Universidade de Aveiro
2021

Rui Emanuel Simões da Silva

Controlo e comunicação distribuídos para UAVs

Distributed control and communication for UAVs

Universidade de Aveiro
2021

Rui Emanuel Simões da Silva

Controlo e comunicação distribuídos para UAVs

Distributed control and communication for UAVs

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 Nuno Miguel Abreu Luís, Professor Adjunto do Instituto Superior de Engenharia de Lisboa.

o júri / the jury

presidente / president	Professor Doutor Artur José Carneiro Pereira Professor Auxiliar, Universidade de Aveiro
vogais / examiners committee	Professor Doutor Rui Lopes Campos Professor Auxiliar, Universidade do Porto
	Professora Doutora Susana Isabel Barreto de Miranda Sargento Professora Catedrática, Universidade de Aveiro

agradecimentos / acknowledgements

Agradeço ao Instituto de Telecomunicações pelo suporte financeiro aquando da realização desta Dissertação, no âmbito do projeto FRIENDS: Fleet of dRones for radIological inspEction, commuNication anD reScue (PTDC/EEI-ROB/28799/2017).
Agradeço à professora Susana Sargento pela oportunidade que me deu, assim como pela disponibilidade e apoio durante o desenvolvimento deste projeto.
Ao Nuno Luís que, como coorientador, também colaborou no desenvolvimento desta Dissertação com os seus conselhos e sugestões.
À Margarida Silva por toda a ajuda fornecida, principalmente durante a fase inicial do projeto, em algumas partes que se mostraram um pouco complicadas.
Ao André Mourato e ao Nuno Ferreira, que embora não tenham colaborado diretamente no desenvolvimento deste projeto, deram-me uma grande ajuda, com conselhos sobre os drones e na realização de testes.

Gostaria também de agradecer a todos os que sempre me apoiaram durante todo o meu percurso académico.
À minha família, principalmente aos meus pais e ao meu irmão, porque sem o seu apoio, não teria sido possível chegar até aqui.
Aos meus colegas, que de perto sempre me acompanharam e apoiaram desde que comecei o meu percurso universitário, principalmente ao Tiago, ao Afonso e ao Vasco.
E também aos meus amigos mais próximos que já me acompanham há imensos anos, e que mesmo estando longe agora, sempre estiveram lá quando foi preciso, nomeadamente o Jorge, José, Diogo e a tantos outros.

Palavras Chave Drones, Missões, Gestão de Frotas, Descentralização, Rede, Android

Resumo

Os drones têm-se tornado mais populares nos últimos anos devido a uma diminuição de custos e a uma gama mais ampla de casos de uso. Podem ser usados para vários fins, incluindo recreação e atividades mais específicas, como monitorização de florestas, agricultura e até mesmo no caso de um desastre natural. São a ferramenta ideal para auxiliar ou até mesmo substituir um ser humano devido ao seu tamanho minúsculo, facilidade de utilização e mobilidade. Os pilotos automáticos foram projetados para permitir que os drones realizem missões sem a necessidade de controlo humano. Muitas ferramentas diferentes foram desenvolvidas para gerir uma frota de drones, seguindo os seus movimentos e enviando instruções.

Uma nova abordagem para estas ferramentas foi apresentada nesta Dissertação. Normalmente, as arquiteturas desenvolvidas são mais centralizadas, o que significa que os drones estão sempre sob o comando de uma estação terrestre. Em vez disso, desenvolvemos um método para que os drones comuniquem uns com os outros e tomem decisões com base em dados em tempo real recolhidos durante as missões, mesmo quando estão fora do alcance da estação terrestre. Isso foi feito criando dois módulos separados, um para os drones e outro para a estação terrestre. O módulo dos drones executa missões, monitorizando a telemetria e a conectividade de rede com o resto da frota, enquanto o módulo da estação terrestre recolhe a telemetria dos drones e gere os drones disponíveis para realizar as missões. Criámos um método para aumentar o alcance máximo da rede da plataforma por meio de drones de retransmissão ativados de acordo com as métricas de rede.

Também desenvolvemos um método para integrar dispositivos Android na plataforma. Por exemplo, uma pessoa pode pedir um drone usando uma aplicação em Android, efetuando o pedido e fornecendo a sua localização.

Tudo o que desenvolvemos foi testado passo a passo para validar o trabalho realizado. Primeiro usámos ferramentas de simulação para verificar o trabalho localmente antes de passar para os testes em hardware real usando drones físicos num ambiente real, com um único drone primeiro e depois com vários drones em cooperação.

Keywords Drones, Missions, Fleet Management, Decentralization, Network, Android

Abstract

Drones have been more popular in recent years due to lower prices and a wider range of use cases. They can be used for various purposes, including recreation and more specific activities such as forest monitoring, agriculture, and even in the event of a natural disaster. They are the ideal tool to assist or even substitute a human due to their tiny size, quick deployment and mobility. Autopilots were designed to allow drones to carry out missions without the need for human control. Many different tools have been developed to manage a fleet of drones by tracking their movements and sending instructions.

A new approach to these tools has been presented in this Dissertation. Typically, the architectures developed are more centralized, meaning that the drones are always under the control of a ground station. Instead, we devised a method for drones to communicate with one another and make decisions based on real-time data collected during missions, even when they are out of range of the ground station. This was accomplished by creating two separate modules, one for the drones and the other for the ground station. The drone module executes missions, monitoring telemetry and network connectivity with the rest of the fleet, while the ground station module retrieves telemetry from the drones and manages the available drones to perform missions. We have created a method of extending the maximum network range of the platform through relay drones activated through a change on the network metrics.

We also devised a method to integrate Android devices into the platform. For example, a person could request a drone using an Android application by placing the order and providing its location.

Everything we developed was tested step by step to validate the performed work. First we used simulation tools to verify the work locally before moving on to real hardware testing using physical drones in a real environment, with single drones first, and then with multiple drones in cooperation.

Contents

Contents i

List of Figures iii

List of Tables v

List of Abbreviations vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Document outline . . . 3

2 Background and Related Work 5
- 2.1 Unmanned Aerial Vehicles . . . 5
- 2.2 Drone components . . . 6
- 2.3 Mission planning and control . . . 9
- 2.4 Additional technologies and other tools . . . 11
- 2.5 Related Projects . . . 14
- 2.6 Summary . . . 18

3 Architecture 19
- 3.1 Base architecture . . . 19
  - 3.1.1 Drone module . . . 20
  - 3.1.2 Ground Station module . . . 21
  - 3.1.3 Simulation tool . . . 21
- 3.2 Requirements . . . 22
- 3.3 Proposed architecture . . . 23
  - 3.3.1 Network module . . . 25
  - 3.3.2 Android module . . . 25
  - 3.3.3 Drone module . . . 25
  - 3.3.4 Ground Station module . . . 26
- 3.4 Summary . . . 26

4 Implementation **27**
4.1 Modules . . . 27
4.1.1 General overview . . . 27
4.1.2 Communication between modules . . . 28
4.1.3 Network Module . . . 33
4.1.4 Android Module . . . 39
4.1.5 Drone Module . . . 42
4.1.6 Ground Station Module . . . 44
4.2 Relay . . . 45
4.2.1 Modules integration . . . 46
4.2.2 Message exchange and format . . . 47
4.2.3 Drone performing a mission . . . 50
4.2.4 Relay Drone . . . 51
4.2.5 Ground Station . . . 57
4.2.6 Routing . . . 58
4.2.7 Relay summary . . . 59
4.3 Rescue . . . 61
4.3.1 Modules integration . . . 61
4.3.2 Android application . . . 62
4.3.3 Drone . . . 65
4.3.4 Rescue summary . . . 66
4.4 Summary . . . 66

5 Experiments and Results **69**
5.1 Simulated experiments . . . 69
5.1.1 Relay mission . . . 70
5.1.2 Android integration . . . 74
5.2 Experiments in a real environment . . . 74
5.2.1 Hardware . . . 74
5.2.2 Testing relay with network integration . . . 75
5.3 Summary . . . 85

6 Conclusion **87**
6.1 Final observations . . . 87
6.2 Future work . . . 88

References **91**

List of Figures

2.1 An example of open and closed frames. 7
2.2 Holybro Pixhawk 4 Power Module (PM07). 7
2.3 Examples of Flight Controllers. 8
2.4 Example of GPS modules. 8
2.5 Raspberry Pi 4 Model B and NVIDIA® Jetson Nano™. 9
2.6 QGroundControl GUI. 10

3.1 Base architecture. 20
3.2 Proposed architecture. 24

4.1 Network Module Implementation. 33
4.2 Android Module Implementation. 39
4.3 The user interface of the application. 41
4.4 Drone Module Implementation. 42
4.5 Ground Station Module Implementation. 44
4.6 Example of a mission scenario. 45
4.7 Modules connection for a mission with relay. 46
4.8 Flow for a Relay Request. 48
4.9 Flow for a Cancel Request. 49
4.10 Drone's different speed zones. 52
4.11 Example scenario for a mission with relay support. 54
4.12 Example scenarios using batman-adv to route information. 59
4.13 Flow for the drone that initiates a mission. 60
4.14 Flow for the drone that performs the relay. 60
4.15 Modules connection for a mission with relay and Android integration. 61
4.16 The user will see one of these messages if one or both of the prerequisites (connection and Global Positioning System (GPS), respectively) are not met. 62
4.17 The user will receive a response to the request depending on the availability of drones. 64
4.18 Request cancelation. 65
4.19 The user will receive this message if the drone cancels the mission. 65
4.20 Flow for the drone performing a rescue mission. 66

5.1 Example of a mission with relay, displaying the paths of the drones. 70
5.2 Graph with the distance between the mission drone and its relay. 71

5.3 Graph displaying the improvement using different Quality of Service (QoS) policies. . . . 73
5.4 One of the drones used in the experiments. . . . 75
5.5 Path planned for the mission drone. . . . 78
5.6 Relay request performed by the first drone. . . . 79
5.7 Path followed by the drones. . . . 80
5.8 Network quality between drone01 and the Ground Station during the mission. 81
5.9 Optimal and real distance between the two drones throughout the mission. 82
5.10 Signal between the two drones throughout the mission. . . . 83
5.11 Telemetry gathered by the different nodes during the mission. . . . 84

List of Tables

4.1 The key parameters in a network measurement. 34
4.2 Parameters list for each connection in a network measurement. 35
4.3 List of parameters in the configuration file. 37
4.4 Supported commands in the Drone Module. 43
4.5 Requests handled by the Ground Station for the relay scenario. 57

5.1 Drones specifications. 74
5.2 Ground Station specifications. 75

List of Abbreviations

AMSL Above Mean Sea Level

API Application Programming Interface

CSMA Carrier Sense Multiple Access

ESC Electronic Speed Controller

FC Flight Controller

FIFO First In, First Out

FW Fixed-Wing

GPS Global Positioning System

GS Ground Station

GUI Graphical User Interface

HITL Hardware-In-The-Loop

HTTP HyperText Transfer Protocol

IoD Internet of Drones

IoT Internet of Things

IP Internet Protocol

JSON JavaScript Object Notation

LAN Local Area Network

MAC Media Access Control

MAVLink Micro Air Vehicle Link

MQTT Message Queuing Telemetry Transport

MR Multi-Rotor

QoS Quality of Service

RC Radio Control

RCL ROS Client Library

RCLPY ROS Client Library for Python

REST Representational State Transfer

ROS Robot Operation System

SDK Software Development Kit

SITL Software-In-The-Loop

TCP Transmission Control Protocol

TDMA Time-Division Multiple Access

UAS Unmanned Aircraft System

UAV Unmanned Aerial Vehicle

UDP User Datagram Protocol

USB Universal Serial Bus

VTOL Vertical Take-Off and Landing

Chapter 1

Introduction

This chapter provides some context for the development of this Dissertation. We begin by presenting the motivations that lead to the work done, and then explain the key goals that were set. Then, we outline the organization of the Dissertation, briefly describing the various chapters.

1.1 Motivation

Unmanned Aerial Vehicles (UAVs), or drones, are pilotless aerial vehicles. They were originally designed to be used in military activities, but in recent years, their applications have broadened, and they are now utilized for a variety of purposes, ranging from personal use to more specialized research.

This increased accessibility enables them to be employed in highly specialized circumstances, such as forest monitoring and emergencies like natural catastrophes and the COVID-19 epidemic. However, this demands the use of several drones, and the ability to control a fleet of drones requires the use of a platform that meets these requirements. To accomplish these objectives, various studies have been conducted. [1], for example, provides a set of functionalities and tools that allows the management of a fleet of drones, as well as features that make developing and debugging new features simpler. In addition, the Ground Station always coordinates the fleet on this platform, which has a centralized architecture. This restricts the platform because a mission is always sent from the Ground Station, which means that if the drone's connection to the ground station is lost, the drone will be unable to receive commands. This is also a concern when we do not have direct eye contact with a drone. We have no method of acquiring data and coordinating a drone's return if we do not know where it is in case of a malfunction that requires manual intervention. The platform's reliance on the ground station limits the situations where drones can be employed and what they might execute.

The primary purpose of this Dissertation is to address these restrictions by designing an architecture that focuses more on drones and less on the ground station. This way, even if the contact with the Ground Station is temporarily lost, a drone may still complete a mission, decreasing its dependency from the Ground Station.

As a starting point, we will use the base platform from [1] to provide a way to deliver

missions. The Ground Station module is responsible for keeping track of which drones are online at any time, assigning tasks, and receiving data from them, such as telemetry, which provides information such as a drone's current location. On the other hand, the Drone Module concentrates on making the interface with a drone as simple as possible, by eliminating low-level user interactions and offering a high-level and user-friendly means of creating missions.

We aim to combine existing characteristics with new ones to allow drones to communicate directly with one another without the intervention of the Ground Station. Because the drones will coordinate missions among themselves, this could expand their capabilities. We also aim to deploy relay drones to increase the platform's network range by acting as intermediary links between multiple endpoints, allowing for longer-distance connections. This will demand a mechanism to evaluate network conditions and determine when these relays are required.

Finally, we plan to incorporate Android smartphones into the platform as a whole. This could be the most challenging part, as the current architecture does not accommodate these devices. The objective is to create an application that allows Android devices to communicate with drones, allowing the former to assign missions.

1.2 Objectives

We established a list of goals that needed to be met in order to attain a more decentralized architecture:

- Create a drone-to-drone communication channel.
- Allow mission code to be performed in the drones rather than at the Ground Station.
- Develop a system that allows drones to make decisions on their own, based on data collected in real time during missions, even when they are not connected to the ground station.
- Create a ground station module to handle requests from drones.
- Integrate a mechanism for assessing the platform's network conditions.
- Integrate Android devices into the platform, by developing an application that allows for a user to assign a mission to a drone.

