5 uint8 VIDEO_STOP_CAPTURE = 6 uint8 NAV_HOTPOINT = 7 uint8 NAV_FOLLOW_TARGET = 8 uint8 NAV_TRACK_SUBJECT = 9 uint8 CAMERA_PANORAMA = 10 uint8 NAV_HOME = 11 #Identifier for monitoring the progress of the command int32 command_id #Identifier of the type of command to execute int32 command_type_id #Parameters that depend on the type of command afarcloud_aux_msgs/NavTakeOffParam p1 afarcloud_aux_msgs/NavLandParam p2 afarcloud_aux_msgs/NavWaypointParam p3 afarcloud_aux_msgs/CameraImagParam p4 afarcloud_aux_msgs/VideoStartCaptureParam p5 afarcloud_aux_msgs/VideoStopCaptureParam p6 afarcloud_aux_msgs/NavHotpointParam p7 afarcloud_aux_msgs/NavFollowTargetParam p8

	afarcloud_aux_msgs/NavTrackSubjectParam p9 afarcloud_aux_msgs/CameraPanoParam p10 afarcloud_aux_msgs/NavHomeParam p11
afarcloud_aux_msg/msg/NavTakeOffParam	float64 minimum_pitch #[0,45] decimal degrees float64 yaw_angle #[0,180] decimal degrees float64 latitude #[-90, 90] decimal degrees float64 longitude #[-180,180] decimal degrees float64 altitude #[0,10] meters
afarcloud_aux_msg/msg/NavLandParam	float64 minimum_target_altitude #[0,10] meters float64 precision_land_mode #[0,10] meters float64 desired_yaw_angle #[0,180] decimal degrees float64 latitude #[-90, 90] decimal degrees float64 longitude #[-180,180] decimal degrees float64 landing_altitude #[0,5] meters
afarcloud_aux_msg/msg/NavWaypointParam	float64 hold_time #[0,3] seconds*10 float64 acceptance_radius #[0,3] meters float64 trajectory_control #[0,3] meters float64 desired_yaw_angle #[0,180] decimal degrees float64 latitude #[-90, 90] decimal degrees float64 longitude #[-180,180] decimal degrees float64 altitude #[0,5] meters
afarcloud_aux_msg/msg/CameraImagParam	float64 yaw_angle #[0,180] decimal degrees float64 pitch_angle #[-90, 90] decimal degrees uint8 photo_count 1 #1=shoot a single photo. Time interval and total time ignored uint8 time_interval 0 #seconds
afarcloud_aux_msg/msg/VideoStartCaptureParam	uint8 video_stream_id #0 for all streams float64 freq #[0,5] Hz
afarcloud_aux_msg/msg/VideoStopCaptureParam	float64 video_stream_id #0 for all streams
afarcloud_aux_msg/msg/NavHotpointParam	#Start point uint8 NORTH=0 uint8 SOUTH=1 uint8 WEST=2 uint8 EAST=3 uint8 NEAREST=4 #Orbit type uint8 ALONG_CIRCLE_LOOKING_FORWARDS=0 uint8 ALONG_CIRCLE_LOOKING_BACKWARDS=1 uint8 TOWARDS_HOT_POINT=2 uint8 AWAY_FROM_HOT_POINT=3 uint8 CONTROLLED_BY_REMOTE_CONTROLLER=4

