Ana Margarida Oliveira da Costa e Silva

Framework de planeamento de missões para frotas de drones interligados

A mission planning framework for fleets of connected drones

Universidade de Aveiro
2021

Ana Margarida Oliveira da Costa e Silva

Framework de planeamento de missões para frotas de drones interligados

A mission planning framework for fleets of connected drones

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica da Doutora Susana Isabel Barreto de Miranda Sargento, Professora Catedrática do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor André Braga Reis, Investigador no Instituto de Telecomunicações.

o júri / the jury

presidente / president

Professor Artur José Carneiro Pereira
professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Professora Doutora Teresa Vazão
professora associada do Departamento de Engenharia Electrotécnica e de Computadores do Instituto Superior Técnico

Professora Doutora Susana Isabel Barreto de Miranda Sargento
professora catedrática do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço ao Instituto de Telecomunicações e à Fundação Portuguesa para a Ciência e Tecnologia pelo suporte financeiro através de fundos nacionais e quando aplicável cofinanciado pelo FEDER, no âmbito do projeto FRIENDS: Fleet of dRones for radIological inspEction, commuNication anD reScue (PTDC/EEI-ROB/28799/2017).

À professora Susana Sargento, agradeço todas as oportunidades que me proporcionou durante o meu percurso académico e que culminaram no desenvolvimento deste projeto. Obrigada também pela disponibilidade para me acompanhar semanalmente e por toda atenção cedida.

Ao André Reis, que no papel de co-orientador se mostrou bom conselheiro e me prestou apoio nas fases mais fracturantes do desenvolvimento, mas a quem não posso deixar de agradecer pelo trabalho de mentoria que antecedeu esta dissertação. O conhecimento e as boas práticas que transmitiu foram sem dúvida marcantes no meu percurso de engenheira.

Àqueles com quem trabalhei no Instituto de Telecomunicações, André Martins, Bruno Areias e Nuno Humberto, que embora não estando presentes durante esta dissertação, tiveram impacto neste trabalho. Agradeço também ao Rui Lopes, com quem não trabalhei diretamente, mas que em várias ocasiões cedeu do seu tempo para me aconselhar. Aos colegas de investigação Mourato, Gonçalo e Nuno, pois sem a sua disponibilidade não teria sido possível demonstrar a vertente prática da minha dissertação.

Também agradeço àqueles que, não tendo contribuido diretamente para este trabalho, demonstraram o seu apoio durante o meu percurso académico:

Ao André Pinho, o meu colega de eleição nos trabalhos a pares, com quem aprendi a primar pela excelência e a exercer o espírito crítico em todo o nosso trabalho. Foi para mim um exemplo a seguir e com quem tenho muito gosto de ter trabalhado durante 5 anos. Ao Diego, que se juntou à nossa *dream team* e connosco fez parte das aventuras que começavam como um simples trabalho de grupo e que não irei esquecer.

Aos que me acompanham desde a 1ª semana de aulas, Hugo e Chaves, porque não foram apenas colegas de sala de aula, mas também fizeram sempre sentir a sua amizade incondicional nos altos e baixos deste caminho.

A todos os que sobreviveram comigo a estes 5 anos: Brunos, Daniel, Dinis, Francisco, Miguel, Pousa e Rodrigo, e também ao engenheiro honorário Fred.

Aos que estiveram comigo no associativismo neste ano, em especial ao Rafael Direito e ao Vasco Ramos. É certo que o tempo de isolamento teria sido mais difícil sem esta constante que eram os projetos da AETTUA.

Aos meus afilhados Francisco, Paulo, Fábio, João, Daniel, Miguel e Hugo, a quem tentei transmitir aquilo que sei, mas de quem recebi tanto mais de volta.

Ao Afonso, Alexandre, Bernardo, Gonçalo e Inês, os amigos de há mais de uma década, sempre presentes para celebrar novos marcos.

Aos meus pais, pois é inegável que seria impossível chegar aqui sem o seu apoio. Obrigada pelo vosso incentivo à excelência académica desde cedo.

Ao José Rosa, pela sua paciência para ouvir todas as minhas reclamações, hesitações, e muitas outras indecisões. Não chegaria tão longe se não soubesse que tinha alguém a amparar-me. Obrigada pelo apoio, mesmo quando não sei demonstrar como o agradecer.

Palavras-chave

veículos aéreos autónomos, drones, plataforma de controlo, linguagem de descrição, missões

Resumo

A utilização de drones aéreos tem-se vindo a popularizar à medida que estes se tornam mais acessíveis, quer em termos económicos quer em usabilidade. Atualmente, estes veículos são capazes de apresentar dimensões reduzidas e uma boa relação de custo-benefício, o que potencia que diversos serviços e aplicações suportados por redes de drones aéreos estejam a emergir. Alguns cenários que beneficiam da utilização de drones aéreos são a monitorização de situações de emergência e catástrofes naturais, a patrulha de áreas urbanas e apoio às forças policiais e aplicações turísticas como a transmissão de vídeo em tempo real de pontos de interesse. É comum que o controlo do drone esteja dependente de intervenção humana nestas situações, o que requer profissionais especializados no seu controlo. No entanto, nos últimos anos têm surgido diversas soluções que possibilitam o vôo autónomo destes veículos, minimizando a interferência manual.

Perante a enorme diversidade de casos de aplicação, muitas das soluções existentes para o controlo autónomo focam-se em cenários específicos de intervenção. Existem também plataformas de planeamento genérico de missões, mas que na sua maioria apenas permitem missões constituídas por conjuntos lineares de pontos a ser percorridos. Estas situações traduzem-se num suporte a missões que é pouco flexível.

Nesta dissertação propomos uma infraestrutura modular passível de ser utilizada em cenários variados, possibilitando o controlo autónomo de uma frota de drones aéreos num contexto de missão e a sua monitorização. Esta plataforma tem dois componentes principais, um integrado no computador a bordo do veículo e o outro no controlo terrestre. O primeiro permite a comunicação com o controlador de vôo para que se possa recolher diversos dados de telemetria e enviar instruções de movimento para o drone. O segundo permite monitorizar esses dados e enviar os comandos remotamente, possibilitando também um planeamento robusto de missões com múltiplos drones. Uma missão pode ser descrita num script que o módulo terrestre interpreta, enviando os comandos para os veículos atribuídos. Estas missões podem descrever diversos caminhos, modificando o comportamento dos drones de acordo com factores externos, como a leitura de um sensor. Também é possível definir plugins para serem reutilizados em várias missões, como por exemplo, integrando um algoritmo que garante que todos os drones mantêm a conectividade.

A solução foi avaliada em cenários com um único drone e com a colaboração de múltiplos drones. Os testes foram executados em ambiente simulado e também num ambiente com drones reais. O comportamento observado nas missões é semelhante em ambos os cenários.

Keywords

autonomous aerial vehicles, drones, control platform, user-friendly description language, missions

Abstract

The usage of aerial drones has become more popular as they also become more accessible, both in economic and usability terms. Nowadays, these vehicles can present reduced dimensions and a good cost-benefit ratio, which makes it possible for several services and applications supported by aerial drone networks to emerge. Some scenarios that benefit from the use of aerial drones are the monitoring of emergency situations and natural disasters, the patrolling of urban areas and support to police forces, and tourist applications such as the real-time video transmission of points of interest. It is common for the control of the drone to be dependent on human intervention in these situations, which requires professionals specialized in its control. However, in recent years, several solutions have emerged that enable the autonomous flight of these vehicles, minimizing manual interference.

Taking into account the enormous diversity of use cases, many of the existing solutions for autonomous control focus on specific scenarios. Generic mission planning platforms also exist, but most of them only allow missions consisting of linear waypoints to be traversed. These situations translate into a mission support that is not very flexible.

In this dissertation, we propose a modular infrastructure that can be used in various scenarios, enabling the autonomous control and monitoring of a fleet of aerial drones in a mission context. This platform has two main components, one integrated into the onboard computer of the vehicle, and the other one in the ground control. The former allows the communication with the flight controller so that it can collect telemetry data and send movement instructions to the drone. The latter allows to monitor this data and send the commands remotely, also enabling robust mission planning with multiple drones. A mission can be described in a script that the ground module interprets, sending the commands to the assigned vehicles. These missions can describe different paths, modifying the behaviour of the drones according to external factors, such as a sensor reading. It is also possible to define plugins to be reused in various missions, for example, by integrating an algorithm that ensures that all drones maintain connectivity.

The solution was evaluated in scenarios with a single drone and with the collaboration of multiple drones. The tests were performed in a simulated environment and also in an environment with real drones. The observed behaviour is similar in both scenarios.

Contents

List of Figures

List of Tables

Acronyms

AMSL	above mean sea level
API	application programming interface
CLI	command-line interface
DDS	Data Distribution Service
DSL	domain-specific language
FANET	Flying Ad-hoc Network
FC	flight controller
GPS	Global Positioning System
GS	ground station
GUI	graphical user interface
HITL	Hardware In The Loop
HTTP	Hypertext Transfer Protocol
I2C	Inter-Integrated Circuit
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
NED	North-East-Down
NTP	Network Time Protocol
PID	proportional-integral-derivative
RC	radio control
REST	representational state transfer
ROS	Robot Operating System
SBC	single-board computer
SDK	software development kit
SITL	Software In the Loop
SPI	Serial Peripheral Interface

TCP	Transmission Control Protocol
UAV	unmanned aerial vehicle
UDP	User Datagram Protocol
USB	Universal Serial Bus

Chapter 1

Introduction

In this chapter, we introduce the context and topics of this Dissertation. We start by presenting a brief motivation for this work, which is followed by the delineation of the goals we want to achieve, and what contributions have been originated with this work. To conclude, this chapter also provides the document outline, summarising the main contents of each chapter.

1.1 Motivation

Unmanned aerial vehicles (UAVs), most commonly referred to as drones, have become a prominent subject in the past years. Initially targeted for military applications, its usage has been popularised as they have also become more accessible for the regular citizen.

Their affordability, reduced size, and flexibility contributed to numerous research efforts towards innovative solutions involving fleets of networked UAVs. They can be used in a wide range of scenarios, such as monitoring emergency situations and natural catastrophes, patrolling and surveillance of urban areas, and tourism applications. Many works mostly address the implementation, analysis, and improvement of autonomous and remote drone control systems, as well as algorithms related to path planning, applied to a specific scenario. However, most solutions that offer a mission planner provide limited functionality. These missions usually comprise a fixed set of waypoints, not being adaptable at runtime in response to events, and usually provide limited multi-drone collaboration.

To overcome these restrictions, this Dissertation proposes a mission planning framework for drone fleets. This framework will be supported by an underlying system for remote drone control, which will also be developed in this context. The major novelty of this work compared to other similar platforms is its ability to allow the declaration of a non-linear mission, which may lead to a different outcome based on the environment and concurrent multi-drone collaboration. Although this solution aims to have a generic approach towards missions, it should also provide enough functionalities, such that it is available to implement specific use cases in different areas.

1.2 Goals and contributions

This Dissertation aims to create a framework in which it is possible to autonomously control a networked fleet of drones, following a non-linear mission. These are the goals that we plan to achieve:

- Implement a drone control module that is capable of receiving high-level commands and instructing the drone to execute them
- Develop ground station software that interfaces with the drone module to submit commands and retrieve telemetry data
- Allow the interpretation and execution of a mission plan
- Provide mechanisms for multi-drone collaboration in a mission
- Allow defining how the mission should respond to different events and conditions
- Support the definition of background restrictions, such as maintaining connectivity, that does not have to be explicitly stated in the mission
- Test the missions in a simulated environment, and demonstrate the validity of the developed solution in a scenario with real drones

Motivated by these goals, the solution that we propose resulted in the implementation of the mentioned drone and ground station modules. As for the mission support, we defined a domain-specific language (DSL) that can describe a mission plan and implement the necessary components for its interpretation and execution. A plugin system for these missions was proposed and implemented. Two plugins were also developed to provide specific functionality: one that handles a drone's replacement when it reaches a low battery level, and another one that dispatches relay drones to maintain connectivity. Additionally, we also developed a tool to easily launch simulator and drone module instances, which facilitates the process of setting up a test environment.

This Dissertation contributed to the FRIENDS (Fleet of dRones for radIological inspEction, commuNication anD reScue) project [1]. So far, the drone and ground station modules developed in the context of this Dissertation were used during field tests, but it is planned to also use the mission planning features in future trials of this project. Also, in this project's scope, the work of this Dissertation contributed to an article titled *Radiological Scouting, Monitoring and Inspection using Drones*, which has been submitted to the international journal MDPI Sensors. Additionally, this work is also included in a paper being prepared, focused on the remote drone control platform; a third paper is planned to be written after the conclusion of this Dissertation, which will present the mission planning framework.

1.3 Document outline

The content of this Dissertation is organised into six chapters.

Chapter 2 (Background and Related Work) introduces the background context of this Dissertation. The relevance of UAVs and scenarios in which UAVs have been employed are also presented. There is a brief analysis of available open-source and commercial solutions for autonomous and remote UAV control, which is followed by the description of some research works on that same subject. The flight stack and robotic middleware to be used in this work are also mentioned. We conclude this chapter by listing some aspects lacking in previous works and proposing the improvements that we would like to include in our platform.

Next, in chapter 3 (Architecture), we provide an overview of the architecture of our solution. The global requirements of the platform are presented, followed by an architectural proposal to discuss those issues. The communication between the drone and ground modules is also addressed.

Chapter 4 (Implementation) follows with a thorough description of the implementation of the drone and the ground station modules. On the drone side, we explain the flight controller (FC) integration, how to launch the module, the format of the exchanged messages, the allowed commands, and the shared telemetry fields. On the ground side, we describe the several internal components and the available endpoints. The last section of this chapter addresses the simulation and test environment, scripts and utilities developed for that end, and the results of some conducted tests.

After presenting the base platform implementation, we address the mission planning framework in chapter 5 (Mission Support). It starts with a brief explanation of the reasoning for using a DSL for mission description, followed by the definition of the requirements for mission support. We explain the features that allowed us to build the DSL seamlessly and how the different requirements are met by the DSL. We also provide a few examples to demonstrate how to write a mission plan, in varying complexity levels.

In chapter 6 (Experiments), we present the experiments that we conducted in order to validate this work. Some of these experiments were executed in a simulated environment, while others involved the usage of physical drones in real flights and missions.

Finally, chapter 7 (Conclusion) summarises the developed work and make some final remarks on the results. We also present some ideas for future work.

Chapter 2

Background and Related Work

In this chapter, we present the background of this work, as well as the related work, through which we can contextualise this Dissertation. First, we start by describing UAVs, their relevance, and scenarios in which they are employed. Afterwards, we address autonomous and remote drone control, in which we present existing solutions and research efforts towards this autonomous control. We also approach the flight stack to be used during the development of this work, as well as the middleware for component integration. We conclude by discussing the works that are presented and how this Dissertation fits in this context.

2.1 Unmanned Aerial Vehicles

An unmanned aerial vehicle is an aircraft which does not have a human pilot on board. UAVs are usually integrated into a system that, besides the vehicle itself (or multiple vehicles), is comprised of a ground-based control system, and the communication channel between the UAVs and the ground station. Although UAVs were mainly used in a military setting originally, they have proliferated and thrived, and are now employed to serve the population across different domains. Regardless of the context, UAVs are able to make processes faster and more flexible, while also improving the precision and cost-efficiency. To leverage this, UAVs have also become more affordable in recent years, which leads to the emergence of a wide array of drone-based solutions.

2.1.1 UAV relevance and prevalence

UAVs combine three relevant technological aspects - data processing, autonomy and boundless mobility [2]. They enable the analysis and access to certain spaces that would not be possible otherwise, such as situations in which the environment poses a threat to human beings or is not easily accessible by ground. In a scenario that involves a rescue operation, UAVs have several advantages compared to other vehicles. UAVs can be rapidly deployed to shorten the delay for a rescue operation, access areas inaccessible by road, and, more importantly, have the flexibility to adapt the network according to the current needs [3].

The number of developed drones has increased significantly in the past few years, namely in civil and commercial platforms. Progress is expected in this last sector, as new developments in robotics, computer vision, and other technologies are being achieved. While these civil-drones are not as developed as systems used for military applications, studies and advances in the development of technology can offer professional systems with new, improved and promising features [4].

The relevance of UAVs in today's society can be attested by their prevalence in the media. Although lately the subject of drones has been revolving around legislation and privacy issues that arise with their usage[1], often we also note their adoption and deployment by entities such as in law enforcement. The following are real examples. With the aid of a drone, border patrol agents rescued two men that were attempting an illegal entry and got lost and injured[2]. A search and rescue drone has also been tested to replace police helicopters to help save whales[3]. Police forces have also used drones to monitor protests in England[4]. In Portugal, drones have also been deployed, such as to help firefighters monitor the area of a fire[5], or during the COVID-19 pandemic to monitor and warn citizens that broke the confinement while in state of emergency[6] [7].

2.1.2 Use cases and applications

UAVs can intervene in numerous scenarios ranging from agriculture, tourism, surveillance and sensing missions, logistics transportation, weather monitoring, fire detection, communication relaying, and emergency search and rescue.

Among these vehicles' various applications, achieving high-speed wireless communications is expected to become important in future communication systems [3]. UAV-aided wireless communication offers a solution to provide wireless connectivity for devices without infrastructure coverage due to communication infrastructure failure caused by natural disasters or to extend and improve coverage around saturated base stations [5]. Amazon provides the Prime Air[8] service, in which fully autonomous UAVs deliver packages to its customers in thirty minutes or less. In [6], a UAV is used to inspect facilities and structures to identify cracks in buildings, saving a human inspector from potential risks. For tourism purposes, [7] describes a route selection scheme for drones to provide virtual tours on touristic sites,

[1]https://observador.pt/2020/08/02/lei-vai-ser-mudada-para-facilitar-uso-de-drones-por-parte-da-policia/

[2]https://www.cbp.gov/newsroom/local-media-release/border-patrol-drone-successfully-rescues-two-illegal-aliens

[3]https://www.mirror.co.uk/news/politics/drone-tested-search-rescue-wales-23339872

[4]https://www.theguardian.com/uk-news/2021/feb/14/drones-police-england-monitor-political-protests-blm-extinction-rebellion

[5]https://observador.pt/2020/01/23/incendios-bombeiros-de-vila-real-usam-drone-para-ajudar-nas-operacoes/

[6]https://www.publico.pt/2020/04/02/politica/noticia/psp-vai-utilizar-camaras-portateis-drones-estado-emergencia-1910782

[7]https://www.noticiasdeaveiro.pt/aveiro-nove-autos-levantados-durante-fiscalizacao-da-psp-na-cidade-com-ajuda-de-drone/

[8]https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011.

using a 360° video camera to capture the surroundings. A UAV was adapted in [8] to act as a semi-autonomous pesticide sprayer for agricultural applications. In [9] it is presented a method for decentralised control for multiple UAVs aimed at area patrolling. There are many scenarios involving natural disasters, such as [10], in which multiple UAVs are dispatched to perform rapid assessment after an earthquake. Another comparable set of scenarios are those related to environmental monitoring. In a previous Dissertation [11], [12], a solution for monitoring and delineating forest fires was proposed. This Dissertation also contributes to the project [1], which aims to enable a fleet of drones to perform a radiological inspection and mapping of an environment.

2.2 Autonomous and remote UAV control

It is sufficient for some applications to have a human operator controlling the UAV. However, this is not always ideal, since, at times, the task could be easily automated, or the required behaviour could be difficult to execute manually. Although controlling the drone through a radio control (RC) may be considered remote control, this can be extended further - for example, by using a local platform that can communicate with the drone, or even reaching a drone that is in a different location through the Internet. Autonomy can be considered at different levels. Remotely sending an action to be performed by the drone can be considered an autonomous task, since after receiving the command, the drone moves by itself without human interference. A higher level of autonomy would include having these commands being sent automatically, without having the user to provide each one individually. Instead of having an external component to trigger these actions, these could be generated from within the drone, resulting in a distributed decision-making in a fleet of multiple drones. At the highest autonomy level, it is possible for the drone to perform actions that it was not explicitly instructed to, but were deemed necessary, such as avoiding a collision. This section presents several works that address remote and autonomous UAV control.

2.2.1 Open-source and commercial solutions

There are multiple solutions for remote UAV control and mission planning that are available to use. Some of these solutions are free and open source, while others are commercial and closed source. These can be used for generic tasks that involve moving the UAVs, or in some cases they may target a specific use case. In terms of supported hardware and flight controller, these may vary.

QGroundControl

QGroundControl[9] is a free open-source ground control station platform associated with the Dronecode project[10], that can be connected to any vehicle that uses the same protocol,

[9] http://qgroundcontrol.com/.
[10] https://www.dronecode.org/.

MAVLink, to communicate. It can also be used to setup and configure the vehicle. It supports mission planning for autonomous flights, which allows setting a sequence of commands beforehand and then executing them. It contains a map where it is possible to see the vehicle position, flight track and more information, and video streaming is also available. Multiple vehicles can be managed simultaneously, but they cannot collaborate in a mission. Figure 2.1 shows the graphical user interface (GUI) of QGroundControl. Other platforms have similar interfaces.

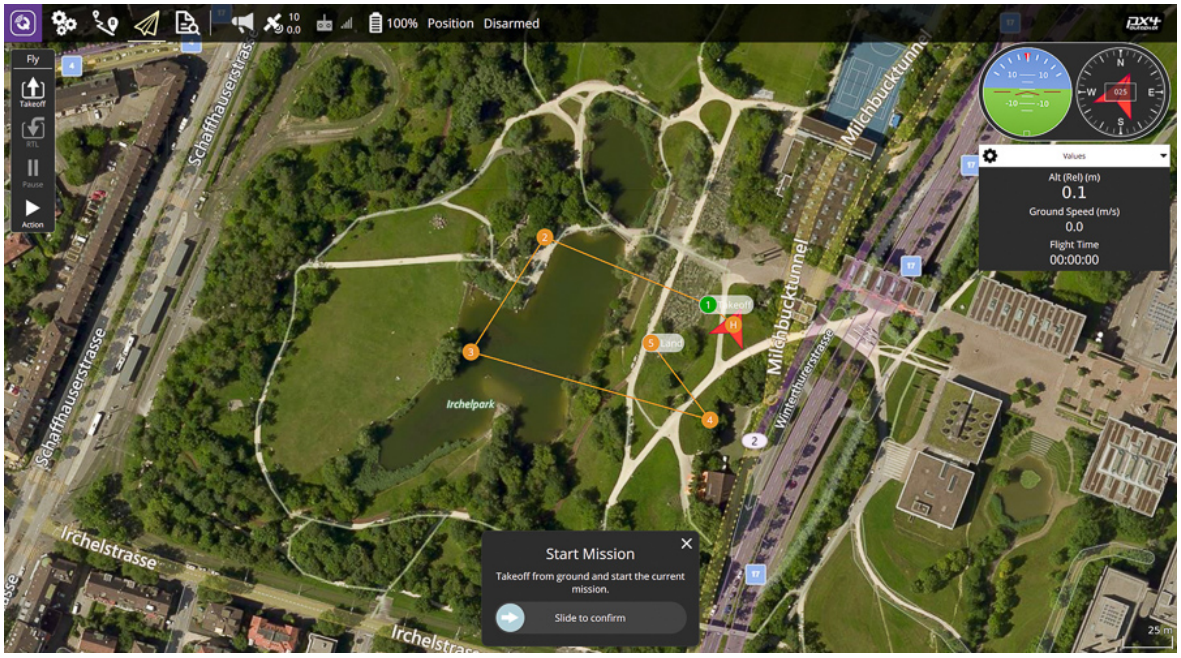


Figure 2.1. QGroundControl graphical user interface

Mission Planner

Mission Planner[11] is part of the ArduPilot[12] project and is another free and open-source ground station application. In terms of features, it is very similar to QGroundControl, and it is also compatible with MAVLink autopilots. It also allows configuring the vehicle parameters, planning and loading missions into the autopilot, and visualising the connected UAVs in a map interface.

FlytBase

FlytBase[13] is an enterprise drone automation platform that enables UAV deployment in cloud-based business applications. FlytBase supports DJI drones and flight controllers with either PX4 or Ardupilot autopilots. The solution provided by FlytBase has two major components: FlytOS and FlytCloud[14]. FlytOS is a custom Linux distribution providing

[11]https://ardupilot.org/planner/docs/mission-planner-overview.html.
[12]https://ardupilot.org/index.php/about.
[13]https://flytbase.com/.
[14]https://flytbase.com/platform/.

APIs and SDKs for building high-level drone applications, which is deployed at the edge and contains time-critical components: collision avoidance, precise navigation, and object tracking. Computation heavy tasks are offloaded to FlytCloud, which provides real-time access to control, telemetry, and video from the UAVs. This cloud connectivity also enables the remote operation of the drone fleet.

Additionally, the user can also acquire FlytNow, a web-based ground control station, which can also be launched from a smartphone to manage the drone fleet. It also includes a mission planner, from which the user can create a mission by adding the desired waypoints in a map and customising parameters such as the mission altitude and speed. With FlytNow, it is also possible to integrate specific solutions, such as deliveries, emergency response, or surveillance.

DroneDeploy

DroneDeploy[15] provides commercial drone solutions, aimed mostly at image capture, mapping, and surveying. It supports a defined set of drones[16], most being DJI drones, and there is no support for MAVLink-compatible autopilots. Flight plans can be planned and shared using the mobile application, which can also be used for real-time drone mapping. DroneDeploy provides photogammetry features, along with 3D map reconstruction. Additionally, they also provide specific solutions aimed at agriculture, construction, and inspection, among other areas.

2.2.2 Control platforms

Besides these free and commercial solutions, there have also been research efforts towards autonomous and remote drone control. These offer very distinct approaches, some focused on specific use cases, while others provide more generic functionality. In recent years, the number of available solutions has increased [13], [14].

The work in [15] proposes an architecture for controlling a drone remotely. Through the usage of multiple cellular carriers, the authors maintain drone operation even outside the communication area of a cellular carrier. No support for multi-drone interaction or collaborative missions is provided.

The ROLFER (Robotic Lifeguard for Emergency Rescue) project in [16] proposes a drone control solution which aims to provide immediate rescue and life-saving services. This is accomplished by enabling distressed swimmers to call a drone by tapping a button on a smartwatch. Its applications are limited to rescue support scenarios, and it provides no multi-drone support and no mission planning capabilities. However, experimental results in real flights demonstrated the effectiveness of this system.

Other examples of platforms with drones executing automated tasks include [17], in which the drones focused on automated indoor inspections, but it does not handle takeoff

[15]https://www.dronedeploy.com/.

[16]https://support.dronedeploy.com/docs/recommended-and-supported-drones.

and landing. The used autopilot was the Pixhawk, which communicates via the MAVLink protocol. This solution was tested in a realistic scenario.

The work in [18] outlines the communication and control systems for a single UAV in difficult environmental conditions. Similarly, it also uses the Pixhawk autopilot. However, this work did not present any experimental flight results.

Research in [19] resulted in the development of a cost-efficient platform enabling longer-duration missions for persistent mobile aerial surveillance. This is accomplished by using intelligent battery health management and drone swapping to maximise the time during which a zone may be covered without interruption. This mechanism enables a set of quadcopters to take turns hovering above a location. It was tested on the field, using three drones and five batteries, achieving a total flight time of 54 minutes with four vehicle swaps. Its communication range, however, is limited to half a mile.

On the topic of Flying Ad-hoc Networks (FANETs), the work in [20] presents an implementation study of an ad-hoc network between drones, but lacks a detailed analysis of communication delays, and does not consider the possibility of cellular uplinks. In [21], the authors developed a networking framework for FANETs, where multiple vehicles, either aerial or ground, cooperate in a mission. Each vehicle broadcasts the navigation information, including the position, speed, and direction. This information is also used in the ground station for anti-collision and monitoring. The work in [22] proposed a framework in which a group mobility model is applied to mission planning, with consideration for the network connectivity. The FANET was simulated in Matlab, but there were no experiments conducted in a real scenario.

As discussed in the previous section, there are many applications for drone fleets, and among those, several approaches have been studied for patrol and surveillance tasks with aerial drones. A set of strategies to plan patrol routes and distribute surveillance of areas by several drones simultaneously is presented in [23], considering factors such as uncertainty, time, and communications, and showing that an optimised solution can reduce the need and frequency of communications in 50 times. The different strategies were tested in a simulated environment. Algorithms for route planning and navigation in urban scenarios with complex 3D topographies, based on the A-star and Ant Colony Optimization algorithms were proposed in [24]. This work aims to identify the optimal patrolling path, given a set of patrolling positions. Further research efforts include trial studies to assess the performance of the solution in realistic patrolling tasks. Centralised and decentralised approaches to planning patrol routes with multiple drones are presented in [25], aiming to reduce total energy consumption and maximising flight time. The authors considered that a centralised approach was not feasible in real-world scenarios due to the need of constant communication between the vehicles and the ground station. The algorithm was not validated in real flights.

Multi-UAVs surveillance platforms have also been researched in the last years [26][27], which aim to control multiple UAVs in a set of different tasks. They provide the possibility for UAVs to distinctly receive different tasks and missions, with specific surveillance-based tasks in an IoT environment. In [28], a remote management architecture for drone fleets

was developed, with the specific goal of aiding the management of solar power plants and object tracking. The whole system was designed to serve that specific purpose. With image processing and automated remote control of the UAV, it is possible to generate autonomous missions for the inspection of defects in solar panels. The used drones were powered by a PixHawk flight controller.

2.2.3 Mission planning

The work in [29] describes a high-level architecture for the design of multi-drone systems, including mission planning. The architecture is composed of several blocks: communication and networking, coordination, and sensing, while also integrating with the UAV hardware. The platform also contained a user interface that allowed the user to define the area to be monitored in a map, which is sent to the ground station which contains the mission planning and control modules. The mission planning component takes the high-level tasks defined in the user interface, and breaks it down into individual flight routes for the individuals UAVs. This path planning control is centralised, which allows replanning the routes and adaptive coordination. It is also mentioned that decentralised mission planning strategies are available, although those were not further elaborated. The system was demonstrated in several real-world applications, which included assistance during a disaster, documenting progress in a construction site, and participating in a fire service drill. This work is revisited in [30], which adopts the same architecture. An interesting aspect that was included is a new case study, which involved a search and rescue mission. During mission execution, UAVs could be repositioned to serve as a relay. This allowed maintaining the connection between the detecting UAV and the ground station. The centralised and distributed decisions were also clarified: navigation and video streaming are fully autonomous, collision avoidance is distributed, and the path planning may be either centralised in the ground station or, in some cases, in one of the UAVs.

A system supporting complex mission definition, planning and execution is proposed in [31]. This project provides a platform for infrastructure inspection using UAVs. The mission definition aims at improving planning efficiency by setting waypoints, using high-level mission definition primitives. The user has access to a GUI, where it is possible to use a map tool to select the central target location, and then it is possible to choose the inspection type. However, it provides no support for multi-drone interaction, and it has a limited communication range using Wi-Fi.

Other works, such as [32], presented methods to generate trajectories and missions for multiple drones, satisfying a set of given Signal Temporal Logic requirements. This approach can be used as an offline path planner, and the resulting missions are static, unable to be adjusted at runtime. However, this work was validated both in a simulated environment and with real UAVs.

In [33], a model for mission planning in outdoor material delivery with UAVs is proposed. Mission plans have to consider several aspects, including different weather conditions, payload capacity, energy capacity, fleet size, and the number of customers visited by a UAV.

The proposed solution presents a model that takes the previous considerations into account.

An embedded decision-making module for autonomous UAVs missions is proposed in [34]. This work's novelty is that the module allows choosing a recovery action when there is a failure, generating a new mission plan. According to sensor data, the failures are defined and may lead to changes in trajectory, emergency landings, or decreasing the speed.

The connection of abstract task definition at a mission level with the control functionalities in autonomous missions is addressed in [35]. Since this work also approaches heterogeneous vehicles, a common ground was found by defining parametrized tasks such as *fly-to*, *take-off*, *scan-area*, or *land*. The behaviour of these actions was implemented for each platform. An experiment was conducted, in which a scan and search mission is executed. The user selects an area to scan (that could represent an area in which a missing person could be located) in a GUI, which will generate a sequence of waypoints. While the UAV executes the scan, the user will monitor the live video. Upon noticing a potential person, the user can choose to pause the mission to manually control the drone, and then resume the mission or cancel it entirely. This mission was performed with a real UAV in an indoor site.

A language aimed at solving collaboration among robots and communication with humans was proposed in [36], called Situation Information Exchange and Interpretation Language (SIEIL). This language's purpose is to provide a common language specification to command and control robotic forces. SIEIL uses a context-free grammar to describe the possible actions, subjects of those actions, and other variables. However, those expressions are transformed into RDF/XML documents to be interpreted as commands by the intervening robots.