We want to achieve our objectives by improving the present Drone and Ground Station modules by adding new communication channels and functionalities. After that, we will need to create two new modules: one for the network and one for Android. Then, to allow the platform's current capabilities to grow, combine these with what already existed to expand functionalities and the number of scenarios in which they can be applied, testing them in a real-world environment with drones.

1.3 Document outline

This Dissertation is divided into five chapters.

Chapter 2, Background and Related Work, provides context for the Dissertation, including how Unmanned Aerial Vehicles (UAVs) have grown in importance in recent years, and how they are now used, as well as a detailed description of a drone's main components. Following that, we go over some of the tools that may be used to plan and manage missions for drone fleets, and the essential technologies used to create these platforms. Finally, we describe related Unmanned Aerial Vehicle (UAV) studies and provide a brief description of some of their limitations, as well as how the work developed in this Dissertation aims to overcome them.

Chapter 3, Architecture, outlines the most significant modules established in [1] that served as a basis. We next list and explain the requirements that must be met to achieve our objectives, and we conclude by proposing an architecture, defining the various modules and how communication is performed.

Chapter 4, Implementation, discusses how the various modules presented throughout the Dissertation are implemented. First, we describe the overall implementation of modules communication and then go over each one separately. Then, we describe the scenarios in which these modules were evaluated, and how the various components interacted.

Chapter 5, Experiments and Results, explains the various experiments conducted to test and validate the implementation, first in a simulator and then in a real-world scenario using real hardware. Finally, we examine the data and evaluate what is working well and what needs improvement.

Chapter 6, Conclusion, provides a brief overview of the work completed, some remarks, and a list of potential future ideas.

Chapter 2

Background and Related Work

In this chapter, we will discuss the research that was conducted prior to the development of the Dissertation, including describing UAVs, its applications, the many components that make up the UAVs, as well as discussing available control platforms. We will also discuss how the work in this Dissertation made use of the platform developed in [1], which provided a set of tools that made both setting up drones for flying and developing new features much easier.

2.1 Unmanned Aerial Vehicles

An Unmanned Aerial Vehicle (UAV), often known as a drone, is an aircraft that does not have a human pilot, crew, or passengers on board. UAVs are a component of a larger system known as an Unmanned Aircraft System (UAS), which also includes a ground-based controller and a communication system with the UAV [2]. It can be commanded remotely or autonomously.

Drones were first designed to be used in military missions that were too risky for humans to perform. Drone experiments began in the 1920s, following World War I [3], and were eventually upgraded to the first cruise missile variants, such as the Fieseler Fi-103 [4], the German V-1 flying bomb, employed during World War II. Fast forwarding a few years to the Cold War era, the United States deployed drones as reconnaissance vehicles to spy on Soviet territory, and the Israeli Army later used them in the 1980s [5]. The MQ-1 Predator[1], the first iteration of UAVs with heavy military armaments, was deployed by the United States in the Balkans in 1995. However, as technology progressed and hardware components became affordable, drones began to be commercialized as a recreational product and for personal use, being widely used in various scientific areas.

Applications for UAVs

Drones can be used for a variety of reasons, since their characteristics, such as small size, easy and quick deployment, and mobility, make them a better alternative to other

[1] https://www.deagel.com/Support%20Aircraft/Predator/a000517

vehicles [6]. They enable deployment in regions where smaller movements are impossible to access by traditional techniques or where human lives are at risk. They are also used in monitoring missions, such as checking for fires in forest areas and even assisting in firefighting[2].

Other uses for UAVs include:

- Agricultural applications include crop monitoring and dusting[3];
- Filmmaking, by giving new angles of coverage not generally available with standard cameras[4];
- Monitoring pollution and analyzing air quality with sophisticated sensors[5];
- Using thermal cameras to monitor crowds during the COVID-19 pandemic[6].

2.2 Drone components

Several drones are also used to research new approaches. These drones differ from the ones typically sold for recreational use, in that they are purchased in pieces and assembled by researchers, allowing for the attachment of specific equipment required for investigations. We will go through these components in more detail on the next pages.

Drone Frame

The frame of a drone (see Figure 2.1) is the drone's main structure. It is where the drone's arms are kept, each with its own motor and propeller. It can be a "closed" piece, which means the rest of the components, including sensors and other electronics, are stored inside of it; this is common in recreational drones because it is more user-friendly. The problem of these drones is that they usually cannot be upgraded with a new gear. They can also be an "open" piece, which means that the components are held in a central wheel, and the user can add more sensors as needed.

[2] https://newyork.cbslocal.com/2021/09/30/fdny-drones-2/

[3] https://spectrum.ieee.org/chris-andersons-expanding-drone-empire

[4] https://www.theatlantic.com/technology/archive/2014/02/the-future-of-sports-photography-drones/283896/

[5] https://journals.vgtu.lt/index.php/Aviation/article/view/3004

[6] https://www.forbes.com/sites/stevebanker/2020/06/11/is-the-future-of-drones-now/?sh=633a76333284

Figure 2.1: An example of open[7] and closed[8] frames, respectively.

Distribution Board

The Power Management Board (see Figure 2.2) is a vital part of the drone's wiring circuit. It is a Power Supplier as well as a Power Distributor. It distributes power straight from the battery to the remaining components, including the Flight Controller (FC) and Electronic Speed Controller (ESC), as well as informing the FC on the amount of energy provided.

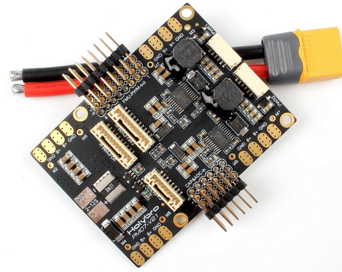

Figure 2.2: Holybro Pixhawk 4 Power Module (PM07)[9].

Flight Controller

The Flight Controller (see Figure 2.3) is the device that runs the flight software. It accepts commands from the Radio Control (RC) or any sort of software running autopilot, and delivers them to the drone's ESC that control motors' speed, which then sends them straight to the motors by adjusting their speed. It also retrieves information from the drone's state, such as GPS coordinates, and provides it to the user.

[7] https://www.dji.com/pt/flame-wheel-arf/feature
[8] https://www.dji.com/pt/phantom-4-pro?site=brandsite&from=mobile_nav
[9] https://docs.px4.io/master/en/power_module/holybro_pm07_pixhawk4_power_module.html

Figure 2.3: Examples of Flight[10]Controllers[11].

GPS Module

The GPS module (see Figure 2.4) is a standard GPS module that is necessary for autonomous flights, as well as remote control in any flying mode that requires a worldwide position, usually supporting global navigation satellite systems (GNSS[12]). Most of them come with a FC and an integrated compass, used for determining the direction the drone is facing (heading) and the proper direction to face when a new destination coordinate is entered (bearing).

Figure 2.4: Examples of GPS[13]modules[14].

Onboard computer

To execute any form of a mission, the devices that are running it must be directly connected to the Flight Controller. Because it is not possible to fly a drone while attached to a conventional computer, the solution relies on the use of an on-board computer (see Figure 2.5). A common example is any type of Raspberry because it is inexpensive and simple to use, with the ability to perform any type of task as long as it is not too heavy. However, if we want to conduct any kind of task that requires more computer power,

[10] https://www.dji.com/pt/naza-m-v2
[11] https://docs.px4.io/master/en/flight_controller/pixhawk4.html
[12] https://www.euspa.europa.eu/european-space/eu-space-programme/what-gnss
[13] See footnote 10
[14] https://docs.px4.io/master/en/GPS_compass/

such as image processing, we need something better, such as a NVIDIA® Jetson Nano™, because it has more computational power with enhanced CPU, GPU, and memory. Typically, these devices run a lightweight version of an operating system, such as Ubuntu arm64[15].

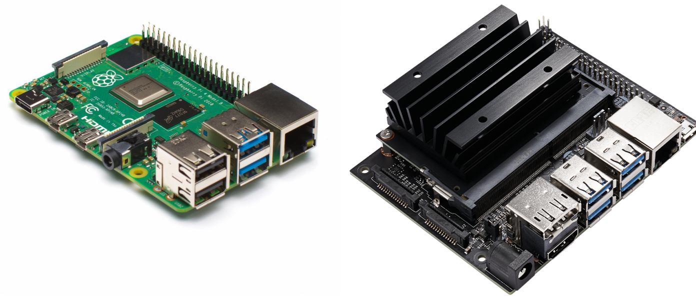

Figure 2.5: Raspberry Pi 4 Model B[16]and NVIDIA® Jetson Nano™[17], respectively.

The components above are the ones that, when combined, allow us to have a viable drone for our tests. We will now go over different platforms for managing a fleet of drones.

2.3 Mission planning and control

Various tools are available on the market for setting up missions and controlling drones, ranging from free and open-source software that allows to transmit commands and track their movements to more specialized software created for particular applications.

QGroundControl

QGroundControl[18] is a free open-source application, member of The Dronecode Foundation[19], that provides full flight control and mission planning for drones that communicate via the Micro Air Vehicle Link (MAVLink)[20] protocol. It is particularly useful for both end-users and developers. It provides a graphical user interface with a map that shows the drone's global location, as well as information like battery level and flight mode. It also allows a user to give simple commands such as take off, coordinates, and landing, as well as complete missions with multiple tasks. Apart from that, when connecting a FC, it gives a number of options, such as the sort of drone frames[21] that may be used, as

[15] https://ubuntu.com/download/server/arm
[16] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
[17] https://developer.nvidia.com/embedded/jetson-nano-developer-kit
[18] http://qgroundcontrol.com/
[19] https://www.dronecode.org/
[20] https://mavlink.io/en/
[21] https://docs.px4.io/v1.9.0/en/getting_started/frame_selection.html

well as allowing to calibrate the embedded sensors and presenting a list of configurable parameters provided by the Flight Controller.

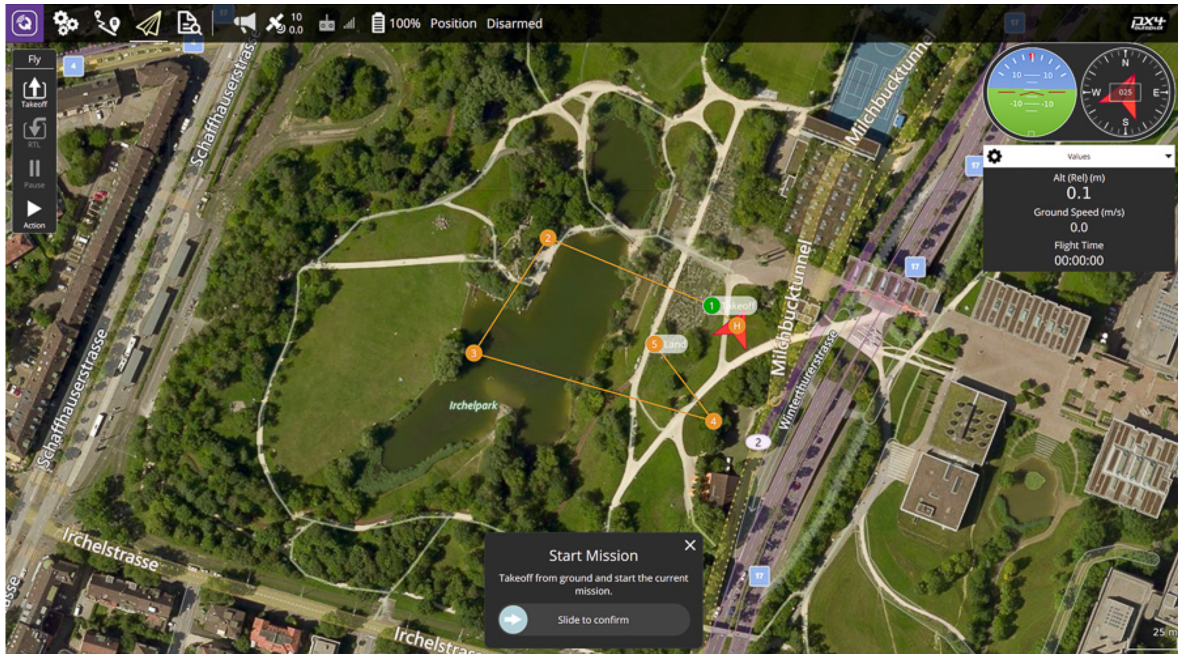

Figure 2.6: QGroundControl Graphical User Interface (GUI)[22].

DroneKit

DroneKit[23] is a free open-source platform that includes a Software Development Kit (SDK) and a web Application Programming Interface (API) for developing apps for drone systems that use the MAVLink[20] communication protocol. Path Planning, Autonomous Flight, and Live Telemetry Recovery are all possible. It is split into two parts: DroneKit Python is a Python API that allows to plan and design missions for use in onboard companion computers (covering high-level situations like Computer Vision, 3D modeling, and so on), but it can also be used by Ground Station (GS) apps to provide commands to the drones performing the missions. DroneKit-Android is the second part, which provides an API that can be used to control any type of UAV, such as control copters, planes, and ground vehicles, in a manner similar to the first part, but this time for Android devices.

Mission Planner

Mission Planner[24], which is part of the ArduPilot[25] project, is likewise a free and open source platform. It sets waypoints, sends commands to a fleet of drones, analyzes mission

[22] https://docs.qgroundcontrol.com/master/en/index.html
[23] https://dronekit.io/
[24] https://ardupilot.org/planner/
[25] https://ardupilot.org/

logs, and customizes a variety of autopilot settings. It also comes with a fully complete Software-In-The-Loop (SITL) simulation that simulates the drone's point of view for a more realistic experience. It is very similar to QGroundControl in terms of graphical user interface (see Figure 2.6).

Precision Hawk

Precision Hawk[26], in contrast to the products mentioned above, provides enterprise-level tools. It is primarily aimed at accumulating massive amounts of aerial data for future study. It enables users to fly the drones themselves with the necessary hardware, using its integrated platform, or to hire a skilled team for this purpose, as well as a team of data scientists for aerial intelligence. It is also developer-friendly, offering users ways for integrating their platform with previously existing operations. One of the most widely used study fields in which their technologies could be valuable is agriculture, because their combination of specialized sensors and data collection enables for more effective and efficient farming decisions.

2.4 Additional technologies and other tools

Beyond the above-mentioned control platforms, there are a few other related technologies and relevant tools worth mentioning.

MAVLink

MAVLink[20] is a messaging protocol that is used in a variety of UAV systems around the world and is managed by the Dronecode Foundation[19]. It was founded in 2009 by Lorenz Meier[27]. However, because it is a free and open source project, many others have contributed to its growth throughout the years. It can be used for communication between drones and external nodes (e.g. GSs), or even between onboard components in drones and other vehicles. It follows a typical publish/subscribe format with point-to-point communication. Some of the characteristics that distinguish this protocol are that it is very efficient and lightweight, working in applications with very limited network bandwidth, and that it is very reliable, working properly in communication channels with high noise/latency and providing methods for packet authentication, corruption, and packet loss. MAVLink 2[28], an improved version of MAVLink, offers improved security and flexibility while being backwards compatible with the original one.