	uint8 USING_INITIAL_HEADING=5 float64 latitude #[-90, 90] decimal degrees float64 longitude #[-180,180] decimal degrees float64 orbit_altitude #[MIN_RADIUS, MAX_RADIUS] meters float64 radious #meters float64 angural_velocity 20 #degrees/s bool is_clock_wise true #true/false uint8 start_point #[0, 4] uint8 orbit_type #[0, 5]
afarcloud_aux_msg/msg/NavFollowTargetParam	#Heading of the drone uint8 TOWARD_FOLLOW_POSITION=0 uint8 CONTROLLED_BY_REMOTE_CONTROLLER=1 float64 latitude #[-90, 90] decimal degrees float64 longitude #[-180,180] decimal degrees float64 altitude #[0,10] meters uint8 heading #[0,1]
afarcloud_aux_msg/msg/NavTrackSubjectParam	#Track mode uint8 TRACE=0 uint8 PROFILE=1 uint8 SPOTLIGHT=2 uint8 SPOTLIGHT_PRO=3 uint8 QUICK_SHOT=4 uint8[2] corners #[0, 1] - [1, 1] uint8 track_mode #[0, 4]
afarcloud_aux_msg/msg/CameraPanoParam	#Panorama mode uint8 FULL_CIRCLE=0 uint8 HALF_CIRCLE=1 uint8 panorama_mode #[0,1]
afarcloud_aux_msg/msg/NavHomeParam	bool landing true #home/puase
afarcloud_aux_msg/msg/AbortMisionParam	#Optional parameters float64 latitude #[-90, 90] decimal degrees float64 longitude #[-180,180] decimal degrees float64 altitude #[0,10] meters
afarcloud_aux_msg/msg/CommandReport	#Command status int32 COMMAND_NOT_STARTED=1 int32 COMMAND_RUNNING=2 int32 COMMAND_FINISHED=3 int32 COMMAND_FAILED=4 int32 COMMAND_ABORTED=5 #Identifier of the mission command

	int32 command_id #Report of the mission command int32 command_status_id
afarcloud_aux_msg/msg/Observation	string resource_id string observed_property int32 result_time #Epoch format 1558086914 float64 result_value string result_uom

User manual

A.1 Installations needed and versions used

Table 11. Programs

Webots [38]	Webots R2023b (Date June 28, 2023)
ROS2 [39]	Humble hawksbill distribution
Python	Python 3.10.12
Setup tools [40]	58.2.0

Table 12. PC characteristics

OS	Ubuntu 22.04.3 LTS (64-bit)
Graphics	Mesa Intel® HD Graphics 630 (KBL GT2)
Processor	Intel® Core™ i7-7700 CPU @ 3.60GHz × 8
Memory	16.0 GiB

Table 13. Virtual environment

Python	Python 3.10.12
Django	1.11.5
Django REST Framework	3.6.4
Psycopg2	2.9.9
Pytz	2023.3. post1

A.2 Compilation

After carrying out the installation tutorial for ROS2 [39] and Webots [38], a folder and file structure like this appears:

```
Home/
└── ros2_ws/
      ├── build/
      │   └── …
      ├── install /
      │   └── …
      ├── log/
      │   └── …
      └── src/
           └── …
```

Nex step is adding the code to the structure, pasting the folder afarcloud_docker to the src folder, getting something like this:

```
Home/
└── ros2_ws/
      ├── build/
      │   └── …
      ├── install /
```

```
│   └── …
├── log/
│   └── …
└── src/
    ├── build
    ├── install
    ├── log
    └── webots_ros2
        ├── …
        └── afarcloud_docker
            ├── afarcloud_aux_2_msgs
            ├── afarcloud_aux_msgs
            ├── afarcloud_msgs
            ├── EDGE
            │   ├── env
            │   │   └── django_env
            │   │       ├── bin
            │   │       │   ├── activate
            │   │       │   └── …
            │   │       ├── fleetManager
            │   │       │   ├── build
            │   │       │   ├── edge
            │   │       │   │   ├── …
            │   │       │   │   ├── migrations
            │   │       │   │   ├── models.py
            │   │       │   │   ├── serializers.py
            │   │       │   │   ├── urls.py
            │   │       │   │   └── views.py
            │   │       │   ├── fleetManager
            │   │       │   ├── install
            │   │       │   ├── log
            │   │       │   ├── db.sqlite3
            │   │       │   └── manage.py
            │   │       ├── include
            │   │       ├── lib
            │   │       ├── lib64
            │   │       └── pyenv.cfg
            │   └── ros2_broker
            │       └── …
            └── UnmanedVehicle
                ├── webots_ros2_mavic_afarcloud
                │   └── …
                └── tester
                    └── …
```

To compile and execute ROS2 the whole scenario, that allow programs to start, a command prompt must be opened in the 'ros2_ws' folder and the following commands must be executed:

Second and third terminal:

```
source /opt/ros/humble/setup.bash
. install/setup.bash
```

To build the new ROS2 programs and messages defined in the 'afarcloud' folder (only once):

```
colcon build
```

After building, two folders appear in 'ros2_ws/build' and 'ros2_ws/install' with the same name as the modules: 'ros2_broker' and 'webots_ros2_mavic_afarcloud', which must be deleted after each modification and before the next building to avoid update errors in the executables.