Although [37] refers to underwater vehicles, it is still relevant in this context, as it covers available task description languages for the vehicles. This work proposes a language that is easy to understand. The proposed notation provides a hierarchical structure consisting of simple and composite statements, the latter being a container for additional statements. A simple statement may either be an assignment or a comparison, and is composed of three tokens: a key, a sign and a value. The task description is easily understandable by humans and may contain responses to events, such as finding an object or setting a time limit for the task. However, collaboration was not addressed, with the mission being sent to each vehicle to be executed individually.

Similarly, [38] also suggests an approach to mission planning for underwater vehicles by formalising the task description. In this case, the language is based on the GeoJSON standard. This standard is a format for encoding geographic data structures, such as a point, a line string, or a polygon. Like in [37], the statements are described in key-value pairs, but will be JavaScript Object Notation (JSON) formatted and include the geometry definitions of GeoJSON, such as coordinates or polygons. It also does not address collaboration between multiple vehicles.

In a previous Dissertation, [39], and additionally in [40], a platform for communication and multi-drone control with autonomous mission support was presented. The architecture was composed of the drone side and the ground station components. It contains a mission planner that enables the user to configure a mission in a web interface, selecting the desired

actions. By monitoring the UAV's telemetry, it is possible to verify anomalous situations, such as a low battery level. In that case, the current UAV will be replaced. During a mission, a UAV can request for another drone to collaborate, which will cause the remaining waypoints to be distributed among the participating vehicles. However, this platform is custom-made, and is not able to interact with other platforms in the literature.

2.3 Related technologies

The interaction between the UAV, the user, and other software components requires the usage of appropriate technologies for that end. Those technologies should cover communication, integration and interoperability. Some of the works presented in the previous sections provided information concerning the integration with the drone's hardware.

2.3.1 Flight stack

A flight controller is a board coupled to the drone which allows it to perform movements. Commands may be sent to the FC, either through a RC or programmatically, which the FC then converts in controls for the Electronic Speed Controllers, controlling the motors. This allows controlling the drone's flight and stabilising it. FCs usually integrate several onboard sensors, which have to be calibrated to maintain altitude and navigate safely. The FC needs to be connected to a single-board computer (SBC) to control the FC programmatically, which will run software capable of sending the commands.

Major drone manufacturers, such as DJI[17], fabricate their own flight controller. However, some of those rely on proprietary firmware and provide no SDK or limited means of controlling the drone remotely. The latest generation of FCs developed by DJI shows more potential in remote interaction; however, those still rely on closed-source firmware and are more expensive. Some drone manufacturers provide flight controllers that can be programmed with open-source firmware.

A few open-source autopilot firmware solutions are available, such as dRonin[18], Ardupilot[19], and PX4[20]. From these, we will favour PX4. The PX4 Autopilot provides guidance, navigation, and control algorithms for autonomous drones, along with estimators for attitude and position. PX4 is also part of the Dronecode[21] project, which is hosted under the Linux Foundation. As previously mentioned, QGroundControl is also part of that project, as well as Mavlink[22], a lightweight communication protocol for UAV systems, and MAVSDK[23], a MAVLink library providing an application programming interface (API). Although this flight stack can be run in several FCs, they officially support one in particular, the Pixhawk 4[24].

[17]https://www.dji.com/pt.
[18]https://dronin.org/.
[19]https://ardupilot.org/.
[20]https://px4.io/.
[21]https://www.dronecode.org/.
[22]https://mavlink.io/en/.
[23]https://mavsdk.mavlink.io/develop/en/index.html.
[24]https://docs.px4.io/master/en/flight_controller/pixhawk4.html.

This FC is widely adopted in research projects, which was mentioned in several works [8], [17], [18], [28], [41].

2.3.2 Middleware for robotics

Nowadays, UAVs, or robots in the generic sense, are designed and built with more broad goals in mind, not always focused on performing a specific task. With this assertion, we are not claiming that they are not used for executing a specific task, but that the same drone type could be a fit for multiple scenarios. These robotic systems may be comprised of several different components, both hardware and software-wise. Different components may depend on heterogeneous software modules to be controlled, developed in different programming languages and requiring different communication protocols to be interacted with. This modular design has its advantages, but we also have to consider how to integrate all these different modules into a cohesive system. The integration can be eased by making use of middleware software to which we can delegate communication, interoperability, and configuration of those modules [42]. Middleware systems can be employed in the context of distributed systems to reduce development time and redundancy, by providing commonly needed services and functionalities.

In the context of robotics, this has converged into one particular middleware [43] in the past years - Robot Operating System (ROS)[25]. It is possible to find several examples of UAV systems, such as [44]–[46], and other works that were previously mentioned, that rely on ROS for a myriad of tasks. ROS is an open-source set of software libraries and tools that leverage distributed robot software.

ROS

The Robot Operating System is a framework targeted for writing robot software. It is comprised of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour.

The middleware part of ROS is the communications infrastructure, which is a low-level message passing interface for intra and inter-process message exchange. Apart from message passing, it is possible to record and replay messages through the *rosbag* tool[26].

In ROS, a component that performs a determinate computational task is called a *node*[27]. A single application may be comprised of several nodes that exchange ROS messages between them in order to communicate. The messaging pattern adopted by ROS is the publisher-subscriber model; a given node may publish information to a certain topic[28], while a different node will subscribe to that topic to retrieve the data. Multiple nodes may publish and subscribe to the same topic, and a single node may also publish and subscribe to different topics. These topics must respect a defined message type, such as an integer message, a string

[25]https://www.ros.org/.
[26]http://wiki.ros.org/Bags.
[27]http://wiki.ros.org/Nodes.
[28]http://wiki.ros.org/Topics.

message, or a custom message type. The several nodes that are part of a single application can be packaged using ROS's build tools, which allow them to be easily shared.

In order to use ROS, it is required to run *roscore*[29] - a collection of nodes and other programs that are pre-requisites of a ROS system. Without it, ROS nodes will not be able to communicate. Among other modules, *roscore* will start the ROS master[30], which provides naming and registration services to the rest of the nodes, enabling the nodes to locate one another. The node discovery mechanism is centralised in the machine that is set to be running the ROS master.

ROS2

Initially, the development of ROS was guided towards a specific use case, which had characteristics such as working on a single robot, no real-time requirements, and assuming excellent network connectivity[31]. Since ROS later expanded and was adopted in applications of several different domains, there came a need to meet the needs of the community better. This included addressing new use cases, such as teams of multiple robots, real-time systems, non-ideal networks, and production environments. This required significant changes in the ROS API; as such, a new framework was created - ROS2.

ROS2 borrows many concepts from its predecessor, but also brings some major improvements and is being actively developed. ROS2 still uses nodes and the publisher-subscriber pattern to communicate. ROS1 used a custom serialisation format, a custom transport protocol, and a custom central discovery mechanism[32]. Unlike it, ROS2 has an abstract middleware interface, through which serialisation, transport and discovery is provided. All implementations of this interface are based on the Data Distribution Service (DDS) standard, which enables ROS2 to provide various Quality of Service policies to improve communication over different networks. This is the biggest difference compared to the previous version - while ROS's discovery mechanisms rely on a central master, ROS2's nodes are essentially peer-to-peer, enabling a distributed node discovery.

Since so many packages developed for ROS1 were already widespread and stable, the *ros bridge*[33] package was developed to allow communication between nodes from the different ROS versions and ease the transition to ROS2. Using the most recent ROS2 version will not be an impediment to collaboration with other modules that use the previous version.

2.4 Discussion

This chapter approached many diverse subjects related to UAVs usage. As a first conclusion, we are able to agree that in general, UAVs are an interesting research subject that can be a solution for many different use cases. Their usage has gained traction on media, and this

[29]http://wiki.ros.org/roscore.
[30]http://wiki.ros.org/Master.
[31]https://design.ros2.org/articles/why_ros2.html.
[32]https://design.ros2.org/articles/changes.html.

increase in popularity is also reflected in the amount of commercial and non-commercial solutions available. The scenarios in which drones are employed are not only diverse, but also incredibly relevant for today's society, since those are often in favour of the citizens' well-being and safety.

Overviewing all the presented solutions and research works, we can formulate a few conclusions. Most solutions focus on a particular task instead of providing solutions for a more generic setting. For example, surveillance missions are developed to detect specific behaviour, such as identifying a human being. While research works usually tackle one problem at a time, when looking at commercial solutions, it may be possible to solve multiple issues with the same platform, but it may come at a high monetary cost since those features may be part of different packages.

The offered mission planning frameworks were often very strict, mostly addressing waypoint traversing or area coverage. Although most missions can be summarised as a sequence of waypoints to be traversed, the definition of those could be more dynamic. For example, instead of manually selecting a set of waypoints from a map, those could be programmatically defined in relative coordinates. Different strategies could be used in area surveillance and mapping, but those algorithms are usually part of the platform's solution, which is not easily replaced.

It may also be interesting to follow different flight paths in response to a sensor reading or other event. This means that the missions could be dynamic, and the actual course of action would be decided at runtime. An issue that was addressed in some solutions but not always present was fault tolerance - being able to progress even though one or more drones encountered an issue. For example, if one drone is running low on battery, it should not jeopardise the whole mission - it should be possible to replace it.

In many solutions, it seemed that network connectivity was not a major concern. The range at which a drone can be placed should not constraint our missions, as such, it is relevant to implement measures that ensure that the UAVs can always maintain connectivity. This probably ties with the fact that a considerable amount of the research works were an architectural proposal or validated in a simulated setting, not dealing with the real-life scenario's constraints.

Multi-drone missions were often simplified or not addressed at all. A major point of interest in having a fleet of drones is the collaboration between the vehicles. Collaboration implies that, not only more than one drone is present in the mission, but they are also working together towards the same goal. This may be present in the form of synchronisation, such as having one drone perform a task only after another drone has finished, autonomous replacement of faulty drones, or division of a task into subtasks.

Looking at these conclusions, we want to implement a solution that addresses the aforementioned issues. This leads to building a platform that: 1) can be used in different contexts; 2) offers flexible mission planning, with missions that will dynamically adjust to the environment; 3) supports multi-drone collaboration; 4) is able to respect relevant constraints such as maintaining network connectivity; and 5) can be validated in a real scenario. It is

ambitious to tackle all these different fronts at once in a single solution, which may require simplifying some of these constraints.

The knowledge gathered by researching related works can be applied into the design of the mission framework that will be developed in this Dissertation. The platform should be modular, separating the concerns of the ground station and the drone. Although fully distributed control is more compelling and provides more flexibility, it is also more challenging, as such, it would not be wise to attempt to implement such a solution as a first approach to the problem. Instead, the groundwork should be a system with centralised decision-making in the ground station. However, this should be developed without locking the system to that implementation, avoiding that the decentralisation would require redoing most of the work. As some works suggested, describing mission plans in an adequate language can lead to interesting results. Those languages introduced flexibility to the mission description that was not possible through a GUI.

This platform should target a wide range of drones. As such, instead of focusing on fully-fledged drones that relied on proprietary software, this solution will be designed for community-developed hardware. As in many of the presented works, we will make use of the Pixhawk 4 flight controller. Since we have access to those flight controllers, we opted to use the full PX4 flight stack, including the PX4 autopilot. However, other MAVLink compatible FCs could also potentially work.

The usage of ROS to communicate and integrate several components was also prevalent. By using ROS, we are able to compartmentalise our solution, allowing different ground station software to be used with our drone software or different drone software to be used without ground station software, given that the message format is respected. This is particularly useful when trying to integrate with different research projects. Although the decision process is centralised, this is still a distributed system, as such, usage of ROS2 is more adequate. It is still possible to interface with projects using the first version of ROS, by configuring the rosbridge.

2.5 Summary

In this chapter, we confirmed that UAVs are relevant in today's society, backed by the prevalence in the media and by mentioning several scenarios that can be solved with the use of drones. We then presented existing commercial and open-source platforms, and research work that also proposed related solutions and mission planning software. Use cases involving drones are very diverse, often focusing on improving specific tasks. However, there are several aspects that could be improved since there is a lack of generic platforms to solve a broad set of problems. The existing mission planning solutions are not flexible enough, without much support for multi-drone collaborative missions and constraint definition. We conclude by proposing the implementation of a platform that focuses on those issues.

Chapter 3

Architecture

In this chapter we propose an architecture for a drone fleet mission planning framework. This architecture comprises a drone-side module that handles the interaction with the flight controller, and a ground-side module for remote control of the drones and mission execution. As a starting point, we present the requirements that the platform has to comply with. We propose a global architecture which is followed by the description of the internal components of both modules. We also cover the communication and interaction between modules.

3.1 Requirements

Before proposing an architecture, we need to have a set of well-defined requirements that we will fulfil with this new platform. We aim to build a mission planning framework that can be used for a wide range of goals instead of specific, limited sets of scenarios. The requirements that we will define in this section are directed towards providing generic functionality. Some scenarios might have specific requirements: for example, during a fire, the drone should move away from the fire and identify its perimeter; on an earthquake, the drones should position in an area to provide an autonomous communication platform near the rescue forces and the victims. Instead of embedding a particular behaviour in the global system architecture, we delegate that responsibility to the missions that require this behaviour, and only provide the base features necessary to support it, such as retrieving sensor readings that will give the source data to proceed with the specific mission.

3.1.1 Telemetry acquisition

Flight controllers incorporate on-board and external sensors, such as accelerometers, gyroscopes, magnetometers, barometers, and Global Positioning System (GPS) modules. These sensors allow the drone to stabilise itself in both manual and autonomous flight modes, but are of increased importance when seeking a fully autonomous flight.

While monitoring a UAV, whether under autonomous or manual flight, it is useful to retrieve sensor data such as velocity, acceleration, heading, and current GPS coordinates. The

FC also provides other useful information such as the current flight mode, an estimate of the remaining battery, and overall sensor health.

All this data should be retrievable in real time and be made available to both the user and the ground station software.

3.1.2 Autonomous navigation

One of the most crucial features that the platform has to enable is to control the drone without using a RC. The UAV must respond to commands such as arm, disarm, take off, land, move, and stop, which should then be executed autonomously. To sum up, the user should only need to request the required drones to perform a specific command without further input.

These commands also need to be versatile. For example, not only should it be possible to send the drone to a specific GPS coordinate, but also to instruct it to, e.g., move 10 meters forward, go 5 meters down, or rotate the heading 45° clockwise.

With the implementation of this layer of abstraction, the degree of mastery that is required to operate a UAV is lowered, since now the user only needs to understand a set of higher-level commands.

3.1.3 Sensor integration

We previously mentioned the available on-board sensors, which can be tightly integrated into the drone module, since they are part of the FC. However, external sensors can also be coupled to the UAV – those could be a camera, a temperature sensor, a radiation detector, or any other sensor that can interface with the drone hardware. These sensors may not be relevant in the context of most missions, but are vital for navigation decisions in specific scenarios. Therefore, interaction with these sensors needs to be supported to adjust the navigation according to the received data.

3.1.4 Multi-drone control

Many scenarios are only possible to accomplish by controlling multiple UAVs simultaneously. This architecture should be compatible with managing a fleet of several drones, monitoring each drone's telemetry and independently commanding the drones. In the context of mission planning, this is also a fundamental feature, since some missions will require the collaboration of various drones.

3.1.5 Mission support

Controlling a drone through higher-level commands (instead of direct, manual interaction) can improve the drone's autonomy, but it still requires the user to continuously input the desired actions. Sending each action individually is a repetitive task that should also be automated to accommodate complex tasks seamlessly.

In this context, the concept of a mission can be summed up as a sequence of commands that are executed one after the other. However, to only support a linear succession of commands

can be quite limiting – the framework should also provide the means to define certain trigger conditions that can lead the drone to separate execution paths. We can succinctly define this requirement as supporting the interpretation and execution of a sequence of commands that may be altered in response to an external factor. It should be possible to develop solutions for a wide range of specific use cases while using the same generic primitives, such as the previously described commands.

Since the ultimate goal of this Dissertation is precisely the mission planning framework, there are numerous requirements for complex, versatile mission support that have to be addressed. Such requirements will be thoroughly detailed in chapter 5.

3.2 Proposed architecture

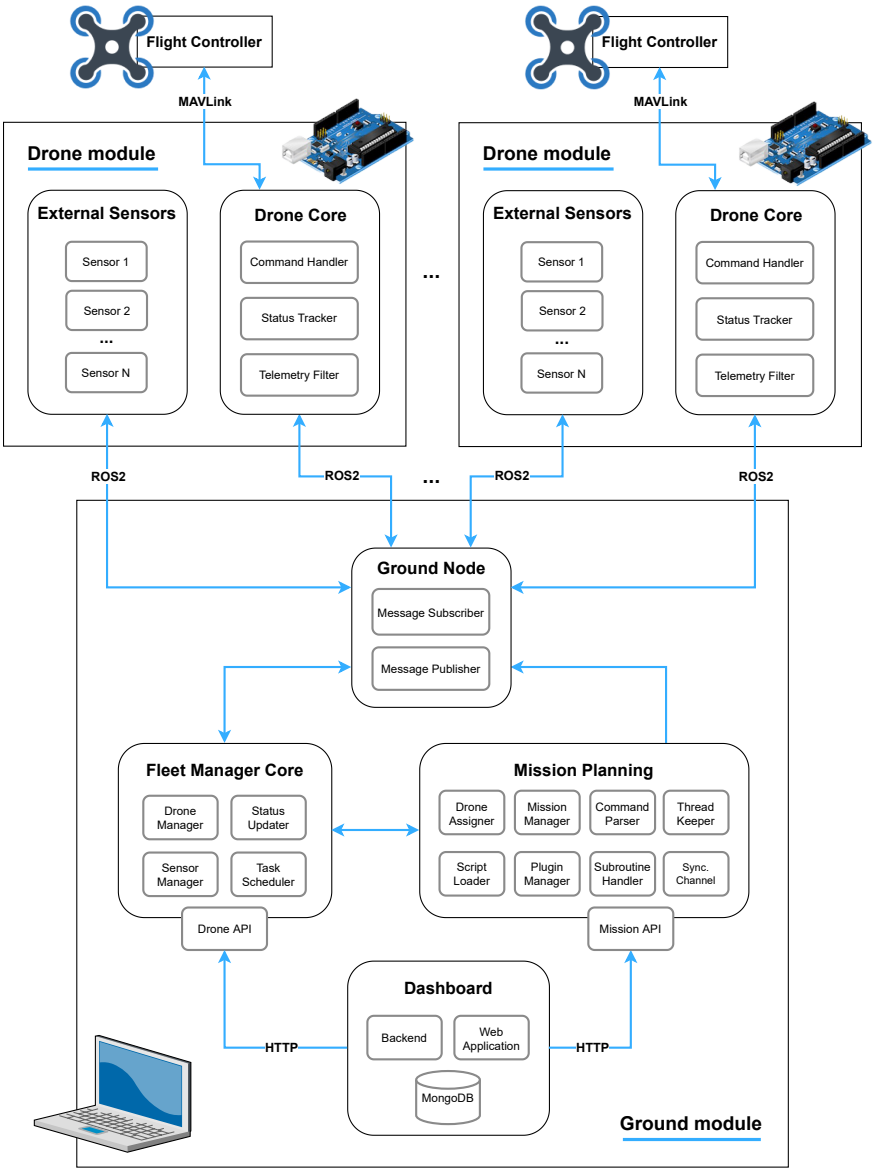


FIGURE 3.1. Global system architecture

To satisfy these requirements, we designed the architecture that can be seen in figure 3.1, in which we have two major components: the **drone module**, which will run in the drone's companion computer, and the **ground module**, which is launched in the ground station computer. Multiple UAVs can be connected in this platform.

The communication of both entities is possible through the publishing and subscription of ROS messages, which are briefly documented in section 3.3. In this architecture the drones do not exchange information with each other directly – this would require drone-to-drone wireless links, and is outside the scope of this Dissertation. As such, the responsibility of coordinating multiple drones is centralised in the ground station.

Each of the modules encloses several components, which are explained in detail later on, in chapter 4. These components satisfy the requirements proposed in section 3.1 and their main features are succinctly described.

3.2.1 Drone module

The UAVs are equipped with a flight controller and a companion computer. Both devices can communicate through the MAVLink protocol, which allows to receive telemetry data and send instructions to the autopilot. The drone module contains software that runs in the companion computer and serves as a bridge between the ground station and the flight controller.

Drone core

The drone core package contains the drone-side logic. By integrating with a MAVLink library, it is possible to retrieve the drone's telemetry data and execute several supported commands. This component interacts with external systems by employing a ROS node that publishes telemetry and status messages and subscribes to command messages.

The main components of this package are:

- **Command Handler** - Parses received commands and instructs the FC to execute them; it notifies the status tracker that a new command is starting or failed.
- **Status Tracker** - Informs the ground station of status changes; these can involve tracking when a command starts or finishes or other events such as the FC disconnecting.
- **Telemetry Filter** - Periodically fetches the current telemetry data and filters the useful parameters; the filtered data is then published according to the telemetry message format.

External sensors

External sensors can be connected to the companion computer using a wide array of hardware protocols, such as Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI) or even Universal Serial Bus (USB). The steps to retrieve data from a sensor and make that data usable

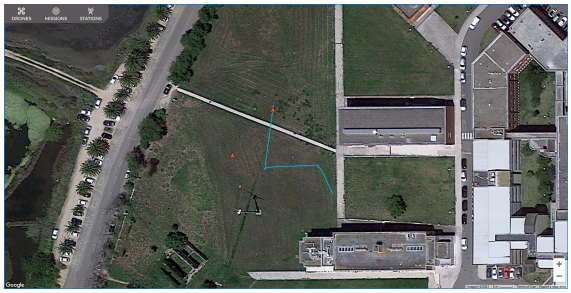
vary substantially from sensor to sensor. To address this, each type of sensor is connected to a ROS node that acts as an interface capable of publishing that sensor's data in a message format that is accepted by the ground station.

3.2.2 Ground module

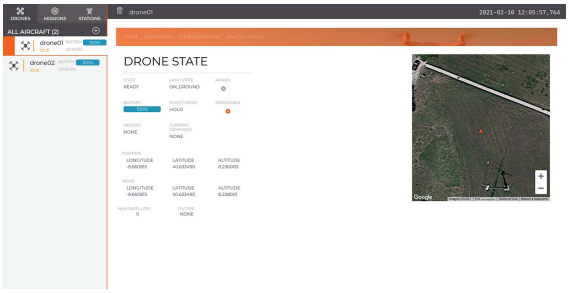
The ground station module is the central point for managing connected drones. It is responsible for providing endpoints where the user can retrieve the telemetry data, request that a drone executes a command, or send a mission to be performed.

Dashboard

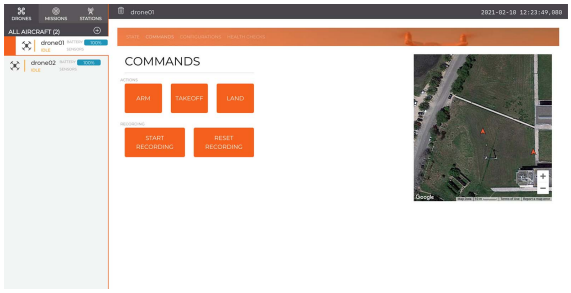
The author of this Dissertation did not develop the dashboard software, but it integrates the dashboard into the platform. This more intuitive GUI can be used to conveniently send commands or to consult the current drone status instead of having to manually issue Hypertext Transfer Protocol (HTTP) requests. The information displayed in the interface is updated by automatically sending HTTP requests to the ground station.



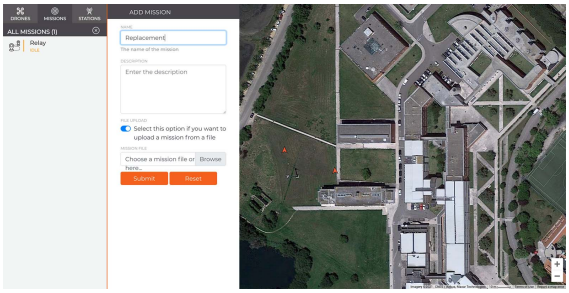
(A) Drone path



(B) Drone telemetry data



(C) Send a command to a drone



(D) Submit a mission script

FIGURE 3.2. Dashboard preview

Figure 3.2 shows a collection of screenshots of the dashboard. We can see the drone's current location in the map and the path it has traversed, examine telemetry data and send commands to the selected drone. It is also possible to submit a mission script file through this interface.

Ground node

To communicate with available UAVs, the ground station has a ROS node which owns the required publishers and subscribers. This component aims to provide central access to the ROS API for the remaining internal components.

Fleet manager core

All core functionality that enables the interaction with the drone is part of this package. The fleet manager core is compose of four main components:

- **Drone Manager** - Drone telemetry and data is stored here, which may be updated and retrieved by the other components. When a command request arrives, the drone manager will evaluate if the drone can perform it based on the current data. It also provides the endpoints for external applications to interact with the fleet: querying for drone data and sending commands.
- **Status Updater** - The main task of this component is to process telemetry and status messages and update the data stored in the drone manager. Consequently, this includes registering a new drone when one is detected. By inspecting the received messages, the status updater is also responsible for cancelling a mission if something goes wrong with one of the drones.
- **Sensor Manager** - Receives sensor messages, stores the latest sensor data, and notifies the drone manager when it detects a new sensor that is connected to a drone.
- **Task Scheduler** - A few scheduled tasks will run in the background to monitor possible timeouts. Crucially, this component is responsible for noticing when a drone exceeds a given threshold for the time elapsed since the last received telemetry message. Timeouts can also be enabled for each connected sensor, to attempt to detect sensor malfunctions.

Mission Planning

This component interprets and executes the missions that it receives through a REST endpoint. The mission functionality is divided among these components:

- **Drone Assigner** - Assigns drones to a mission according to the requested requirements, if any. When assigning multiple drones at once, the order in which they are allocated should be optimised by a priority. Similarly, it will also revoke a drone if that is solicited during a mission.
- **Script Loader** - Receives a mission file, notifies the mission manager of the new mission, and launches the script.
- **Mission Manager** - Registers starting missions, keeps track of the mission status and is responsible for requesting pending thread termination when a mission concludes with an error.

- **Plugin Manager** - Loads mission plugins and launches a plugin script when called in a mission.
- **Command Parser** - Parses and validates the sequences of commands present in a mission. It sends the command to be executed and waits until it is concluded before allowing the script to progress.
- **Subroutine Handler** - Allows the user to start a new subroutine from within a mission. It is also possible to consult if a subroutine is still running or wait until it is finished. Subroutines are essential for multi-drone scenarios since they enable the concurrent execution of multiple commands.
- **Thread Keeper** - Keeps track of the threads for running missions and manages their lifecycle. When a new subroutine is created, the thread keeper will launch a new thread to run it, and when a mission finishes, either successfully or with an error, it will stop all pending threads related to that mission.
- **Synchronisation Channel** - Central point for thread synchronisation. It provides a mechanism to halt execution until a command is concluded or a new telemetry message is received.

3.3 Module communication

As previously mentioned, both modules communicate by exchanging ROS messages. This way, the only requirement to build an application that can interact with either of the modules is a working ROS distribution, which is simpler than if a communication module had been custom-built. This decoupling is advantageous as it allows the implementation of one of the modules to be swapped without affecting other modules that expect its messages, as long as the message format itself remains unchanged. Since ROS usage is widespread in the robotics community, our ground station may be distributed independently of the drone module and combined with different UAV software (or vice versa, distributing the drone module instead).

This section further clarifies aspects of the module communication, first by declaring the supported message types, the properties they have to respect according to the requirements, and then by demonstrating the system's data flow.

3.3.1 Message types

We defined four types of messages that will be used to exchange data between the drone and ground modules: telemetry, command, status, and sensor messages. Each of these messages serves a specific purpose, and as such, they will carry different data; however, there are some characteristics that they will have in common.

Since at the time of this Dissertation, ROS2 did not natively support instantiating a single subscriber for multiple topics or subscribing using wildcards, our approach was to have a single topic for each message type regardless of the drone in question. This approach spares

us from the overhead and memory footprint of instantiating a new publisher/subscriber for each drone. However, this comes at a cost: we cannot distinguish which drone sent a given message. To solve this, we have to ensure that all messages include a field identifying the drone.

After executing a mission, we may want to analyse how this mission has performed. We will most likely need to merge information between telemetry, command, status, and sensor data. If we want to retrieve meaningful information from the data, we need to sort the various messages temporally. To allow this, we have to include a timestamp in each message.

Finally, these messages have to be categorised by type so they can be filtered. In ROS, this can be achieved by sending each message type to a particular topic, which ensures that the subscriber knows what kind of message structure it will receive.

In sum, all of these messages have to:

- Include the **drone identifier** to distinguish the drone that has sent the message.
- Include a **timestamp** to sort the messages by time.
- Be published to the **topic** of that message type to be correctly filtered.

Telemetry messages

Telemetry messages are used to share telemetry data with the ground station or other applications. The drone module will publish these messages periodically at a configurable frequency. Telemetry data is retrieved through the use of a MAVLink library, which provides a large number of parameters. Since some of those values may be of less interest to the user, MAVLink messages from the FC should be filtered so that only the most relevant are published to ROS.

Command messages

Command messages are sent by the ground station to a specific drone, requesting that it executes a command. Besides providing the command name in the message, some commands also accept additional parameters. Those can either be optional parameters (such as providing the takeoff height) or mandatory (like coordinates for a move command). These parameters must be included in the message, but can be omitted when a default value exists.

Status messages

The drone module sends status messages to notify the ground station of certain events. These can be either a command status change or a component notification, such as the flight controller disconnecting. Some commands may only result in a success/failure response, while others may send a "start" message and later conclude by being cancelled, failing, or finishing successfully. Other internal events relevant to share with the ground station will also be sent in a status message.

Sensor messages

Sensor modules must conform to a standard when publishing sensor messages, so that the ground station can interpret data from any sensor without having to know the specifics of how that sensor is interfaced. Besides the drone ID and the timestamp, the message should also include the sensor type (for example, temperature), and the value that the sensor has reported.

3.3.2 Data flow

Both modules, drone and ground station, communicate through ROS messages. The user or application will send HTTP requests to interact with the drones (either to send commands or to monitor telemetry data), and the drone module running in the SBC will communicate with the flight controller through MAVLink messages. Figure 3.3 summarises the message exchange, exemplifying a scenario in which there are two connected drones, one of which is equipped with a temperature sensor.

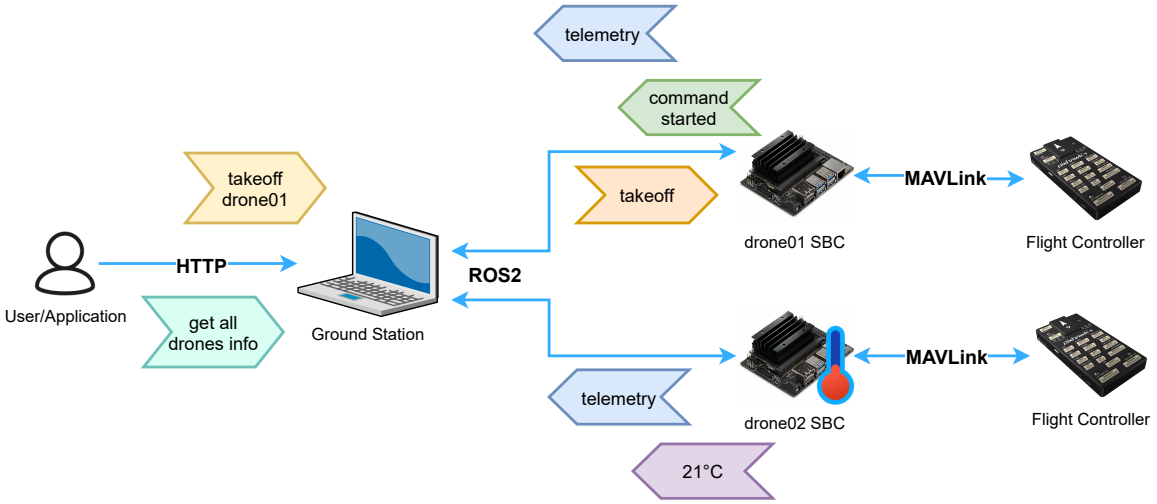


FIGURE 3.3. Data flow diagram

Drone telemetry can be monitored by requesting it to the ground station, which will return the most recent telemetry data. The drones continuously publish telemetry messages – remote update requests are not required.

When sending a request for a takeoff command, the ground station will direct it to the drones as a command message. As soon as the command is accepted, the drone module will send a status message, indicating it has started. After the drone concludes the command execution, it will also send a status message for the finished command.

In the presence of a sensor module such as a temperature sensor, a sensor message is also periodically published with the current temperature reading.

3.4 Discussion

This chapter started with the description of the platform's requirements. We will discuss how those are effectively addressed by the architecture we presented.

Telemetry acquisition is handled by the *Telemetry filter*, which is forwarded to the ground station through ROS2 messages. The user can monitor the telemetry data by sending the appropriate HTTP requests, or by consulting the dashboard.

The *Command Handler* from the drone module can communicate with the FC to execute a command. The ground station can also receive a command, validate the fields, and forward it to the drone module. This enables autonomous navigation to a certain extent since the drone can be controlled without using a RC.

We defined the format in which the sensor software has to publish sensor messages. If this format is respected, the ground station will be able to understand the sensor data and make it available for the user and the missions.