MAVSDK

MAVSDK[29] is a higher-level tool, also managed by the Dronecode Foundation[19], that is directly tied to MAVLink[20], since it includes a set of libraries for a variety of pro-

[26] https://www.precisionhawk.com/
[27] https://github.com/LorenzMeier
[28] https://mavlink.io/en/guide/mavlink_2.html
[29] https://mavsdk.mavlink.io/main/en/index.html

gramming languages that may be used to interface with any vehicle that uses MAVLink as its communication protocol. This tool comes with an API that allows a user to manage vehicles and includes features like mission planning, vehicle control, and telemetry gathering.

PX4

The Dronecode Foundation[19] supports PX4[30], a professional open source autopilot. Its features include control over a variety of aircraft, including fixed wing, multicopters, and Vertical Take-Off and Landing (VTOL), support for a variety of hardware for vehicle control and sensors, and a variety of flight modes, all of which are backed up by numerous safety features to protect the vehicle in the event something goes wrong. PX4 is made up of two main layers[31]: the flight stack and the middleware. The Flight Stack is the actual Flight Control System, consisting of a collection of algorithms for autonomous drone guidance, navigation, and control. Fixed wing, multirotor, and VTOL airframe controllers, as well as attitude and position estimators, are all included. On the other hand, middleware encompasses any form of communication channel, both internal - between software and hardware - and external - between the device and the rest of the world (with a Ground Control Station, for example). PX4 also supports both SITL, by simulating the software in a computer and Hardware-In-The-Loop (HITL), by running simulations on a real flight controller, with a list of fully compatible simulators[32]. Last but not least, as part of the Continuous Integration Program[33], developers can integrate PX4 with new hardware by modifying existing algorithms.

ROS1 and ROS2

The Robot Operation System (ROS)[34] is a collection of libraries and tools for creating robot-based applications. Despite the fact that its primary goal was to be utilized by robots, ROS is now used in a wide range of hardware. It was first launched in 2007, but much has changed since then, and ROS2 was eventually released to address these changes, taking the finest features of ROS1 and improving them. The decision not to upgrade ROS1 and instead build a new framework from the ground up was made due to the intrusive nature of the additional required functionalities, which could utterly disrupt an already functioning system. With the usage of rosbridge[35], it is possible for nodes to communicate with one other even if one uses ROS1 and the other uses ROS2, and even between different ROS distributions. To record the messages that go through the existing topics, we use rosbags[36]. A node in ROS is a component that performs a certain computational work and follows a publisher/subscriber model, which means that messages are published to topics and received by anyone who has subscribed to those

[30] https://px4.io/
[31] https://docs.px4.io/master/en/concept/architecture.html
[32] https://docs.px4.io/master/en/simulation/
[33] https://docs.px4.io/master/en/test_and_ci/
[34] https://docs.ros.org/en/rolling/
[35] http://wiki.ros.org/rosbridge_suite
[36] https://docs.ros.org/en/foxy/Tutorials/Ros2bag/Recording-And-Playing-Back-Data.html

topics. A node can subscribe to as many topics as it wants, and publish to as many topics as it wishes. One of the most significant differences between ROS2 and ROS1 is that ROS1 required the running of a ROS master node with a collection of nodes and programs that are pre-requisites of a ROS-based system, known as roscore[37], in order for the ROS nodes to communicate and work properly, whereas ROS2 does not, evolving from a centralized architecture to a peer-to-peer architecture. This also means that ROS2 no longer requires creating XML launch files[38], instead requiring simply coding the nodes' routines in the programming language of choice, utilizing the supplied API and ROS Client Library (RCL), such as rclpy in Python and rclcpp in C++.

Gazebo

Gazebo[39] is a free open source robotics simulator developed as part of The Player Project[40]. With Gazebo, we can test any kind of algorithm and even teach Artificial Intelligence. It combines a complex physics engine with high-quality graphics, including 3D rendering of environments, textures, shadows, and lighting, as well as a GUI to help with the job. For environmental, sensor, and robot control, Gazebo provides an API that permits direct access using plugins that can be custom developed by the user. It is also feasible to incorporate more specialized hardware, such as contact sensors and 3D cameras, to provide sensor data for more complex projects. The best thing about Gazebo is that it is fully integrated with both ROS[41] and ROS2[42], allowing to simulate a robot using ROS messages with the help of libraries that act as a bridge between Gazebo's C++ API and ROS' services. Because Gazebo is fully compatible with PX4's Flight Stack[43] and MAVLink[20] messages, this simulator can be used to test UAV missions and algorithms.

jMavSim

PX4[30] and the MAVLink[20] protocol are supported by jMAVSim[44], an open-source simulator. It is simpler than Gazebo[39], as it just simulates UAVs with four motors (quadrotors). It only shows the drone in a simulated environment with telemetry data[45], and it does not handle advanced environmental factors like shadows or lighting. It supports basic commands like takeoff, flight, and landing, making it an excellent choice for testing smaller missions and/or algorithms. Because of all of these, it is far lighter than Gazebo, making it the best existing alternative, particularly when the device executing the simulator lacks the necessary performance or when multiple instances of a simulation must be run at the same time.

[37] http://wiki.ros.org/roscore
[38] https://roboticsbackend.com/ros1-vs-ros2-practical-overview/#Why_ROS2_and_not_keep_ROS1
[39] http://gazebosim.org/
[40] http://playerstage.sourceforge.net/
[41] http://gazebosim.org/tutorials?tut=ros_overview
[42] http://gazebosim.org/tutorials?tut=ros2_overview
[43] https://docs.px4.io/master/en/simulation/gazebo.html
[44] https://github.com/PX4/jMAVSim
[45] https://docs.px4.io/master/en/simulation/jmavsim.html

2.5 Related Projects

In recent years, there has been a significant increase in the use of UAVs in a variety of research fields, with UAVs being used to replace traditional methods [6]. Several studies have also been carried out in order to discover new ways to improve the autopilot features, as well as the ability to control several drones with platforms. We will take a look at a few of these now.

In [7], an UAV was used to examine multiple field plowing techniques. The UAV would fly through the field with a depth-sensing device called an RGB-D sensor installed on it. It was possible to classify the three different plowing depths that were tested, which were plane, 25 centimeters, and 50 centimeters deep, using this data, leading the authors to conclude that this method could be used as a terrain classifier, making it faster and easier than traditional methods, which are done by hand or occasionally with the use of satellite images.

Still in the agricultural field, UAVs with the function of plant protection were tried in [8]. These drones took full advantage of the PX4[30] firmware and featured a Pixhawk 4[11] placed on the drone that ran the autopilot software. The plant protection UAV control system was divided into two parts: the flight control system and the spray system. The results revealed that the control system could meet a plant protection UAV's spray control requirements.

The work in [9] describes a test of a new UAV design. It was able to improve the aircraft's performance by removing the most complex components of a Fixed-Wing (FW) aircraft and adding two extra side-motors, creating a hybrid drone between Multi-Rotor (MR) and FW. PX4[30] firmware was used in the tests, together with QGroundControl[18] to select the desired trajectory and Gazebo[39] to display the UAV model in 3D.

Because drones may only stay in the air for a short period of time due to battery limitations, [10] addresses how to improve trajectory planning to extend this duration. The authors created a machine learning model that estimates the battery's lifespan based on several flying situations, and a variety of other variables that could occur in real-world circumstances. It was feasible to infer that the models and algorithms constructed were capable of achieving near-optimal performance by completing the set of a mission's target sites in the lowest possible time to improve battery duration using rigorous testing, both in simulators and in real life.

A hybrid nonlinear control system in a quadcopter drone was tested in [11]. When faced with uncertainties, such as any form of disruption during a flight, both internal, such as the movement of the UAV's multiple rotors, and external, such as wind gusts, a drone's trajectory tracking can become quite inaccurate. In order to compare these results to conventional control systems, this experiment used a nonlinear predictive model and a fuzzy[12] feedforward compensator[46], as well as a lot of computer simulation. The authors found that the performance of this hybrid feedback fuzzy feedforward autopilot could undoubtedly execute accurate trajectory tracking in the face of uncertainty, while also enhancing overall stability based on the data analysis from these simulations.

[46] https://www.sciencedirect.com/topics/engineering/feedforward-control

Smart cities, which make use of the Internet of Things (IoT)[47], have become increasingly significant as technology has advanced in recent years. As a result, the Internet of Drones (IoD)[48] arose, which is a combination of drones and IoT. Drones, as we all know, need a GPS module to navigate on their own. However, these GPS modules are often of poor quality, resulting in inaccurate geolocation. To address this, a study published in [13] proposed a cooperative drone position measurement approach (CDPM). This entails every drone that is online being connected to each other and constantly exchanging their position and measuring distance between each other, as well as applying algorithms to determine their own location based on this data. This work concluded from the simulations conducted in a real-world scenario that, using CDPM as a positioning device for self-drone positioning, taking advantage of the sensors installed in other drones, increased localization accuracy.

A platform for controlling multiple drones was suggested in a previous Dissertation, [14], and also in [15], allowing more inexperienced users to plan, execute, and monitor complex missions that require drone cooperation, taking full advantage of the autopilot features. Drone Side and Ground Side were the two halves of the solution. A Mission Handler was featured on the Drone Side, which would receive tasks and send them to the Flight Controller. The Ground Side consisted primarily of a Drone Manager and a Mission Planner, which would send the user's missions to the drones, and a Telemetry Analyzer, which would receive telemetry data from the drones, allowing the user to track the progress of the missions, as well as have data to analyze in the event of a debugging process, such as if a drone crashed.

Various institutions and businesses use drones to accomplish duties such as surveillance and inspection. To help with this, the research presented in [16] suggests a microservice-oriented architecture that includes methods for organizing and optimizing drone deployment. The goal is for a single operator to manage a fleet of drones using tools that automatically assign, coordinate, and control the fleet to achieve a given purpose. Drones are viewed as a service in this architecture. The authors validated this architecture by testing it in an emergency scenario, specifically a multi-vehicle incident. The operator in this scenario can select the mission type (for example, "Recognition"), the location and radius of the operation zone, and the landing site for the drones. Following the operator's selection of these options, the system would gather available resources and develop flight plans, which the user could subsequently deploy.

The work in [17] offers an algorithm that employs Reinforcement Learning to allow a fleet of drones to monitor a particular area autonomously. Drones can move to the target area and coordinate themselves without having to set up a flight path in advance, thanks to this algorithm's ability to discover the optimum path by recognizing obstacles and avoiding collisions. One of the most potential applications is monitoring forest wildfires by providing virtual reality to firefighters. The authors concluded that the algorithm has 80% success probability based on simulations.

In recent years, one of the most common applications for drones has been as a delivery system. Drones deliver packages to customers considerably more quickly than

[47] https://www.iotforall.com/what-is-internet-of-things

[48] https://www.ericsson.com/en/blog/2021/6/internet-of-drones-sky-is-not-the-limit

traditional ways. However, one of these systems' flaws is that, when the drones are near populated areas, they frequently lose GPS and even communication with the controlling entity. The work in [18] proposes a mechanism for allowing drones to accomplish specific activities, even while the constraints described are in effect to avoid this issue. They do it by processing images to identify safe landing zones and potential obstructions. The researchers used a real drone to conduct testing through landing and take off without any human intervention. The implementation turned out to be a success in the end. The work in [19] depicts a similar scenario, but the drones will be delivering meals. Customers' waiting time is extremely limited in this scenario. The utilization of charging stations across the system was studied and how they affected the system's overall performance.

With the advancement of current technologies, 5G has emerged, and the number of infrastructures that use it is steadily growing. The work in [20] suggests to use a combination of drones and 5G in Critical Infrastructure Preventive Maintenance as a Service (PMaaS). The idea is to deploy drone swarms to help with monitoring, offloading tasks, and media processing in these infrastructures.

The work in [21] researched how to use drones to detect gas leaks. For this, the researchers used a swarm of drones controlled by a ground control station to mount gas sensors that monitor industrial air pollution. They also built a coverage algorithm that automatically elaborates and assigns paths to each drone. The drones perform a mapping of the gas concentration while on a mission. The tests were carried out in a simulator and a real-life scenario with a gas leak. Because the drones could move to the correct location and detect the leak, the software and hardware were validated.

Drones are employed in rescue missions to assist in regions that would be difficult for a human to reach or to automate this process due to their quick deployment. The research undertaken in [22] looked into how drones could be used in these situations. The drones would use RGB cameras to evaluate live streaming data sets such as object tracking and face recognition, allowing them to coordinate their actions. Drones would fly over some areas of interest and utilize these tools to detect situations where assistance is required. Even though no actual tests were conducted, this study looked at the various capabilities that drones can bring in realistic scenarios.

A mission planning framework is presented in another Dissertation, [1]. This framework is divided into two parts: a drone control module that receives commands and instructs the drones to carry out those commands, and GS software that serves as an interface with the drone module for submitting commands and receiving telemetry data, as well as simulation tools for testing new features in development. This Dissertation also addressed the issue of support for multi-drone cooperation missions, which had not been addressed in previous solutions. The drones are constantly controlled by the GS under this architecture, and they always need to be connected to it in order to receive commands, which can limit the mission's ability to work properly in some circumstances. Drone-to-drone communications and implementing mechanisms that would allow drones to complete missions even when they were not connected to a GS were planned as part of the future work of this student's Dissertation, both of which were addressed in this Dissertation.

One drone may not be sufficient for a single mission. A connection between the

drones and the Ground Station is required when using multiple drones. To overcome this problem, the research published in [23] looked into implementing a new routing protocol between the network's various nodes. The authors did this by using the location of the drones and their flight itineraries. The authors put their technology produced in real drones to validate the features. They also examined more specific scenarios like drone replacement, such as when a drone runs out of battery and needs to return to base or when a drone collides. Finally, it was determined that the protocol developed outperformed previous routing protocols to ensure quick route updates and high, consistent throughput.