To execute the full system these actions should be run in order.

A.3 Postman configuration

In this design the cloud control and access is simulated with a mock server in postman, the API of this server is documented in: https://documenter.getpostman.com/view/30533683/2sA2r8145N, which can be used to create it in Postman, but in summary it is needed a mock server with this operations:

- POST {{URL}}/pose
- POST {{URL}}/battery
- POST {{URL}}/observation
- POST {{URL}}/image
- POST {{URL}}/state_vector
- POST {{URL}}/mission_report
- POST {{URL}}/alarm

After crating the mock server, the URL must be substitute in:

```
afarcloud_docker\EDGE\env\django_env\fleetManager\edge\views.py
(line 20): cloud_url
```

The manage of the system is made from the Postman app, all the Django server operations are in this API: https://documenter.getpostman.com/view/30533683/2sA2r8145F, some of them are used only internally, although, all of them can be done externally. The most important ones are POST plans, latest_state_vector and event, which are expected to be used from the cloud, the examples in the documentation can be used to test the scenario.

A.4 Execution

A.4.1. EDGE execution

The components in the EDGE can be executed directly in the host computer or in Docker, as commented at the beginning of this document.

In host

First terminal

Activate virtual environment:

In home/ros2_ws/src/afarcloud_docker/EDGE/env/django_env folder:

```
source ./bin/activate
```

Run Django server

In home/ros2_ws/src/afarcloud_docker/EDGE/env/django_env/fleetManager folder:

```
python manage.py runserver
```

Second terminal

Run ROS2 Broker

In home/ros2_ws/:

```
ros2 run ros2_broker broker
```

In Docker

Before the execution it is needed to create the network and the volume to keep the data after execution with:

```
docker network create --subnet=192.168.100.0/24 --
gateway=192.168.100.1 --ip-range=192.168.100.128/25 --
driver=bridge ros2_network
docker volume create django_volume
```

First terminal

Build and run Django server docker:

In home/ros2_ws/src/afarcloud_docker:

```
docker build -f Dockerfile_django_server -t
django_server:1.0 .
```

docker run -v django_volume:/server --name django_server -p 8000:8000 -p 1111:1111 --network ros2_network --ip=192.168.100.130 -t django_server:1.0

Second terminal

Build and run ROS2 broker docker:

```
docker build -f Dockerfile_broker -t ros2_broker:1.0 .
docker run --name ros2_broker -p 2222:2222 --
ip=192.168.100.131 --network ros2_network -t
ros2_broker:1.0
```

Or both together executing only in one terminal without seeing the logs:

```
docker-compose -f docker-compose.yml build
docker-compose -f docker-compose.yml up -d
```

And to see the logs:

Firs terminal

```
docker logs -f django_server
```

Second terminal

```
docker logs -f ros2_broker
```

A.4.2. UAV simulation execution

Third terminal

In home/ros2_ws/:

```
ros2 launch webots_ros2_mavic_afarcloud robot_launch.py
```

Wait for some second until Webots is launched successfully. After that you can see the Webots simulator running with the Mavic 2 Pro robot model in an example world, it is still waiting for plans.

A.4.3. Testing module execution

To execute the test, instead of the Mavic drone and its execution command, the commands needed is:

```
ros2 run tester uav_tester
```

After that you will see in the tester prompt some debugging information about the node, publishers' and subscribers' initialization, and the information of a mission received in the moment it is received as well, for example. And then the set of menus to set actions.

A.4.4. Postman

After all these execution you can use Postman to make a post request with one of the example JSON as body, this request will be received by the Django server and resent through a socket to the ROS2 Broker that extracts the information of the JSON to a message ROS2 and publish

it into ‘mission’ topic; this message will be received by the drone, which is subscribed to the topic, and it extracts the information of the message and execute the commands with the corresponding parameters on the message, periodically publishing the mission report and state vector and after the finish of a command execution a set of command report messages.