Multi-drone support is not the responsibility of any single component in particular, but each drone's SBC will be running a separate instance of the drone module. From the ground station, it is possible to filter telemetry data by each drone, and also submit a command to a specific one. Simultaneous control of multiple drones is delegated to the mission planning framework.

Mission support was not extensively explored in this chapter, but the architecture contains components that leverage the mission planning framework. This will be further detailed in chapter 5.

Overall, the architecture we propose is able to accomplish the presented requirements through the components that comprise it. We will move forward into implementing it.

3.5 Summary

This chapter proposed an architecture for a mission planning framework for fleets of aerial drones. Through this platform, a user can monitor, control, and effectively choreograph the fleet's behaviour. These functionalities are leveraged by two different modules: the drone module and the ground module. Both modules will communicate through ROS2 messages - sharing telemetry, status, sensor data and also command requests. The core modules will enable the drone to perform commands autonomously without being manually controlled by the user. The support for mission planning is also a major component in this architecture, and it enables the representation and execution of complex non-linear missions involving multiple UAVs.

Chapter 4

Implementation

In this chapter, we describe the implementation of the platform presented in the previous chapter. Although the final outcome of this Dissertation is the development of a mission planning framework, a reliable drone control platform needs to be implemented before progressing. The mission implementation will be addressed in chapter 5.

We will begin by documenting the drone module's implementation and the messages it exchanges with external components. Subsequently, we describe the ground station module implementation, excluding the mission planning features, which will be detailed in chapter 5. After presenting the software implementation, we introduce the simulation and test environment that is used to validate the platform's base features.

4.1 Drone module

Autonomous UAVs are usually equipped with a SBC, which are more computationally powerful than the flight controller. These computers travel with the vehicle and enable the communication with the FC, through software and tools developed for that purpose. Furthermore, we can use that software to remotely control the drone and retrieve the data produced by the FC, such as the GPS data, to make flight decisions based on those readings. Besides the communication with the FC, the software in the SBC is also responsible for providing an interface that enables external entities, such as a ground station (GS) module, to interact with the drone.

We developed a drone module that acquires telemetry data in real-time and instructs the autopilot to execute a command, doing so by exchanging MAVLink messages with the FC. External modules can interact with the UAV by sending ROS messages. While this was designed with our GS module in mind, there was also a secondary goal of building the module in a way that would allow it to be easily integrated with other ground control platforms. Since all the libraries we need for this module are available and well tested in C++, we decided to use it for the implementation. C++ is a highly performant language and, when used correctly, yields a low memory footprint, which is very important when writing software that will be executed in a SBC, as the hardware is usually not as powerful as

a personal computer.

4.1.1 Flight controller integration

As mentioned in chapter 3, we will use the MAVLink protocol to interact with the FC. To include it in our implementation, we could use the base MAVLink library for sending and receiving MAVLink messages. However, since it is only used to marshal and unmarshal MAVLink messages, this would require to develop a significant amount of code just for parsing and encoding the MAVLink packets, which is not ideal since higher-level solutions are already available.

One such library is MAVROS[34], a ROS package that enables MAVLink communication through ROS. This is not the most fitting solution since, at the time this work was developed, MAVROS did not support ROS2, and would have required a ROS2/ROS1 bridge that would have introduced avoidable complexity and overhead. Another disadvantage is that this would strongly couple our implementation with ROS, with the consquence that if we wished to swap ROS later on with another framework for communication, we would have to rewrite most of the drone control.

The solution that was selected for this implementation is MAVSDK, a MAVLink library that provides an API to programmatically interact with a drone's FC, allowing access to telemetry and UAV control[35]. This library is well-tested and provides bindings for multiple programming languages, such as C++, Python, Java, and Go. Autopilots can be connected with a Transmission Control Protocol (TCP), User Datagram Protocol (UDP), or serial connection. Overall, this library was deemed the most straightforward to use in comparison with the other options, while still being robust and lightweight. For these reasons, we opted to use MAVSDK in our implementation.

MAVSDK's functionalities are divided across multiple plugins, each providing a different set of features. At the core, these plugins work by receiving or sending the appropriate MAVLink messages[36], while we only need to invoke higher-level methods. To build our drone module, we imported three plugins: the **telemetry**, **action**, and **offboard** plugins.

Telemetry plugin

As the name suggests, the telemetry plugin provides vehicle telemetry and state information,[37], which we use for data acquisition. This API supplies a wide range of information, such as battery state, flight modes, sensor calibration, GPS position, and several other fields. The access to the telemetry data is provided through each available field's corresponding methods, and the retrieval can be done either synchronously or asynchronously.

[34]https://docs.px4.io/master/en/ros/mavros_installation.html.
[35]https://mavsdk.mavlink.io/develop/en/index.html.
[36]https://mavsdk.mavlink.io/develop/en/guide/general_usage.html.
[37]hhttps://mavsdk.mavlink.io/develop/en/guide/telemetry.html.

Action plugin

The action plugin can be used to command the UAVs[38]. It includes actions such as *arm*, *disarm*, *takeoff*, *land*, *return to home position and land*, or *go to* a location. Besides sending these commands to the FC, the plugin can also be used to set some default values, such as takeoff altitude, return to launch altitude, and maximum speed.

Offboard plugin

The offboard plugin is also used to command a UAV, but instead of sending discrete actions, it allows the control with position, attitude or motor commands[39]. For example, with this mode, the user could control the drone by providing the velocity in North-East-Down (NED) coordinates.

4.1.2 Drone core package

When we presented figure 3.1 in chapter 3, we referred to the component that controls the drone as *drone core*. This name was chosen because the package provides the "core" drone features for interacting with the drone. Within *drone core*, the MAVSDK library assures communication with the autopilot, and ROS2 is used for exchanging messages with the ground station, configuring the module's parameters, and logging.

Launch and configuration

By building the source code with *colcon*, the ROS2 build tool, an executable for this package will be created. Without further configurations, it is possible to launch the executable through a command-line interface (CLI) by running the command in example 1, in which `DRONE_ID` is the chosen drone identifier.

Example 1 Base drone module launch command

```
ros2 run fleetman_drone drone_controller -n DRONE_ID
```

The ROS node has a set of parameters that can be customized using the ROS parameter API. Following the ROS parameter API documentation[40], we could provide a configuration file when launching the application to customize the parameters. This is achieved by creating a file equivalent to example 2 and launching the module with the command in example 3.

In that command, `cfg.yml` has to be an adequately formatted ROS2 parameter file. It is worth noting that the `DRONE_ID` will also be the ROS2 node identifier. As such, for the program to work correctly, the provided `DRONE_ID` must match the node in the parameter file.

[38]https://mavsdk.mavlink.io/develop/en/guide/taking_off_landing.html.
[39]https://mavsdk.mavlink.io/develop/en/guide/offboard.html.
[40]https://index.ros.org/doc/ros2/Tutorials/Parameters/Understanding-ROS2-Parameters/.

Example 2 ROS2 parameter file

```
DRONE_ID:
    ros__parameters:
        param1: 3
        param2: 'foo'
```

Example 3 Drone module launch command with ROS parameter file

```
ros2 run fleetman_drone drone_controller -n DRONE_ID
--ros-args --params-file cfg.yml
```

The ROS2 suggested parameter loading approach is not intuitive for the user. In order to simplify this process, a bash script, `run_drone_controller.sh`, was developed. This approach of manually writing a bash script to launch a ROS node was chosen instead of the ROS2 launch system because the latter involves editing a Python script,[41] whenever a parameter or other setting needs to be changed. This approach has its own drawbacks however, since a less experienced user could unknowingly leave the launch script with syntactic errors. Besides this, the ROS2 launch API has been changing throughout ROS2 versions, which would be an additional consideration that would have to be taken into account when upgrading the ROS2 version.

This script takes two arguments, the `DRONE_ID` and optionally a YAML configuration file. It works by transforming the user-provided configuration into a compatible ROS parameter file. In example 4 we observe how to launch the application with a configuration file by using this script.

Example 4 Script to launch drone module

```
./run_drone_controller.sh DRONE_ID cfg.yml
```

If the user runs the script without providing a configuration file, the default parameters will be used instead. The user can also supply a configuration file that does not contain all of the parameters, in which case only the undefined parameters will take the default value. The example 5 demonstrates a valid configuration file that could be provided when launching the software.

[41] https://index.ros.org/doc/ros2/Tutorials/Launch-system/.

Example 5 Drone module configuration

```
port: serial:///dev/ttyACM0
telemetryTopic: /telem
commandTopic: /cmd
statusTopic: /status
telemetryRateMs: 1000
acceptanceRadius: 0.5
takeoffAltitude: 15.0
returnAltitude: 15.0
maxSpeed: 5.0
```

The implemented configurable parameters, their purpose and their default values are listed in table 4.1. A parameter worth clarifying is that the `port` can represent either a TCP, UDP, or a serial connection. The string format is `udp://[bind_host][:bind_port]` for a UDP connection, `tcp://[server_host][:server_port]` for a TCP connection, and `serial://[device]` for a serial connection,[42]. When using our platform with a simulated drone, we launch the drone module with the default simulator port, which is `udp://:14540`; when using the physical drones, we use the serial port as `serial:///dev/ttyACM0`.

TABLE 4.1. Drone configuration parameters

Parameter	Type	Default value	Description
port	string	udp://:14540	The protocol and port where it is possible to connect to the flight controller
telemetryTopic	string	/telem	Topic where the telemetry messages will be published
commandTopic	string	/cmd	Topic where the system will retrieve the drone commands from
statusTopic	string	/status	Topic where the system status and action completion will be published
telemetryRateMs	integer	1000	Rate at which telemetry messages will be published, in milliseconds
acceptanceRadius	double	0.5	Distance to a given waypoint, in meters, at which it is considered reached
takeoffAltitude	double	15.0	Relative takeoff altitude in meters above ground
returnAltitude	double	15.0	Relative minimum altitude the drone has to reach when in return to launch mode
maxSpeed	double	5.0	Default maximum speed when the commands do not specify one

[42]https://mavsdk.mavlink.io/develop/en/guide/connections.html.

Error handling

At runtime, the software may encounter some errors - whether it was impossible to find the flight controller on the specified port, one of the on-board sensors needs calibration, or a command failed to execute. When the system cannot connect to a flight controller at startup, the program will exit and print the error message after a few seconds. Other errors that may arise will be logged using the ROS2 logger, which can be seen in the terminal where the drone program was launched or on the respective `/rosout` topic of the ROS2 node. This allows us to examine these errors later on by recording the published messages. When a requested command fails, the ground station is also notified of the failure through status messages.

Message exchange and format

Each ROS node communicates with the ground station through a subscriber for reading commands, a publisher for telemetry data, and a publisher for status messages. These three message types have each their own message format, and both the drone and the ground modules must comply with it; otherwise, it is not possible to communicate.

The exchanged ROS2 messages are of the type `std_msgs/String`, structured in JSON format. The software uses the JsonCpp library,[43], to parse and create JSON strings in C++, since it is already an installed dependency from ROS. A few relevant aspects to consider are that: these strings are case sensitive and must be in lower case; the order in which the parameters are presented is irrelevant; and, as in any valid JSON, string values must be wrapped in quotes. As mentioned in chapter 3, the `timestamp` and `droneId` attributes have to be included in every message, and the latter must be the one defined at launch time.

Example 6 shows how the base ROS message should be formatted. The remaining parameters will be defined according to the message type.

Example 6 Base message

```
{
    "droneId": "drone01",
    "timestamp": 1609323920,
    "...": "...",
}
```

Utility methods

Throughout this module's development, it became apparent that some common operations had to be executed multiple times. To simplify this process, we developed some utility classes that contain these methods, which are mainly of two types: drone utilities and coordinate utilities.

[43] https://github.com/open-source-parsers/jsoncpp.

The drone utilities' class mainly comprises methods to convert states, commands, and telemetry from a string to an enumeration and vice versa. It also allows retrieving the current timestamp, which is needed for every published message. Additionally, it also contains a method that, provided a JSON node and the desired publisher, converts the JSON object to a string, creates a ROS message with the content set to that string, and publishes it in the right topic. These methods help mostly the serialization, but also the deserialization of some of the fields of the JSON messages.

The coordinate utilities provide the functionality related to coordinate calculations. The most fundamental operations include angle normalization and conversions between degrees and radians. Besides those, there are also more three methods to aid on coordinate operations:

- Given the current coordinates and the target coordinates, calculate the drone's bearing that allows it to be headed towards the new set of coordinates (equation 4.1);
- Given two sets of coordinates, calculate the distance between them using the haversine formula (equation 4.2);
- Given the current coordinates, the bearing, and a distance value, calculate the target coordinates (equation 4.3).

$$\arctan^2\left(\sin(\Delta lon)\cdot\cos(lat_2), \cos(lat_1)\cdot\sin(lat_2) - \sin(lat_1)\cdot\cos(lat_2)\cdot\cos(\Delta lon)\right) \tag{4.1}$$

$$2r\arcsin\left(\sqrt{\sin^2\left(\frac{lat_2 - lat_1}{2}\right) + \cos(lat_1)\cos(lat_2)\sin^2\left(\frac{lon_2 - lon_1}{2}\right)}\right) \tag{4.2}$$

$$\begin{aligned} lat_2 &= \arcsin\left(\sin(lat_1)\cdot\cos(dist/r) + \cos(lat_1)\cdot\sin(dist/r)\cdot\cos(bearing)\right) \\ lon_2 &= lon_1 + \arctan^2\left(\sin(bearing)\cdot\sin(dist/r)\cdot\cos(lat_1), \cos(dist/r) - \sin(lat_1)\cdot\sin(lat_2)\right) \end{aligned} \tag{4.3}$$

4.1.3 Command execution

The drone module subscribes to command messages sent by the ground station to the topic defined in the `commandTopic` parameter. These messages are parsed with JsonCpp, and the corresponding command will be executed through MAVSDK. If the command is unknown or additional parameters are missing, it reports an error instead. In example 7, we have the base command message. The `mode` attribute indicates whether the command is part of the `action` or `offboard` plugins of MAVSDK, or if it is a `custom` command. The `custom` mode refers to commands that are supported but were not implemented directly in MAVSDK. Additionally, some commands may accept more mandatory or optional parameters. Valid commands and their additional parameters are presented in this section.

Example 7 Base command message

```
{
    "droneId": "drone01",
    "mode": "control_mode",
    "cmd": "command",
    "timestamp": 1590512277423,
    "...": "..."
}
```

Action commands

There are six supported action commands, listed in table 4.2. If a `takeoff` or `return` command does not provide an altitude parameter, the default parameters configured at launch will be used instead. The altitude parameter is always in meters, but it is represented in a different referential depending on the command. In the context of a `goto` command, the altitude is in meters above mean sea level (AMSL), for a `takeoff` it is meters above ground level, and for a `return` it is meters above the altitude of the home position. The `yaw` angle is positive for clockwise direction while looking from above, with 0° being North. The drone must be armed before executing any of the other commands.

Table 4.2. Action commands

Command	Required parameters	Optional parameters	Description
arm	-	-	Arm the drone
disarm	-	-	Disarm the drone
takeoff	-	`alt` *(m)*	Fly to the takeoff altitude
land	-	-	Land in the current position
goto	`lat` *(°)* `lon` *(°)*	`alt` *(m)* `yaw` *(°)* `speed` *(m/s)*	Fly to the provided position
return	-	`alt` *(m)*	Move to launch position and land

In terms of behaviour, it is relevant to clarify the `return` command. The altitude provided to the return to launch command represents the minimum height relative to the takeoff position altitude that the drone has to achieve before moving to the launch position. For example, suppose that before taking off, the drone was at an altitude of 8 meters AMSL, and the `return` command is requested with an altitude of 10 meters while it is at 13 meters AMSL. In that case, the drone will first rise to 18 meters AMSL and then move to the launch position before landing. However, the lowest return altitude that is accepted is 10 meters. If the user requests less than ten meters, the drone will still rise 10 meters before moving to the launch position.

The MAVSDK API requires an altitude parameter with any `goto` command. To allow the user to request the drone to move to a location while maintaining the current altitude, we retrieve the current altitude value from the telemetry and send it instead. When the `yaw` parameter is omitted, it will also be replaced with the current heading. Another feature that we provide with the `goto` command is that we accept that the user sends the `yaw` parameter as a null value. In that case, we calculate the drone's bearing that allows it to head towards the target position, using the methods in the coordinate utilities.

If the user or an external application aims to send an `arm` command to the drone, the message could be formatted as in example 8. For the `disarm` and `land` commands, the message would follow the same pattern, just requiring the `cmd` field to be adjusted accordingly. If the `takeoff` and `return` commands are to be sent using the default altitude values, they can follow the previous format; otherwise, they also have to include the `alt` field as in example 9.

Example 8 Arm command message

```
{
    "droneId": "drone01",
    "mode": "action",
    "cmd": "arm",
    "timestamp": 1590512277423
}
```

Example 9 Takeoff command message

```
{
    "droneId": "drone01",
    "mode": "action",
    "cmd": "takeoff",
    "alt": 10,
    "timestamp": 1590512277423
}
```

Since the `goto` command requires multiple parameters, when providing a value for all of the fields, it would be similar to example 10. We note that the `alt` and `speed` fields could be omitted and that the `yaw` field could either be omitted or present a `double` value instead. After receiving this command, the drone will move at five meters per second to the given `lat, lon` coordinates, at an altitude of 15 meters AMSL and headed towards the destination.

Example 10 Goto command message

```
{
    "droneId": "drone01",
    "mode": "action",
    "cmd": "goto",
    "lat": 40.633667,
    "lon": -8.660522,
    "alt": 15,
    "yaw": null,
    "speed": 5,
    "timestamp": 1590512277423
}
```

Offboard commands

Table 4.3 lists the available offboard commands, as well as additional parameters and their units. The `start_offboard` and `stop_offboard` commands, as the names suggest, are single-purpose commands for starting and stopping the offboard mode. The remaining commands can all receive additional parameters, but none are mandatory, since by default they will assume a null value in that component. Unlike the `action` commands, these may not have a definite end condition. For example, when requesting a velocity body command as in example 11, the drone will indefinitely move in a circle, or in example 12, it will keep moving Northwest at 10m/s. In those examples, the drone will only stop moving if the user sends a `stop_offboard` command. According to MAVSDK, each offboard command had to be preceded by a `start_offboard` mode command. To avoid sending two commands, we start the offboard mode automatically when one of the remaining offboard commands is received.

Example 11 Velocity body command

```
{
    "droneId": "drone01",
    "mode": "offboard",
    "cmd": "velocity_body",
    "forward": 20,
    "yaw": 30,
    "timestamp": 1590512277423
}
```

Example 12 Velocity NED command

```
{
    "droneId": "drone01",
    "mode": "offboard",
    "cmd": "velocity_ned",
    "north": 10,
    "east": -10,
    "timestamp": 1590512277423
}
```

TABLE 4.3. Offboard commands

Command	**Parameters and units**	**Description**
start_offboard	-	Start offboard mode
stop_offboard	-	Stop offboard mode
position_ned	`north` *(m)* `east` *(m)* `down` *(m)* `yaw` *(°)*	Set position in NED coordinates and yaw
velocity_ned	`north` *(m/s)* `east` *(m/s)* `down` *(m/s)* `yaw` *(°)*	Set velocity in NED coordinates and yaw
velocity_body	`forward` *(m/s)* `right` *(m/s)* `down` *(m/s)* `yaw` *(°/s)*	Fly to the provided position
attitude	`roll` *(°)* `pitch` *(°)* `yaw` *(°)* `thrust` *(0 to 1)*	Set attitude in roll, pitch, yaw and thrust
attitude_rate	`roll` *(°/s)* `pitch` *(°/s)* `yaw` *(°/s)* `thrust` *(0 to 1)*	Set attitude rate in roll, pitch, yaw and thrust

Custom commands

With the previous messages, we provide direct access to existing MAVSDK commands. The offboard commands provide an advanced level of control, especially if we want to move the drone without knowing the absolute target coordinate. This thorough level of control comes at a price - those commands are not as easy to understand by a new user. There is also no obvious way to cancel a command that is executing, which means that if the user wanted to stop the current command, the only way would be through the RC or by sending another command. To address this, we also accept three new commands that can be interpreted by the drone module, which we list in table 4.4.

Table 4.4. Custom commands

Command	Required parameters	Optional parameters	Description
cancel	-	-	Cancel the current command
turn	`deg` *(°)*	-	Perform a clockwise yaw rotation
move	-	`x, y, z` *(m)*	Move along the x, y, z axis

To implement the `cancel` command, we execute a `start` offboard command followed by a `stop` offboard command. This causes the drone to stop the current command, whether it is an action or an offboard command. The `turn` command sums the provided angle value to the current bearing of the drone, which allows the user to rotate the drone orientation without having to provide an angle relative to North. If the drone currently is headed at 45°, after receiving a turn command of 45°, it will rotate and face East. Finally, the `move` command allows the user to request that the drone moves along the `x` axis (right), the y axis (forward), and/or the `z` axis (up). The calculation of the target position is done with the methods provided by the coordinate utilities. This representation may be easier for the user to visualize, since providing NED coordinates for a movement is not as trivial when the drone is facing a different direction.

Command priority

As already mentioned, without sending an `arm` command or manually arming the drone through the RC, most commands will be rejected. A `stop` offboard mode command will change the drone's flight mode, which in turn causes any running command to be cancelled, even if it is an `action` command. As a general rule, a command that is already executing will be cancelled if the user sends a new command. The exception to this rule is the `goto`, and consequently, the `move` command. These commands will only override the execution of another command of the same type and will be queued for execution if sent while the drone is taking off. Otherwise, if sent during a landing or returning to launch position, they will be rejected. If multiple `goto` commands are sent while the drone is taking off, the previous ones will be canceled and only the last one will execute after the takeoff.

Command completion

When a command is sent through MAVSDK, the only confirmation that is received is whether the command was successfully received or not. There is no direct method that we can call to verify if the command has finished the execution. We must let the user know that a command has been completed; as such, we have to resort to a polling strategy to verify it. This is not done by the command handler, but by the status tracker instead.

4.1.4 Status and command updates

The status tracker will notify external modules of events that happen in the drone module. These concern two distinct types of events: those related to tracking a command's status or detecting other system-related events. When a message reports a command's status, it will contain a `command` field to identify the command, as in example 13. Consider that the status concerns an event other than a command update. In that case, the triggering component will be identified in the `component` field instead, as in example 14.

Example 13 Base command status

```
{
    "droneId": "drone01",
    "command": "...",
    "state": "...",
    "timestamp": 1590512277423,
    "...": "..."
}
```

Example 14 Base system status

```
{
    "droneId": "drone01",
    "component": "...",
    "state": "...",
    "timestamp": 1590512277423,
    "...": "..."
}
```

Command completion

In table 4.5, we list all the states that can be present in a command status message, as well as the commands they may affect. In the "Commands" column, `action` refers to all `action` commands, except `arm` and `disarm`, and `offboard` refers to all `offboard` commands except `start_offboard` and `stop_offboard`, while `all` represents all existing commands. Since the `custom` commands are an interface for `action` or `offboard` commands, they are not depicted in the table. A `move` command is replaced by `goto` and `turn` is replaced by `velocity_ned`.

Table 4.5. Command status messages

State	Commands	Description
start	`action, offboard`	Started execution
stop	`offboard`, except `position_ned`	Offboard mode was stopped
cancel	`action, position_ned`	Command was cancelled
finish	`action, position_ned`	Finished execution
queued	`goto`	Queued until takeoff finishes
success	`arm, disarm, start` and `stop` offboard	Command succeeded
failure	all	Command failed to start execution

The `stop`, `cancel`, and `finish` commands may cause some confusion regarding their differences: because most of the `offboard` commands will execute infinitely, it does not make sense ever to consider them as finished. As such, for whichever reason may cause one of those commands to stop, the `state` of the message will be `stop`. A `cancel` command can be sent not because the user does not want that command to "finish", but because it is the way to stop it effectively. For the remaining commands, since they have a concrete end goal, it is clear that they have finished execution or that their execution was cancelled.

Example 15 Arm command successfully executed message

```
{
    "droneId":"drone01",
    "command":"arm",
    "state":"success",
    "timestamp":1590512277423
}
```

At the most fundamental form, the format of these messages is as in example 15, which informs that an `arm` executed successfully. In some circumstances, status messages may provide additional information so that the receiver better understands what the drone is currently doing. That is the case of a starting `goto` command, in which the target position is also provided, as seen in example 16. Similarly, we have in example 17 a `takeoff` that started and also stated the target altitude, which is also present when a `return` command starts. This behaviour is only present for these commands when they start, as there would be no additional value in providing them when they have finished.

Example 16 Starting go to command

```
{
    "droneId":"drone01",
    "command":"goto",
    "state":"start",
    "coords": {
        "alt" : 9.06500053,
        "lat" : 40.6338702,
        "lon" : -8.6602946
    },
    "timestamp":1590512277423
}
```

Example 17 Starting takeoff command

```
{
    "droneId":"drone01",
    "command":"takeoff",
    "state":"start",
    "altitude": 10,
    "timestamp":1590512277423
}
```

System status

Status messages also provide information on relevant events. Currently, the only `component` that is used is the "`system`" component, which represents local autopilot events. In the future, it could be used to notify of the sensor state, in which the `component` attribute would have the sensor ID or type. For the `system` component, the possible states are listed in table 4.6. As of now, none of these messages includes additional parameters, so they are all similar to example 18, with the only difference being the value of `state`.

Example 18 Flight controller connect message

```
{
    "droneId":"drone01",
    "component":"system",
    "state":"connect",
    "timestamp":1590512172217
}
```

TABLE 4.6. System status messages

State	Description
connect	Flight controller connected to port
disconnect	Flight controller disconnected from port
disarm	Drone was automatically disarmed
return	Drone started an automatic return to launch
stop_offboard	Offboard was automatically stopped
start_manual	Drone started being manually controlled
stop_manual	Drone stopped being manually controlled

The `connect` event will be triggered at the launch of the module and later on if the FC is reconnected. The `disconnect` event may happen if the cables connecting the SBC and the FC are accidentally moved or when the FC is reset. The `disarm`, `return`, an `stop_offboard` commands are notified through a system status message instead of a command status messages when these are executed without being requested by the user. The autopilot may

be configured using external applications to automatically disarm after landing or return to launch if no RC is detected. In those cases, the user will be warned of the behaviour with these status messages. If the drone module is running when the user starts controlling the drone with the RC, the system sends the `start_manual` message. If an arm command is sent through the platform after the drone was being controlled manually, the `stop_manual` message is sent instead.

4.1.5 Telemetry acquisition

Telemetry data is filtered from the MAVSDK library, as it contains more than twenty different fields, and not all of them are relevant in this scope. Internally, the drone module has access to all fields, but only those we chose are published to the external modules in the topic defined in the `telemTopic` parameter. In table 4.7, we have the transmitted telemetry fields. The formatted telemetry message is equivalent to that of example 19. Most of these fields are shown exactly as they are provided by the MAVSDK API, but some are slightly altered.

Table 4.7. Telemetry fields

Field	**Type/Values**	**Description**
armed	`boolean`	Whether the drone is armed or not
battery/remaining_percent battery/voltage	`float` `float`	Remaining battery, scaled from 0 to 1, and current voltage
flight_mode	`[unknown, ready, takeoff, hold, mission, return_to_launch, land, offboard, follow_me, manual, altctl, posctl, acro, stabilized, rattitude]`	Current autopilot flight mode
gps_info/satellites gps_info/fixType	`integer` `[no_gps, no_fix, fix_2d, fix_3d, fix_dgps, rtk_float, rtk_fixed]`	GPS fix type and number of satellites in use
heading	`float`	Yaw angle in degrees
healthFail	`[gyrometer, accelerometer, magnetometer, level, levelPos, globalPos, homePos]`	List of the current health failures (may be empty)

home/alt
home/lat
home/lon
	`float`
	`double`
	`double`
		Position from drone takeoff

landed_state	`[unknown, on_ground, in_air, taking_off, landing]`	Current landing state of the drone

position/alt
position/lat
position/lon
	`float`
	`double`
	`double`
		Current position

speed	float	Speed among all vectors

The telemetry data provides the attitude of the UAV, including the pitch, roll, and yaw angles. The value that we publish as the `heading` is the yaw angle. MAVSDK provides the health failures as separate fields, each associated with a `boolean` value indicating whether there is a problem. Most of the times all health checks will be fine, to avoid publishing a series of values that are constantly `false`, we group all of those fields into a list of failures, which will be empty when all checks have passed. Otherwise, it will contain the current failures. We can retrieve the current velocity of each NED coordinate from the telemetry data. When monitoring the drone's performance, it is not particularly relevant to see the velocity separated by direction in most circumstances. As such, we calculate the speed, including all of the components and publish it in the `speed` field.

Example 19 Telemetry message

```
{
    "droneId":"drone01",
    "armed":false,
    "battery":{
        "remaining_percent":0.98,
        "voltage":12.15000057},
    "flight_mode":"hold",
    "gpsInfo":{
        "fixType":"fix_dgps",
        "satellites":16},
    "heading":-0.2981671095,
    "healthFail":["accelerometer"],
    "home":{
        "alt":487.9980164,
        "lat":47.3979674,
        "lon":8.5431335},
    "landed_state":"on_ground",
    "position":{
        "alt":9.0650005340576172,
        "lat":40.633870299999998,
        "lon":-8.6602946000000003 },
    "speed":0.02999999933,
    "timestamp":1586998439212
}
```

4.1.6 Sensors

Multiple sensors may be connected to the SBC. Due to the heterogeneity that those sensors may present, the platform should not be responsible for integrating each of them. Instead, each of these sensors should have software running in the SBC capable of retrieving the current readings and publishing them in a format that is understood by the ground station. In example 20 we observe the base sensor message. The `type` identifies the sensor type, and the `value` identifies the current reading. This might not fit every type of sensor; as such, this may be reviewed in the future.

Example 20 Base sensor message

```
{
    "droneId":"drone01",
    "type":"...",
    "value":"...",
    "timestamp":1590512172217
}
```

Temperature sensor

In the context of this Dissertation, we did not use any physical sensor connected to the SBC. To allow the testing of sensor features in the future, we developed a simulation of a temperature sensor that complied with the message format.

The temperature simulator reads a configuration file with the properties of the environment, which will determine how the algorithm will calculate that temperature value, and its recording at the current GPS coordinates. In this context, the current coordinates are those published by the UAV where this sensor would be attached. Example 21 demonstrates how we could configure the simulator with a a simulated fire radiating in a round shape from a single origin.

Example 21 Configuration of a simulated temperature sensor with one heat source

```
droneId: drone01
telemTopic: /telem
sensorTopic: /sensors
rate: 200
environmentTemp: 20
heatSources:
  - coords:
      lat: 40.633400
      lon: -8.660815
    radius: 50
    temp: 60
```

The `droneId` field indicates from which drone's location the sensor should calculate the current temperature, and `telemTopic` indicates the topic where telemetry is published. The temperature values are published in the topic defined in `sensorTopic` at the rate specified in the `rate` field, in milliseconds. One or more `heatSources` can be defined, in which we can specify the coordinates of the source, the radius in which it affects the temperature, and the temperature at the centre. When the drone is outside the radius of any heat source, the temperature will be that of `environmentTemp`.

When we have a single heat source, the temperature at the core will be the temperature defined in the configuration. As we move further away from it, it will decrease linearly until the distance is equal to the radius; from that distance on, it will present the environment temperature. When the configuration provides multiple heat sources, the current location's temperature will be calculated for each of them, and the final result will be the highest of them. This method is not an accurate model of an actual forest fire, but since this is not the main focus of this work, it serves as an adequate approximation.

4.1.7 Missions

At the moment, the drone module does not contain any component responsible for the mission management. The ground side fully handles the mission control. Since this platform has been built from the start, implementing distributed mission control would increase the complexity. Another positive aspect of using ROS, specifically ROS2, is that node discovery is an automatic procedure. If we were to decentralize the mission control, we would not need to implement a new mechanism to allow drones to communicate with each other, because they can already communicate. This will be a topic for future work.

4.2 Ground station module

The ground station is the entry point from which the user interacts with the drone modules. This can be done through HTTP requests to the provided endpoints. It is possible to send commands to a specific drone, send mission scripts, and monitor drone data. Since the last section covered the messages exchanged between the drone and the ground station thoroughly, those will not be detailed.

4.2.1 The Spring framework

The Spring framework[44] is an application framework for Java. Spring reduces the development time of an application by providing dependency injection and supporting Java Database Connectivity (JDBC), along with other features. However, Spring required a lot of configurations to be made, which lead to the creation of Spring Boot[45]. In Spring Boot, the application can be autoconfigured, reducing the amount of work needed to get a simple application running. Another useful feature is that it includes an embedded web server.

By using Spring Boot, building the skeleton of the application was a trivial task. Without much effort, it was possible to implement our representational state transfer (REST) API.