The work in [24] deals with a scenario in which a drone is conducting a mission and must send data to a Ground Station, mainly live video data. This requires a continuous connection between the two ends. Relay drones are used when the drone is too far away from the operator and loses communication. These connect to increase the network's maximum range. Drones should be placed evenly spaced in open environments. However, this is usually not true when near high obstacles or uneven terrain, because the topography will limit the relays' ability to send data. This study proposes a new distributed protocol to enhance throughput (the rate at which information is sent through a network) and packet delivery ratio (ratio of packets successfully received) for the latter case. Each relay used software to determine the ideal placement by evaluating the Packet Delivery Ratio and sharing those values with the rest of the links. The experiments demonstrated that, moving a drone further away from the mid-point between links can increase network quality, indicating that asymmetric links can be beneficial. The gains in packet delivery and throughput were 40% and 300%, respectively. [25] adds to this research by contrasting two medium access protocols for this scenario: Carrier Sense Multiple Access (CSMA) and Time-Division Multiple Access (TDMA). The same authors used the same scenario to test the relay implementation in [26]. The goal was to optimize the network once more by determining the ideal placement for each drone. However, this research focused on establishing two separate approaches to accomplishing this: one centralized and the other distributed. The centralized method was implemented in its entirety in the Ground Station, taking advantage of the potential for increased computing capability. The GS did all of the math and only sent the results to the drones. In the distributed approach, each drone would track its neighbors' locations and compute the necessary metrics independently. The investigators used simulations to replicate a real scenario as closely as possible. The data analysis determined that both approaches completed their tasks successfully, converging from an equidistant to an asymmetrical placement with an increase in the end-to-end packet delivery ratio of up to 15%. However, because the drones have less powerful hardware, the calculations took 1.4 times longer on average in the distributed method than the centralized one. Nonetheless, the distributed method provides a mechanism for lowering platform overhead and controlling drones more naturally.

2.6 Summary

This chapter showed how UAVs are important now, by replacing the way various tasks were done in the past. We also discussed some of the existing control platforms on the market, as well as research done to improve present features in the autopilots' system and control platforms involving several drones.

The most significant point to emphasize is that the work done in [1] served as a basis for the work done in this Dissertation, which used many of the capabilities offered by the previously built framework to make development substantially easier. This Dissertation focused on some of the existing platform's limitations and attempted to address them, most notably the fact that the current architecture was centralized, which meant that drones only received commands from the GS, which could be a problem if this connection was lost due to network issues or the drone being too far away. This issue was addressed presenting a more decentralized architecture in which drones may make their own decisions based on certain conditions without relying on the GS. These issues are addressed in greater depth in the following chapter.

Chapter 3

Architecture

In this chapter, we will discuss the architecture that existed previously as part of another Dissertation, as well as the improvements that were proposed in this Dissertation to make the drones more independent of the GS. We describe the two separate modules that comprised the platform, one on the drones and the other on the GS, as well as how they communicate with one another and the new modules that were developed.

3.1 Base architecture

We must first summarize the work that served as the basis for this Dissertation before moving on to the proposed architecture. The MSc Thesis in [1] provided a framework for managing missions with several drones, as well as support tools for developing additional capabilities. We will look at some of its most important characteristics. Figure 3.1 shows a simplified version of the architecture presented in [1]. We did not cover several additional features in-depth in this Dissertation. This is because that portion of the architecture was not modified for this Dissertation: some sections of the base platform were not affected by the development of a more decentralized approach, and as a result, they are still working as before. We will only examine the most important features: the Drone Module and the Ground Station Module. These modules provided functions that were quite useful for this Dissertation. While we did not change the original code, some parts were integrated with other modules, and new components were introduced.

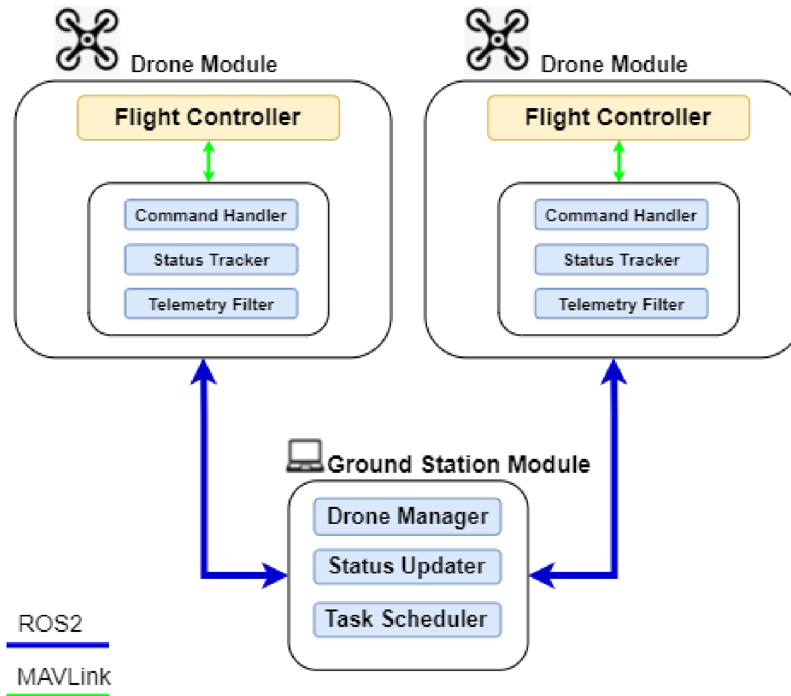

Figure 3.1: Base architecture.

The base architecture provided a more centralized approach, meaning the Ground Station commands everything and should always be connected to the fleet. The GS would not only manage and receive data from the fleet. but also send each instruction that made up a mission to the drones. This is the main limitation of this approach and a concern, since the GS will be unable to send those instructions if connectivity is lost at any point during a mission.

3.1.1 Drone module

The drone module includes software that acts as a link between the ground station, the onboard computer, and the flight controller of the drone. The MAVLink protocol connects the drone's FC and onboard computer, allowing the drone module to utilize those capabilities to transmit commands directly to the FC as well as receive telemetry from it, which can be accessed by the GS monitoring the flights. The drone module communicates with any external node via a ROS2 node, which subscribes to a topic to receive commands and publishes telemetry and status messages. This module can be used in real hardware (on a drone's onboard computer) or in combination with simulation software to emulate drone behavior for testing and debugging.

The drone module is comprised of the following parts:

- **Command Handler** - Subscribes to the **/cmd** topic to receive commands and orders the Flight Controller to perform them, as well as notifying the status tracker when a command begins, ends or fails.
- **Status Tracker** - Sends messages to the **/status** topic to notify the GS and the

mission running on the onboard computer that a certain command has initiated or completed, as well as other key events like the FC disconnecting.

- **Telemetry Filter** - Filters and publishes the most essential telemetry parameters received from the Flight Controller to the **/telem** topic, which can be used by the Ground Station and other drones, on a frequent basis.

3.1.2 Ground Station module

A ROS2 node in the Ground Station is utilized to communicate with the UAVs. The GS played a larger role in the previous architecture since it was more centralized, therefore only a couple of its characteristics were used in the current architecture, and only those will be explained:

- **Drone Manager** - This component stores telemetry and data from the drones so that other components can update and/or retrieve it. This module included a node that let external applications to interface with the UAV, including sending commands, however this feature was not required for this Dissertation.

- **Status Updater** - This module uses the ROS2 node to receive data from the drones and updates the information in the Drone Manager.

- **Task Scheduler** - This module is critical because it detects probable drone timeouts by verifying when a drone stops transmitting telemetry messages. If this happens, the user will be notified that something went wrong.

This module also had a component for mission planning, such as sending commands to the drone through an external application using a Representational State Transfer (REST) API or via HyperText Transfer Protocol (HTTP) using a dashboard, both of which were not used for this Dissertation.

3.1.3 Simulation tool

One of the most crucial parts from the previous Dissertation was the simulation tool. It is not a part of the central architecture, but it is essential to mention it. This component may launch numerous drone modules using a launch script, simulating a real-world scenario. PX4[30] firmware is used for simulation support[1], while MAVLink[20] is used for communication. They can be placed anywhere in the world and act just as if they were in a real-life situation. jMAVSim[2] can also be used for a more immersive experience. This tool made debugging much easier during this Dissertation's development process because it allowed us to test everything on a single computer before moving on to actual hardware in a real environment.

[1] https://docs.px4.io/master/en/simulation/
[2] https://docs.px4.io/master/en/simulation/jmavsim.html

3.2 Requirements

Before we go over the proposed architecture, we need to talk about what we will need to make it function. There is a list of requirements that must be fulfilled in order for the system to function properly.

Autonomous navigation

The ability for a UAV to move without the use of a Radio Control is one of the most obvious requirements. When a drone receives an instruction, it should carry it out without the need for human intervention. To meet this demand, each drone includes a mounted Flight Controller with autopilot capabilities. Only the most basic commands were required for this Dissertation, such as arming, taking off, landing, returning home, and traveling to a specific coordinate. However, someone should always be on hand with a RC to engage manual control in the event that something goes wrong or the drone behaves abnormally.

Telemetry gathering

The usage of GPS coordinates is one of the most basic components of today's drone systems. They must use the installed GPS module to obtain current coordinates in order to fly autonomously, as these will be used to calculate where to go next when a new destination is given. Driving with a RC in any flying mode that holds the drone in the same position (staying in the same place while not receiving any commands), such as Position Mode[3], requires the GPS as well. Apart from that, it is crucial to keep track of the drones' positions during the flight to see if anything goes wrong.

Network monitoring

Drones require monitoring the network with the different nodes to know who they can communicate with. A network measuring method was added to the platform for this Dissertation, which was previously absent. Each drone performing a mission will run software in the background that checks the network's strength with the Ground Station and/or other drones. This information could be utilized for logging purposes only, or it could be used to influence the mission's progress.

Communication

Since the platform is composed of different nodes and modules working together to accomplish a mission, they need to communicate. A drone has two types of communication. The first one is internal, between the onboard computer and the Flight Controller, where the onboard computer sends commands to the FC and receives telemetry. As explained before, this part of the communication is done using the MAVLink protocol

[3] https://docs.px4.io/v1.12/en/getting_started/flight_modes.html#position-mode-mc

and is physically done through a Universal Serial Bus (USB) cable, connecting both devices. The second type is external between the drone and other devices in the same network (other drones or the Ground Station). This part of the communication is done through ROS2, with each device in the network running a ROS node. The Message Queuing Telemetry Transport (MQTT)[4] protocol is used to communicate from and to Android devices when they are integrated. The different nodes communicate using an ad-hoc network in real-world circumstances. The benefit of using this type of network is that all two devices need to communicate with each other is to be within range of each other, making it the best option for our scenarios.

Android Integration

We will also need the means to integrate Android smartphones with the new platform. An Android smartphone needs to connect with the rest of the platform, or at least a part of it, which may be the Ground Station or a single drone. We will develop an Android Application that allows a user to perform requests and receive feedback from the drones to achieve this.

3.3 Proposed architecture

In order to meet the previously described requirements, we propose the architecture illustrated in Figure 3.2.

Figure 3.2 depicts a simplified form of the architecture, using the architecture proposed in [1] as a starting point. The components listed in blue are those that already existed, whereas the components marked in green are those that were added for this Dissertation.

There are two main modules in this architecture: the Drone Module and the Ground Station Module, both described in detail in the Section 3.1. The Drone Module is made up of everything that runs on the onboard computer, including the drone module software and mission scripts, and the Flight Controller, which is wired to the onboard computer and receives commands from it as well as sending telemetry from the drone, using the MAVLink protocol for this. The Ground Module software and the scripts for managing requests from drones created in this Dissertation operate in the device that serves as Ground Station.

As previously stated, MAVLink is used to communicate between internal components of a drone (such as the FC and onboard computer). In contrast, ROS2 is used to communicate between drones and the ground station. However, unlike the base architecture, this one allows drones to connect directly with other drones using ROS2. Drones will now be able to read the mission status of other drones, telemetry and network strength monitoring, which will influence the missions' evolution. Drones can now execute missions without needing a constant connection to the Ground Station since they can now make decisions based on data received straight from other drones.

[4] `https://mqtt.org/`

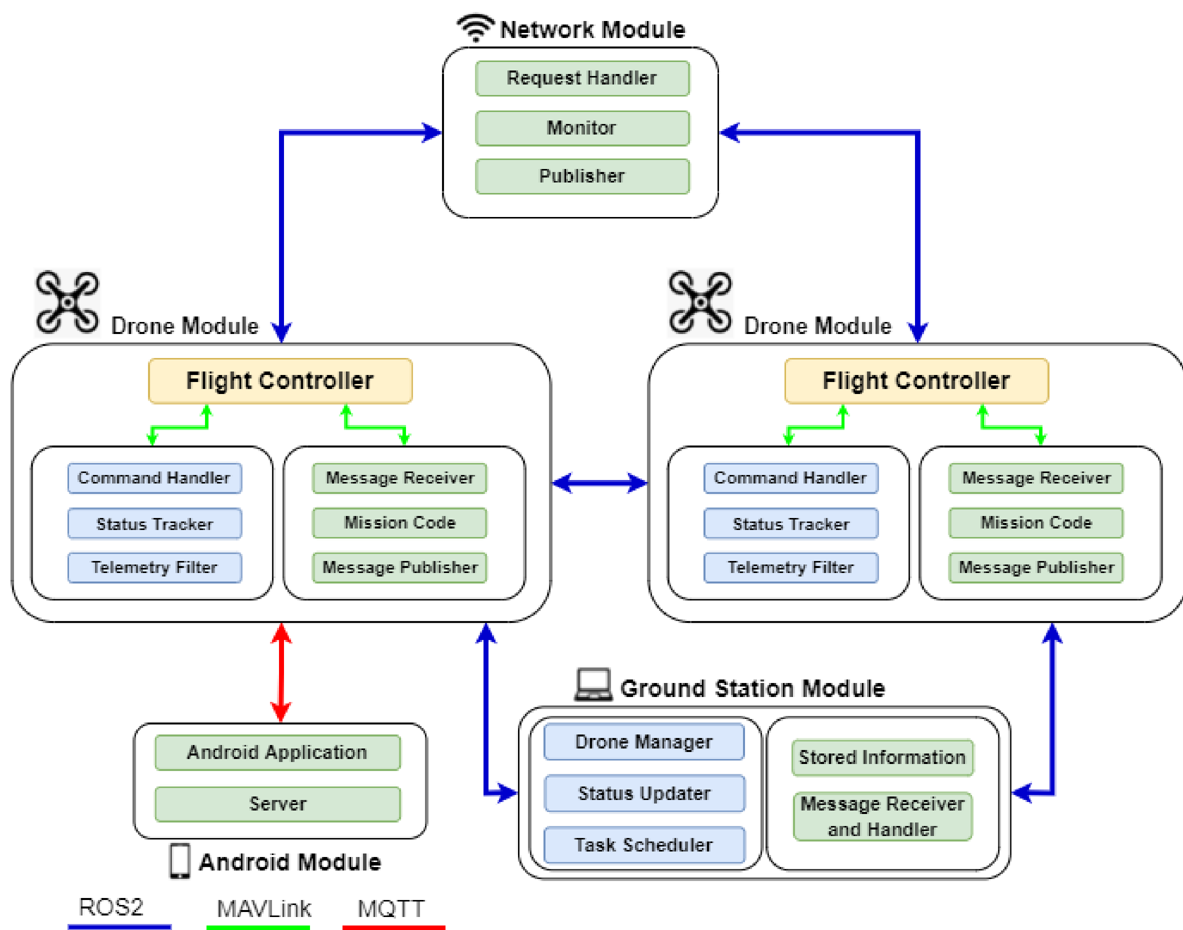

Figure 3.2: Proposed architecture.

A network module was also created. This module allows drones to interact with other drones or the Ground Station to evaluate network conditions. It sends packets to a target node on a routine basis, calculating various metrics that can be used to examine the network's status. This information can play a more significant role in some cases because one may program a drone to do something specific if a connection to a particular node is lost or below a configured threshold.

Android device integration is also a new feature. Aside from the existing ROS2 communication, a new communication protocol called MQTT has been established to communicate with these devices. In addition, an application was developed to enable this communication using a simple interface, so a user can use it to send and receive messages from a drone.

The new components made in the Drones and Ground Station modules and the new modules developed in this Dissertation are described below.

3.3.1 Network module

- **Monitor** - Monitors the network, computing various metrics such as signal strength to assess the network's status between drones and the Ground Station. This module is critical since the drones will use it to ensure that connectivity does not fail. It can be run in just some of the nodes or all of them, depending on the scenario.

- **Request Handler** - Receives requests for specific connections to be monitored and forwards them to the Monitor.

- **Publisher** - Publishes the results of the measurements in a specified ROS2 topic.

3.3.2 Android module

- **Android application** - The Android application will allow an Android device to communicate with the rest of the platform via the MQTT protocol.

- **Server** - The MQTT core is launched on the server, allowing other nodes to publish and subscribe to this broker. It is possible to put it in a drone or the Ground Station.

3.3.3 Drone module

- **Mission Code** - The mission code now runs directly on the drones, eliminating the Ground Station's dependency. This comprises everything that goes into creating a mission, from a basic set of target coordinates to more complex computations.

- **Message Receiver** - During the course of a mission, listens for messages from the Ground Station and/or other drones for crucial information that could affect the mission's outcome.

- **Message Publisher** - During some stages of missions, the publisher's responsibility is to inform or make requests to other nodes of the platform, such as drones or the Ground Station.

3.3.4 Ground Station module

- **Stored Information** - Drone-related information, such as which drones are available to perform missions.
- **Message Receiver and Handler** - Receives and replies to requests from drones, updating the Stored Information.

Message format

A new drone-drone communication point has been established to allow drones to communicate with one another. The drones communicate by publishing and subscribing to messages using a designated topic in ROS2. A consistent message format was utilized to keep everything in order. Messages are sent in JavaScript Object Notation (JSON) format and must include the following fields:

- The sender's identifier (which might be a drone or the GS).
- The receiver's identifier, the node in which the message is destined (which might also be a drone or the GS).
- The type of the message, defining the purpose of the message.
- The date and time the message was sent.
- Because the type is insufficient to send all essential information, some messages have additional specific fields.

3.4 Summary

In this chapter, we looked at the platform's architecture. The work done in a prior Dissertation, on the other hand, was extremely important because it served as a foundation for the work done in this Dissertation. Thus, even though the author of this Dissertation did not create it, its main elements were discussed, as well as the improvements made to enable a more decentralized platform, allowing drones to be more independent from the Ground Station.

We also described the new modules created in this Dissertation (Network and Android) that are now part of the architecture, their components, and purpose.

We will look at how these modules were implemented in more detail in the upcoming chapter.

Chapter 4

Implementation

The implementation of the various modules will be discussed in this chapter. First, we will go over a general overview, and then we will describe each module in-depth, including what it is composed of, what it does, and how it works. Then we will go over some of the scenarios created to test these modules, including their components and how they operate in these scenarios.

4.1 Modules

Four modules form the platform - Drone, Ground Station, Network, and Android. Depending on the scenario, they may work individually or together.

4.1.1 General overview

The several developed modules give a range of new platform functionalities. It is worth noting that we are using the architecture illustrated in the Figure 3.2 as a reference. Each module has a set of interfaces that allow it to communicate with other modules.

The Drone Module includes a ROS2 interface that allows it to communicate with any other module that uses the same protocol, such as other drones or the Ground Station. This is performed by posting messages on topics and subscribing to them. Physically, we can accomplish this by using a built-in wireless interface in the onboard computer. However, if such does not exist, it is possible to use a USB wireless adapter instead. The onboard computer must, then, be configured to connect automatically to the platform's network upon booting. Therefore, they can communicate with each other as long as they are connected to the same network. The Flight Controller is also connected to the onboard computer via a wired USB connection. It can communicate in this manner. This is accomplished by using MAVLink, which is the standard method for this type of hardware. Through this link, it may transmit commands directly to the Flight Controller and receive information from it, such as telemetry from the drone. We do not need to know the low-level commands required to make a drone fly because [1] gives a form to complete this task easier. Instead, we merely need to send a message to a specific topic, which the drone module will gather and convert into the required data before sending it

to the Flight Controller. The mission code might also be executed by the drone's onboard computer, making the mission run entirely by the drone.

A ROS2 interface is also available on the Ground Station module to connect it to the network. We can quickly run a set of software with this module that connects to the network, and promptly identify any drones online by verifying published telemetry. We can also use this software to monitor the commands that each drone receives, which can be helpful in debugging. Sending commands is also possible. Finally, the Ground Station could be utilized to control certain aspects of drone missions. For example, it can receive and respond to drone messages, such as requests. Even in a more decentralized architecture, the Ground Station should always handle these requests because it is the module that oversees the entire fleet. Another significant feature of this module, also included in [1], is displaying any error messages sent by a drone. This could happen, for example, if a drone refuses to obey an order for some reason.

The network module is a brand-new module with the primary goal of integrating with other modules to monitor current network conditions. It also communicates with the rest of the platform through ROS2. Because it is aimed to be used inside a drone or Ground Station, it does not require any additional physical connections as it is a single script. This software does a network assessment on a selected target connection in the background and uses the information to produce specific metrics that are then published to a topic. This data could be gathered by the same node performing the evaluation or by multiple nodes in the platform. What they do with it is determined by the scenario.

The Android module is also new, and it contains significant differences from the other ones. One significant difference is that, unlike the other modules, it does not connect with the rest of the platform using ROS2. MQTT is used instead. This is because ROS2 connectivity with these systems is challenging to develop. MQTT is similar to ROS2 in that it employs a publish and subscribe model. However, a node must be linked to a server to conduct these actions. A server can be located in either the Ground Station or a drone. The fact that Android does not support ad-hoc connections creates an issue. As a result, a separate mechanism through a wireless technology, such as Wi-Fi, is required to facilitate communication. This complicates the platform because it requires two networks: a standard ad-hoc connection for the drones to communicate with one another and with the Ground Station, and a Wi-Fi or similar type of network for the server to link with the smartphone. The Android device can recognize and connect to that network if the server's node also runs an Access Point. Another option is to link both ends to a pre-existing network, as long as the Android system recognizes it. The nodes can now exchange messages using the MQTT protocol because they are connected to the same network.

Since different modules are implemented, it is essential to explain how we implemented their communication before moving on to a more detailed description of each module.

4.1.2 Communication between modules

In practice, the author of this Dissertation had the choice of using Java, Python, or C++ to accomplish this. C++ was initially dismissed because it is the language in which

there was less experience with, followed by Java because Python was easier to work with, so ROS Client Library for Python (RCLPY) was used.

The most commonly used library throughout development was RCLPY. It comes with utilities that make it simple to send and receive messages via ROS2 using Python. ROS2 uses topics to publish and subscribe. This means that when a node publishes a message to a certain topic, it will be received by anyone who has subscribed to that topic. Of course, this is only possible if the subscriber is in the same network as the publisher and can communicate with it.

A simple example of how we publish a message is shown below.

Example Send a message using RCLPY

```python
from rclpy.node import Node

node = rclpy.create_node("demo_node")
publisherCmd = node.create_publisher(String, "foo", 10)

msg = String()
msg.data = "bar"
publisherCmd.publish(msg)
```

In this example, the message "bar" is published to the topic "foo". The first step is to build a node. One Python script must have at least one node to post messages. Then, we establish a publisher that specifies the type of message that will be published on that topic, which in our case is String. We also supply the topic's name and the QoS profile. Quality of Service refers to a set of parameters that alter how a publisher or subscriber behaves while carrying out their duties. Using the value 10 as an argument uses the default settings, which satisfy our purpose and do not necessitate a thorough understanding of how they work. We need to declare a String and the data we want in the attribute *data* after setting up the publisher. Then, finally, we use the publisher's `publish` method to publish the message by passing it as an argument.

Below, there is an example of how to subscribe to a topic.

Example Receive a message using RCLPY

```python
class DemoSubscriber(Node):
    def __init__(self):
        name = "demo_subscriber"

        super().__init__(name)
        self.subscription = self.create_subscription(String,"foo",
        self.listener_callback,10)
```

```
8
9     def listener_callback(self, msg):
10        print(msg.data)
11
12 def main():
13    demo_subscriber = DemoSubscriber()
```

To subscribe to a topic, we must first construct a class that contains the subscription's configuration. We supply the type of message, the name of the topic, the callback function name, and, once again, the QoS settings using the method `create_subscription`. The callback function is a sequence of instructions that are executed each time a message is received. In this example, we merely print the message's content. After configuring everything, we call the class in the `main` function to start the subscription.

These two implementations are essential to the architecture's success. This code is used to allow communication between the various nodes that use ROS2. A drone can send a message to another drone, and the other drone, if subscribed, will be able to receive it and respond as needed. The messages delivered might be merely informative or control messages that, upon reception, could change something in the receiver's mission, depending on the situation. However, this part only covered the ROS2 communication.

Let us now look at sending and receiving messages via the MQTT protocol because the Android device will only interact in such way. The MQTT broker functions in the same way as ROS2. One of its key disadvantages is that it is based on a centralized architecture. This means that one node must be connected to a server to publish or subscribe to messages. Paho-MQTT[1] was used to enable this, providing tools to interact with the MQTT broker in Python. Below is an example of code for how a drone could use this library to send a message. The Android application will be able to receive this information in this manner.

Example Send a message through MQTT using Python

```
1 import paho.mqtt.client as mqtt
2
3 client = mqtt.Client("demoClient")
4 client.connect("192.168.1.1")
5 client.publish("foo","bar")
```

This library's publishing procedure is simple. First, create a client and give it a name. Then, by providing an Internet Protocol (IP), we utilize the `connect` method to connect to the server, which will only work if the server is up and running. After that, we call the `publish` method, passing the topic and message to be published as parameters. The client publishes "bar" in the topic "foo" in the example above.

[1] https://pypi.org/project/paho-mqtt/

The procedure to subscribe to a topic is, just like in ROS2, a little more complex.

Example Receive a message through MQTT using Python

```python
import paho.mqtt.client as mqtt

def MQTTHandler(client, userdata, message):
    print(message.payload)

def main():
    client = mqtt.Client("demoClient")
    client.connect("192.168.1.1")
    client.on_message = MQTTHandler
    client.subscribe("foo")

    client.loop()
```

The subscriber follows the same steps as the publisher. First, create a client and connect it to the server as the initial step. Then, using the `on_message` method, we must specify a callback function, just like in ROS2. Then we select which topics we want to be subscribe to. Finally, we call the `loop` method to begin the subscription process. In this example, when a message is received, the client prints the content of that message by reading the attribute *payload*.

This completes the MQTT communication for the drone and the GS. We will now show how to perform the same mechanism using an application. Again, the same library was used, but this time for the Android application. Because listeners must be used to perform the connections to the server and ensure that messages are sent, the actual code is more complex. The application was developed using Android Studio[2] and Java language.

Example Send a message through MQTT using the Android application

```java
import org.eclipse.paho.android.service.MqttAndroidClient;
import org.eclipse.paho.client.mqttv3.MqttClient;

String clientID = MqttClient.generateClientId();
client = new MqttAndroidClient(getApplicationContext(),
"tcp://192.168.1.1:1883", clientID);

String message = "bar"
client.publish("foo", message.getBytes(), 0, false);
```

[2] `https://developer.android.com/studio`

On the Android side, we use a similar procedure. First, we declare a client identifier and associate it with a client. The IP address is then used to connect to the server, which requires us to supply the protocol, Transmission Control Protocol (TCP), and the port on which the server is executing. To send the message, we must first define it and then use the `publish` method to send it. The topic's name, the content of the message, the QoS settings, and a retained flag are passed to this method. The supplied QoS profile was 0 because that is the default, and this was a simple interaction that required nothing more. The retained flag merely tells the server if it wishes to keep the published message if another node does not receive it right away, which was not required for our app.

To finalize the inter-module communication, we show how to subscribe to and receive messages through the app.

Example Receive a message through MQTT using the Android application

---

```
1  import org.eclipse.paho.client.mqttv3.IMqttActionListener;
2  import org.eclipse.paho.android.service.MqttAndroidClient;
3  import org.eclipse.paho.client.mqttv3.MqttClient;
4
5  String clientID = MqttClient.generateClientId();
6  client = new MqttAndroidClient(getApplicationContext(),
7  "tcp://192.168.1.1:1883", clientID);
8
9  client.subscribe("foo",0,null, new IMqttActionListener() {...} )
10
11 client.setCallback(new MqttCallback() {
12   public void messageArrived(String topic, MqttMessage message) {
13     if (topic.equals("foo")) {System.out.println(message.getPayload())}
14 }
```

We connect the client to the server to receive a message. The `subscribe` method is then used to provide which topics we want to receive messages from and the QoS profile and user context. The topic is "foo," and the rest of the arguments are set to default. Finally, we create a client callback and use the `messageArrived` function to indicate what happens when a message from the "foo" topic is received. For demonstrating purposes, we only print that message.

This completes the communication between modules implementation. Using these libraries and methods, we can use a module to communicate with other modules. These were only simple examples, and the messages sent and actions conducted in the actual implementation are more complex, as covered in the rest of this chapter. Following this, we have a detailed description of each module, including its purpose and implementation.

4.1.3 Network Module

The Network Module is used to determine the connectivity between the platform's nodes, drones, and the Ground Station. This module can be used just to keep track of connectivity or, in some cases, it can even influence the mission's outcome, as discussed later in this Dissertation. It is structured as follows and can be linked with various modules (one entry point in the **Request Handler** and one exit point in the **Publisher**).