4.2.2 Configuration

Like the drone module, the ground station can also receive a configuration file. The configuration file is a `properties` file that extends the Spring Boot configuration, so it can be customized to edit Spring specific parameters. This may be useful for changing the server address and port, or configuring a different database. Example 22 contains a possible configuration of the properties that are specifically related to the ground station. Not all of the properties need to be provided; in their absence, the ground station will use the default value instead.

[44] https://spring.io/.
[45] https://spring.io/projects/spring-boot.

Example 22 Ground station configuration

```
fleetman.ros2.node=ground
fleetman.ros2.topic.cmd=cmd
fleetman.ros2.topic.telem=telem
fleetman.ros2.topic.status=status
fleetman.ros2.topic.sensor=sensors
fleetman.drone.batteryLevel.critical=0.15
fleetman.drone.batteryLevel.low=0.30
fleetman.drone.timeout=15000
fleetman.sensor.temperature.timeout=10000
fleetman.mission.error.battery.return=false
```

Table 4.8 describes these properties and their default values. Although we only mention the temperature sensor, if other sensor types were available, it would be possible to configure those sensors' timeouts. The implications of each of those properties are explained in the appropriate sections.

Table 4.8. Configurable ground station properties

Property	Default value	Description
fleetman.ros2.node	ground	ROS2 node name
fleetman.ros2.topic.cmd	cmd	Topic to publish commands
fleetman.ros2.topic.telem	telem	Topic to retrieve telemetry
fleetman.ros2.topic.sensor	sensors	Topic to retrieve sensor data
fleetman.ros2.topic.status	status	Topic to retrieve status messages
fleetman.drone.batteryLevel.critical	0.05	Battery level at which the drone cannot be used
fleetman.drone.batteryLevel.low	0.15	Battery level at which the user is warned of low battery
fleetman.drone.timeout	5000	Milliseconds since last received message for a drone timeout
fleetman.sensor.temperature.timeout	0	Milliseconds since last message for temperature sensor timeout
fleetman.mission.error.battery.return	true	Whether the drone returns after reaching critical battery level during a mission

4.2.3 ROS2 message handling

In order to interact with the drone module, the ground station must have a ROS publisher to send commands and two ROS subscribers, one to retrieve telemetry data and another one for status messages. The GroundNode class, which represents a ROS2 node, was created as a Spring Component, meaning that it will be instantiated automatically at the application startup. This class was also configurable to be runnable, which means that it can provide a method to be executed in a separate thread. In the run method, we will instruct RCLJava to *spin* this node, which is required to receive messages. Since this is running in a separate thread, it will not block other parts of the program's execution. Publishers and subscribers can be attached to this node. Two additional classes, MessagePublisher and MessageSubscriber, were created for that purpose.

Message Publisher

The message publisher only contains one publisher, the command publisher. It exposes a single method, `publishCommand`, which will take an already JSON formatted command and send it as a ROS message. Commands are sent when the ground station receives an HTTP request to send a command or during mission execution.

Message Subscriber

Two subscribers are included in the message subscriber: the telemetry data subscriber, and the status message subscriber. When a telemetry message arrives, it is converted to a Telemetry object and forwarded to the Status Updater, which will handle the data.

If it is a status message instead, the callback will first distinguish what kind of event the message represents before calling the appropriate Status Updater method. Some messages will only be logged in the ground station, such as when an `arm` command succeeds, while others will trigger an update to the drone state, like when a `go to` command starts.

4.2.4 Drone Manager

The drone manager provides access to two repositories storing data for all connected drones, which can be retrieved or updated by other components. One repository contains telemetry data, kept as the drone sends it without any modifications. The other one contains the drone state data, a more compact version of the telemetry data combined with information from the status messages.

Drone state data

Although the telemetry messages that the ground station receives already contain a subset of the telemetry data, not all fields have equal relevance at runtime. As such, we also provide drone data that results from combining telemetry and status messages for more streamlined drone data access. These are the drone data fields:

- `state` - drone's state
  - `unknown` - the drone has timed out
  - `error` - the drone has errors
  - `manual` - the drone is in manual mode
  - `active` - the drone is executing a command
  - `hold` - the drone is hovering on air
  - `ready` - the drone is landed and ready to be commanded
- `currentCommand` - the command the drone is executing
  - If the drone is taking off, the target altitude is shown
  - If the drone is moving, the target coordinates are shown
- `errors` - list of errors that are currently affecting the drone
  - Critical battery level
  - Failed health check
- `warns` - list of warnings that are currently affecting the drone
  - Low battery level
  - Flight controller disconnected
- `sensors` - list of detected sensors attached to this drone
- `missionId` - the mission this drone is currently assigned to

Retrieving and updating data

The drone manager provides methods for external components to retrieve drone data - whether it is to respond to an HTTP request, to check if the drone has timed out or to be used during a mission. It also supports requesting a single drone's data or multiple drones that match the parameters, such as current state, or available sensors. The drone manager also updates the drone data, which is most often requested by the status updater when it receives new telemetry data. This update can also be triggered by other events: when there is a new sensor attached to the drone, when the drone is revoked from a mission, or when the drone has timed out.

Sending commands

Although the drone manager does not communicate with the message publisher to submit a command, it is consulted when a command request is received. By verifying the drone's state, it can reply if the command is accepted. It can reject a command for several reasons:

- The drone is in error state
- The drone is being manually controlled
- The drone has timed out
- The drone is assigned to a mission, but the command does not come from the mission

As a failsafe, the `return` command can always be executed, even if one of the aforementioned conditions is verified. This ensures that the drone can always return home safely when possible.

4.2.5 Status updater

The status updater processes telemetry and status messages and updates the drone manager's data according to what was received. This represents an essential part of the core functionality of the ground station module.

Registering new drone

Since drones will automatically send telemetry updates as soon as the drone module is running, it is simple to detect a new drone. When examining telemetry data, the status updater will check if the drone is already present in the drone manager. If not, it will be registered and an event will be logged.

Updating drone state and telemetry

When a new telemetry message arrives, the status updater immediately updates the drone manager's telemetry data. After this, it will compare the current telemetry with the current drone data to decide if this last one has to be updated. For example, if the telemetry presents a failed health check but the drone data does not list any errors, then it will be updated. The same analysis will be made regarding battery level and warnings. The drone state is also updated according to the current telemetry data. If the drone was in an `unknown` state before receiving the message, we assume it was able to re-establish the connection; this event is then logged. Another situation that may occur is when the drone enters into an error or manual state while being on a mission, in which case the status updater will request the mission manager to terminate the mission.

Command tracking

Status messages mainly serve the purpose of tracking when a command starts, stops, fails or is canceled, but are also relevant to log drone events that have no other consequence on the drone status. For example, when the manual mode begins or ends, the flight controller is connected or disconnected, or the drone was automatically disarmed, and the ground station will log that event. If a command fails while the drone was in a mission, this mission is terminated.

4.2.6 Sensor handling

Sensor handling is currently simplified since it can become a complex task, which was out of this Dissertation's scope. The enabled sensor types are `temperature` and `camera`. Currently, it only receives the sensor data that is published to the `sensors` topic and stores it locally. This data can then be retrieved by the user or during a mission. We consider that a drone has a certain sensor type attached after receiving one sensor message from it.

Although we only support sensors that are associated with a drone at the time, this could be easily adapted to integrate field sensors. The drone ID would have to be omitted, but a new parameter should be added to identify each sensor, so that multiple sensors of the same type could be used simultaneously.

4.2.7 Scheduled tasks

Spring Boot allows to create tasks that are executed in fixed time intervals. In our case, this is useful to monitor timeouts, but this feature could also be used for other monitoring activities.

Drone monitor

The drone module will send periodic telemetry messages that will reach the ground station without significant delay under normal circumstances. If a drone crashes or loses connectivity, telemetry messages may no longer be received. The drone monitor will regularly go through each connected drone's telemetry and compare the last telemetry message's timestamp with the current timestamp. If the time difference between both timestamps exceeds the one defined on the `fleetman.drone.timeout` parameter on the configuration file, the drone is considered in an `unknown` state.

Sensor monitor

For sensor monitoring, the timeout threshold has to be configured by sensor type (for example, under `fleetman.sensor.temperature.timeout`). By default, the timeout is set to 0, which results in never alerting that the sensor has timed out. This behaviour was implemented because it may not make sense for a specific sensor to timeout, as it may send messages sporadically. In case the sensor timeouts, it will be removed from the drone's list of available sensors, which may result in a mission being cancelled if it was relying on that sensor.

4.2.8 Missions

Some details regarding the missions have already been mentioned when relevant to explain part of the component's behaviour. However, since that is the block with higher complexity and the main focus of this Dissertation, it will be fully explored in chapter 5.

4.2.9 Logging

The Logback framework[46] is used to handle the logging. Several events will be logged at runtime, such as the drone's status updates or mission status. If a terminal is attached, it is possible to read these logs as they are produced. However, the logs will also be automatically saved to a file. Since we are using ROS, we also use the `rosbag` tool, which can be executed in the GS computer to record published messages. This is useful to inspect telemetry data at a later occasion while crossing it with the ground station logs.

4.2.10 API Endpoints

The platform provides four HTTP endpoints that are related to the drones. This is the interface through which the user and the dashboard can interact with the drones.

TABLE 4.9. /drone GET endpoint

/drone GET		
Description	List the requested data from all discoverable drones that match the filter	
Response	200 - return the requested drone data *(json)*	
Request parameter	**Parameter value**	**Description**
state *(query)*	string - `unknown, error, manual, active, hold, ready`	Filter by drone state
sensors *(query)*	[string] - `camera, temperature`	Filter by connected sensors
data *(query)*	string - `telem, info, sensors`	Filter the returned data; return all if absent

Table 4.9 contains the details of the endpoint through which we can retrieve the drone data. If the user does not want to retrieve the data from all drones, it is possible to filter it by the current drone state or attached sensors. When no `data` parameter is provided, it will return all data for the drones - telemetry, filtered drone data, and the latest sensor data. If we have the ground station running locally at port 8001, a request such as `localhost:8001/drone?sensors=temperature` will have a response similar to example 23.

[46] http://logback.qos.ch/.

Example 23 Response to a GET request on the /drone endpoint


```
{"info": {
    "droneId": "drone01",
    "state": "active",
    "currentCommand": {
      "description": "Takeoff with
                          altitude 3.0m",
      "timestamp": "2021-01-23 16:46:12,070"
    },
    "sensors": ["temperature"],
    "assignableToMission": true,
    "mission": null,
    "errors": [], "warns": []
  },
  "telem": {
    "droneId": "drone01",
    "armed": false,
    "position": {
      "lat": 40.6334901,
      "lon": -8.660815,
      "alt": 10.3220005},
    "heading": -4.566989899,
    "speed": 0.02,
    "home": {
      "lat": 40.63349,
      "lon": -8.6608147,
      "alt": 8.196},
    "flightMode": "takeoff",
    "landState": "taking_off",
    "battery": {
      "voltage": 12.150001,
      "percentage": 1.0},
    "gpsInfo": {
      "satellites": 10,
      "fixType": "fix_3d"},
    "healthFailures": [],
    "timestamp": "2021-01-23 16:46:16,509"
  },
  "sensors": [{
    "type": "temperature",
    "timestamp": "2021-01-23 16:46:16,572",
    "value": 51.985058
  }]}
```


The endpoint described in table 4.10 is similar to the previous one, with the difference that it is already targeted to a specific drone. The `data` parameter can also be used to filter the response data. If the user sends a request with a `droneId` that is not available, the ground station will reply with a 404 error message.

Table 4.10. /drone/droneId GET endpoint

/drone/{droneId} GET		
Description	List the requested data from this drone in particular	
Response	200 - return the requested drone data *(json)* 404 - no drone with the given `droneId` found	
Request parameter	**Parameter value**	**Description**
droneId *(path)*	string	The drone ID
data *(query)*	string - `telem, info, sensors`	Filter the returned data; return all if absent

To send a command to a drone, the user has to send a request to the endpoint described in table 4.11. The command has to be included in the request body, formatted in JSON, similarly to what was demonstrated in section 4.1.3. Looking at an example, such as example 10, the

difference is that it does not have to provide the `droneId` field, since it is already present in the request path, and the `timestamp`, which is concatenated by the ground station. If the request body is not properly formatted or refers to a command that does not exist, the ground station will reply with a 400 error message. When the given `droneId` is not registered, the message will be a 404 instead. As it was previously explained, in some situations the drone may be unable to perform the command, such as when it is in error state, in manual mode, or during a mission, which will lead to a 409 response. Finally, if all conditions are met, the ground station will answer with a 200 message to indicate that the command was successfully submitted to the drone, although it does not guarantee that the command was successful.

TABLE 4.11. /drone/droneId/cmd POST endpoint

/drone/{droneId}/cmd POST		
Description	Send a command to this drone	
Response	200 - command successfully submitted 400 - malformed command in request body 404 - no drone with the given `droneId` found 409 - the drone can not currently execute this command	
Request parameter	**Parameter value**	**Description**
droneId *(path)*	string	The drone ID
(body)	json	The command, properly formatted in JSON

The user can read the ground station logs for the drones with the endpoint in table 4.12. To filter the returned logs, the user can either provide an array of `droneId` or of `loglevel`. The response of a request to this endpoint is the matching log messages as plain text.

TABLE 4.12. /drone/logs GET endpoint

/drone/logs GET		
Description	List the logged messages that match the request	
Response	200 - return the requested drone logs *(plain text)*	
Request parameter	**Parameter value**	**Description**
droneId *(query)*	[string]	Filter the logs by `droneId`; return all if absent
loglevel *(query)*	[string] - info, warn, error	Filter the logs by logging level; return all if absent

In example 24, we have a possible response to this request, without using any filtering parameter.

Example 24 Response to a GET request on the /drone/logs endpoint

```
2021-01-24 17:04:15,369 INFO drone01 - Registered
2021-01-24 17:04:15,532 ERROR drone01 - Failed health check
2021-01-24 17:04:15,534 INFO drone01 - Established connection with ground station
2021-01-24 17:04:15,109 INFO drone01 - Start manual mode
2021-01-24 17:04:15,109 INFO drone01 - Flight controller system connected
2021-01-24 17:04:15,940 INFO drone02 - Registered
2021-01-24 17:04:15,948 ERROR drone02 - Failed health check
2021-01-24 17:04:15,948 INFO drone02 - Established connection with ground station
2021-01-24 17:04:15,927 INFO drone02 - Start manual mode
2021-01-24 17:04:15,927 INFO drone02 - Flight controller system connected
2021-01-24 17:04:30,741 INFO drone02 - Stop manual mode
2021-01-24 17:04:30,927 INFO drone01 - Stop manual mode
2021-01-24 17:04:32,419 INFO drone01 - Detected temperature sensor
2021-01-24 17:05:24,925 INFO drone01 - Success on arm
2021-01-24 17:05:27,193 INFO drone01 - Start takeoff with altitude 10.0m
2021-01-24 17:05:34,383 INFO drone01 - Queued go to 40.634421, -8.660437 at 15.0m
2021-01-24 17:05:37,885 INFO drone01 - Finish takeoff
2021-01-24 17:05:37,885 INFO drone01 - Start go to 40.634421, -8.660437 at 15.0m
2021-01-24 17:06:04,525 INFO drone01 - Finish go to 40.634421, -8.660437 at 15.0m
```

4.3 Simulation, test environment and validation

It is important to have a simulation pipeline to avoid damaging the UAVs while the platform is not stable before validation. Since it is composed of several modules, each with their own dependencies, there should also be an effective and simple method to launch these components. In this section, we describe how we prepared our test environment.

4.3.1 Docker images

Docker is an open platform that allows us to run and distribute our applications efficiently[47]. With Docker, we can package and run an application in a relatively isolated environment, which is called a container. With this isolation, we are able to run multiple containers simultaneously on the same host machine. A container can be defined as a unit of software that packages code, dependencies, and configurations[48]. By using a container, we can be sure that the application will run reliably across different computing environments. Docker is available across operating systems, distributions, and architectures, and the containerised software will run regardless of the infrastructure. A Docker container image is a software package that includes everything needed to run our application, such as the code itself, system tools, system libraries and settings. These container images will become containers at runtime. We can share a Docker image by creating a `dockerfile`, which is a file that includes

[47]https://docs.docker.com/get-started/overview/.
[48]https://www.docker.com/resources/what-container.

the steps required to build the image. With that file, other users can successfully build the Docker image in their system.

Using Docker, we can package two different images - the drone image and the ground station image. We include the ROS2 distribution, the MAVSDK library, their respective dependencies, and the drone module's source code in the drone image. The image has the drone code already built and also two example configuration files that can be used. For the ground station image, we install the ROS2 distribution, the ROS Client Library for Java and Spring Boot, and build the ground station module. This image has also a configured Network Time Protocol (NTP) server, so that the SBCs can use the ground station to synchronise their clocks in a real scenario.

4.3.2 Launch scripts

Docker has simplified the distribution of a ready-to-use environment; however, the image requires configuring some parameters to be launched, which may not be trivial for an unexperienced user. To solve this, we also developed two scripts for launching these Docker images: `launch_drone_container.sh` and `launch_ground_container.sh`. Examples 25 and 26 demonstrate a possible way to run those scripts. The CLI arguments to launch the drone module are listed in table 4.13 and the ground station in table 4.14.

Example 25 Drone launch script

```
./launch_drone_container.sh drone01 -c drone_cfg_serial.yml
```

Example 26 Ground launch script

```
./launch_ground_container.sh -c ground.properties -b gs/bags -l gs/logs
```

TABLE 4.13. Drone image launch script arguments

Argument	Argument type	Description
<drone-id>	positional *(string)*	This drone's ID
-c, --config	optional *(file path)*	Drone module configuration file
-s, --serial --no-serial	optional *(boolean)*	True if the connection to the FC is through a serial port. Ignored if a configuration file is provided.
-h, --help	optional	Print the help message

When the user provides a configuration file, it has to be formatted like the one previously presented in example 5. The `serial` argument is used when the user does not provide a configuration file, but wants to use the default configuration with a FC connected through a serial port.

Table 4.14. Ground station image launch script arguments

Argument	Argument type	Description
`-b, --bags`	optional *(dir path)*	Directory to save recorded rosbags
`-c, --config`	optional *(file path)*	Ground station properties file
`-l, --logs`	optional *(dir path)*	Directory to save ground station logs
`-h, --help`	optional	Print the help message

The ground station image script can be launched without any additional arguments but will use the default configurations, and it will not record any data while it is running. The properties file has to match what was demonstrated in example 22. If the `bags` argument is provided, the `rosbag` tool will also be launched and record all ROS2 messages, which will be saved in that directory. In the presence of the `logs` flag, the ground station and drone logs will be written to disk as well.

4.3.3 UAV Simulation

UAV simulators allow to use the PX4 autopilot code to control a simulated vehicle, with which we can interact with as if it were a real drone[49]. The PX4 software supports Software In the Loop (SITL) and Hardware In The Loop (HITL) simulation. HITL simulation requires a physical FC board to run the simulation, which was not always available during the development of this Dissertation. As such, we opted for a SITL simulator.

There are multiple SITL simulators that are compatible with the PX4 flight stack. However, we require simulating either quadcopter or hexacopter vehicles and support for multi-vehicle simulation. After analysing the possibilities, we considered two options - jMAVSim[50] and Gazebo[51]. Although Gazebo is the one that provides the most features, it also makes it heavier to run on the host machine. This led us to choose jMAVSim instead, as it is lightweight, provides the essential features, and was simple to install and configure[52]. However, Gazebo would have been a better choice if we also wanted to test obstacle avoidance and computer

[49]https://docs.px4.io/master/en/simulation/.
[50]https://docs.px4.io/master/en/simulation/jmavsim.html.
[51]http://gazebosim.org/.
[52]https://docs.px4.io/master/en/simulation/jmavsim.html.

vision algorithms[53].

With jMAVSim, we can simulate one ore more quadcopter UAVs. If we configure the drone module to connect to a device on the UDP port, we can control the simulated vehicle. At launch, jMAVSim can be configured to start the simulation at a given coordinate, which is useful if we want to foresee if our missions could inadvertently cross a tree or a building. By default, jMAVSim launches a GUI where we can see the simulated drone move and change a few settings such as wind speed, but it can also be launched in headless mode if we are not interested on those settings.

4.3.4 Simulation launcher

When launching jMAVSim for more than one vehicle, we also have to launch an equal number of drone module instances with the corresponding port configured. This task becomes time-consuming when we are testing a scenario with multiple UAVs in which we restart the simulator and the drone modules often. To increase productivity and the ease of simulator launching, we developed an application responsible for that management. The application has to launch a docker image with the simulator, configure the simulation parameters, and launch the drone modules with the drone configurations. Docker provides an API to interact with the Docker daemon, called the Docker Engine API, and software development kits (SDKs) for Go and Python[54]. With the SDKs, we can programmatically build and launch Docker images. We chose the Go programming language for developing this tool.

Example 27 Configuration of a simulation of two drones

```
- simulator:
    showGUI: false
    coords:
      lat: 40.633490
      lon: -8.660815
      alt: 8.2
  fleetman:
    id: drone01
    telemetryRateMs: 200
    maxSpeed: 5
    takeoffAltitude: 5
- simulator:
    showGUI: false
    coords:
      lat: 40.633347
      lon: -8.660358
      alt: 8.4
  fleetman:
    id: drone02
    telemetryRateMs: 200
    maxSpeed: 5
    takeoffAltitude: 10
```

The first time the application runs, it pulls the PX4 Docker image and builds the PX4 firmware and jMAVSim. The following executions do not require this step, since it will be installed already. The user can provide a YAML file with the simulator and drone configurations, such as in example 27, which contains the configuration of two simulated UAVs. This tool will launch the PX4 Docker image and run the simulator instances, configured

[53]https://docs.px4.io/master/en/simulation/gazebo.html.
[54]https://docs.docker.com/engine/api/.

according to the `simulator` configuration, and then launch the Docker images for the drone module, with the `fleetman` configurations.

The `simulator` configurations allow to choose whether the simulator GUI is displayed, through the `showGUI` parameter, and define the location where the drone will spawn, with the `coords` parameter. Considering that the user does not provide a `coords` parameter, the simulation will start with coordinates close to Instituto de Telecomunicações, with each drone being a few meters apart from the previous one. The `fleetman` configurations are equivalent to those that are usually provided in the drone module configuration file, but with the addition of the `id` parameter, corresponding to the drone's id, and the removal of the `port` parameter, since it will be automatically configured.

It is also possible to launch this without providing a configuration file, in which case it will only launch jMAVSim instances, and the user will have to start the drone modules manually. Through the CLI, the user can also provide the `-n` argument, which indicates how many simulator instances to launch, and `-i`, whether or not to show the GUI.

4.3.5 Feature validation

At this point, we have Docker images that provide an environment with an installation of our modules, a simulation environment for the UAVs, and tools to launch and configure these modules quickly. We can now validate the current features of the platform, without resorting to the physical drones. Validating the platform's current state is essential to ensure that it is safe to progress to the more complex tasks that the missions will enable.

Method

Using the ground station launch script, we started the ground module while also saving the logs and the ROS data. With the simulator launcher tool, we launched the drone simulator and the drone module with the configuration in example 28. We can visualise the drone behaviour with the QGroundControl application. To send the HTTP requests to the ground station, we used Postman, an application that, among other features, can act as an API client[55]. We sent a series of commands to the simulated drone, monitored the ground station logs, and used the telemetry data to evaluate the performance later.

Example 28 Test simulation configuration

```
- simulator:
    showGUI: false
    coords:
      lat: 40.633490
      lon: -8.660815
      alt: 8.2
  fleetman:
    id: drone01
    telemetryRateMs: 200
    maxSpeed: 5
    takeoffAltitude: 3
```

[55] https://www.postman.com/.

Results

With these tests, we were able to verify that the current features of the platform present a consistent behaviour in a simulated environment. These experiments can be divided into two different flights, with a landing happening between them. Figure 4.1 shows image captures of the drone effectively moving in jMAVSim during one of these flights.

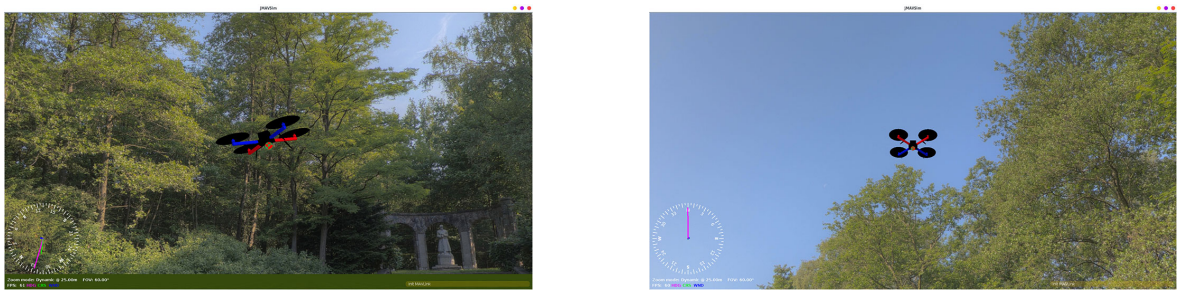
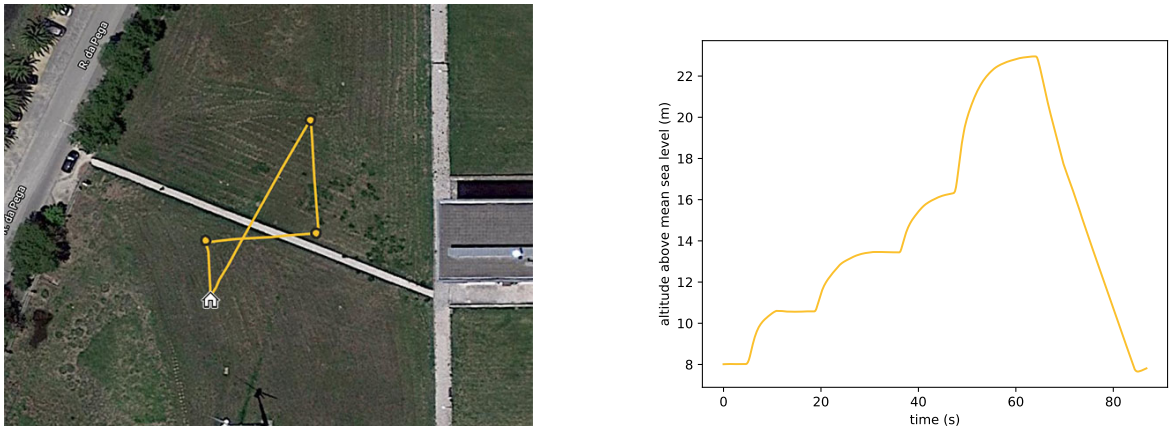


FIGURE 4.1. jMAVSim image captures

For the first flight, in example 29, we started by requesting a command while the drone was not armed. After that, we sent an `arm` command, followed by a takeoff with the default altitude. We sent two move commands: to move 10 meters forward, and then to go 20 meters right and 3 meters up. After reaching that last location, the drone turned 180° degrees and repeated the last move command. This flight was finished with a return to launch command, with an altitude of 15 meters. In figure 4.2, we observe the path that the drone traversed and the altitude throughout the flight. The results are consistent with the requested commands.



(A) Drone path

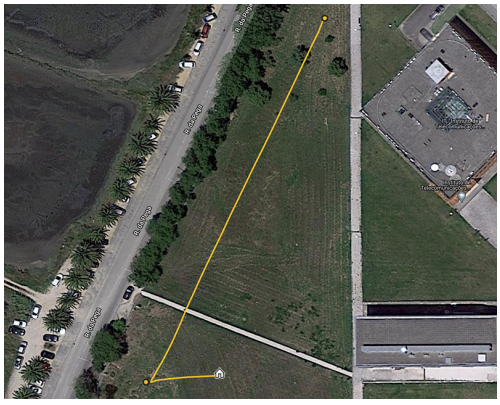
(B) Drone altitude

FIGURE 4.2. Results of the first set of commands

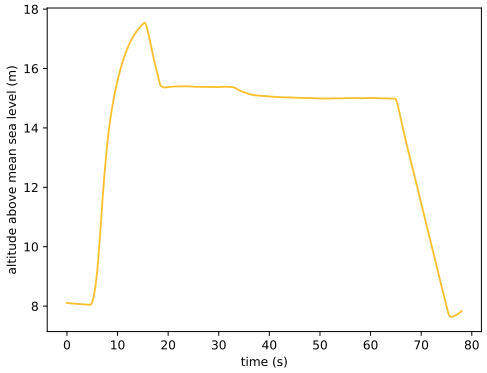
Example 29 Ground station logs of the first flight

```
2021-01-19 23:54:38,263 INFO - drone01 - Registered
2021-01-19 23:54:38,374 INFO - drone01 - Flight controller system connected
2021-01-19 23:55:02,359 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:02,453 WARN - drone01 - Failure on takeoff: disarmed
2021-01-19 23:55:04,232 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:04,862 INFO - drone01 - Success on arm
2021-01-19 23:55:05,903 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:05,987 INFO - drone01 - Start takeoff with altitude 3.0m
2021-01-19 23:55:10,876 INFO - drone01 - Finish takeoff
2021-01-19 23:55:12,617 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:12,682 INFO - drone01 - Start go to 40.633579, -8.660825 at 10.56m
2021-01-19 23:55:17,980 INFO - drone01 - Finish go to 40.633579, -8.660825 at 10.56m
2021-01-19 23:55:20,464 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:20,582 INFO - drone01 - Start go to 40.633591, -8.660591 at 13.57m
2021-01-19 23:55:30,993 INFO - drone01 - Finish go to 40.633591, -8.660591 at 13.57m
2021-01-19 23:55:32,699 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:32,894 INFO - drone01 - Start velocity ned
2021-01-19 23:55:34,096 INFO - drone01 - Finish velocity ned
2021-01-19 23:55:37,524 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:37,898 INFO - drone01 - Start go to 40.633773, -8.660603 at 16.44m
2021-01-19 23:55:48,009 INFO - drone01 - Finish go to 40.633773, -8.660603 at 16.44m
2021-01-19 23:55:48,988 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:55:49,119 INFO - drone01 - Start return to launch with altitude 15.0m
2021-01-19 23:56:28,853 INFO - drone01 - Finish return to launch
```

The logs of the second flight are presented in example 30. We started by arming the drone and requesting a `takeoff` of 10 meters. We immediately sent a `land` command, followed by a `cancel`. The drone had then to go to two different locations, sent through a `goto` command. The second `goto` command was sent when the drone was close to the first waypoint. When the drone reached the second location, it landed. The path and altitude of this flight are presented in figure 4.3. They are again consistent with the commands sent to the drone.



(a) Drone path



(b) Drone altitude

Figure 4.3. Results of the second set of commands

Example 30 Ground station logs of the second flight

```
2021-01-19 23:56:36,105 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:56:36,833 INFO - drone01 - Success on arm
2021-01-19 23:56:39,846 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:56:39,973 INFO - drone01 - Start takeoff with altitude 10.0m
2021-01-19 23:56:49,869 INFO - drone01 - Finish takeoff
2021-01-19 23:56:51,041 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:56:51,069 INFO - drone01 - Start land
2021-01-19 23:56:54,345 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:56:54,357 INFO - drone01 - Cancel land
2021-01-19 23:56:55,862 INFO - drone01 - Success on cancel
2021-01-19 23:57:03,016 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:57:03,087 INFO - drone01 - Start go to 40.633474, -8.661052 at 15.38m
2021-01-19 23:57:08,701 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:57:08,794 INFO - drone01 - Cancel go to 40.633474, -8.661052 at 15.38m
2021-01-19 23:57:08,797 INFO - drone01 - Start go to 40.634421, -8.660437 at 15.0m
2021-01-19 23:57:38,628 INFO - drone01 - Finish go to 40.634421, -8.660437 at 15.0m
2021-01-19 23:57:40,585 INFO - Received request [POST /drone/drone01/cmd]
2021-01-19 23:57:40,629 INFO - drone01 - Start land
2021-01-19 23:57:53,840 INFO - drone01 - Auto disarmed
2021-01-19 23:57:53,844 INFO - drone01 - Finish land
2021-01-19 23:58:22,726 WARN - drone01 - Stopped receiving messages from drone
```

4.4 Summary

We successfully implemented the base features of the architecture proposed in chapter 3. With the current platform, we can remotely control a UAV without requiring a RC, using our ground station and drone software.

The drone module, which relies on the MAVSDK library for integration with the autopilot software, can send commands to the FC and retrieve the telemetry data. By exchanging ROS2 messages, it can communicate with the ground station to receive the commands, share the telemetry data, and notify it of drone events. The integration of sensors requires that the user develops software that can read the sensor data and publish it correctly. We provided an example with the implementation of a simulated temperature sensor.

The user and external applications can communicate with the system through HTTP requests to the ground station. The API endpoints were developed with the Spring Boot framework. With those, it is possible to submit the commands to the drone, retrieve telemetry data and drone state, and monitor the system state.

We package both software modules in Docker images, which allows to distribute a working environment and installation. To aid the software validation through simulated drones, we also developed an application that can launch and configure the jMAVSim simulator and the drone module. Finally, we tested the current state of the platform in a simulated scenario, obtaining consistent results on the actions sent to the drones and the travelled paths.