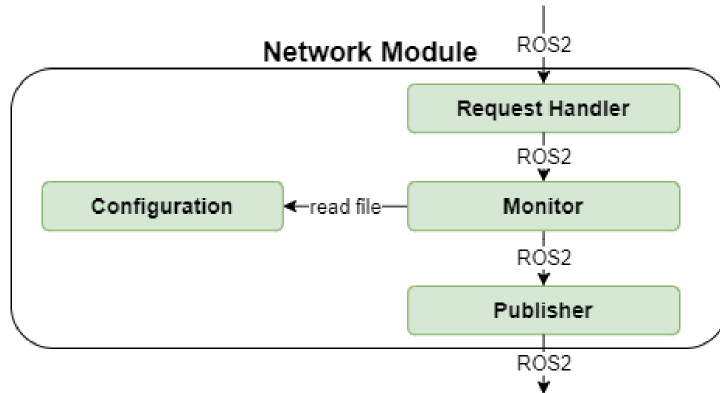

Figure 4.1: Network Module Implementation.

When the module is deployed, the **Monitor** executes the algorithms to perform the calculations, acquires the network's quality, and publishes the results in a ROS2 topic at a set rate. It begins by reading a configuration file and monitoring the connection specified in that file. At any time, we can submit a request to a designated ROS2 topic to update the connection we wish to monitor, which will be picked up by the **Request Handler** and forwarded to the **Monitor**. The latter stops monitoring the prior request when it selects the new one, yet a single request can have multiple connections. Because the monitor utilizes network probes, we must first comprehend what they are. Although the meaning of Probe varies, it is often understood to be a network monitor. They are commonly used in many networks throughout the world, and they are used to monitor traffic coming in and out in order to detect any problems that may arise, as well as to improve performance[3].

It is not required to utilize Probes in such a complex method in this Dissertation, but we must use them to check for a connection with a certain node. A probe, for example, could be used by a drone to test its connection with the GS. A script called **Network.py** was written to accomplish this. It is important to highlight that this study was done for PEI (Projeto em Engenharia Informática) by another group, not by the author of this Dissertation. This Dissertation's author just merged their work with the rest of the platform that had been constructed.

Every existing node on the platform has an IP and Media Access Control (MAC) address, just like any other device. This script sends a series of packets to a certain IP and

[3]https://www.helpsystems.com/resources/articles/everything-you-need-know-about-network-probes

MAC address, and utilizes the time it takes for them to arrive and bitrate to determine the signal strength to that node and measure network quality. The measurement's result is then published to the **/sensor/network/info** topic. The message is sent in JSON format ("attribute" : "value") and looks like this:

Example Simple JSON message format for network measuring

```
{
  "droneId": "drone01",
  ... ,
  "value": [{
    "droneId1": [{
      Information about connection between 'droneId1' and node X,
      Information about connection between 'droneId1' and node Y,
      }],
    "droneId2":  [{
      Information about connection between 'droneId2' and node Z,
      }]
  ]}
}
```

Some parameters were left out because they were unnecessary. The following are the most significant:

Table 4.1: The key parameters in a network measurement.

Parameter	**Definition**
droneId	drone that has requested the message
value	contains a list of the different measures that have been requested by the drone in the attribute "droneId". This "value" attribute has a list of JSON messages as its value, with each element of that list having the source node as an attribute and a list of one or more target nodes to whom the connection will be tested as its value.

Let us look at a full example to see how this works. In this case, the drone with the identifier "drone01" has requested a connection test with the node "groundStation", which has the MAC address *f4:f2:6d:0d:c0:7f* and the IP *10.1.1.103*.

Example A complete example of a JSON message format for network monitoring


```json
{
  "droneId": "drone01",
  "sensorId": "real",
  "type": "network",
  "timestamp": 1627033747604,
  "value": [{
    "drone01": [{
    "Station": "f4:f2:6d:0d:c0:7f",
    "IP": "10.1.1.103",
    "Latency": "1.82 ms",
    "Signal": "43.46",
    "TxByte": "593626",
    "RxBit": "54.0 MBit/s",
    "TxBit": "54.0 MBit/s",
    "Relay": "groundStation",
    "NetworkQuality": "HIGH"
    }]
  }]
}
```


The result contains a number of parameters that will be used to assess the network's strength.

These parameters have the following meanings:

Table 4.2: Parameters list for each connection in a network measurement.

Parameter	**Definition**
Station	The MAC address of the node that was used to test the connection from "drone01."
IP	The IP address of the node that was used to test the connection from "drone01."
Latency	The average time it took for a packet to arrive from the source to the destination (in milliseconds)
Signal	The connection's signal strength, defined by a value between 30 and 90 (a lower value indicates a better connection)
TxByte	Number of bytes transmitted so far to test this connection
RxBit	Bitrate of bits being received
TxBit	Bitrate of bits being sent
Relay	The name of the node that was used to test the connection from "drone01".

NetworkQuality	Takes into account all relevant criteria and assigns a network quality rating of Low, Medium, or High

The following method is used to determine network's quality:

```
1 if (Signal) < 60 and (Latency) < 50 and (TxBit) > 5:
2     NetworkQuality = "HIGH"
3 elif (Signal) <= 75 and (Latency) < 100 and (TxBit) > 3:
4     NetworkQuality = "MEDIUM"
5 else:
6     NetworkQuality = "LOW"
```

The outcome is calculated using the signal strength, latency, and bitrate of the bytes transmitted, as shown above.

Configuration File

The Network.py script is run with:

```
python3 Network.py -c Network.yml
```

To run, we must provide a file (**Network.yml**) with a network configuration as an argument. Here is an example of a file like this:

Example Configuration file

```yaml
droneId: drone01
sensorTopic: /sensor/network/info
infoTopic: /sensor/network/requests
telemTopic: /telem
rateHigh: 200
rateMedium: 300
rateLow: 400
groundStation_IP: "10.1.1.103"
groundStation_MAC: "f8:d1:11:08:2e:b1"
interface: "wlan0"
```

The following are the definitions of the parameters:

Table 4.3: List of parameters in the configuration file.

Parameter	**Definition**
droneId	The identifier of the drone that is running the script, the same drone that will be making the queries
sensorTopic	The topic for which network measurement results will be published
infoTopic	Custom connections measurements are requested for this topic:
telemTopic	The topic in which drones communicate telemetry, so that the script may check if the drone that is requesting connections is up and operating
rateHigh	If the network quality is high, the interval of time (in milliseconds) in which messages will be sent out
rateMedium	If the network quality is medium, the interval of time (in milliseconds) in which messages will be sent out
rateLow	If the network quality is low, the interval of time (in milliseconds) in which messages will be sent out
groundStation_IP	IP address from the GS node
groundStation_MAC	MAC address from the GS node
interface	The network interface of the device that the script will use to test the network connection

When this script is launched, it will continue to measure and publish the results for the connection between the drone that is running the file and the GS, using the IP and MAC defined in the **groundStation_IP** and **groundStation_MAC** parameters, respectively. It does so at a set rate, which is determined by the rates defined in the configuration file and the current network's quality:

```
1  if quality == "HIGH" and self.rate != self.rate_high:
2      ...
3      self.rate = self.rate_high
4
5  if quality == "MEDIUM" and self.rate != self.rate_medium:
6      ...
7      self.rate = self.rate_medium
8
9  if quality == "LOW" and self.rate != self.rate_low:
10     ...
11     self.rate = self.rate_low
```

The higher the network's quality, the more messages it will publish; this is done to prevent the network from becoming overloaded when the quality is low.

Measuring a different network

Although the script maintains publishing these statistics, we can request a network test on a network other than the one connecting the drone to the GS at any moment. This is accomplished by publishing the request to the **sensorTopic** parameter's specified topic, in this example **/sensor/network/info**. The IDs, IP addresses, and MAC addresses of the nodes in which we want to evaluate the network must be included in the message:

Example Simple JSON message format for a custom network measuring


```json
{
  "droneId": "...",
  "connectionsId": [ ... ],
  "connectionsIp": [ ... ],
  "connectionsMac": [ ... ]
}
```


The parameters *connectionsId*, *connectionsIp* and *connectionsMac* contain a list with the group of nodes in which we want to measure the network. In order for everything to work properly, all three lists must have the same size.

An example of the drone01 requesting a network evaluation with the drone02 and drone03 is shown below:

Example A complete example of a JSON message format for a custom network measurement request


```json
{
 "droneId": "drone01",
 "connectionsId": ["drone02" "drone03"],
 "connectionsIp": ["10.1.1.101" "10.1.1.102"],
 "connectionsMac": ["ff:ff:ff:ff:ff:ff" "ee:ee:ee:ee:ee:ee"]
}
```


When sending this message, the **Network.py** script will receive it and perform the required calculations. When this task is performed, it is published in the appropriate topic. The message will look like this:

Example A complete example of a JSON message format for a custom network measurement response


```
{
  "droneId": "drone01",
  ...
  "value": [{
  "drone01": [{
    "Station": "ff:ff:ff:ff:ff:ff",
    "IP": "10.1.1.101",
    ...
    "Relay": "drone02",
    "NetworkQuality": "HIGH"},
    {
    "Station": "ee:ee:ee:ee:ee:ee",
    "IP": "10.1.1.102",
    ...
    "Relay": "drone03",
    "NetworkQuality": "HIGH"
    }]
  }]
}
```


4.1.4 Android Module

The Android Module allows Android devices to be integrated with the rest of the platform. For example, an Android application can be used to connect with the Ground Station or drones and perform tasks such as requesting a drone for a specific location.

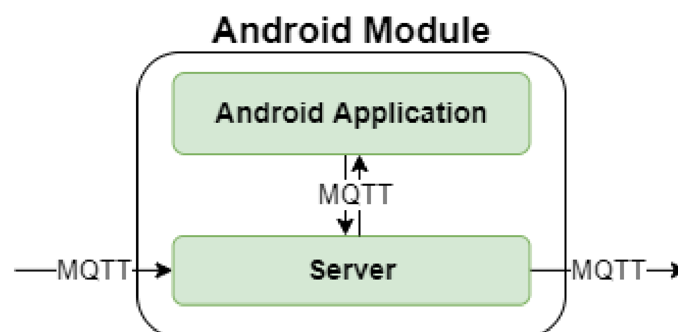

Figure 4.2: Android Module Implementation.

The Server and the Android application are the two components of this module. However, one can use the Server to connect this module to others.

Because ROS2 on Android has some limitations, a new method of connecting with the rest of the system was used. The MQTT protocol will be used by this module, since it is a lightweight mechanism of communications utilizing a publish/subscribe structure.

Unlike ROS2, which is a point-to-point broker, MQTT is a centralized broker, which means that it needs a server to connect clients to. This server can be run on either the GS or a drone.

Server

The **Mosquitto**[4] broker was used to run the server because it implements the MQTT protocol and is the most extensively used message broker for this purpose. Furthermore, it is quite simple to use. After installing, running this command on Linux will start the server:

```
sudo systemctl start mosquitto
```

The following command can be used to send a message to the topic "bar," which will result in the message "foo" being published to that topic:

```
mosquitto_pub -m "foo" -t "bar"
```

We can subscribe to the topic "foo" using the following command, and each message received will be displayed in the terminal:

```
mosquitto_sub -t "foo"
```

Although Mosquitto comes with a set of preset configurations that work, it is preferable to utilize a custom configuration. The below is the one that was used:

Example Mosquitto configuration format 1

```
bind_address 192.168.X.X
port 1883
allow_anonymous true
```

The MQTT server's IP and port are defined in the first two lines. The IP is the address of the server's machine, and the port is generally 1883, which is the standard for MQTT communications.

The third line makes it possible for any device to connect to the MQTT server and publish/subscribe without requiring authentication.

However, both first lines can be replaced with a single one:

[4] https://mosquitto.org/

Example Mosquitto configuration format 2

```
listener 1883 0.0.0.0
allow_anonymous true
```

We do not have to specify the IP address because 0.0.0.0 will do it for us. However, this configuration had a drawback because the Android device would not connect to the server, therefore the first one was utilized instead.

Android Application

The commands presented before allow us to interact via MQTT using a Linux terminal; but, in order to integrate this connection with an Android system, we must use a specialized library. A small application was created in Android Studio using the **paho-mqtt** library to make this possible.

The user is provided with two text boxes and a button when the application first opens. The IP address of the server is entered in the first text field. This is the IP of the server node, the same IP of the machine that runs the server. The phone's ID is the second. The smartphone, like the drones, needs to have a unique identification because it will be used to communicate with the system.

The rest of the interface comprises a single button for requesting or canceling a drone, as we can see in Figure 4.3.

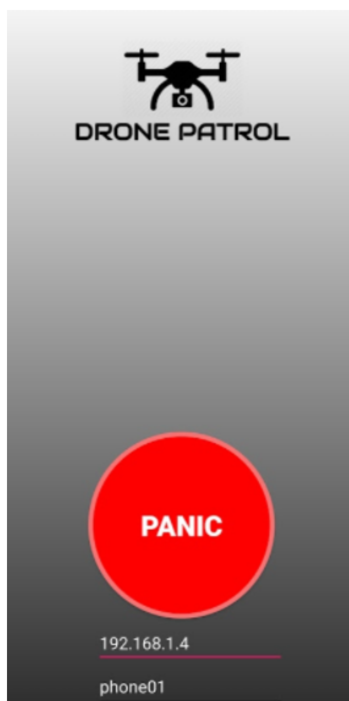

Figure 4.3: The user interface of the application.

There was only one scenario in which the application was used. As a result, the format of the messages sent and received and how the application interacts with the rest of the system are covered in that scenario's explanation.

4.1.5 Drone Module

The Drone Module comprises the different components that allow the drone to fly autonomously and be monitored.

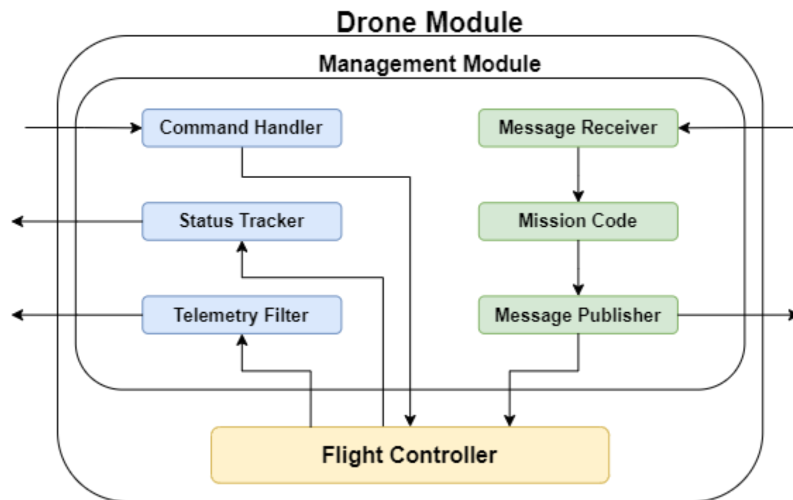

Figure 4.4: Drone Module Implementation.