Chapter 5

Mission Support

Having implemented the features described in the previous chapter, we can now interact with the drone by sending direct high-level commands to it. Although this already delegates most of the control to the drone, the user still has to manage every action one-by-one, which is not ideal when dealing with more complex scenarios and may lead to human error.

This chapter addresses this issue by presenting a solution that allows the description of multi-drone concurrent missions. We start by outlining the overall mission support definition, its description language, and implementation. We conclude by presenting several mission examples.

5.1 Context

To develop a system capable of interpreting and executing missions, we have to balance two factors when defining how the user will describe these missions: how difficult it is for the user to describe the desired mission plan successfully, and what level of mission complexity can be achieved with the method. A GUI is an intuitive medium for the user to describe several mission steps. However, it is less flexible when defining restrictions, such as acting upon a sensor reading, or when it is required to coordinate several entities.

People and organisations that own drone fleets often come from a technological background and may already have programming experience. Considering this, we propose a solution in which the user describes the mission flow through a scripting language developed specifically for this purpose - a DSL. A mission solely based on following given waypoints can be written without much programming knowledge or even generated using other interfaces, while also enabling the usage of control flow statements for more advanced missions. Many navigation algorithms have already been developed and could be easily translated into this language, which would make it straightforward to test, comprehend and modify such algorithms. The purpose of this language is to provide flexibility - it should be possible to write simple scripts without a deep understanding of the framework; however, it should not restrict an experienced user from implementing complex logic.

5.1.1 Requirements

Given that the mission framework and the DSL should support contrasting levels of mission complexity, there are several features that it should leverage. These are the requirements that have to be addressed.

Commands and telemetry

The most crucial feature that has to be implemented in the mission support framework is integrating the previously developed features: sending commands to a drone and accessing the telemetry data. Sending a sequence of commands to a drone and obtaining the drone's current location has to be a trivial task for the user, as those are the base actions upon which more complex behaviour is built. Mission plans are mostly comprised of a set of waypoints or movements, which sometimes may only be retrievable at runtime. As such, the platform should provide utility methods that facilitate the coordination of operations and calculations, such as calculating the distance between two coordinates or retrieving a coordinate 10 meters ahead of the drone. These utilities decrease the redundancy and repetition throughout different scripts, as those operations are useful in a wide range of scenarios, avoiding writing a similar method in multiple circumstances.

Drone assignment and revokation

It should be possible to assign a drone to a running mission at any time - although in many scenarios there will be at least one drone assigned at mission start, it may also be relevant to dispatch additional drones later on. Similarly, it should also be possible to remove a drone from the mission if it is no longer required, allowing it to be allocated to another mission. The user has to be able to choose which drone is being assigned to the mission, but in case there is no available drone that meets the requirements, the mission should not proceed. Drones that are not assigned to the mission should be considered out of scope when sending commands or reading the telemetry data to avoid interference with other running missions.

Multi-drone and synchronisation

Another important aspect is the support of multi-drone missions, and coordination and synchronisation strategies to enable those. It is necessary to be able to control several drones in a single mission independently. Besides, it should also be possible to coordinate tasks that require multiple drones, or that have sequential steps distributed among different drones that must be synchronised.

Multiple execution paths

As previously mentioned, it should be possible to describe a mission that may follow different execution paths. This means that running the same mission script could yield different results according to the context in which it is being executed. For example, a secondary drone could

be summoned to replace another one in response to low battery. This can be achieved by allowing decision-making statements (if-then-else, switch), looping statements (for, while) and branching statements (break, continue, return) to be part of the scripts.

Sensor support

One factor that may influence the execution path of a mission is a sensor reading; for example, if the temperature is too high, the drone's path could be diverted to avoid damage. If the framework enables reading sensor data from within the mission context, it is possible to implement these decisions.

Mission constraints

Some circumstances should lead to a mission failure, as when one of the drones detects a health failure, the battery reaches a critical level, or moves too far from the fleet, risking losing connectivity. These constraints can be verified through the mission script, but it is impractical to inspect those conditions before every single action. As such, to avoid cluttering the mission script, some of these common concerns should either be automatically solved, or immediately stop the mission from progressing without any user input.

5.1.2 Supporting a DSL

Considering the previous requirements, we need a versatile solution to cover them while also providing functionality directed towards this specific context. Writing a custom language and interpreter from scratch would be too time-consuming and require manually implementing basic features and integrating with the existing platform. Using an existing scripting language and building a custom DSL on top of it is a better approach, leveraging what the base language already offers and extending it with the necessary additional features. With an embedded DSL, we can use the underlying language to provide basic syntax and semantics, such as variable declaration, conditional expressions, and loop constructs.

The language we selected to build the DSL is the Apache Groovy programming language[56]. It is compatible with the Java syntax and compiles to Java Virtual Machine (JVM) bytecode, which leads to the integration with the existing GS codebase being completely seamless. Although it is compatible with Java syntax, Groovy's grammar is more flexible. Some of the features that contribute to this are optional semicolons at the end of each line, employing type inference, more straightforward array and map initialisers, consistent relational operators, implicit getters and setters, and default and named parameters[57][58]. These traits make it less verbose, more intuitive, and less error-prone for those with less experience with programming languages. In later sections, more of these features will be explored to improve our DSL.

[56]https://groovy-lang.org.
[57]https://groovy-lang.org/differences.html.
[58]https://groovy-lang.org/style-guide.html.

Many features present in Groovy make it easy to hide complex logic behind a more comprehensible and domain-specific API. The feasibility of Groovy as a DSLs is evidenced by the fact that it has been adopted by a number of widely-used software projects. Jenkins, an automation server, allows defining continuous delivery pipelines with a custom DSL based on Groovy[59]. Gradle, a build automation tool (which is used in this project to build the GS software), uses a Groovy-based DSL[60] to specify build scripts.

In conclusion, Groovy is a viable option to support the development of our DSL, given the easy integration with the existing software, the flexible syntax that can leverage intelligible domain-specific semantics, and proven usage in building DSLs. In section 5.2, we will address the development of the DSL.

5.1.3 Mission pipeline

To summarise the flow of a mission, we illustrate it in figure 5.1. This is a very simplified representation and is not architecturally complete.

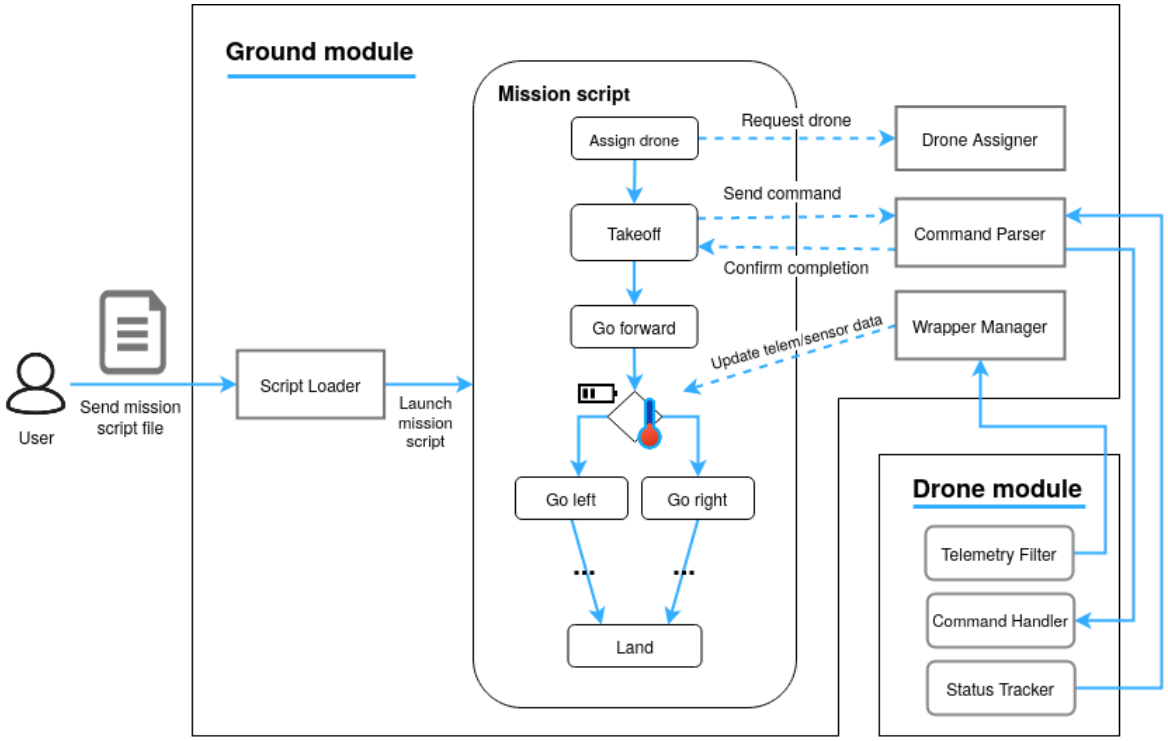


Figure 5.1. Mission pipeline

After writing the mission script, the user submits it to the platform. This script is then loaded and launched. There should be a drone request at some point in the script, which will assign the drone to this mission, preventing it from being allocated to another one. After the assignment, the script will contain command requests, which will be directed to the drone. The script's next instruction will only execute after the command completion is confirmed,

[59]https://www.jenkins.io/doc/pipeline/steps/workflow-cps.
[60]https://docs.gradle.org/current/userguide/writing_build_scripts.html.

whether it is another command or not. The mission flow may take different paths in response to the current state, whether related to a sensor reading or telemetry data. From the mission script, it is always possible to retrieve the most recent data. After all instructions from the script have been fulfiled, the mission concludes.

5.2 A domain-specific language for drone missions

Considering the previous requirements, we will extend our platform to interpret and execute missions. These missions are scripts written in Groovy that provide high-level methods that allow users to send commands to the drones. This section details the development process of the framework and presents the resulting DSL.

5.2.1 Groovy integration

As previously mentioned, Groovy's flexibility allows to build a more fluid DSL. This section will start with the description of some features of Groovy that were relevant to improve the intuitiveness of the DSL. These features are thoroughly documented and explained in[61] and [47].

Command chaining

Groovy allows to omit parentheses around the arguments of a method call, which means that both of these formats are valid and will yield the same result:

```
move(up) , move up
```

This feature is extended by using command chains, in which the dot between chained calls can also be omitted, leading to both of these instructions having the same meaning:

```
move(up).and(right)
move up and right
```

With these features, the DSL is brought closer to natural language, which is beneficial for less experienced users, since the knowledge required to understand the semantics of the instruction is related to the domain.

Named arguments

Usually, when providing parameters to a method, these are sorted by the order in which they were given. This may not pose a problem in most situations, but it requires the user to be aware of the correct argument order and memorise it. In our context, this is not a very convenient approach and could potentially lead to the user committing mistakes. Groovy supports named arguments in method calls, which can solve this issue as it is easier to

[61] https://groovy-lang.org/dsls.html.

remember what property we are referring to instead of the order in which we should provide it.

```
move(40.6339, -8.6605, 15, 45, 5)
move(lat: 40.6339, lon: -8.6605, alt: 15, yaw: 45, speed: 5)
```

Comparing these two method calls, we can observe that the latter is more straightforward since the arguments can be arranged in any order. This works by converting that list of arguments into a key-value map. As a consequence, this does not force providing all of the parameters, which is also convenient if some of the arguments are optional and have default values.

Categories and operator overloading

In the domain of UAVs, when referring to commands like moving to a relative position, we often refer to measurements like distance or speed. It is important to make sure that these measurements are being communicated in the same units to avoid unexpected (and possibly dangerous) outcomes.

To address this, we can use a mechanism in Groovy that allows to augment existing classes with additional methods, which is called a "category". This concept is similar to inheritance, with the difference that no existing behaviour of the class is modified - there is no need to create a subclass, just extending the original one with the additional behaviour. With categories we can extend, for example, numbers, to provide behaviour that we need in the context of our DSL, but only within the scope in which we decide to use it, without polluting the external environment with superfluous methods.

```
takeoff 50.cm
move forward: 10.meters
```

These examples are possible by extending the Number class with the `cm()` and `meters()` methods, which when called on a number, will return a distance object. By using this feature, we can guarantee that the user is aware of the distance they are requesting. However, categories will not suffice to represent speed units coherently. Groovy allows to overload operators in our classes by implementing the corresponding methods. We can take advantage of this feature to implement a division operator between distances and time units. By doing this, we are able to describe speed as expected:

```
move forward: 10.meters, speed: 10.m/s
```

Whenever a distance is required and the user provides a number without specifying the unit, it will be interpreted as meters.

Method injection and delegation

We will assume a scenario in which we want to assign a drone to a mission and request that it moves left. Since these actions are unrelated, they are each defined in a different class. Additionally, the concept of "left" is part of a list of allowed directions, thus being present in a separate enumerate. This would lead to the code having an appearance similar to this:

```
droneAssigner = new DroneAssigner()
commandParser = new CommandParser()
drone = droneAssigner.assign
commandParser.move drone, Direction.left
```

This is a very verbose approach and exposes the internals of our framework to the user. To solve this, we can use method injection. When we launch a Groovy script, we have to provide what will be the script's binding - an object where the instances of variables referenced through the script are stored. We can initialise a single instance of the `DroneAssigner` and the `CommandParser`, which will be injected to the script's binding before launch. It can also be used to inject constants such as the directions, creating a direct correspondence between `left` and `Direction.left`.

```
drone = droneAssigner.assign
commandParser.move drone, left
```

This already improves how the script looks; however, even though we can now access the `droneAssigner` and `commandParser` without having to instantiate it explicitly, it is still necessary to reference them in order to call one of their methods. By using delegation, we can import all methods of a certain class to be used directly in another class. If we delegate the methods of the `droneAssigner` and `commandParser` to our script's base class, these will be available to use in the scripts developed by the user.

```
drone = assign
move drone, left
```

5.2.2 Mission launching and management

When the user submits a mission file to the platform, which is accomplished by sending the corresponding file to an HTTP endpoint, the *Script Loader* will also receive it and is responsible for loading it. It will start by requesting that the *Mission Manager* initialises it, by generating a mission ID, which is followed by generating the script's binding, defining the blacklisted imports and methods, and setting the base script class. The *Script Loader* then launches a thread in which it notifies the *Mission Manager* that the mission started and then evaluates the script, executing the instructions.

The *Mission Manager* stores the mission status - registering when a mission started, when it finished, how it finished, or what drones were involved. These mission properties are

detailed in table 5.1. Besides providing this data when requested, it also logs the events. The internal components that are part of the mission framework also interact with the *Mission Manager*, whether to request a mission to be stopped, or to retrieve data, such as confirming if a mission is still running or for what mission is a certain drone working. There is also an HTTP endpoint from which these values can be retrieved through a `GET` request.

Table 5.1. Mission properties

Property	**Description**
missionId	ID of this mission
status	Current status of the mission
cause	The cause of mission termination, when not successful
start	Timestamp of mission start
end	Timestamp of mission end
plugins	Enabled plugins
activeDrones	Drones that are currently active in this mission
usedDrones	Missions that were used by this mission
usedDrones	Drones that were used by this mission

A mission can be in one of the five states described in table 5.2. The mission will be in the `pending` state after the *Script Loader* has requested a mission ID, but before it launches the mission script. A concluded mission can either be in `finished`, `failed`, or `cancelled` states.

Table 5.2. Mission status

Status	**Description**
pending	Mission has been initialized but the script has not started
running	Mission is currently running
finished	Mission finished successfully
failed	Mission had a failure during execution
cancelled	Mission was cancelled

In table 5.3, we present all possible causes of mission conclusion, with the majority being related to mission failure. Most of the failures are not triggered by external events, resulting from a runtime exception instead. The *Exception Handler* is responsible for logging the exceptions and for requesting the mission and corresponding threads' termination. In order to cancel a running mission, the user can send a `DELETE` request to the HTTP endpoint, providing the corresponding mission ID.

Table 5.3. Mission conclusion causes

Cause	**Status**	**Message**
success	finished	success
user	cancelled	user request
manual	cancelled	manual mode activated
return	cancelled	automatic return to launch detected
requirements	failed	not enough drones to fulfill mission requirements
cmd_exec	failed	failure in command execution
cmd_parse	failed	failure in command parsing
plugin_error	failed	plugin encountered an error
plugin_param	failed	illegal or missing plugin input parameter
plugin_missing	failed	plugin is not loaded
plugin_disabled	failed	plugin is not enabled
timeout	failed	drone timeout
sensor	failed	sensor timeout
script	failed	malformed script
revoked	failed	referenced revoked drone
drone_error	failed	drone entered error state
security	failed	mission script called a blacklisted method or import
irreplaceable	failed	requested replacement of drone that can not be replaced
unexpected	failed	unexpected exception

5.2.3 Drone data management

With the platform developed in the previous chapter, we can retrieve the most recent telemetry data for all available drones. Not all telemetry data is relevant in the context of a mission. Some of the drone data information could be useful; for example, the drone state and current command provide more useful information about the drone than the flight mode and land state telemetry data. To better suit the mission scope, we created a wrapper model that merges the relevant information of both telemetry and drone data, as well as sensor readings from available sensors, which resulted in the drone attributes in table 5.4 being accessible during a mission.

Table 5.4. Drone wrapper data

Attribute	**Description**
id	ID of the drone
state	drone state
armed	whether the drone is currently armed
cmd	current command
cmd.target	target of the command if it is either a `takeoff` or a `goto/move`
position	current GPS coordinates
home	GPS coordinates before takeoff
battery	remaining battery percentage
heading	direction at which the drone is headed
speed	current speed
sensor	sensor readings

As soon as a drone is assigned to a mission, an object encapsulating these parameters is created. The data can be easily accessed during a mission by accessing the attribute in the corresponding drone's object, such as `drone.id`, `drone.cmd`, `drone.cmd.target`, `drone.position.alt`, or `drone.sensor.temperature`. The *Wrapper Manager* manages this data, updating the values after each telemetry message and keeping track of the Drone Wrappers associated with each mission and clearing them when the mission is over. The sensor readings are also updated when the sensor sends a new value. If the drone is revoked from the mission before it ends, these values will no longer be accessible - the only available

data is from drones within the mission scope.

5.2.4 Drone assignment

A drone has to be assigned to a mission before we can control it. The *Drone Assigner* provides the methods to do it in a mission script by using the `assign` verb and binding it to a variable:

```
drone = assign 'drone01'
```

If a drone with id `drone01` is currently available, it will be assigned to this mission, and the `drone` variable will reference a drone wrapper object. We could access the drone's battery with `drone.battery` or the drone's altitude with `drone.position.alt`.

As a starting point, this form of assignment provides enough functionality, but it can be improved. For example, the mission might not be restricted to be executed by that specific drone, and any available drone could suffice. There could also be some restrictions regarding which drone should be assigned, such as what sensors the drone needs to perform the task, or it could be useful to retrieve the drone that is closer to a particular coordinate. Additional forms of the previous command are allowed to enable dynamic control over which drone is assigned.

```
1 drone1 = assign any
2 drone2 = assign temperature
3 drone3 = assign [temperature, camera]
4 drone4 = assign lat: 40.6339, lon: -8.6605
```

In line 1, the script will request for any available drone to be assigned. The second and third lines demonstrate how to request for a drone with one or more sensors. The last case shows how to request the drone that is closer to the provided coordinate. Excluding this last case, in which the closest drone will always be picked, the chosen one will be the one that has the least sensors while still fulfiling the requirements. When multiple drones fit the criteria, the one with higher battery percentage is assigned to the mission.

```
(drone1, drone2, drone3) = assign 'drone01', any, temperature
```

By wrapping the left-hand side of the declaration with parentheses, we are able to assign multiple drones in the same line of code. When using this feature, the order in which the assignments are processed is sorted according to the requirement:

1. Specific drone id
2. Sensors, with the priority sorted according to number of sensors
3. Closer to a coordinate
4. Any drone

Since the mission will be cancelled if it is not possible to satisfy all requirements, this sorting order will attempt to maximise the drone distribution, avoiding what could be an avertable mission failure.

Besides handling the drone assignment, the *Drone Assigner* can also be used for two more tasks: replacing and revoking a drone from a mission. We may want to replace a drone during a mission if, for example, we detect that its battery is running low. However, the drone will be classified as irreplaceable if assigned through the drone ID or by requesting a specific sensor. In this case, the replacement could yield unwanted results, such as if the drone contains a sensor required for the mission that we do not manage. An attempt to replace an irreplaceable drone will result in a mission error. The replacement drone is the available drone that is closest to the location where the previous drone stopped. If we no longer need one of the drones to proceed with the rest of the mission, we should revoke it, allowing it to be used by another mission that could be running simultaneously. Once the drone is revoked, it is no longer possible to send a command or read its telemetry within the mission, unless it is assigned again. Both tasks can be executed in a straightforward manner - either by calling `replace drone` or `revoke drone`.

5.2.5 Command handling and execution

The most crucial feature of the framework is the possibility to send commands to the drone. These are interpreted by the *Command Parser*, which contains the method definitions of all provided command verbs. In this context, the commands that we support are the high-level ones - this includes the `action` commands and the `custom` commands. All commands employ a similar formula, comprised of `[command] [droneId] [options]`, with the main difference being the possible additional parameters and how those can be provided. Upon receiving a method call, the provided parameters are validated according to the command. If there are any irregularities with the command, the mission will fail. Otherwise, a JSON string is built containing all the required fields, which is then sent to the drone in a ROS message through the *Message Publisher*.

Thread progress

As an example, we consider that the user writes a mission script after the drone assignment contains several consecutive commands. If the script execution progressed after sending the first command, it would be immediately cancelled by the following command, which would repeatedly happen until the end of the script - only the last command would be effectively executed. This can be avoided if each command call results in halting the thread's execution while waiting for the command completion. After sending the command, the *Command Parser* will lock the current thread until it is notified that the command has finished, which happens after the *Status Updater* receives a status message stating it.

Utilities

Sometimes it may be required to calculate the distance between the drone and a particular coordinate in order to make a decision. We may want to find the coordinates 10 meters ahead of the drone or calculate the bearing between two drones. In chapter 4, we implemented the algorithms for those calculations drone-side, so we have to rewrite those in Groovy. It should be noted that the drone-side calculations are more accurate, since the drone is able to access the current telemetry data directly. In contrast, we are dependent on the telemetry update rate when operating from the ground station. However, if we want to send the drone to a position that is not relative to itself, there is no other way to do it except from the ground station. With this in mind, we can always force waiting for a telemetry update before doing one of these operations.

To use the `distance` method, we have to provide two coordinates, whether they are manually defined, a drone, a command (if it is a `move` command), or a drone's home position. Instead of providing the first position, the user can call the method upon it, which will yield the same result.

```
1 (drone01, drone02) = assign any, any
2 coords = [lat: 40.634125, lon: -8.660336]
3 dist1 = distance drone01, drone02
4 dist2 = drone02.cmd.distance(coords)
5 dist3 = coords.distance(drone01.home)
```

If we want to calculate the `bearing` between two coordinates, the method can be called similarly to the `distance` one, but in this case, we just have to provide both positions. This will return the absolute bearing angle between two points.

```
1 (drone01, drone02) = assign any, any
2 coords = [lat: 40.634125, lon: -8.660336]
3 bearing1 = bearing drone01, drone02
4 bearing2 = coords.bearing(drone02)
```

Finally, it is also possible to calculate a `target` position based on the original position, the distance and the bearing. If the user wants to provide a bearing relative to the drone, it has to add the current drone heading to the provided angle. One circumstance in which this might be useful is when we want one drone to move towards another, which can also be achieved by calling the `bearing` method. As a convenience, if we call `left` on a drone, it will return the target position left of the drone. These can also be chained, which will return the final position if each movement had been executed consecutively. This can be useful to calculate the waypoints of a mission before execution.

```
1 (drone01, drone02) = assign any, any
2 coords1 = target drone01, 10.m, 45.deg
```

```
coords2 = target drone01, 10.m, drone01.heading + 45.deg
coords3 = drone01.target(10.m, drone01.bearing(drone02))
coords4 = drone02.left(5.meters).right(10.meters)
```

Arm, disarm, land, and cancel

The `arm`, `disarm`, and `land` action commands and the `cancel` custom command do not take additional arguments. As such, they can be executed by simply calling it in the `[command] [droneId]` format. The drone's FC might be configured to disarm it once landed, so this has to be taken into account while developing the scripts.

Takeoff and return to launch

The `takeoff` and `return` commands may be issued while providing a target altitude. Otherwise, it will use the drone's default takeoff altitude. The keyword that is used to indicate a return to launch command is `home`.

```
takeoff drone01
takeoff drone01, 50.cm
takeoff drone01, 10
home drone01, 15.meters
```

Turn

As a starting point, we allowed the `turn` command to receive a value after the `droneId`. This could be the angle, or what direction the drone would be turning to.

```
turn drone01, 45.deg
turn drone01, left
turn drone01, north
```

Since it was possible, we decided to allow a more versatile description of this command, by introducing the `by` and `to` keywords. This means that turning the drone "by" x degrees will rotate the yaw angle by that amount, while turning it "to" x degrees will rotate it to that angle, relative to North. Also, we introduced the concept of turning to a particular position.

```
turn drone01 by 90.deg
turn drone01 to 90.deg
turn drone01 to west
turn drone01 to drone02
```

Move

Both the `goto` action command and the `move` custom commands are referenced through the keyword `move`, since "goto" is a reserved keyword in Java (and consequently in Groovy) and the sentence "go to drone" does not mean that we are moving the drone, but moving something to the drone. The `move` command could be issued by using the `[command] [droneId]` format, followed by the arguments, which could be either corresponding to the drone's `goto` command or the `move` command itself. The forward/backward, left/right, up/down parameters will be converted to the `xyz` parameters of the `move` command. As before, the `alt`, `yaw` and `speed` parameters are optional.

```
move drone01, lat: 40.634, lon: -8.660, alt: 18.m, yaw: 45.deg, speed: 5.m/s
move drone01, forward: 12.m, up: 3.m, left: 5.m, speed: 3.m/s
```

Once again, we also provide an alternative that is more fluid. The `goto` command parameters can be provided after the "to" keyword. In the `move` command, the first direction has to be followed by the keyword "by" (such as "down by:"), while the remaining ones do not require the "by". In terms of implementation, this is needed because the first direction will be interpreted as a method call, while the remaining ones (and the "by") will be interpreted as parameters. In both cases, the speed can be specified after the "at" word, but it is restricted to be the first parameter of the command.

```
move drone01 at 5.m/s to lat: 40.6342, lon: -8.6614, alt: 17.4.meters
move drone01 at 10.m/s forward by: 5.m right: 10.m
```

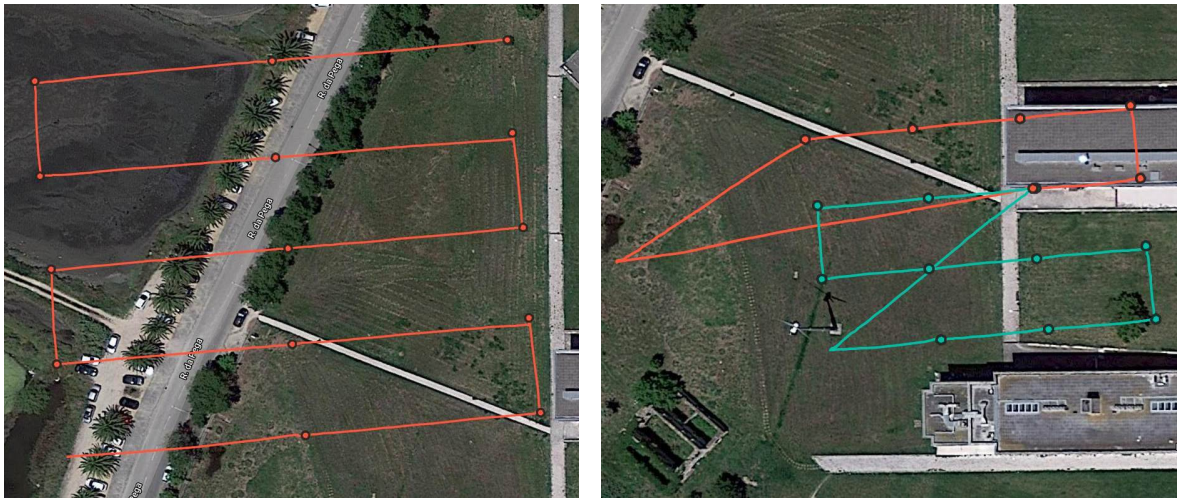
Examples

To demonstrate how to use these commands to build a mission, we will present two mission script examples. An algorithm that is commonly used is the mapping of an area, which is described in script 5.1. An initial coordinate may optionally be provided so that the drone first moves to that location before everything else. The drone will then proceed to traverse the area from left to right, bottom to top, with the provided `x` and `y` dimensions, and dividing the path according to the requested number of steps. With a value of 2 in `steps_x`, the drone will stop once halfway through the horizontal traversal. With a value of 2 in `steps_y`, the drone will change the traversal direction twice, traversing a horizontal length of `x` three times. In this example, we used a length of 100 meters, a width of 80 meters, two horizontal steps and four vertical steps. The resulting path by the end of the mission is depicted in figure 5.2a.

Script 5.1 Mapping

```
drone01 = assign any
arm drone01
takeoff drone01
map_area(drone01, 100.m, 80.m, 2, 4)
land drone01

def map_area(drone, x, y, steps_x, steps_y, init_pos=null) {
    def step_sz_x = x / steps_x
    def step_sz_y = y / steps_y
    if (init_pos != null)
        move drone to init_pos
    (steps_y+1).times { traversal ->
        if (traversal > 0)
            move drone right by: traversal.even ? step_sz_y : -step_sz_y
        turn drone to traversal.even ? right : left
        steps_x.times { move drone forward by: step_sz_x }
    }
}
```



(a) Drone path of the mapping mission

(b) Drone path of the replacement mission

Figure 5.2. Drone path after executing the missions

By slightly modifying the mapping algorithm to verify the battery before sending a new move command, we are able to implement a simple version of drone replacement, as demonstrated in script 5.2. When the battery of the first drone is below 40%, it is sent to the home position, and a new drone takes its place to finish the mapping. In figure 5.2b, we see

the result of this mission - the drone in green traversed the first waypoints, while the drone in red went through the last ones. The mapping algorithm was requested with three horizontal and vertical steps.

Script 5.2 Replacement during mapping

```
def move_or_replace(movement) {
    if (drone.battery < 0.4) {
        def curr_pos = [lat: drone.position.lat, lon: drone.position.lon,
                        alt: drone.position.alt, yaw: drone.heading]
        run {home drone}
        revoke drone
        summon_drone()
        move drone, curr_pos
    }
    move drone, movement
}
```

5.2.6 Multi-drone missions and concurrency

As previously explained, a running command will halt the execution of the thread until it is completed. A consequence of this behaviour is that, if the user wants to control two drones in the same mission script, then they will take turns executing commands, without the possibility of running commands in parallel. This restriction would not be appropriate for multi-drone missions.

To solve this, we have to allow the user to define subroutines, which can be executed concurrently. A subroutine is a mission thread - whether it is the primary mission thread or one that is later created - running a set of mission instructions. Any failure that happens during its execution will also lead to the failure of the mission. After launching a subroutine, it should be possible to stop it, wait for its conclusion, or verify if it is still running.

Subroutine control

The handling of subroutines is split between two components: the *Subroutine Manager* and the *Thread Keeper*. The former leads the higher-level operations that are provided to the user: it allows the user to launch a subroutine, stop it on command, wait for its conclusion, and verify the running status. The latter is responsible for actually running and stopping the threads, cancelling a command if the thread that requested it was also stopped, and stopping all threads related to a mission that concluded. In sum, the *Subroutine Manager* contains the methods that are the interface for the user to control subroutines, which will, in turn, call the appropriate actions defined in the *Thread Keeper*.

When the user requests for a subroutine to be launched during a mission, providing the code that should be executed in it, the *Subroutine Manager* starts by generating the subroutine ID, which will be stored on the list of running subroutines. The provided code is wrapped with the *Exception Handler*, which will be able to catch any malfunction during the subroutine execution that should lead to a mission failure. Finally, it is sent to the *Thread Keeper*, which will launch the actual thread. The subroutine ID is returned after the call to run it, which should be saved in the script if necessary to manage it later on, such as waiting for its conclusion or stopping it. The subroutine ID is also equivalent to the thread name, which will contain the mission ID followed by a random string of alphanumeric characters. This allows the platform to stop all the threads of a mission efficiently without creating additional data structures to manage it. With that ID, it is also possible to verify if a request to stop or wait on a particular subroutine comes from the same mission. The subroutine ID has to be provided in order to stop a subroutine, wait for its conclusion, or verify if it is running.

The implementation of these subroutines led to the creation of three other keywords - `run`, `stop` and `wait`. The `run` method receives a Groovy Closure, launches it through the *Subroutine Manager*, and returns the subroutine ID. A closure is similar to a method and can be defined as a block of code that can take arguments, return a value, and be assigned to a variable[62]. Any block of code can become a closure if it is wrapped in curly brackets, which means that a method can be converted into a closure simply by encapsulating the call into the brackets. The `wait` method receives a subroutine ID and will halt the script execution until the corresponding thread terminates. The `stop` method receives a subroutine ID and will stop the execution. With the subroutine ID, it is possible to call the `running` or `finished` methods on it to verify the running status.

```
1 subroutineId1 = run {println "hello!"}
2 subroutineId2 = run {println "bye!"}
3 if (subroutineId1.running)
4     stop subroutineId1
5 wait subroutineId2
```

Script 5.3 contains an example of a mission in which two drones simultaneously arm, takeoff, move and land. We observe that dispatch is defined as a method, while `land_drone` is a closure. However, as both can take arguments, they have to be wrapped in curly braces when provided to the `run` method; otherwise, we would be sending the result of the closure call instead of the closure itself. Instead of providing a method or a previously defined closure to be executed, we can wrap an instruction in a closure, such as in lines 13 and 14. This is particularly useful when we only aim to submit a single command that is not repeated.

[62]https://groovy-lang.org/closures.html.

Script 5.3 Takeoff-move-land with two drones

```
def dispatch(drone) {
    arm drone
    takeoff drone
}
def land_drone = { drone ->
    land drone
}
(drone01, drone02) = assign any, any
(takeoff1, takeoff2) = run {dispatch(drone01)}, {dispatch(drone02)}
wait takeoff1, takeoff2
move1 = run {move drone01 forward by: 10.meters}
move2 = run {move drone02 left by: 5.meters}
wait move1, move2
(land1, land2) = run {land_drone(drone01)}, {land_drone(drone02)}
wait land1, land2
```

Synchronization

We might want to synchronise two or more subroutines during a mission, guaranteeing that some tasks are only performed after a previous one concludes. Besides, we may need to poll a drone's status, which would be better achieved by waiting for a telemetry update instead of calling a `sleep` method for an indeterminate amount of time. Previously, we mentioned that a thread's execution is halted after a command is sent until it receives confirmation that the command finished. This update also needs to be synchronised.

To introduce concurrency into the platform without increasing the complexity level, we used GPars[63], a concurrency and parallelism framework for Groovy and Java. This library provides high-level abstractions to write concurrent and parallel code in Groovy. The main feature provided by GPars that we employ is Dataflow variables, which allow thread-safe data exchange between threads. When a thread accesses a dataflow variable, it will be blocked until another thread has set its value. This is useful to create a communication channel between threads, in which a subroutine may wait for another one to unlock it. We provide a `channel` variable for each mission, in which the different subroutines can set and get values.

In script 5.4, we have defined two different subroutines, one for each drone. The first subroutine will set the value of `channel.done` after the drone has finished taking off, which will cause the second subroutine to unlock and launch the second drone. This is an example of how a similar type of synchronisation could be achieved. The value that is used to set the variable may have a purpose, as when trying to share data between threads, or could be disposable data, when used solely to unblock the other threads.

[63] http://www.gpars.org/.

Script 5.4 Multi-drone synchronisation

```
routine1 = {
    arm drone01
    takeoff drone01, 5.m
    channel.done = true
    6.times {
        move drone01 left by: 12.m, forward: 3.m
    }
    land drone01
}
routine2 = {
    channel.done
    arm drone02
    takeoff drone02, 2.m
    6.times {
        move drone02 right by: 30.m, forward: 5.m
    }
    land drone02
}
drone01 = assign 'drone01'
drone02 = assign 'drone02'
(r1, r2) = run routine1, routine2
wait r1, r2
```

A similar approach is used to notify that a command has completed or that the telemetry has been updated. To manage the synchronisation at the platform level, we developed the *Synchronisation Channel*. It provides methods that simplify this process for the other components, such as the *Status Updater* that unlocks the commands and telemetry. When the *Command Parser* calls a method to wait for the command, it will attempt to retrieve the value of a Dataflow variable containing the droneId and the command. In turn, the *Status Updater* will set the value of that variable when it receives a command completion message, marking it as `true` when successful and as `false` when unsuccessful. At every received telemetry message, the *Status Updater* calls the method for unlocking the telemetry, which involves setting the value of the Dataflow variable corresponding to that drone's telemetry and immediately clearing it again. This will unlock all threads that were already waiting for the latest telemetry, while also blocking any new threads that request it just after the update. During a mission, the user can force the script to wait for a telemetry update by merely including a `wait [droneId]` instruction.

5.3 Mission plugins

Most of the mission framework requirements have been met so far, but this still lacks support for mission constraints. Those can be addressed in the mission script itself, such as the drone replacement in script 5.2, but that is not ideal for two reasons. One consequence is that it demands the repetition of the same algorithms through different missions scripts. Another one is that some algorithms that solve those constraints, such as a relay algorithm, may be necessary for inexperienced users, but too complex for them to implement.

To reduce the amount of code that is reused across multiple missions and to allow verifying certain conditions in the background, we introduced the concept of mission plugins. Plugins are mission scripts that can be enabled from within other scripts to provide additional functionality. Besides providing the actual functionality they are aimed to, the plugin scripts will also include the plugin configuration which is loaded at GS startup.

Several types of plugins could be implemented: (1) plugins that expose methods such as mapping to be used during the mission; (2) plugins that monitor each drone's status and act upon it; (3) plugins that transform a command into subcommands (such as to implement obstacle avoidance); (4) plugins that retransmit telemetry and drone status to external components in different formats; (5) plugins that process and transform sensor data, and many other functionalities. All these different types of plugins would be useful; however, implementing and testing all of those would not be possible during this dissertation. We decided to implement the monitoring plugins, which allow demonstrating how to integrate plugins into the framework and at the same time, support mission constraints.

Monitoring plugins will analyse each drone's data after a telemetry update, which may lead to additional actions, such as assigning another drone to the mission or revoking the current drone.

5.3.1 Configuration

In order to communicate how the user should launch the plugin, such as what parameters have to be provided, each plugin file should provide a plugin configuration as in example 31. The available parameters are documented in table 5.5, which are mandatory, with the exception of the `input` and `vars` parameters. The plugin ID is used to identify what plugin the user wants to enable or disable during a mission. If multiple plugin files contain a configuration with the same ID, the last one to be loaded will overwrite the previous ones. Currently, the only available type is `monitoring`, since it is the only one that was implemented. Some plugins may require that the user provides input parameters; if so, the parameter data type has to be declared, as well as any default value if applicable. The `vars` parameter initialises internal variables required by the plugin, which belong to the mission scope instead of the plugin scope. Any global variables declared outside will be shared by the plugin throughout all missions. The `callback` parameter, in the case of a monitoring plugin, indicates the closure that is called upon receiving a drone's telemetry message. That closure has to accept one parameter, which will be the monitored drone.

Table 5.5. Plugin configuration parameters

Parameter	Description
id	ID of the plugin
type	type of the plugin
input	input parameters of the plugin provided by the user
vars	variable initilisation in this mission scope
callback	entrypoint callback to be executed

Example 31 Basic plugin configuration

```
def main_closure = { drone ->
    println "drone: ${drone.battery}"
}
plugin {
    id  'base_plugin'
    type monitoring
    input foo: int,
          bar: String,
          deft: [String, "default value"]
    vars  count: 0
    callback main_closure
}
```

This example plugin would simply print out each drone's current battery level when a telemetry message was received. In a regular mission script, this plugin could be enabled and disabled by running the following commands:

```
enable 'base_plugin', foo: 5, bar: 'hello'
disable 'base_plugin'
```

5.3.2 Implementation

The plugin scripts are loaded from a directory after starting the ground station software, and the plugin configuration is validated and then stored. When a mission script enables a monitoring plugin, a new binding will be created for the plugin during that mission. That binding is based on a regular mission script binding, with the addition of all variables defined in the `input` and `vars` parameters, with the `input` values taking the value provided by the user or the default value when one exists. This binding is then used as a delegate of the callback defined in the plugin configuration - this means that it is possible to access the

binding's variables from within that closure. Since this closure is shared among subroutines of this plugin-mission pair, the assigned variables are segregated from other missions using the same plugin. There is a caveat - the plugin script cannot define methods, these have to be replaced by closures since these can be stored in the binding, while method references will disappear. The callback closure is stored, and then a new subroutine will be launched for each active drone. This subroutine will run the same loop while the drone is assigned to that mission, in which the closure is called, and then it waits for a new telemetry message from that drone. If a new drone is assigned to the mission after the plugin is enabled, a new subroutine will also be launched to monitor it as well.

When a drone is revoked from the mission, the corresponding plugin subroutine is stopped. As soon as a mission is finished or the plugin is disabled, all plugin subroutines for that mission are also stopped.

5.4 Mission examples

This section presents a set of scenarios that will be demonstrated and the mission scripts that were developed to deploy them using the previous DSL. These scenarios are diverse in terms of the mission framework features that they demonstrate.

5.4.1 Fire perimeter tracing

UAVs are typically equipped with sensors which may be used to gather data to analyse at a later stage, or to make flight decisions based on those readings. As mentioned in section 5.2.3, it is possible to have access to a drone's sensor readings from within the mission, given that the sensor data is published accordingly to the format described in chapter 4.

Temperature sensors are a common type of sensor and can be used to detect if the environment temperature exceeds a certain threshold. Among other scenarios, these can be useful to detect a fire. To demonstrate an example of how to use sensor readings in a mission script, we used the temperature sensor simulator presented in 4, and designed a mission in which the drone would start by mapping an area until it detected high temperatures. Once those temperatures surpassed the defined threshold, the drone would start tracing the fire perimeter.

Script 5.5 contains the initialisation of the mission. A target temperature of 40° is defined, which will be the target temperature which we will be tracing around. After assigning a drone with a fire sensor to this mission, it will begin by mapping the area, using the algorithm previously described in script 5.1. Once the temperature surpasses 37° (used just as an example), it triggers the fire tracing algorithm. When the fire tracing starts, the drone's position is stored, so that it is possible to calculate if the drone has already traced everything and returned to the beginning.

Script 5.5 Fire tracing initialization

```
target_temp = 40, start_pos = null
drone = assign temperature
arm drone
takeoff drone, 3.m
map = run {map_area(drone, 30.meters, 40.meters, 2, 3)}
while (map.running) {
    if (drone.sensor.temperature > 37) {
        start_pos = drone.position
        stop map
        trace(drone)
        break
    } wait drone
} home drone
```

The fire tracing is based on a perimeter tracking algorithm [48] using proportional-integral-derivative (PID) control. To delimit the area, we control the drone's angular velocity by adjusting the angle direction in response to the detected temperature, with a positive angle when the temperature surpasses the threshold, and a negative angle otherwise. However, the direction calculation is refined by using the PID controller. In script 5.6, we have part of this algorithm. The angle of the next movement is calculated, and the drone is sent to that target position. When the distance between the drone and the initial position is lower than 5 meters after it has already moved further from it, we consider that the fire has been traced, which will cause the drone to return to the home position.

Script 5.6 Moving drone to the next location

```
def trace(drone) {
   def p = 8, i = 0.01, d = 10
   def error = null, error_sum = 0, deg
   boolean first = true
   while (true) {
      (error, deg) = calc_angle(p, i, d, target_temp, error, error_sum)
      error_sum += error
      wait drone
      move drone at 10.m/s to drone.target(4.meters, drone.heading + deg)
      if (first) {
          if (drone.distance(start_pos) > 7.meters)
              first = false
      } else if (drone.distance(start_pos) < 5.meters)
          break
   }
}
```

The PID controller equation is depicted in equation 5.1. It calculates an error value (which in our case translates to the desired angle) as the difference between the target temperature and the measured temperature. It corrects it based on proportional, integral, and derivative terms. The values that were used for the `p`, `i`, and `d` components were chosen empirically, by adjusting these values until it produced decent results. As such, these values may not represent the optimal path correction but are acceptable.

$$\begin{gathered} error(t) = temp(t) - v_{ref} \\ angle(t) = P \cdot error(t) + I \cdot \int error(t)dt + D \cdot \frac{d}{dt} error(t) \end{gathered} \tag{5.1}$$

Although we are using this perimeter tracking algorithm in the context of a fire, it could be applied in other scenarios with different sensors. For example, it could also be adapted for obstacle avoidance, given that it is possible to detect the distance between the objects and the drone.

5.4.2 Drone replacement

In some circumstances, a drone may need to be replaced during a mission, for example, if the battery is lower than the required level to return to the launch position safely. Previously in script 5.2, we presented a lazy approach to the drone replacement. In script 5.7, we have a plugin script for drone replacement at low battery, which is more refined.

Script 5.7 Plugin for drone replacement

```
def calcNecessaryBattery = { drone ->
    drainage * drone.distance(drone.home)
}
def is_returning_home = { drone ->
    drone.cmd == home ||
        (drone.cmd == move && drone.cmd.distance(drone.home) < 50.cm)
}
def main = { drone ->
    if (!is_returning_home(drone)
        && calcNecessaryBattery(drone) > drone.battery - min_battery)
    {
        replace drone
    }
}
plugin {
    id      'replacement'
    input    min_battery: [float, 0.25],
             drainage: [float, 0.0002]
    callback main
}
```

The `min_battery` parameter represents the minimum battery level that the drone should reach. The `drainage` parameter indicates how much battery is drained by moving the drone one meter. The default `min_battery` value is `0.25`, which means that the drone should return to the home position when it is expected that executing it would leave its battery level at 25% (as an example). The default value of `0.0002` for the `drainage` parameter means that we are assuming that the drone's battery level will drop by 1% after moving for 50 meters. These values are merely speculative, since the studies on the drone's battery progression during a flight are still ongoing.

We determine the amount of battery necessary to return to the home position by multiplying how much battery is required to move one meter by the current distance to the home position, plus the minimum battery level. If the required battery level to reach the home position with minimum battery is greater than the current battery, the drone is replaced. If the drone is already returning to the home position when it reaches that battery level, it will not be replaced.

The battery level in jMAVSim decreases linearly through time after being armed, independently of the command it is executing, and will not go lower than a predefined value. Once the simulated drone lands and disarms, the battery will go up to 100% again. By default, the minimal battery percentage is 51%, and the interval to drain all the battery is 60 seconds. These values are not adequate to test this scenario since the battery would stop decreasing after less than 30 seconds. As such, we changed the simulator battery parameters, with the minimal percentage being 38%, and the full drainage interval set to 180 seconds, which means that we have around 110 seconds until the battery reaches 38% and stagnates. Drainage value that we may choose while testing do not hold any particular meaning, since we are not dealing with accurate battery simulation. Those are only an example to demonstrate the algorithm within the battery simulation time window.

5.4.3 Relay bridge

When executing a mission, it is expected that the drone will need to move far from the ground station. If the drone moves beyond the point where it still can maintain connectivity, we will no longer be able to monitor it. Since the mission control is centralised, we will not be able to send further commands through the platform. To avoid this situation without impacting the mission's behaviour, we can send an auxiliary drone to stand between the main drone and the ground station and act as a relay. As the drone moves further away, this could be done recursively, forming a bridge of drones between the main drone and the ground station. To implement this behaviour, we developed a plugin script that handles the monitoring of the distance to the ground station, dispatching drones when necessary.

Script 5.8 contains the relay plugin configuration. It has three parameters that have to be configured - `gs_coords` sets the ground station coordinates; `target_dist` defines the approximate target distance to maintain; `step` is the minimum distance between the next and current target positions to update the drone's current target. Additionally, two parameters have default values, which are related to the takeoff altitude of the relay drones - `base_alt` is

the lower takeoff altitude that a drone may have, while the `alt_diff` indicates the difference between the drones that fly lower and higher. There are three internal variables - `n_drones` counts the number of drones in relay duty, `being_followed` contains the drones that are being followed by a relay drone, and `pending_drones` will reference relay drones that are no longer required but are still assigned to the mission.

Script 5.8 Relay plugin configuration

```
plugin {
    id 'relay'
    input target_dist: Distance,
          step: Distance,
          gs_coords: Map,
          alt_diff: [Distance, 3.m],
          base_alt: [Distance, 5.m]
    vars being_followed: [],
         n_drones: 0,
         pending_drones: [:]
    callback monitor_gs_distance
}
```

We will advance to explain the relay algorithm. Compared to the previous examples, this one has a much higher complexity. As such, it will be analysed in shorter sections so that it is easier to understand, but we will ommit some segments of code which are not as relevant.

Script 5.9 Dispatch new drone

```
def dispatch_drone = { requirement ->
    def drone
    if (pending_drones.isEmpty()) {
        drone = assign requirement
        arm drone
        takeoff drone, n_drones.even ? base_alt + alt_diff : base_alt
    } else
        drone = reassign_relay_drone()
    n_drones++
    return drone
}
```

Script 5.9 is called to assign a new relay drone to the mission. The takeoff altitude is alternated, with the first drone flying at an altitude equal to the baseline plus the provided altitude difference. The following drone will take off to the baseline altitude, and this will alternate for each drone that joins the relay bridge. This reduces the chance of drone collision. It may happen that a new relay drone is required, but there is still one that recently stopped

being necessary and is in the middle of returning to its home position. In that case, instead of finishing removing that drone of the mission and requesting a new one, it will return to relay duty. When a relay drone is no longer necessary due to the followed drone getting close to the ground station, it is sent to its home position.

Script 5.10 Monitor movement towards ground station

```
def is_moving_further = { drone ->
    drone.cmd == move
            && drone.distance(gs_coords) < drone.cmd.distance(gs_coords)
}
def is_returning_to_gs = { drone ->
    drone.distance(gs_coords) < target_dist && !is_moving_further(drone)
}
```

Two auxiliary closures exist to aid in calculating where the followed drone is moving, which are available in script 5.10. We consider that the drone is moving further from the ground station if it currently is executing a move command and its distance to the GS is less than the distance of its command to the GS. If the drone is not moving further from the ground station and the distance to the GS is less than the target distance, then we consider that it is returning to the GS.

Script 5.11 Monitor distance to ground station

```
def requires_relay = { drone ->
    def takeoff_time = n_drones.even ? 15 : 10
    drone.cmd == move && drone.cmd.distance(gs_coords) > target_dist
      && drone.distance(gs_coords) + takeoff_time*drone.speed > target_dist*0.75
}
def monitor_gs_distance = { drone ->
    if (!being_followed.contains(drone.id) && requires_relay(drone)) {
        run { follow(drone) }
        being_followed += drone.id
    }
}
```

After a telemetry update, the distance between a drone and the ground station is verified. This is accomplished by the `monitor_gs_distance` closure in script 5.11, which is provided as the callback of this plugin. If a relay drone is not yet following the current drone, it will verify if one should be dispatched. A new relay drone will be dispatched if the current drone's command is beyond the target distance from the GS, and if the distance that the drone would be from the GS if another one were dispatched at that moment is greater than 75% of the target distance. If that condition holds, another relay drone will be dispatched, and the current one will be added to the list of drones that are already being followed.

Script 5.12 Calculate next position

```
def calc_new_target(other, self) {
    def new_target, dist = self.distance(other.position)
    if (dist < target_dist && is_moving_further(other)) {
        if (dist > target_dist / 2)
            new_target = other.target(other.distance(gs_coords) / 2,
                                                    other.bearing(gs_coords))
        else
            new_target = self.position
    } else
        new_target = other.target(target_dist, other.bearing(gs_coords))
    new_target
}
```

Script 5.12 defines the closures that calculate a relay drone's target position. When the followed drone is moving further from the ground station and the distance between both drones is higher than half the target distance but still lower than the target distance, the target position for the relay drone will be halfway between the GS and the followed drone. If the drones' distance is less than half the target distance, the relay drone's target position is the current one. When the followed drone moves towards the ground station or moves further from it while the distance between both drones is higher than the target distance, the relay drone's target position is at the target distance from the followed one.

Script 5.13 Drone following algorithm

```
def follow(other) {
    def new_target, new_speed, new_bearing
    def self = dispatch_drone(other.position), curr_target = self.position
    while(other.state == active || other.state == hold) {
        if (is_returning_to_gs(other)) {
            cancel self
            being_followed.remove(other.id)
            break
        } new_target = calc_new_target(other, self)
        if (new_target.distance(curr_target) > step) {
            curr_target = new_target
            new_speed = calc_new_speed(other)
            new_bearing = calc_new_bearing(other, self, curr_target)
            run { move self at new_speed to
                lat: curr_target.lat, lon: curr_target.lon, yaw: new_bearing}
        } wait other
    } revoke_relay_drone(self)
}
```

When a relay drone is dispatched, a subroutine is launched in which it will follow the previous drone. The behaviour that it exhibits is that of the `follow` closure in script 5.13. If the followed drone is no longer active, holding its position on air, or it is detected that it is returning to the GS, the relay drone will return to its home position. Otherwise, it will continuously calculate what its target position should be, as explained previously. However, if the new target position is too close to the current target position, with a distance less than the `step`, then the target will not be updated. This avoids the drone frequently cancelling the current command and issuing a new one that is only a few centimetres ahead. Otherwise, a move command to the new target position is issued. The drone's new bearing will be towards the followed drone when moving further from the GS, or towards the GS otherwise. The new speed is relative to the followed drone's speed, but will not surpass 10 meters per second.

5.5 Summary

This chapter addressed the requirements for a mission planning framework and implemented a solution that covered them. This framework is tightly integrated with the platform that we developed in chapter 4. Using Groovy, we were able to develop a DSL for describing a mission for a fleet of aerial drones. This allows the user to write a mission script with commands close to natural language and use the underlying language for the base syntax. In a mission script, the user can assign any number of available drones and instruct them to perform commands according to the restrictions they see fit. Multiple drones may be controlled in the same mission, either working independently or collaborating towards a goal. The same mission may yield different results if executed in different conditions, since the mission script may include decisions subject to the context, such as responding to a sensor reading. The framework functionality was extended by introducing the concept of mission plugins, which allows the user to enable available mission modules to run as a background task during a mission.

Throughout the chapter, small examples were provided to illustrate how to use particular features of the framework. To conclude, we also presented examples with higher complexity to demonstrate how it is possible to use the framework to address specific scenarios. These considered tracing a fire perimeter, replacing a drone during a mission due to low battery, and building a bridge of relay drones to provide connectivity.

Chapter 6

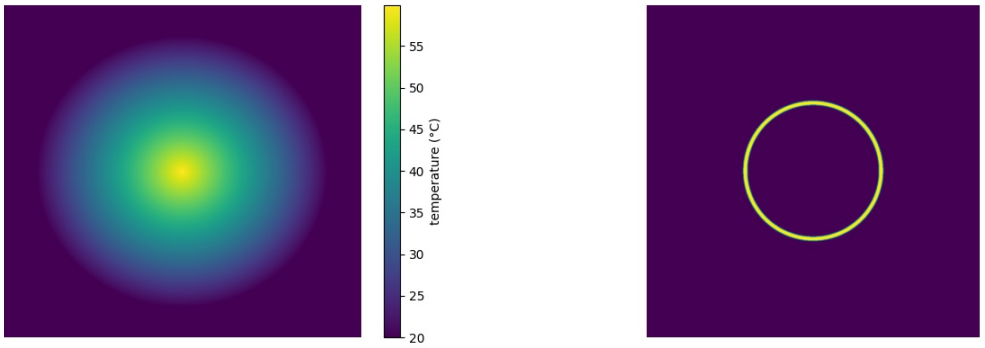
Experiments

The experiments conducted to test the developed mission framework in a simulated scenario and on physical UAVs in a real environment are detailed in this chapter. We start by presenting the results of the last mission examples from chapter 5. We then describe the context of the experiments with physical UAVs, such as the hardware specification of the UAVs and the ground station computer, the scenarios being deployed, and how the tests will be performed. Finally, we present the results of these last experiments, compare those with the simulation, and discuss our findings.

6.1 Simulated experiments and results

The last three mission examples in chapter 5 depicted interesting scenarios which already presented some level of complexity. As such, we will execute those missions and present the results. We used the same simulation setup as in chapter 4, and combined ground station logs with telemetry data stored to generate the relevant plots and figures for each mission.

6.1.1 Fire perimeter tracing

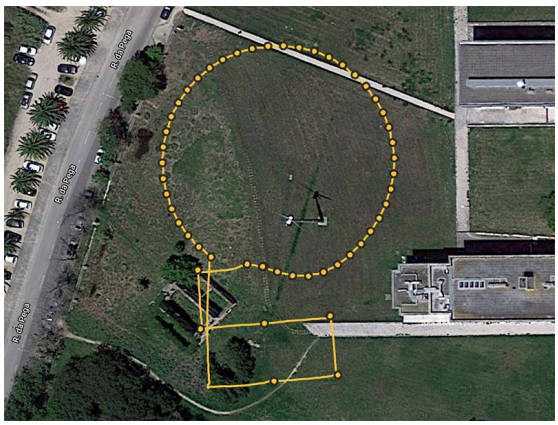


(A) Topology of the fire with one heat source

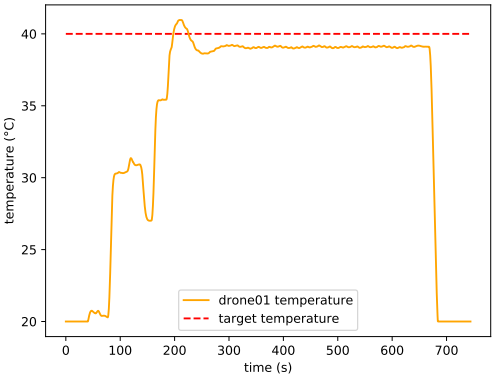
(B) Temperature outline at 40°C with one heat source

FIGURE 6.1. Topology and perimeter of fire with one heat source

We executed the mission described in script 5.5 using the temperature sensor configuration from example 21, with a single heat source. In figure 6.1, we can observe the heatmap that was generated with this configuration. The results of this mission are shown in figure 6.2. The drone is initially mapping the area until the temperature surpassed the threshold, and we can observe that the path correctly describes a circular perimeter afterwards. The registered temperature is also coherent, with a slight increase over 40°C when it first detected high temperatures, and then consistently maintained it just below the target.



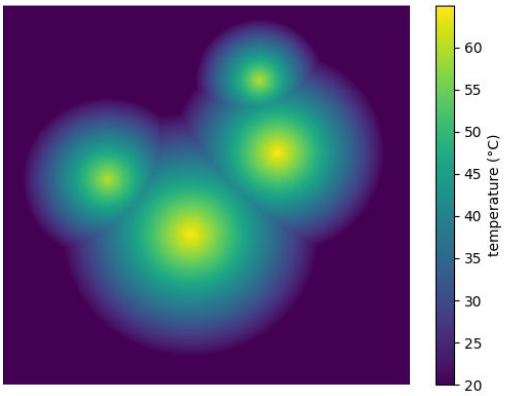
(a) Drone path



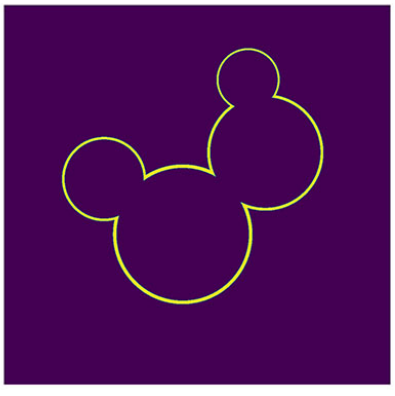
(b) Registered temperature throughout the mission

Figure 6.2. Drone path and registered temperature with one heat source

We also executed another mission in which we modified the temperature sensor simulator configuration to include four contiguous heat sources. This configuration generated the heatmap in figure 6.3, in which we observe a more complex perimeter. It is also possible to notice the shortages in the temperature calculation for multiple heat sources. The transitions should be smoother between them, which would avoid sharp edges like those in figure 6.3b.



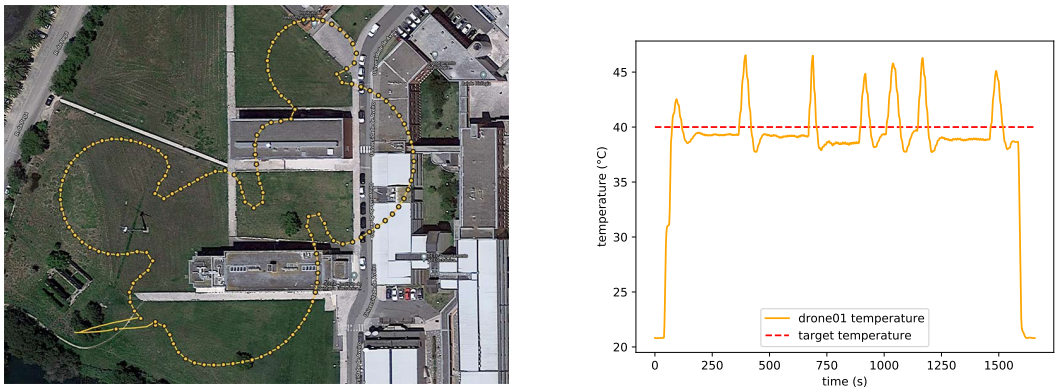
(a) Topology of the fire with multiple heat sources



(b) Temperature outline at 40°C with multiple heat sources

Figure 6.3. Topology and perimeter of fire with multiple heat sources

The results in figure 6.4 demonstrate that, although there is some accentuated error around the edges between heat sources, it is still possible to discern the approximate perimeter of the fire. This can be a result of the simplified temperature simulation, and not of the perimeter tracing algorithm itself. Similarly to the previous example, the temperature surpasses the limit when the abnormal temperature is first detected, but then occurs multiple times after that, at each edge, before it corrects its own path.



(A) Drone path

(B) Registered temperature throughout the mission

FIGURE 6.4. Drone path and registered temperature with multiple heat sources

Overall, the results of the fire perimeter tracing are close to the defined path, considering that the location is not defined, just a set of sensors that provides the location adjustment.

6.1.2 Drone replacement

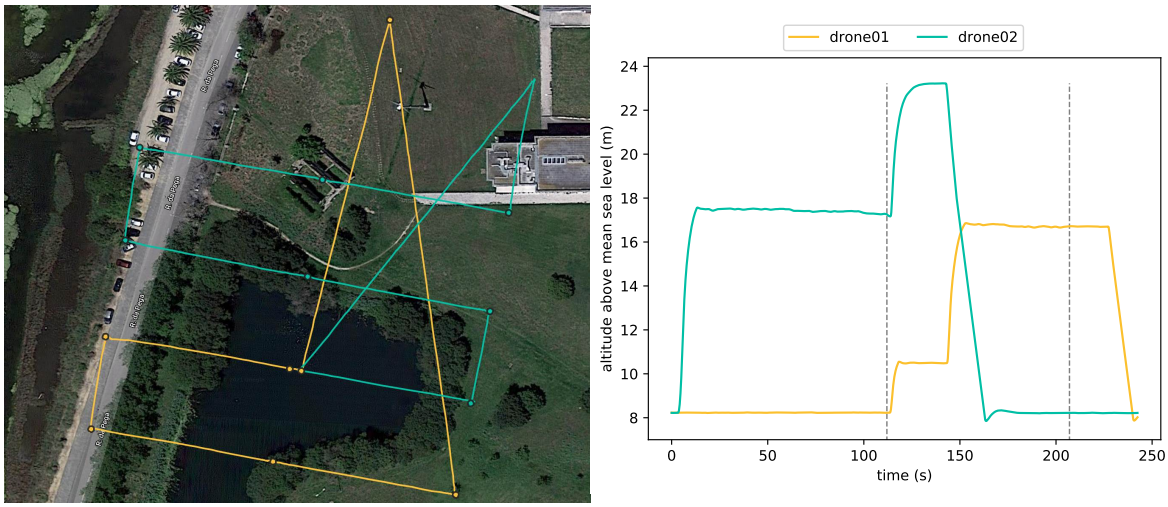
The mission script used to test the drone replacement algorithm through the plugin is depicted in script 6.1. We also used a mapping mission as a base to demonstrate the replacement algorithm. The mapping algorithm is the one used in script 5.1, and the `summon_drone` method will assign any drone, arm it, and order a takeoff. The `drainage` parameter was changed to 0.0015, meaning that we assume that 0.15% of the battery will be used for every meter, roughly 6.66 meters for a 1% drop.

Script 6.1 Mapping mission with replacement plugin enabled

```
1 enable 'replacement', drainage: 0.0015
2 drone = summon_drone()
3 finish_map = run {map_area(drone, 100.m, 75.m, 2, 3, [lat: 40.6330, lon: -8.6604)]}
4 wait finish_map
5 move drone to lat: drone.home.lat, lon: drone.home.lon, speed: 10
6 land drone
```

Figure 6.5 depicts the position variations of both drones assigned to this mission, the first drone used in the mission being depicted as green and the second drone as yellow. The

first dashed vertical line in 6.5b marks the time when the replacement occurred, while the second line indicates when the second drone finished the mapping and started returning to the launch position. The first drone had an altitude increase because it was sent back with the return to launch command, which caused it to raise its altitude to avoid collisions, while the second drone was sent to its home position with the move command.