The Flight Controller is a low-level piece of hardware that connects directly with the drone's electronic components, such as the ESC, and is part of the Drone Module. MAVLink is used to communicate with the FC using a wired USB connection, while ROS2 is used for the rest of the module's internal and external interfaces.

The parts marked in blue represent the modules that were developed in [1], and described in Section 3.1.1, while the parts marked in green represent the components that were developed in this Dissertation.

The Mission Code is the code that contains the list of commands that the drone must follow to execute a mission. Missions come in many different forms. For example, one of the most common instances is a drone carrying a set of sensors (e.g. temperature), mapping a specific area using those sensors to recover data. The Mission Code in this example would be a set of coordinates that the drone had to travel to and a separate script that would run the code that communicates to the physical sensor.

A set of functions was created to support the various commands. The **Command Handler** receives an order by reading a message in the ROS2 topic **/cmd**. These messages have a specific format and are transformed into a low-level instruction and sent directly to the Flight Controller over USB. The RCLPY library is used to send commands. The table below shows the available commands, with the restriction that only the most commonly used ones are supported, as only the most basic ones were required for this Dissertation.

Table 4.4: Supported commands in the Drone Module.

Function name	Description	Parameters
arm_cmd	Arm the drone.	droneId
takeoff_cmd	Arm and takeoff the drone.	droneId alt - altitude (optional)
land_cmd	Land the drone.	droneId
goto_cmd	Send the drone to a specific coordinate.	droneId lat - latitude lon - longitude alt - altitude (optional) yaw (optional) speed (optional)
cancel_cmd	Cancel the current command on execution.	droneId
return_home_cmd	Land the drone in the same location from where it took off.	droneId

Each function converts the command into the equivalent ROS2 message and publishes it to the **/cmd**, which is subsequently picked up by the **Command Handler** and forwarded to the FC. No additional commands were required for the scenarios in which we tested this architecture. Because the scripts that utilize these commands are run in drones, if we do not provide a droneId for these commands, it will use the default droneId, which is the ID of the drone in which the script is running.

If no altitude is specified for the takeoff instruction, the default altitude specified in the configurations (as stated in [1]) is used instead. In this situation, the altitude is the height above ground level.

The altitude is defined in Above Mean Sea Level (AMSL), the speed is defined in meters per second, and the yaw angle is positive for clockwise direction, with 0º being North in the case of the goto instruction. If no yaw is specified, the drone will compute the proper bearing in the direction of the target location, and if no altitude is given, it will use the current. If the speed is not specified, the maximum safe speed of 5 m/s is used.

Another useful function created was waitForTask, because while giving a command to a drone, we may only want to send the following command once the current one has been completed. By providing the name of the command and the state of that command (start or finish) as parameters, the code will listen to the **/status** topic until that message arrives and only then move to the next order.

The code below demonstrates a simple example of sending a mission to a drone using the code.

Example Mission consisting of taking off, going to a coordinate and landing

```
takeoff_cmd()
waitForTask(cmd='takeoff',state='finish')

goto(40.634450026723975, -8.659722852039286, alt=20, speed=5)
waitForTask(cmd='goto',state='finish')

land_cmd()
waitForTask(cmd='land',state='finish')
```

In this mission, the drone will take off from its current location, travel at a speed of 5 m/s to the provided location, and then land.

The **Message Receiver** and **Publisher** were also implemented using the RCLPY library, using the designed methods for subscribers[5] and publishers[6], respectively.

4.1.6 Ground Station Module

The main objective of the Ground Station is to manage the fleet.

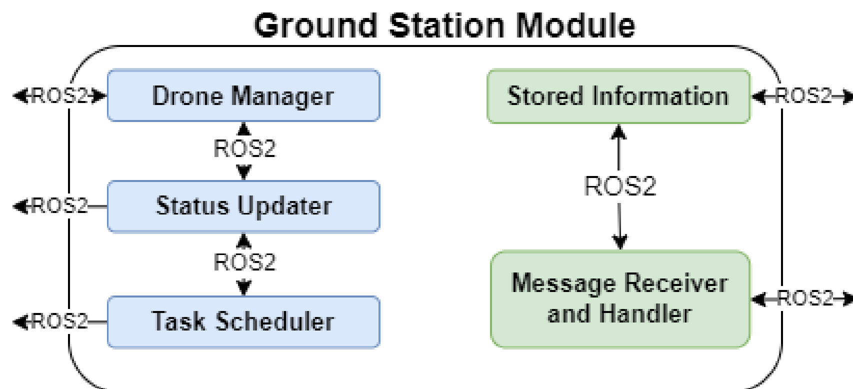

Figure 4.5: Ground Station Module Implementation.

The blue sections are the units developed in [1] and explained in Section 3.1.2, while the green parts are the elements created in this Dissertation.

The **Message Receiver and Handler** listens for requests from drones and replies to those requests using the stored data about the fleet, such as which drones are available to perform specific tasks. The **relay_gs.py** script is used to accomplish this.

Depending on the drone's type of mission and role, the data stored may be divided into different categories. With this more decentralized design, it is not necessary to keep the Ground Station connected to the fleet at all times because the missions will

[5] `https://docs.ros2.org/latest/api/rclpy/api/topics.html#module-rclpy.subscription`
[6] `https://docs.ros2.org/latest/api/rclpy/api/topics.html#module-rclpy.publisher`

proceed normally without it; nonetheless, this connection is required to monitor the drones' location and send commands.

4.2 Relay

The relay was the first scenario in which these modules were tested. When a drone is executing a mission, it is always beneficial to maintain a connection between the drone and the Ground Station to track its position and evaluate data that the drone may be recovering. For instance, if the drone is mapping a particular location and collecting data with a specific sensor, or if the drone has an attached camera and is transmitting video feed. The connection with the GS is not required to make the drones perform the missions because the mission code is running locally on the onboard computer; however, communication is necessary if we need to give further commands from the Ground Station or monitor what is happening. If a drone goes too far away from the GS during a mission, it will not be able to communicate. Therefore, a second drone will be dispatched for this purpose, maintaining a set distance between the first drone and the GS. This second drone will act as a bridge between the first and the GS. Any telemetry and data from the first drone will be forwarded to the Ground Station, and any command from the GS will be forwarded to the first drone. The second drone might also get too far away from the GS, requiring the usage of a third drone, and so on.

Figure 4.6: Example of a mission scenario.

Figure 4.6 represents a scenario in which a relay would be required. Each circle indicates the maximum signal range a node would have if it were positioned in the center.

The A circle symbolizes a drone on a mission, the B circle represents a relay, and the GS represents the Ground Station. Drone A would not communicate with the GS or vice versa if drone B was not stationed there because they are too far apart. This is a concern since, without direct eye contact with the drone and no ability to get its telemetry in the GS, there is no way to know the mission's progress or send a command to the drone in an emergency or to abort the mission.

4.2.1 Modules integration

Three of the designed modules - Drone, Ground Station, and Network - were employed to make the relay possible, and the method they communicate is shown in Figure 4.7.

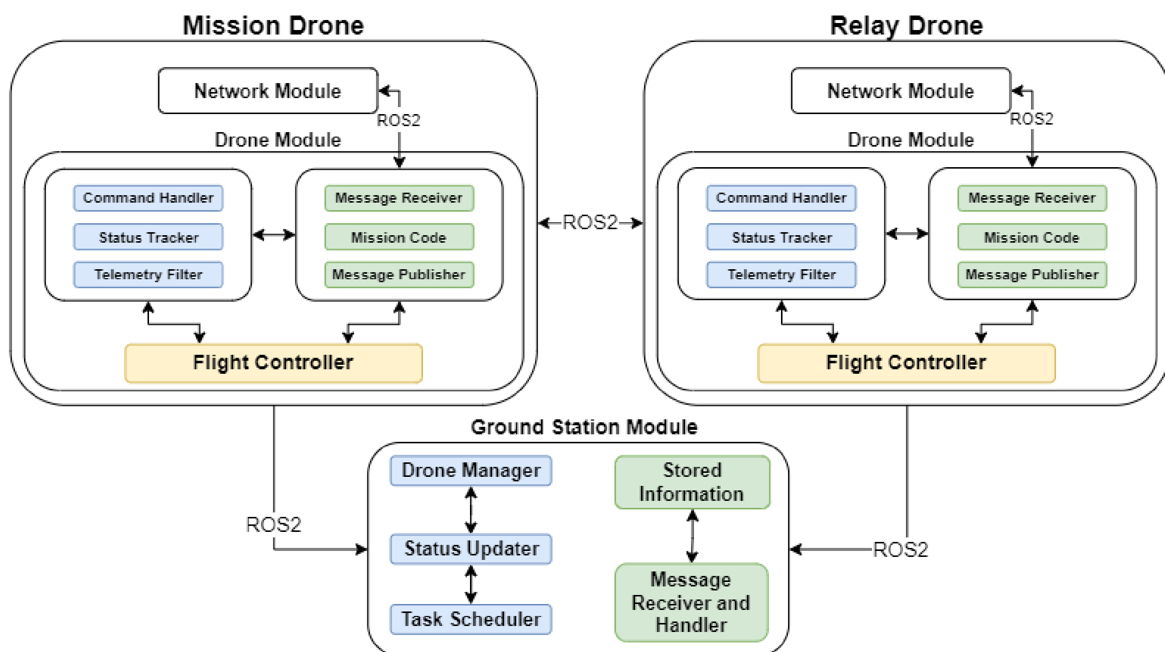

Figure 4.7: Modules connection for a mission with relay.

The modules on the drones are similar; the only difference is in the mission code. The first drone has a code corresponding to its mission (depending on the type of mission it is conducting), while the second drone has a code that allows it to act as a relay. In addition, a network module is active on both drones to keep track of the network. Drones communicate with one another via ROS2 and share data such as status, telemetry, and control messages as needed. ROS2 is also used by drones to communicate with the Ground Station and share vital data.

The Ground Station has the task of managing which drones are available and respond to relay requests.

4.2.2 Message exchange and format

This Section describes how the drone sends a request to the Ground Station when it needs one and how the Ground Station reacts with them, including the various sorts of messages sent and the parameters that make up those messages.

Requesting a relay

For this example, we will assume that a drone (drone A) is sent out with a mission. If at some point, this drone requires a relay, it will send a message to the ROS2 topic **/info** with the following content:

Example Message sent to the GS when a relay is needed

```
{
 "FROM" : "droneA",
 "TO" : "GroundStation",
 "TYPE" : "requestRelay",
 "IP" : "10.1.1.1",
 "MAC" : "ff:ff:ff:ff:ff:ff"
}
```

A simple JSON message with the sender (FROM) being the ID of the drone requesting this relay, the receiver (TO) being the GroundStation and the type "requestRelay". This message also contains the IP and MAC addresses of the drone, because a relay will require this data to be able to monitor the network with drone A. The GS node will receive this message and check if there are any available drones to respond to this call. If there are, it will respond back to drone A with the ID of the drone that was assigned to the relay task in the field "droneAssigned":

Example Response from the GS

```
{
 "FROM" : "GroundStation",
 "TO" : "droneA",
 "TYPE" : "relayAccepted",
 "droneAssigned" : "droneB"
}
```

At the same time, it sends a message to the relay drone, containing the ID, IP and MAC addresses of the drone it must pursue in the field "droneAssigned", indicating that it must take off and follow drone A:

Example Message sent to the relay drone assigning the mission

```
{
 "FROM" : "GroundStation",
 "TO" : "droneB",
 "TYPE" : "relayAssigned",
 "droneAssigned" : "droneA",
 "IP" : "10.1.1.1",
 "MAC" : "ff:ff:ff:ff:ff:ff"
}
```

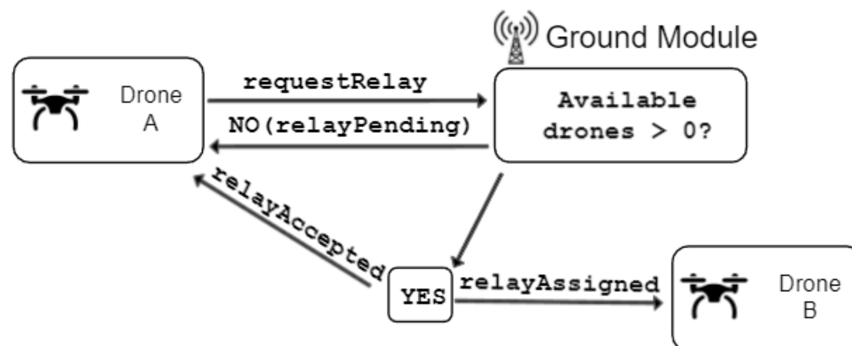

Figure 4.8: Flow for a Relay Request.

Figure 4.8 shows the flow of messages exchanged during a relay request.

Suppose there are no any available drones to complete this task. In that case, the Ground Station sends a "relayPending" message to the drone, indicating that it has received the request and will respond as soon as drones are available.

Canceling a relay

Drone A will continue the mission usually after drone B has been dispatched, and if at some point the relay is not required anymore, it will send a message to the GS to cancel the relay:

Example Message sent to the GS when a relay is not required anymore

```
{
 "FROM" : "droneA",
 "TO" : "GroundStation",
 "TYPE" : "cancelRelay"
}
```

The GS will update the known information, eliminating drone A from the list of drones with a relay drone. In addition, drone A sends a message to drone B, requesting that he be released:

Example Message sent to drone performing the relay to return home

---

```
{
 "FROM" : "droneA",
 "TO" : "droneB",
 "TYPE" : "freeDrone"
}
```

Following reception of this message, drone B will begin its journey back home, and at some point along the way, it will send a message to the GS indicating that it is once again eligible for relay.

Example Message sent to drone performing the relay to return home

---

```
{
 "FROM" : "droneB",
 "TO" : "GroundStation",
 "TYPE" : "freeDrone"
}
```

This message will be received by the GS node, which will update the list of available drones, allowing drone B to be used for another relay mission later, as shown in Figure 4.9.

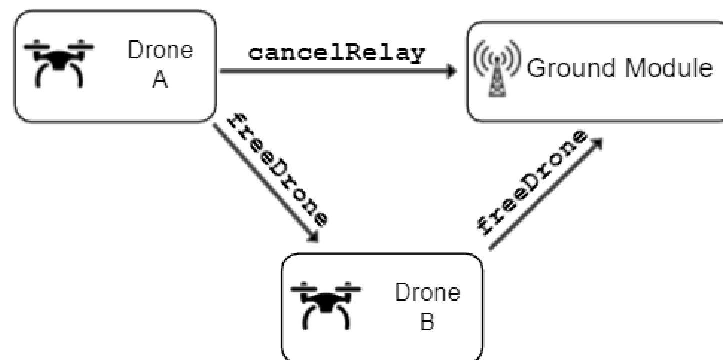

Figure 4.9: Flow for a Cancel Request.