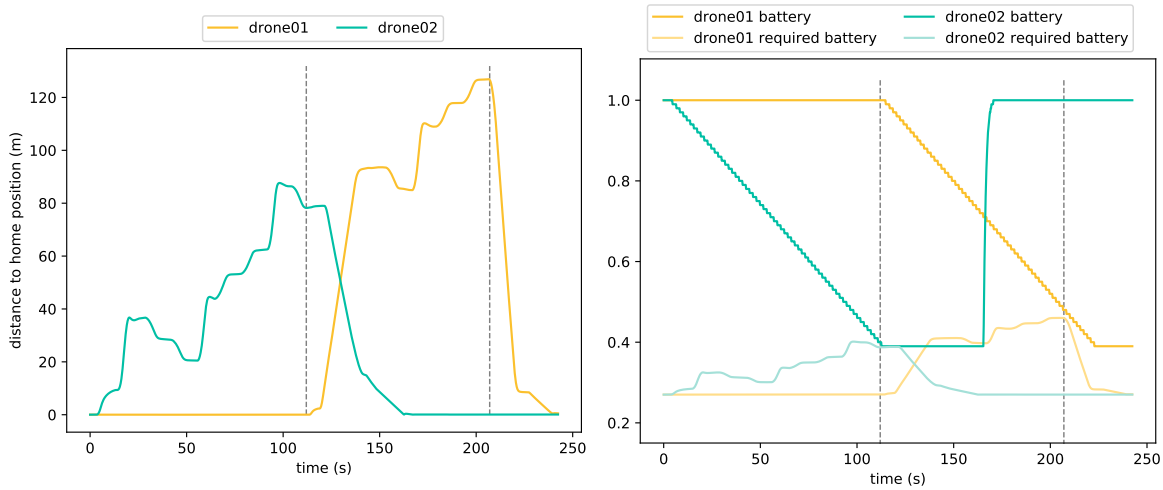


(a) Drone path

(b) Altitude

Figure 6.5. Drone path and altitude during replacement mission

Figure 6.6a shows the distance to each drone's home position, and figure 6.6b includes the battery level throughout the mission and the required battery level to return to the home position at a given time. In both figures, the vertical dashed lines represent the same events as in figure 6.5b. We are able to see how the simulated battery level stops decreasing at around 40%, and in the case of the first drone, increases back to 100% after landing.



(a) Distance to home position

(b) Measured and required battery level

Figure 6.6. Distance to home position and battery level

Looking at figure 6.5, we are able to visualise that the mission was successful in terms of achieving its goal. The area was effectively mapped with two drones, with the same resulting path as if there had been only one. The second drone successfully completed the task of the previous drone. Due to their home positions being relatively far from each other, the drones were not at risk of collision during the replacement. By analysing figure 6.6, we can observe that the replacement was activated at the right moment, when the battery level of `drone02` was lower than the required battery to return home safely. The second drone did not need to be replaced, as it was already returning to the home position when its battery level dropped to values closer to the threshold. The integration of the plugin behaviour into this mission was seamless.

6.1.3 Relay bridge

The relay bridge is an algorithm that brings a lot of value to the platform. By enabling the relay plugin, we are able to maintain connecitivty with the ground station even if the drones move far from it. To test its performance, we developed a mission script in which the main drone was only concerned with mapping a remote area, while the distance to the ground station was continuously monitored by the plugin to dispatch a relay drone when needed. Script 6.2 is the base mission for this test. The relay plugin is enabled with the provided parameters and drone *drone01* is dispatched. It will then execute the mapping algorithm, and once it is over, it will return to its home position and land.

Script 6.2 Mapping mission with enabled relay plugin

```
enable 'relay', target_dist: 50.m, step: 4.m, alt_diff: 5.m, base_alt: 5.m,
                gs_coords: [lat: 40.634125, lon: -8.660336]
drone = assign 'drone01'
arm drone
takeoff drone
map_area(drone, 50.m, 50.m, 1, 3, [lat: 40.632963, lon: -8.660440])
move drone at 3.m/s to lat: drone.home.lat, lon: drone.home.lon
land drone
```

In figure 6.7 we have the resulting paths of each drone by the end of the mission, in which the white circle represents the ground station location. The path of the yellow drone shows the mapping algorithm, while it is possible to observe that each of the relay drones followed the previous one, drawing a path that is roughly similar to the first one's. As we know that the mapped area spans 50 meters, we can visually estimate that the drones were able to maintain an approximate distance of 50 meters as well, since the last horizontal segment of a relay drone is close to the position of the first horizontal segment of the drone it follows.

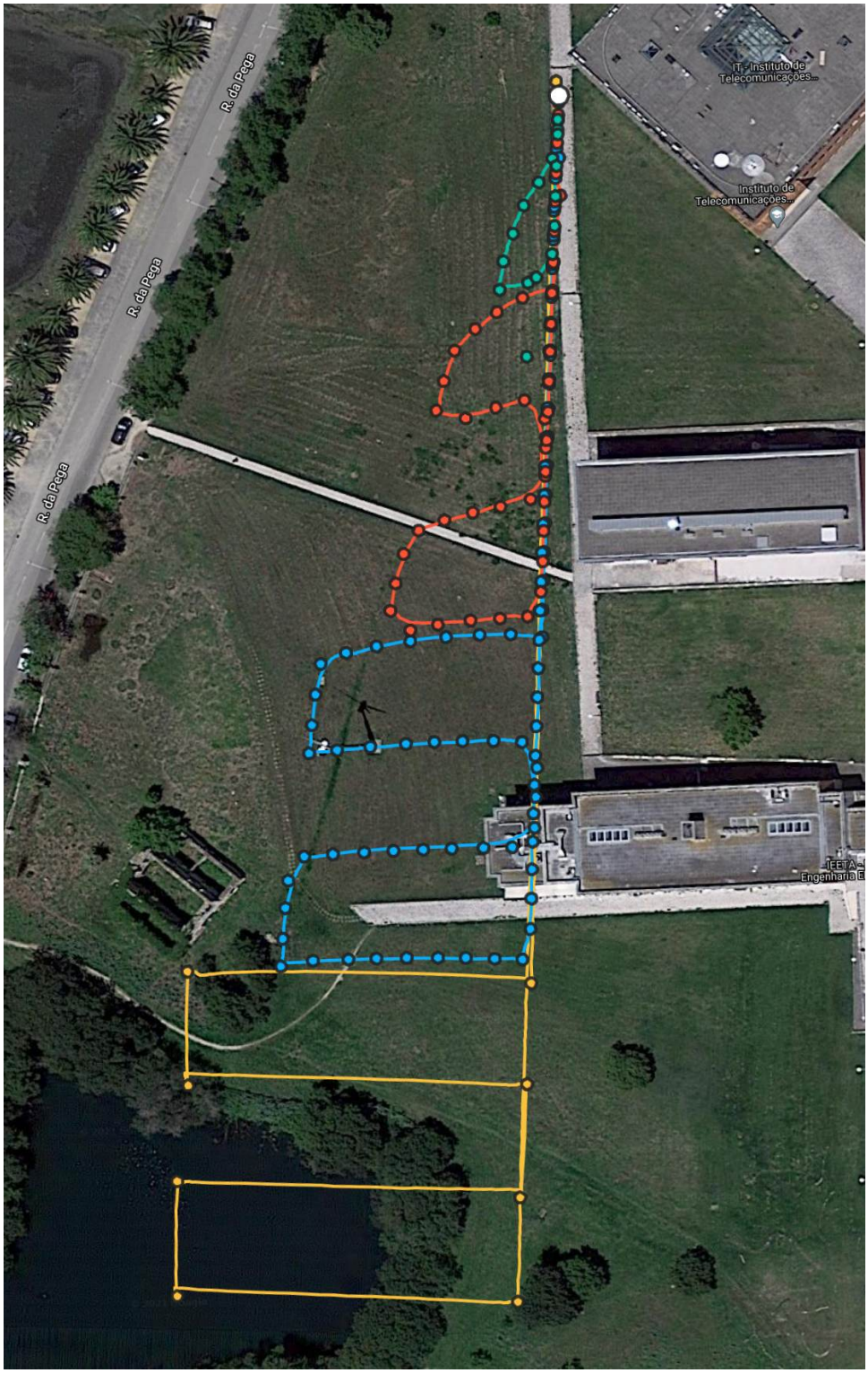
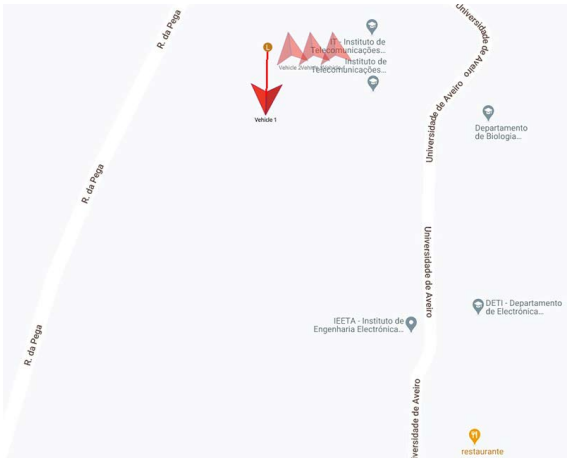
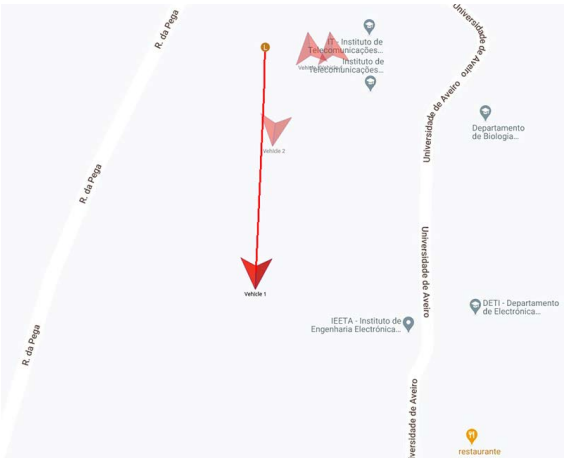


Figure 6.7. Path of multiple drones with three drones serving as relay

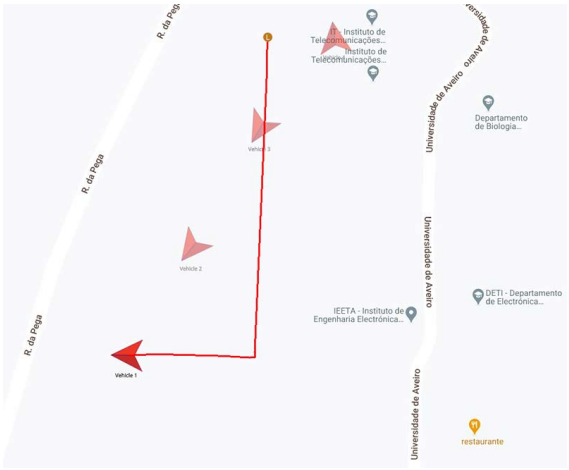
To better demonstrate how the several drones behaved while running the algorithm, we captured images of QGroundControl in key moments while the mission was running, which allows to visualise the progress of the mission state through time, as can be seen in figure 6.8.



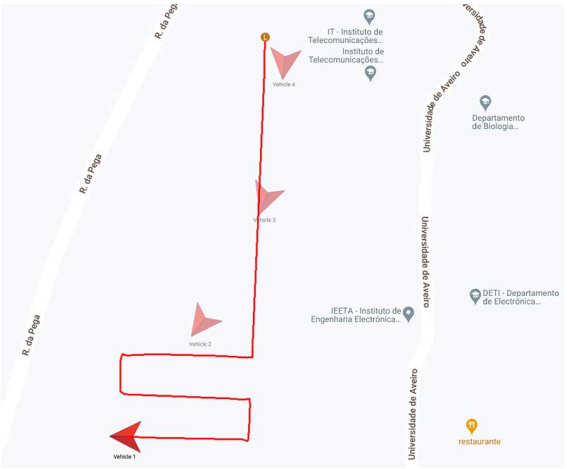
(A) First drone takes off and starts moving towards a waypoint



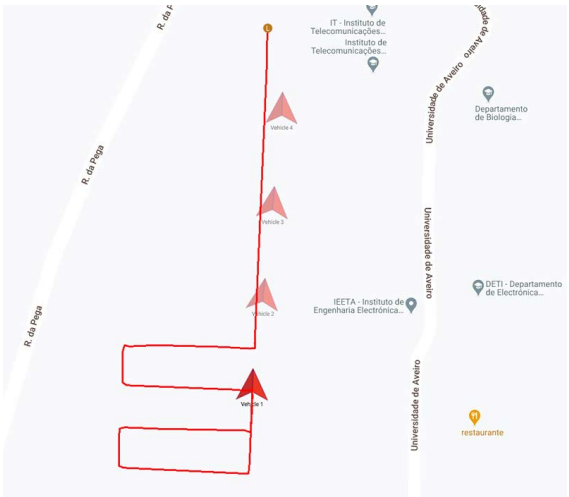
(B) The second drone starts following the first drone



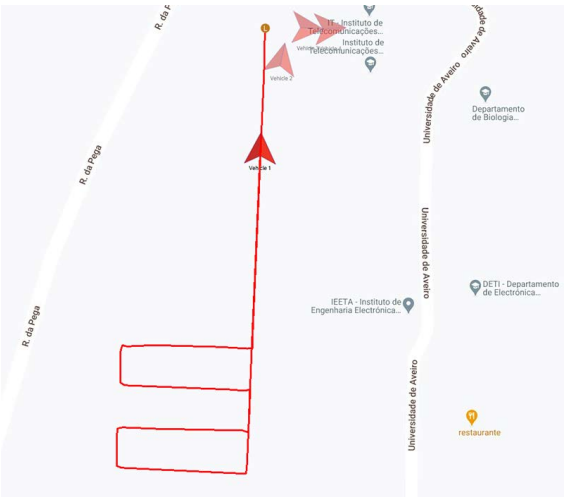
(C) The first drone starts mapping the area and the third drone follows the second drone



(D) The fourth drone follows the third drone



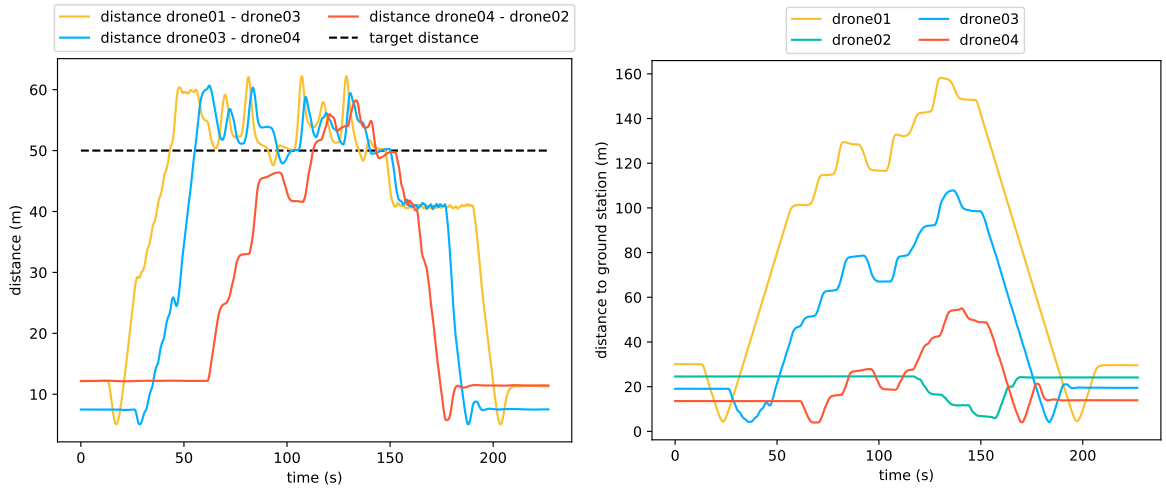
(E) The first drone finishes mapping and returns to launch; the relay drones move towards the ground station



(F) The relay drones land in their home position when the drone they are following is close to the ground station

FIGURE 6.8. Progress of a mission in which one drone is mapping an area and three drones serve as relay

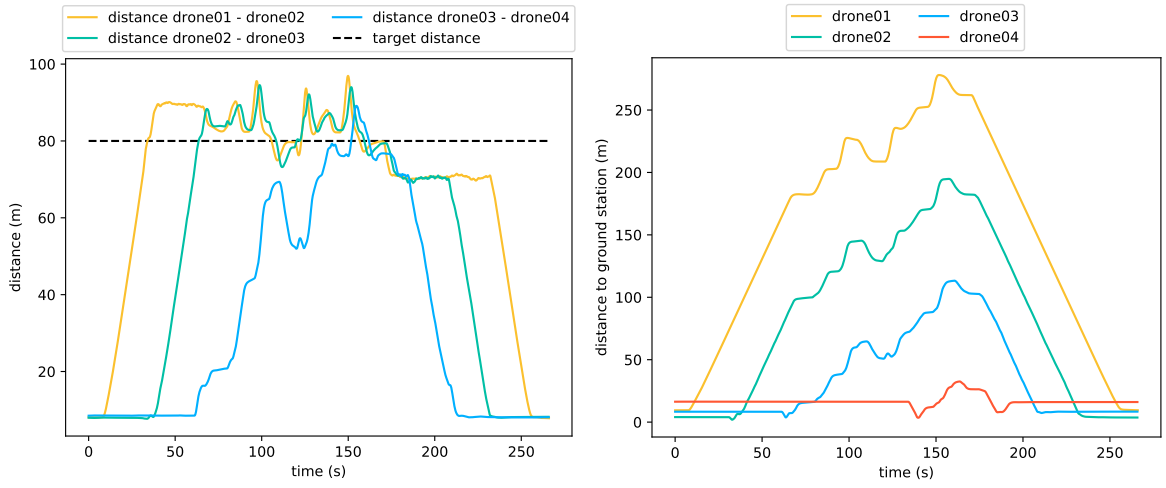
In figure 6.9a, we can observe that, although the distance between drones surpassed 50 meters, the fluctuations did not exceed it by a large margin. The target distance should not be seen as a hard limit that if surpassed would cause the drones to lose connectivity, but as an approximate distance that the drones should keep between each other. Regarding the distance from the ground station, as seen in figure 6.9b, it is apparent that the distance to the relay drones varies proportionally to the main drone's movement. We also executed a similar mission with a longer path and a target distance of 80 meters, which held similar results, as it shows in figure 6.10.



(A) Distance between drones

(B) Distance between drones and ground station

Figure 6.9. Distance between drones and between drones and ground station with a target distance of 50 meters



(A) Distance between drones

(B) Distance between drones and ground station

Figure 6.10. Distance between drones and between drones and ground station with a target distance of 80 meters

We can observe that the target distance is surpassed at the beginning of the mission. It is not trivial to predict the best moment to request another drone in a generic algorithm. One of the issues is that the time it takes for a drone to take off may vary and be influenced by external factors (e.g: wind), and an incorrect prediction may cause the drone to be requested to soon, draining the battery unnecessarily, or the drone may take too long to finish taking off, which may cause the main drone to be well past the target distance. Another issue is that we can not safely assume the path the main drone will take after the current command, or predict if the command will be cancelled by a sensor reading or other user-defined condition without tailoring the algorithm to each scenario. We may request a relay drone, and by the time it is ready to move, the main drone may be returning to launch and conclude the mission, rendering the relay drone unnecessary, or it may start to move so fast that the relay drone will not be able to reach the target distance.

This algorithm was conceived to ensure that a single drone was able to maintain connectivity throughout a mission. As such, it is not very efficient in scenarios with multiple drones executing a mission near each other. Since a chain of relay drones is spawned for each drone that moves too far away from the ground station, if two or more drones move close together, the algorithm will request redundant relay drones. A simple workaround could involve verifying if there is a surrounding drone that is already being followed by relay drones before requesting backup. Although this could minimise the number of summoned drones, it would not place the relay drones in an optimal position. To fully solve this, we would have to develop a more advanced algorithm that would be able to calculate the best distribution based on all drones.

Even though some aspects could be refined, this algorithm provides a good starting point towards guaranteeing connectivity in a mission. Overall, this was still successful, as we could observe that it was possible to fulfil the purpose of maintaining connectivity during a mission with a script. We will assess this in a real scenario in the next section.

6.2 Experiments with physical UAVs

In this section, we provide context on the experiments that we conducted with the physical UAVs. We begin by specifying the hardware that was used during these experiments, followed by presenting the tests cases and scenarios, and explain how we prepared and executed those tests.

6.2.1 Hardware specification

Multiple UAVs were used to test the performance of our solution with physical drones. The physical components differ slightly between some of the drones. However, they follow common specifications, as described in table 6.1. Figure 6.11 depicts the used UAVs.

Figure 6.11. Hexacopters used for our experiments

Table 6.1. Drone hardware specifications

	Drone hardware
Flight controller	Pixhawk 4 + NEO-M8N GPS receiver
Frame	Hexacopter
Radio Transmitter and Receiver	FLYSKY FS-i6 + FLYSKY FS-iA6B
Single Board Computer	NVIDIA Jetson Nano
Operating System	Linux4Tegra
Wireless USB Adapter	TP-Link TL-WN722N

The FC is connected to the SBC's serial port. The wireless USB adapter allows the SBC to be connected to the ground station and the other UAVs through an ad-hoc network. The radio transmitter is used to assert that the drone is stable before starting an autonomous flight. Moreover, during the tests, the radio transmitter was also kept close to the ground station location, to be used in case the drone was at risk and needed manual intervention.

As a ground station, we used a laptop with the ground station modules installed, which has the specifications mentioned in table 6.2. Since the ground station and the dashboard software can be launched with Docker, any operating system could have been used instead.

Table 6.2. Ground station specifications

	Ground station
Processor	Intel Core i7-8565U, 1.8GHz
Memory	16 GB, 2400 MHz
Operating System	Arch Linux 64-bit
Wireless USB Adapter	TP-Link TL-WN722N

6.2.2 Test cases

Previously, we validated the software by executing tests in a simulated environment. These were relevant to rapidly detect and correct some issues while the platform was being developed without damaging the drones. However, it is important to validate that the mission framework is safe and suitable for use in real-life scenarios. These experiment's main goal is to evaluate this platform's functionality when using physical UAVs.

Planned missions

Most of the tests were performed outside at the campus of Universidade de Aveiro. These missions were already tested in the simulated environment; thus, we can easily compare if the performance is similar to the one already observed in the simulations.

1. *Mapping* - Before progressing to more complex missions, we will execute a linear mission involving only one drone to assure that the system is safe to use. In this mission, the drone will map an area of 15 by 15 meters using the algorithm described in script 5.1.

2. *Drone replacement* - This mission is an adapted version of the basic drone replacement algorithm presented in script 5.2, in which the replacement is triggered 40 seconds after the first drone finishes the takeoff. We chose to trigger the replacement with a timeout instead of checking the battery level, because it is not easy to predict how fast it will deplete in a physical drone. In this mission, the first drone will map the area like in the previous mission, but returns to the launch position after the timeout. The mission script instructs the second drone to wait until the first drone starts landing before taking off, which minimises the chance of an unexpected collision. The second drone will then move to the place where the previous drone had stopped, and then proceeds to conclude the area mapping.

3. *Relay bridge* - We tested the relay bridge algorithm that was previously presented, but using less drones. The main drone had to navigate through a set of waypoints scattered throughout the area, while the relay drone followed it when the distance to the ground station increased, with a target distance of 25 meters. We chose this target distance because it allowed us to visualise the algorithm's behaviour without sending the main drone too far from our sightline.

4. *Drone following* - This mission was executed in a different location, in Trancoso. One drone will follow a predefined set of waypoints. The second drone will take off and move closer to the first drone, and after that, it will move to the waypoint the first one is going to, but keeping some distance.

Measurement of the network range

Besides testing the mission performance, we also executed an experiment to retrieve metrics for the relay algorithm. This experiment is relevant to calculate an estimate of the distance

that the UAVs could have from each other or the ground station while still maintaining a reliable network connection. In the future, we can use the results of this experiment to adjust the relay algorithm's target distance more efficiently. These metrics can be obtained by monitoring the network performance while moving a drone further away from the ground station. They can also be used in the future for drone-to-drone distributed missions.

6.2.3 Method

We will clarify the preparation process of the tests, the gathering of the data and results, and the conditions in which we executed the experiments.

Preparation

Each of the missions was tested in the simulator beforehand to preserve the integrity of the drone, which was also useful to retrieve data to which we can later compare the experimental results. In the case of the missions executed on the university campus, these were simulated in coordinates close to the prospective real location. By monitoring the map while simulating the missions, we can anticipate if the provided parameters can lead to a drone hitting a tree or a lamp post.

The tool we chose to monitor the network performance is iPerf3[64], which allows to measure the maximum achievable throughput on an IP network. This tool was also tested in advance to confirm that it provided the expected measurements in a wireless network in different conditions. We also developed some helper scripts to accelerate the experiment configuration on the field.

Data gathering

Similarly to what was done throughout the simulated tests, during the tests in the university campus we recorded the rosbags in the ground station while the missions were running and stored the GS log files. This data will be later combined to generate plots with the results. In Trancoso, the PX4 flight logs were retrieved and analysed instead.

In the experiment in which we measure the network performance, we recorded the rosbags in the ground station, while simultaneously exporting the `iperf` output to a file. This allows to measure the distance from the drone to the GS later on, and compare how the bitrate evolved through time (and consequently, distance).

Setup

Before launching the drone module, the SBCs were mounted on top of the drones and connected to the flight controller. The laptop computer, which was also transported to the field, had the ground station and the dashboard software running. Both the SBCs and the GS were connected to the same ad-hoc network. The ground station was configured to act as

[64]https://iperf.fr/.

a NTP server, and the drone's SBC was synchronised with this server (the SBC does not have a real-time clock and needs to be adjusted on boot). RCs were nearby to be used in case the drone required manual intervention to return safely. To detect any apparent anomaly in the drone's stability, we manually controlled each drone before executing autonomous missions.

Before starting a mission, the drones were placed in an appropriate initial position. Using the dashboard, we submitted the desired mission scripts. While the missions were running, we would carefully watch the drones and the dashboard's map and drone status to monitor the mission progress and possible hazards.

For the network performance experiment, the drone was carried by one person from the ground station location, which was fixed, and continuously walk further away from it in a straight line. Since the ROS2 middleware uses UDP, we configured the `iperf` client in the drone to send a UDP stream, with a target bitrate of 12Mbps. This bitrate is much higher than what we expect from the drone module alone. However, we have to keep in mind that there could be additional network traffic like video streaming or multiple network-intense sensors in a real scenario.

6.3 Results and discussion

Once the proposed experiments were concluded, we analysed the data that was collected and produced the plots and maps presented here. In this section, we present and discuss the obtained results. We will also compare those results with the ones obtained by running the same mission in a simulated scenario. It is important to note that there may be some slight differences in the resulting paths, since the starting coordinates were not exactly the same.

6.3.1 Missions

Mapping

(A) Physical drone's path

(B) Simulated drone's path

FIGURE 6.12. Drone path for the mapping mission

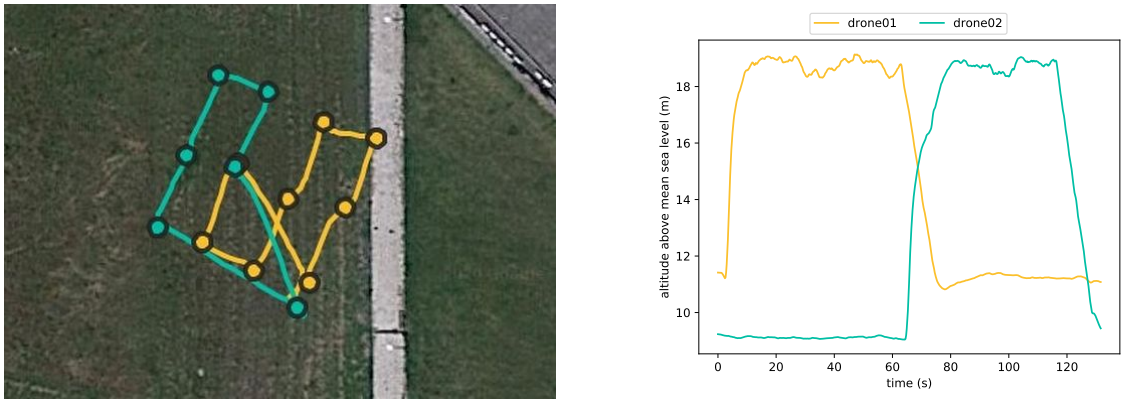
In figure 6.12a, we observe the result of the mapping algorithm. The drone was placed in the lower right corner before the mission started, and was approximately facing what is the left side in this orientation. The area in front of the drone was mapped left-to-right, bottom-to-top, and the drone returned to the launch position, without encountering failures. Comparing to the resulting path of the drone in the simulated scenario, in figure 6.12b, we observe that they are similar.

Drone replacement

The first attempts at running the drone replacement mission were unsuccessful, as shown in figure 6.13. At first, this was caused by a motor that was malfunctioning and had to be replaced. Afterwards, the second drone nearly completed the mission, but right before the last waypoint, we lost connectivity with that drone in particular, and as expected, the ground module automatically cancelled this mission. This incident was likely related to that specific wireless USB adapter, whether it was malfunctioning or it was poorly placed or attached to the drone. The drone was not far from the ground station at that moment, and we did not witness more instances of losing connectivity at such a short distance, from which we conclude that this event was an outlier.

Figure 6.13. Drone path during mapping with replacement - unsuccessful

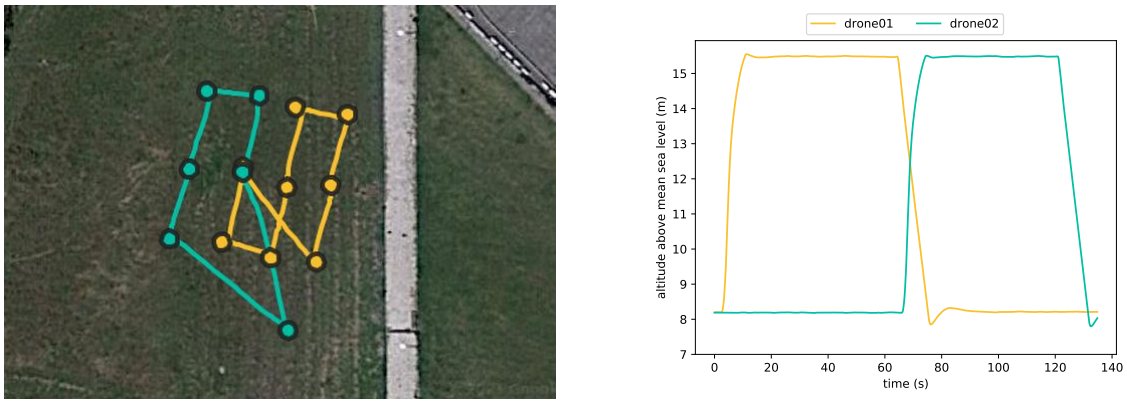
We repeated this experiment on another day and had favourable results, which we portray in figure 6.14. The first drone, portrayed in yellow, returned to the launch position once it reached a waypoint after being on air for 40 seconds. The second drone moves to the position where the previous one stopped and finishes mapping the area successfully. By observing the altitude variation, we can notice that the second drone only started taking off after the first one was landing. Figure 6.15 depicts the results obtained by executing the same mission in a simulated environment. We can observe that the results from both environments are similar in terms of the traversed path. The greatest difference relies on the altitude variation throughout time, which is expected since the simulation provides a constant altitude.



(A) Path of both drones

(B) Drone altitude throughout the mission

FIGURE 6.14. Drone path and altitude variation during the replacement mission



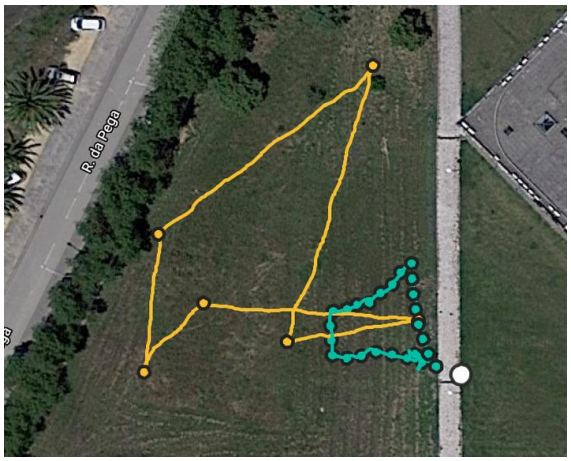
(A) Path of both drones

(B) Drone altitude throughout the mission

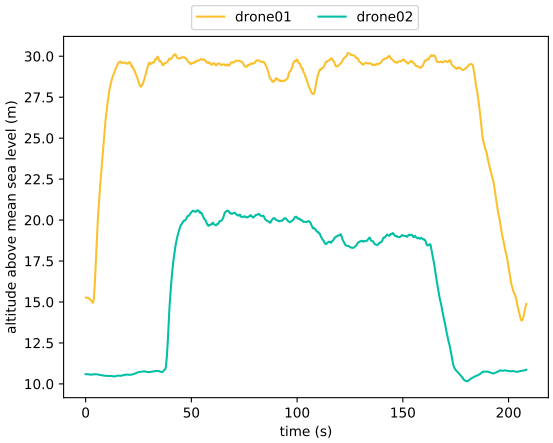
FIGURE 6.15. Results of the simulated replacement mission

Relay bridge

The final mission we executed at the university campus was the relay bridge. The path of both drones is depicted in figure 6.16. In this experiment, we only used two drones, with the yellow path portraying the main drone, which travelled to predefined waypoints, and the green one serving as a relay drone. The white circle represents the ground station location. As we can see, the relay drone maintained an altitude that is considerably lower than the main drone's, which was performed on purpose to prevent possible collisions if one of the drones failed to hold the requested altitude properly. These results can once again be compared to the results of the simulated mission, shown in figure 6.17. There is a slight difference in the simulated drone's path due to the different home position, and as in the previous example, the altitude was also kept at constant levels.

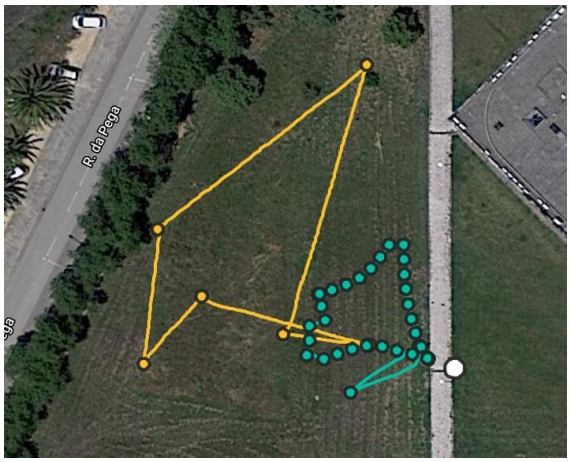


(a) Drone path during relay mission

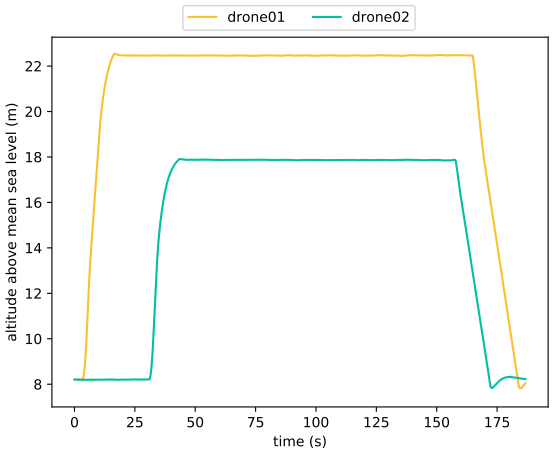


(b) Drone altitude throughout the mission

Figure 6.16. Drone path and altitude variation during the relay mission



(a) Drone path during relay mission

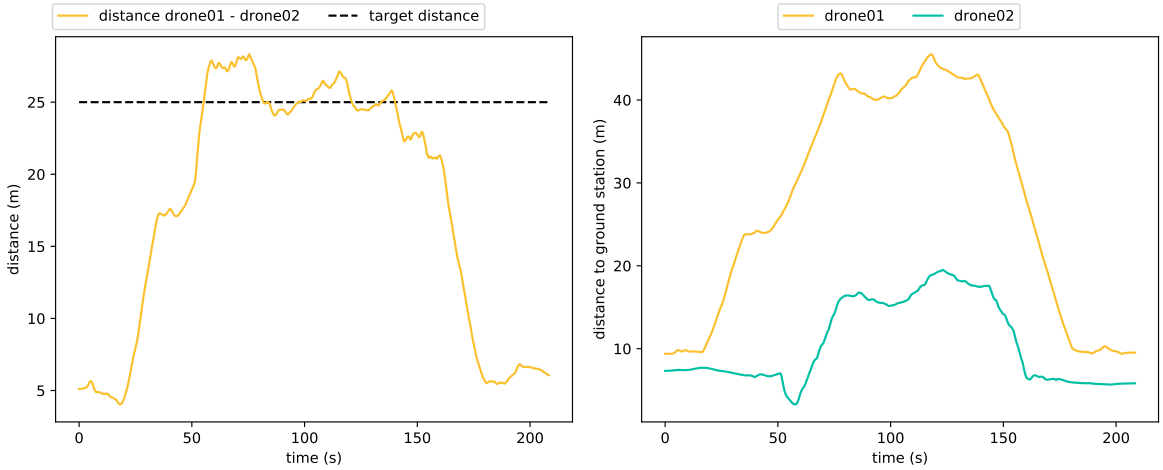


(b) Drone altitude throughout the mission

Figure 6.17. Results of the simulated relay mission

In figure 6.18, the results seem to demonstrate an equivalent behaviour to what we obtained in the previous relay experiment, although with only one relay drone. Comparing to the simulation results where there was also only one relay drone, in figure 6.19, we observe similar values. The difference in the distance between drones is due to the simulated drone being placed further away, compared to the physical drone's original position. This caused it to take longer to reach the target relay position. Overall, the distance between drones was close to the target distance, with some slight fluctuations above that mark. As expected, the relay drone tends to fly close to the ground station when the drone that it is following also moves closer. This behaviour is equivalent to what we saw in the example with three relay drones, in which this also happened to the last relay drone. Although this behaviour did not pose a threat in a simulated environment, it can cause some concern in a real-life scenario if the human operators are also near the ground station, as the drones may fly too close to them. Since the drones were working correctly and the drone kept a considerable altitude

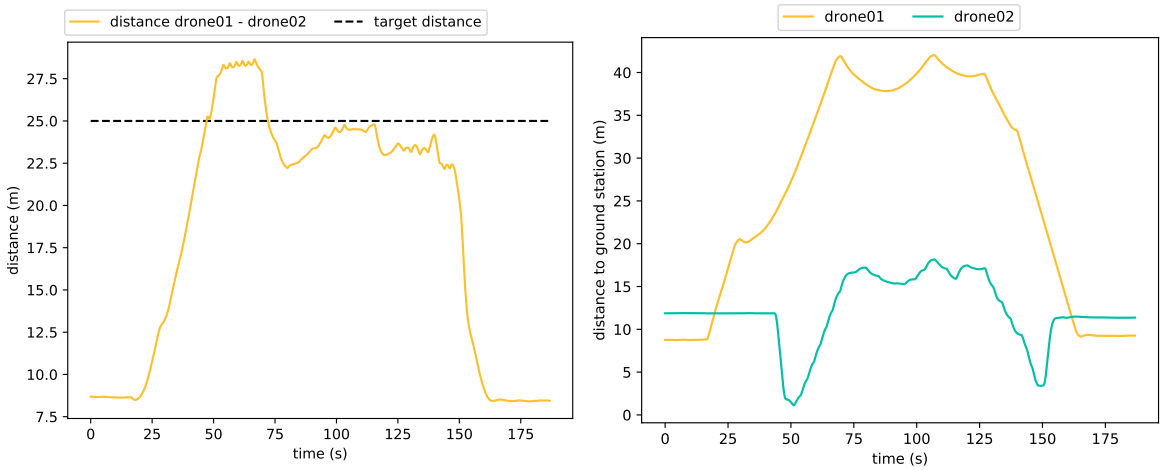
from the ground, we were not at risk, but if the drone had a failure which caused it to fall to the ground, it could have been dangerous. This situation should be taken into account in the future to increase safety in case of malfunction.



(A) Distance between drones

(B) Distance between drones and ground station

FIGURE 6.18. Distances between drones and ground station with a target distance of 25 meters



(A) Distance between drones

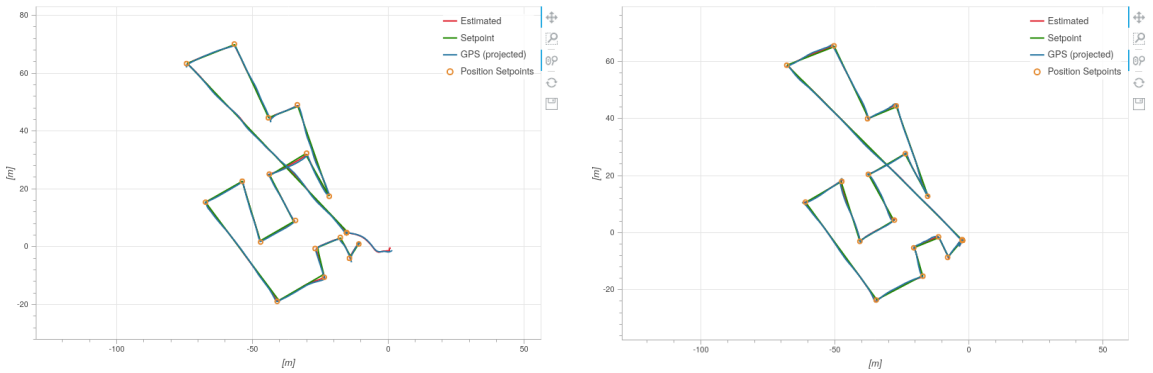
(B) Distance between drones and ground station

FIGURE 6.19. Distances between drones and ground station in the simulated relay mission

Drone following

This last mission was executed in a different location, in Trancoso. Prior to the mission execution, a set of coordinates were collected on the field, which were then used for the mission. The first drone followed the provided waypoints, while the second drone followed it behind. As we can see in figure 6.20, both drones executed an almost identical path. It is possible to observe some slight differences between the paths. Since this mission was executed in a real drone, this is expected, as there is always a small margin of error when moving to

a GPS coordinate. However, looking at the relative positions between the waypoints, those were, in fact, the same approximate path. A video was recorded during this flight[65], in which it is possible to observe the second drone moving to the waypoint where the first drone was previously in.

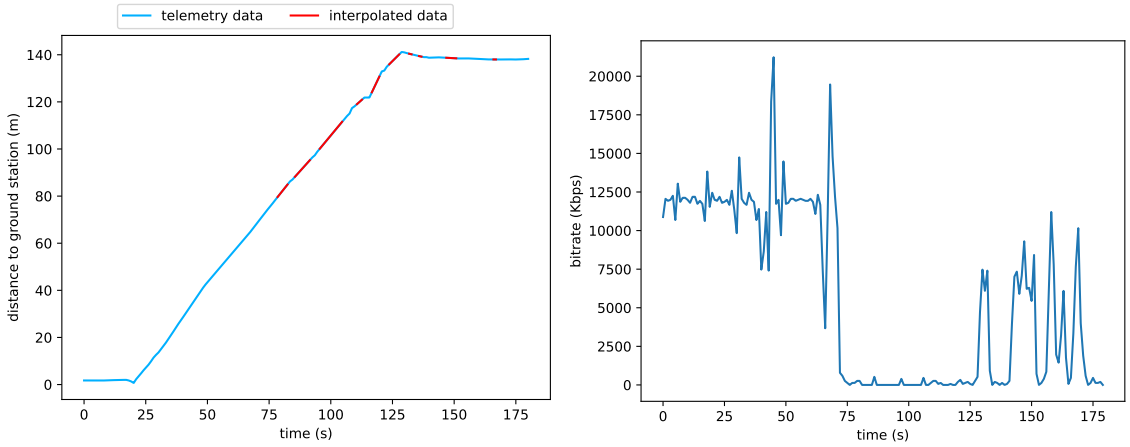


(A) Path of drone02 (B) Path of drone03

Figure 6.20. Path of the drone following mission

6.3.2 Network range

Finally, we analysed the network performance while increasing the distance between the drone and the ground station. Figure 6.21a depicts the registered distance between the drone and the ground station through time, and figure 6.21b depicts the bitrate it was possible to achieve. During some periods, telemetry messages were not received; thus, we implicitly determined the distance based on the previous and next available data.



(A) Distance between the drone and ground station (B) Achieved bitrate

Figure 6.21. Bitrate while drone moves from the ground station

Around the 75 seconds mark, after some instability, the network stopped being able to handle the requested bitrate, which happened when the drone was located around 80 meters

[65]https://www.youtube.com/watch?v=FMkmx-0iOtc

from the ground station. Since the telemetry data was recorded in the ground station, we can also observe that some ROS messages were lost. Fifty seconds later, although there was an increase in the bitrate, it remained considerably unstable, and ROS messages were still occasionally lost.

With this information, we can adjust future missions to be compatible with this limitation. For example, we may improve the relay algorithm to better fit distances of this magnitude. Also, we have to consider that, if we are not running a background application with a high bitrate, it is likely that less ROS messages would be lost.

Ideally, this experiment should have been followed by another relay mission that used a value between 60 and 70 meters as the target distance, combined with the monitoring of the network performance at the same time. However, this mission would have to be executed in a different place since our usual test location did not allow us to run a mission with these characteristics safely. At this distance, the drones would cross trees and buildings in their path. It would have also been relevant to retry this experiment with a lower bitrate. These experiments could not be performed in time to be presented in this Dissertation due to the present restrictions at that time.

6.4 Summary

While running these experiments, we had multiple drones performing autonomous flights successfully with the proposed platform. Although we encountered some issues during the tests, these were related to hardware problems and were solved.

When comparing the results of these experiments with their equivalent in simulation, we can observe that they present the same behaviour. With these results, we can show that this mission framework can be used predictably in real scenarios, given that the fleet has compatible UAVs. Looking back to the requirements that we defined in chapter 3, excluding the sensor integration which we did not demonstrate in a real scenario, we could fulfil and exhibit all of them.

Chapter 7

Conclusion

We conclude this Dissertation by making some final remarks on the work that was presented throughout this document. We also provide suggestions regarding improvements that could be implemented in the future.

7.1 Final remarks

This Dissertation introduced a novel drone control framework and mission planning solution tailored specifically for fleets of aerial drones. Most public, non-commercial solutions offer limited functionality for mission planning, typically consisting of a way to describe a linear mission flow for a predetermined aerial drone, where actions taken at each step are chosen from a non-extensible set. The framework that was designed and evaluated in this work overcomes those limitations and offers flexible mission planning, with non-linear missions that can change the course of a drone at any time during its sequence of actions while being able to request data from its own sensors and reacting to that data.

To accomplish this, we opted for a centralised approach where a ground station is responsible for the management and control of every drone in the fleet. For mission planning, we developed a DSL built on top of the Apache Groovy language. This DSL allows logical conditional plans and provides user-friendly methods to declare common actions such as taking off, moving, turning, retrieving sensor data, landing, and returning to the home position. Multi-drone collaboration is possible by providing access to some concurrency and synchronisation functionalities based on the GPars concurrency framework. With this, different subroutines may be launched to allow concurrent execution of different tasks, and the mission flow of one drone may depend on the state of another one. Missions can be extended by enabling mission plugins. In order to enable the automatic monitoring of events without having to declare them explicitly, we implemented monitoring plugins. This allows the user to import complex behaviour, such as the relay algorithm, without having to understand how it is implemented.

To validate the framework, we executed a number of experiments, both in simulated and real-life scenarios, that were designed to test various combinations of complex missions.

With the perimeter tracing algorithm, we could perceive how to handle a fire scenario. A simulated drone was sent to patrol an area while monitoring the temperature registered by a simulated sensor. After surpassing a certain temperature threshold, the drone would then trace the fire perimeter, which would allow firefighters or property owners to be aware of the fire's extension. The drone replacement plugin can be enabled in a mission script in order to continue a mission even if one of the drones registers a low battery level. By calculating the amount of battery necessary to return a drone to the launch position safely, we can replace it with another drone preemptively. Another plugin was also developed, the relay plugin, is helpful to assure that it is possible to maintain connectivity with the ground station. If a drone moves too far from the ground station, a relay drone will be dispatched to follow it and maintain an adequate distance to the ground station. This is executed recursively, forming a bridge of relay drones. Both of these plugins were also tested in the real scenario and performed similarly to the simulation.

A number of challenges had to be overcome during this Dissertation in order to arrive at this final architecture. The middleware robotics framework that we used, ROS2, was still under active development, which sometimes proved to be difficult. Documentation was not always clear, and not many examples were available. However, some features from ROS that were not already available in ROS2 were released during the development of this work, such as the rosbags, and we were still able to use them. The DSL design took many iterations to arrive at a user-friendly format that still enabled powerful, complex missions to be declared. Much time had to be dedicated to refine the details and achieve the current result. The restrictions that were lived during this Dissertation's development also limited our ability to evaluate the platform in real-life. This was possible to overcome by setting a convenient simulation environment in which we were able to test a number of different scenarios. Nevertheless, we still had the opportunity to conduct a significant set of experiments and confirm the correct behavior in the real world.

Different entities may benefit from this solution. First, the DSL can be targeted at users with different experience levels - less experienced users can easily define missions comprised of simple tasks, such as moving the drones and enabling helper plugins, while more advanced users can use it to develop solutions for complex scenarios. This framework can be particularly useful for fast prototyping of mission algorithms. If we want to approach a new scenario, such as a touristic application, we are able to focus completely on the mission plan since the remaining requirements are already met. This work could potentially be used to replace the need for multiple different solutions in slightly diverging scenarios, given that it is extended to include: more refined motion control, such as obstacle avoidance, improved sensor integration, and a GUI that enables the description of complex missions.

Overall, we were able to address the requirements that we proposed to solve. Comparing to the issues that we addressed in the discussion of chapter 2, we provided 1) a modular platform that allows controlling a drone remotely by sending commands; 2) a mission planning framework that enables the description of missions with several courses of action; 3) support for multiple drones collaboration in these missions; 4) a plugin system to define

monitoring tasks that can cause additional behaviour; 5) several mission examples, which were all possible to run in a simulated environment and, when possible, were also tested in a real scenario.

7.2 Future work

Since this work had to approach very different issues in order to build an integrated solution, we did not have the opportunity to further refine and improve all of the aspects, but it can be used to leverage additional functionalities that could not be addressed in this Dissertation. We identified the following features as relevant future work:

1. *Improve network connectivity*. Different routes could achieve this. To improve the relay algorithm, it should allow more efficient distribution of the drones, with one relay drone serving multiple other drones with optimal placement. It could also be adapted to include the concept of "router drones", which could provide connectivity to a remote ground station. One other aspect would be replacing the decisions based on distance with a more sophisticated approach, which could be implemented as a "network sensor" module running on each drone, responsible for informing the ground station that it may require assistance.

2. *Extend sensor support*. We only tested a simulated temperature sensor and only support embedded sensors assigned to a specific drone. Some sensors might be placed in a static position instead of attached to a drone. These types of sensors should also be accessible in the mission scope. The current sensor model is also very restrictive, as we assume that the sensor sends a single value without further information. It would be interesting if, instead of manually adding new sensor types to the code and assuming it only provides a value, we injected each sensor type's fields and characteristics through a configuration file.

3. *Efforts towards decentralisation*. Although some parts of the mission planning are better handled in a centralised system, some time-critical decisions should be delegated to each drone. It would be relevant to create a drone-side mission module, which could be coupled with a new concept of *task* in mission planning, representing a set of actions that are sent in bulk to the drone.

4. *Collision avoidance*. Collision avoidance, either by preventing crossing other drone's path or colliding with another object, is a crucial feature that needs to exist in such a system. That could avoid potential hazards of drones unexpectedly crashing into a tree or a wall. This could be implemented on two fronts: in the ground station, a plugin could assess the risk of collision between drones through their target coordinates, temporarily stopping one of them and allowing the other one to move; in each drone module, by using adequate sensors to detect objects and circumventing them, which could require mission decentralisation to work effectively.

5. *Implement additional plugin types*. We presented suggestions for several plugin types but only implemented monitoring plugins. Particularly, implementing plugins that expose methods for calling a specific sequence of actions could easily apply monitoring and surveillance algorithms to other missions. This would help less experienced users, since they could easily test those features without developing an algorithm themselves.

6. *Extend mission functionality*. We attempted to make a mission framework that was as generic as possible, but it could be beneficial to survey what specific behaviours are more commonly used and integrate those directly. This would allow users to, for example, request that the drones map an area by calling a *mapping* verb instead of enabling a plugin.

7. *Classify different drone types*. All drones are currently equally treated, but it would be relevant to be able to classify drones as different types. This would allow assigning a drone to a mission based on those characteristics. There could be the concept of an "auxiliary" drone, which would not be usually assigned to a mission, but could be part of the pool of relay drones. In order to integrate with the drone replacement features, when possible, these drones could immediately continue the mission of the drone they were serving to avoid dispatching a distant drone. These drones could also be configured to send less telemetry data while on duty to avoid congesting the connection.

8. *High level interface for complex missions*. Once again, more measures could be taken to help inexperienced users. The existent dashboard allows setting waypoints in a map, which will be converted into a mission which consists of arming the drone, taking off, traversing the waypoints, and landing. However, it would be interesting to investigate ways to visually represent actions, such as defining separate subroutines with multiple drones, reading sensor data, and conditional statements. If this feature is possible to implement, it will considerably enrichen the solution.

References

[1] Instituto de Plasmas e Fusao Nuclear. (2020). "Friends: Fleet of drones for radiological inspection, communication and rescue," [Online]. Available: https://www.ipfn.tecnico.ulisboa.pt/FRIENDS/.

[2] R. Kellermann, T. Biehle, and L. Fischer, "Drones for parcel and passenger transportation: A literature review," *Transportation Research Interdisciplinary Perspectives*, vol. 4, p. 100 088, 2020, ISSN: 2590-1982. DOI: https://doi.org/10.1016/j.trip.2019.100088. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2590198219300879.

[3] T. Yang, C. H. Foh, F. Heliot, C. Y. Leow, and P. Chatzimisios, "Self-organization drone-based unmanned aerial vehicles (uav) networks," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761876.

[4] D. Gonzalez-Aguilera and P. Rodriguez-Gonzalvez, "Drones—an open access journal," *Drones*, vol. 1, no. 1, 2017, ISSN: 2504-446X. DOI: 10.3390/drones1010001. [Online]. Available: https://www.mdpi.com/2504-446X/1/1/1.

[5] F. Oliveira, "Machine Learning for the Dynamic Coordination of a Network of Aerial Drones," Master's Thesis, Universidade de Aveiro, 2021.

[6] S. Choi and E. Kim, "Building crack inspection using small uav," in *2015 17th International Conference on Advanced Communication Technology (ICACT)*, 2015, pp. 235–238. DOI: 10.1109/ICACT.2015.7224792.

[7] E. Skondras, K. Siountri, A. Michalas, and D. D. Vergados, "A route selection scheme for supporting virtual tours in sites with cultural interest using drones," in *2018 9th International Conference on Information, Intelligence, Systems and Applications (IISA)*, 2018, pp. 1–6. DOI: 10.1109/IISA.2018.8633594.

[8] V. P. Subba Rao and G. S. Rao, "Design and modelling of an affordable uav based pesticide sprayer in agriculture applications," in *2019 Fifth International Conference on Electrical Energy Systems (ICEES)*, 2019, pp. 1–4. DOI: 10.1109/ICEES.2019.8719237.

[9] R. R. Zargar, M. Sohrabi, M. Afsharchi, and S. Amani, "Decentralized area patrolling for teams of uavs," in *2016 4th International Conference on Control, Instrumentation, and Automation (ICCIA)*, 2016, pp. 475–480. DOI: 10.1109/ICCIAutom.2016.7483209.

[10] M. Zhu, X. Du, X. Zhang, H. Luo, and G. Wang, "Multi-uav rapid-assessment task-assignment problem in a post-earthquake scenario," *IEEE Access*, vol. 7, pp. 74 542–74 557, 2019. DOI: 10.1109/ACCESS.2019.2920736.

[11] D. Simões, "Forest monitoring through a hybrid system with networks of aerial drones and sensors," Master's Thesis, Universidade de Aveiro, 2019. [Online]. Available: http://hdl.handle.net/10773/29468.

[12] D. Simões, A. Rodrigues, A. B. Reis, and S. Sargento, "Forest fire monitoring through a network of aerial drones and sensors," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2020, pp. 1–6. DOI: 10.1109/PerComWorkshops48775.2020.9156137.

[13] H. Chao, Y. Cao, and Y. Chen, "Autopilots for small unmanned aerial vehicles: A survey," *International Journal of Control, Automation and Systems*, vol. 8, no. 1, pp. 36–44, 2010.

[14] E. Ebeid, M. Skriver, and J. Jin, "A survey on open-source flight control platforms of unmanned aerial vehicle," in *2017 Euromicro Conference on Digital System Design (DSD)*, IEEE, 2017, pp. 396–402.

[15] N. Yamamoto and K. Naito, "Proposal of continuous remote control architecture for drone operations," in *International Conference on Intelligent Interactive Multimedia Systems and Services*, Springer, 2018, pp. 64–73.

[16] E. Lygouras, A. Gasteratos, K. Tarchanidis, and A. Mitropoulos, "Rolfer: A fully autonomous aerial rescue support system," *Microprocessors and Microsystems*, vol. 61, pp. 32–42, 2018.

[17] M. Beul, N. Krombach, M. Nieuwenhuisen, D. Droeschel, and S. Behnke, "Autonomous navigation in a warehouse with a cognitive micro aerial vehicle," in *Robot Operating System (ROS)*, Springer, 2017, pp. 487–524.

[18] G. Crespo, G. Glez-de-Rivera, J. Garrido, and R. Ponticelli, "Setup of a communication and control systems of a quadrotor type unmanned aerial vehicle," in *Design of Circuits and Integrated Systems*, IEEE, 2014, pp. 1–6.

[19] A. Williams and O. Yakimenko, "Persistent mobile aerial surveillance platform using intelligent battery health management and drone swapping," in *2018 4th International Conference on Control, Automation and Robotics (ICCAR)*, IEEE, 2018, pp. 237–246.

[20] I. Bekmezci, I. Sen, and E. Erkalkan, "Flying ad hoc networks (fanet) test bed implementation," in *2015 7th International Conference on Recent Advances in Space Technologies (RAST)*, IEEE, 2015, pp. 665–668.

[21] S. Choi, J. Park, and J. Kim, "A networking framework for multiple-heterogeneous unmanned vehicles in fanets," in *2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN)*, 2019, pp. 13–15. DOI: 10.1109/ICUFN.2019.8806105.

[22] X. Li, J. Li, and J. Chen, "Effective cooperative uav searching using adaptive stgm mobility model in a fanet," in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 2018, pp. 295–301. DOI: 10.1109/BDCloud.2018.00054.

[23] K. Kappel, T. Cabreira, J. Marins, L. Brisolara, and P. Ferreira, "Strategies for patrolling missions with multiple uavs," *Journal of Intelligent & Robotic Systems*, vol. 99, pp. 499–515, 2019.

[24] H. Wang, B. Yan, X. Li, X. Luo, Q. Yang, and W. Yan, "On optimal path planning for uav based patrolling in complex 3d topographies," in *2016 IEEE International Conference on Information and Automation (ICIA)*, 2016, pp. 986–990. DOI: 10.1109/ICInfA.2016.7831962.

[25] T. Cabreira, K. Kappel, L. Brisolara, and P. Ferreira, "An energy-aware real-time search approach for cooperative patrolling missions with multi-uavs," in *Latin American Robotic Symposium*, IEEE, 2018.

[26] J.-H. Park, S.-C. Choi, I.-Y. Ahn, and J. Kim, "Multiple uavs-based surveillance and reconnaissance system utilizing iot platforms," in *International Conference on Electronics, Information, and Communication (ICEIC)*, IEEE, 2019.

[27] C. Hong and D. Shi, "A control system architecture with cloud platform for multi-uav surveillance," in *IEEE SmartWorld*, IEEE, 2018.

[28] S. Rosende, J. Sánchez-Soriano, C. Muñoz, and J. Andrés, "Remote management architecture of uav fleets for maintenance, surveillance, and security tasks in solar power plants," *Energies*, vol. 13, p. 5712, Nov. 2020. DOI: 10.3390/en13215712.

[29] E. Yanmaz, M. Quaritsch, S. Yahyanejad, B. Rinner, H. Hellwagner, and C. Bettstetter, "Communication and coordination for drone networks," in *Ad Hoc Networks*, Springer International Publishing, Dec. 2017, pp. 79–91, ISBN: 978-3-319-51203-7. DOI: 10.1007/978-3-319-51204-4_7.

[30] E. Yanmaz, S. Yahyanejad, B. Rinner, H. Hellwagner, and C. Bettstetter, "Drone networks: Communications, coordination, and sensing," *Ad Hoc Networks*, vol. 68, pp. 1–15, 2018, Advances in Wireless Communication and Networking for Cooperating Autonomous Systems, ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2017.09.001.

[31] J. Besada, L. Bergesio, I. Campaña, D. Vaquero-Melchor, J. López-Araquistain, A. Bernardos, and J. Casar, "Drone mission definition and implementation for automated infrastructure inspection using airborne sensors," *Sensors*, vol. 18, no. 4, p. 1170, 2018. DOI: 10.3390/s18041170.

[32] Y. V. Pant, H. Abbas, R. A. Quaye, and R. Mangharam, "Fly-by-logic: Control of multi-drone fleets with temporal logic objectives," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, Apr. 2018, pp. 186–197. DOI: 10.1109/ICCPS.2018.00026.

[33] A. Thibbotuwawa, G. Bocewicz, Z. Banaszak, and P. Nielsen, "A solution approach for uav fleet mission planning in changing weather conditions," *Applied Sciences*, vol. 9, Sep. 2019. DOI: 10.3390/app9193972.

[34] S. Zermani, C. Dezan, and R. Euler, "Embedded decision making for uav missions," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, 2017, pp. 1–4. DOI: 10.1109/MECO.2017.7977165.

[35] P. Rudol and P. Doherty, "Bridging reactive and control architectural layers for cooperative missions using vtol platforms," in *2017 25th International Conference on Systems Engineering (ICSEng)*, 2017, pp. 21–32. DOI: 10.1109/ICSEng.2017.59.

[36] Y. Wu, G. Bao, X. Xu, and Z. Lei, "Research on a language for commanding and controlling multi-robot systems," in *2014 10th International Conference on Natural Computation (ICNC)*, 2014, pp. 1077–1081. DOI: 10.1109/ICNC.2014.6975990.

[37] A. Bagnitckii, A. Inzartsev, and R. Senin, "Facilities of auv search missions planning," in *OCEANS'11 MTS/IEEE KONA*, 2011, pp. 1–7. DOI: 10.23919/OCEANS.2011.6107164.

[38] A. Pavin and A. Inzartsev, "A geojson-based mission planning language for auv (auvgeojson language)," in *OCEANS 2018 MTS/IEEE Charleston*, 2018, pp. 1–5. DOI: 10.1109/OCEANS.2018.8604643.

[39] N. Paula, "Multi-drone control with autonomous mission support," Master's Thesis, Universidade de Aveiro, 2018. [Online]. Available: http://hdl.handle.net/10773/27846.

[40] B. Areias, A. Martins, N. Paula, A. Reis, and S. Sargento, "A control and communications platform for procedural mission planning with multiple aerial drones," *Personal and Ubiquitous Computing*, Mar. 2020. DOI: 10.1007/s00779-020-01378-3.

[41] K. Priandana, M. Hazim, Wulandari, and B. Kusumoputro, "Development of autonomous uav quadcopters using pixhawk controller and its flight data acquisition," in *2020 International Conference on Computer Science and Its Application in Agriculture (ICOSICA)*, 2020, pp. 1–6. DOI: 10.1109/ICOSICA49951.2020.9243289.

[42] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *2008 IEEE Conference on Robotics, Automation and Mechatronics*, 2008, pp. 736–742. DOI: 10.1109/RWAMECH.2008.4681485.

[43] E. G. Tsardoulias and P. A. Mitkas, "Robotic frameworks, architectures and middleware comparison," *CoRR*, vol. abs/1711.06842, 2017. arXiv: 1711.06842. [Online]. Available: http://arxiv.org/abs/1711.06842.

[44] B. H. Lee, J. R. Morrison, and R. Sharma, "Multi-uav control testbed for persistent uav presence: Ros gps waypoint tracking package and centralized task allocation capability," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017, pp. 1742–1750. DOI: 10.1109/ICUAS.2017.7991424.

[45] H. Hayakawa, T. Azumi, A. Sakaguchi, and T. Ushio, "Ros-based support system for supervision of multiple uavs by a single operator," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 341–342. DOI: 10.1109/ICCPS.2018.00044.

[46] J. Jiang, X. Zhang, J. Yuan, K. Tang, and X. Zhang, "Extendable flight system for commercial uavs on ros," in *2018 37th Chinese Control Conference (CCC)*, 2018, pp. 1–5. DOI: 10.23919/ChiCC.2018.8483362.

[47] D. König, P. King, G. Laforge, H. D'Arcy, C. Champeau, E. Pragt, and J. Skeet, *Groovy in Action*, 2nd ed. USA: Manning Publications Co., 2015, ISBN: 9781935182443.

[48] D. Saldaña, D. Ovalle, and A. Montoya, "Improved algorithm for perimeter tracking in robotic sensor networks," *38th Latin America Conference on Informatics, CLEI 2012 - Conference Proceedings*, pp. 1–7, Oct. 2012. DOI: 10.1109/CLEI.2012.6427231.