Again, this could happen repeatedly, because there could be cases where drone B

gets too far away from the Ground Station and then sends a "requestRelay" message to the GS, causing drone C to begin pursuing drone B, and so on.

4.2.3 Drone performing a mission

To maintain constant contact with the GS, the drone performing the task must continuously monitor the network between itself and the GS, requesting a drone if it becomes unreliable. It is important to remember that the relay drones' primary function is to serve as a link between the drones performing missions and the GS.

Whenever a drone is performing a certain mission, it runs **goto_with_relay.py** and **Network.py** in parallel, as part of the Mission Code. As described in the previous part, **Network.py** will monitor the connection between the drone and the GS. **goto_with_relay.py** will examine the data recovered and determine whether a relay is required or not. The Network Quality parameter is used to do this.

The only value for Network Quality that is appropriate is "HIGH". The values "MEDIUM" and "LOW" are not. As a result, whenever the quality falls below "HIGH", a relay should be initiated. However, because measurements are not always 100 percent exact, we must account for the probability of inaccuracies. For example, we may have a "MEDIUM" measurement in the middle of a series of "HIGH" measurements. To circumvent this issue, a relay will only be activated if a specified number of consecutive measurements have poor network quality. This value is set by the **relayNetworkThreshold** parameter. The application keeps track of how many times a poor quality is detected and resets the counter to zero when a good one is found, triggering a relay whenever the counter reaches the specified value:

```
1 if networkQuality != "HIGH":
2     relayNetworkCounter += 1
3 else:
4     relayNetworkCounter = 0
5
6 if relayNetworkCounter >= relayNetworkThreshold:
7     "send message to the GS requesting a relay drone"
8     ...
```

Of course, this only happens if the drone is not already supplied with a relay. If it does, it will look at the inverse, a set of measurements with a good network, and if it finds such, the relay will be cancelled. This occurs when the drone is too close to the GS, resulting in a good connection.

```
1 if networkQuality == "HIGH":
2     relayNetworkCounter += 1
3 else:
```

```
    relayNetworkCounter = 0

if relayNetworkCounter >= relayNetworkThreshold:
    "send message canceling the relay drone"
    ...
```

4.2.4 Relay Drone

As seen above, the probes are used by the drone performing the mission. Now we can see how the relay drones use them. For relay drones, **relay_drone.py** is the script that contains the mission code. These drone will use the **Network.py** script to monitor their network with the GS after being assigned the relay and taking off. Because a drone with the relay function may have its own relay, it will check for values using the same exact parameter (**Network Quality**) as the mission's first drone.

The Network Probes, on the other hand, have an additional role for these types of drones. The network's quality will determine the distance this drone will maintain from the mission drone, meaning that these drones also monitor the connection with the mission drone. However, because the Network Quality parameter only has three possible values (High, Medium, and Low), it is difficult to determine whether the quality is excessively high or moderately high. We will need a parameter with a larger range of values to do this. The Signal is the better parameter for this. Signal's value can range from 0 to 90, and it is proportional to the distance between the drones, thus the value will be smaller if two drones are closer together and vice-versa, because a lower value means a better connection. This, of course, is what happens in open environments with vast space. However, in areas with uneven terrain and obstacles, this may not be so linear.

The goal is to keep the **Signal** at a level that is neither too high nor too low. Too high indicates a poor connection, implying that the drones are too far apart, which could cause issues if communications deteriorate. Low indicates a good connection; however, too low indicates that the drones are too close to each other, which is unnecessary if the drone can be moved a little further while maintaining a decent connection quality. To do this, the recommended Signal value should be somewhere between a High and a Medium connection (between 55 and 65). This threshold is set by the user in the **relaySignalToKeep** variable, which can be adjusted.

This variable, as well as the actual Signal value, are taken into account whenever a relay drone has to calculate a new coordinate of where to travel (which could happen several times per second):

```
if signalStrength > relaySignalToKeep:
    targetDistance -= distanceAdjustment
elif signalStrength < relaySignalToKeep:
    targetDistance += distanceAdjustment
```

The current value of the **Signal** is represented by **signalStrength** in the code above. If it is higher than the one we want, it means the drone is farther away than the optimal distance, and we should shorten it, and vice versa. The **targetDistance** variable specifies the distance between the relay drone and the drone it is following. The drone will get closer or farther away from the drone it is following as this value is decreased or increased. This variable will be slightly adjusted each time a new coordinate is calculated. The variable **distanceAdjustment** specifies the amount of adjustment. It comes as near to the optimal distance as feasible this way. This variable does not have a constant value and instead fluctuates throughout the mission. The user specifies the first value for this distance in the **targetDistanceDefault** variable at the start of the mission.

Behaviour during relay mission execution

Knowing how the distance changes during a mission makes it simpler to comprehend how a relay drone determines the coordinates for where to go. The relay drone will compute a new coordinate in a loop after taking off, which means that once it calculates one and sets it as destiny, it will instantly calculate the next one, terminating only when the mission is done or a relay is no longer required.

The following are the steps taken by a relay drone to calculate a coordinate:

1. Request a network measurement between itself and the drone it is following;
2. From the preceding request, retrieve the **Signal** parameter;
3. Using the value of the **Signal**, update the **targetDistance** variable using the method previously given;
4. Get the GPS coordinates from the drone it is following;
5. Calculate a new adjusted distance with the **targetDistance**. Only the former will be utilized to perform the final calculations;
6. To make the calculations, use all of the previously specified parameters;
7. Set the resulting coordinate as the destination.

The stages 5 and 6 will be covered in further depth on the following pages.

Stage 5 - Calculating the best distance

Even though **targetDistance** is affected by network measurements in order to obtain the best distance, this parameter will still undergo a minor correction before the calculations are performed. Let us take a look at Figure 4.10 to get a better understanding.

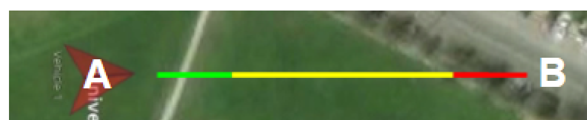

Figure 4.10: Drone's different speed zones.

Assume a drone is at position A and wishes to travel to position B. When we instruct a drone to travel to a specific location, we select the speed at which we want it to travel. However, if the drone is halted or moving at a slower speed than that, it will take a moment to reach it. The same goes for stopping when it gets close to its destination. It begins to slow down before reaching its destination. As a result, this journey is divided into three stages: accelerating (green), traveling (yellow), and slowing down (red).

It is necessary to clarify this because there is a specific moment during a mission that, if it occurs, may be affected by these factors. If the mission drone keeps moving away from the GS, the relay drones' computed coordinates will eventually be near the drone's present position, putting them in the red zone. As a result, the relay drone's speed will equalize with that of the drone it is following, and it will never reach its desired speed. When this point is reached, the relay drone will be unable to go as close to the target as it had intended. For example, even if the drone is aiming for a distance of 65 meters, it may only be able to get closer to 75-80 meters. The only method for the drone to reach this distance of 65 meters is to provide it with coordinates that are less than that distance. This is why the distance requires one more minor adjustment:

```
1 distanceBetweenDrones = Math.calcDistance((myLatitude,
2 myLongitude),(targetLat, targetLon))
3 differenceBetweenDistances = distanceBetweenDrones - targetDistance
4 correctedDistance = targetDistance - differenceBetweenDistances
5
6 # Avoid negative values for correctedDistance
7 if correctedDistance < 0:
8     correctedDistance = targetDistance
```

As we can see, the first step is to determine the distance between the relay drone and the mission's first drone. This is accomplished by utilizing a function accessible within the **Math.py** module, which uses the **geopy** module from Python to compute the distance between two coordinates:

```
1 import geopy.distance
2
3 # Calculates the distance between 2 coordinates and returns the result in meters
4 def calcDistance(coord1, coord2):
5     return geopy.distance.distance(coord1, coord2).km * 1000
```

Following that, we subtract the **targetDistance** parameter from the distance between the two drones, obtaining **differentBetweenDistances**. Finally, we use the difference between **targetDistance** and **differentBetweenDistances** as the final distance parameter, which we call **correctedDistance**. Let us look at an example.

If the relay drone is 80 meters away from the first drone and wishes to achieve a target distance of 65:

$$distanceBetweenDrones = 80$$
$$differenceBetweenDistances = 80 - 65 = 15$$
$$correctedDistance = 65 - 15 = 50$$

In this situation, instead of 65 meters, a distance of 50 meters will be provided. This allows the relay drone to come closer to the first drone, avoiding red zones, and achieve the desired distance of 65 meters. This also works if the relay drone is closer than expected:

$$distanceBetweenDrones = 50$$
$$differenceBetweenDistances = 50 - 65 = -15$$
$$correctedDistance = 65 - (-15) = 80$$

In such cases, a coordinate that is greater than the target distance will be provided.

The only thing left to do is be cautious if the **correctedDistance** returns a negative number. This will occur if the **distanceBetweenDrones** is greater than double the **targetDistance**. In these circumstances, we simply use **targetDistance** as previously and disregard the **correctedDistance**.

Stage 6 - Calculating the next coordinate

Now that we have gathered all the necessary information, we can start the calculations using the scenario depicted in Figure 4.11.

Figure 4.11: Example scenario for a mission with relay support.

Drone A is performing a mapping mission. The Ground Station is marked with **GS**.

Drone B is a relay drone assigned to perform the relay of **A**. So, **B** has to stay in line, between **A** and **GS** within a determined distance of **A**. The red circle around **A** has the radius equal to the distance we want nodes to keep between them (**targetDistance**). There is also another variable, **tolerance**, which is the error allowed to be committed when calculating a new coordinate. If **targetDistance** is 50 meters and the **tolerance** is 2 meters, it means the new coordinate **drone B** is destined to travel to could be anywhere between 48 and 52 meters from **drone A**.

Drone A's coordinates are retrieved from the **/telem** topic.

The first step is to calculate the angle between drone **A** and the **GS**.

This is done by using the arctan function present in the python's **math** module:

```
1 angleInRadians =
2 math.atan2(GS_Latitude - droneA_Latitude,GS_Longitude - droneA_Longitude)
```

This returns the angle we need. In this example, and according to Figure 4.11, it should be around 45º or 0,785 radians.

We can now determine the correspondent coordinate in the red circumference using the angle and these well-known geometric equations:

$x = radius * cos(angle)$
$y = radius * sin(angle)$

But, there is a problem - radius is defined in **meters** and, since we are working with GPS coordinates, the X and Y axis are defined in **degrees**. Unfortunately, it is not possible to directly convert meters to degrees or vice-versa, because Earth is not a perfect sphere, but an **ellipsoid**. In order to do this, we have to calculate the radius, in degrees, that corresponds to a certain distance, in **meters**.

We begin by utilizing a base radius, in this case, the author of this Dissertation used 0.01, which corresponds to around 1 kilometer (depending on the Earth's location) and the formulas previously explained, in which longitude equates to X and latitude to Y:

$lonCorrection = radius * cos(angleInRadians)$
$latCorrection = radius * sin(angleInRadians)$

As a consequence, the existing coordinate in the red circle is obtained. However, the obtained Longitude and Latitude are relative to the center of the circle (the drone itself), therefore we must add them to the drone's coordinates to make them absolute.

$targetLongitude = droneA_Longitude + lonCorrection$
$targetLatitude = droneA_Latitude + latCorrection$

After having a new coordinate (targetLatitude, targetLongitude), we calculate the distance, in **meters**, between this coordinate and the **drone A**, using the previously described **calcDistance** function:

```
min_distance = targetDistance - tolerance
max_distance = targetDistance + tolerance

coord1 = (GS_Latitude, GS_Longitude)
coord2 = (targetLat, targetLon)

distance = calcDistance(coord1, coord2)
```

Finally, we check to see if this coordinate is within the tolerance specified:

```
if distance >= min_distance and distance <= max_distance:
    returns the current coordinate
else:
    radius = radius * 0.975
```

If this condition is met, it suggests we were successful in locating a suitable coordinate. If it is false, the radius is reduced slightly and everything is recalculated with the new radius. This loop can be repeated until a coordinate is obtained, but it is the best way to get an accurate distance. The tolerance should not be less than 2 meters to risk not being able to determine the correct coordinate.

Whenever a new coordinate is found, a new **goto** command is issued (message sent to **/cmd** topic), the **drone B** retrieves **drone A**'s new position (from **/telem**) and does the whole procedure again. At the same time this is happening, the same drone is monitoring the **/info** for any important information.

Returning home

When a relay drone ends its mission and returns home, it uses probes once again.

Because relay drones can undertake multiple missions in a sequence, after completing one, they must notify the GS using the "**freeDrone**" message, as previously mentioned. Because they may be too far away from the GS when a mission concludes, they may not be within communication range, thus we must ensure that they are before delivering this message. We will use a simple mechanism similar to the one we used to cancel a relay for this. While returning home, the drone repeatedly measures the network between itself and the GS and uses the **Network Quality** parameter to determine whether the quality is adequate to send the message:

```
if networkQuality == "HIGH":
    relayNetworkCounter += 1
```

```
else:
    relayNetworkCounter = 0

if relayNetworkCounter >= relayNetworkThreshold:
    "send freeDrone message to the GS"
```

We use the **relayNetworkThreshold** parameter once more to determine how many times the network must be measured with **HIGH** quality before transmitting the message.

After starting its course back home, the drone begins monitoring messages from the Ground Station to check if it is assigned a new mission. The drone may receive this even before landing, meaning it may be on its way back to base when it gets the request, turning around to execute the new relay mission.

4.2.5 Ground Station

In the relay scenario, the Ground Station module is in charge of managing the available drones to conduct relay tasks and respond to requests received from drones. The code for the **Drone Manager** and the **Message Receiver and Handler** are contained **in relay_gs.py**, which allows the GS to operate the fleet. Before running anything in the Drone Module, we must execute this script first.

The stored information is divided into two categories: available drones to perform as relay and mission drones that already have a relay drone. Any requests for which an instant answer is not possible are saved and dealt with as soon as possible. The table below displays a list of possible requests and responses that can be sent to the Ground Station:

Table 4.5: Requests handled by the Ground Station for the relay scenario.

Type	**Meaning**	**Responses**
requestRelay	It is sent by a drone when it requires a relay.	Looks for drones that are available. If any are found, it responds with a "relayAccepted" message and assigns the relay drone the task, updating the stored information; otherwise, it saves the request and responds with "relayPending".
cancelRelay	It is sent by a drone when it no longer requires a relay.	Removes any pending requests from this drone and, if it is present in the list of drones with relay, removes it.
freeDrone	It is sent by a relay drone when it is ready to receive a relay task.	Checks for any pending relay requests. If there are any, responds to the oldest one, updating the stored data; otherwise, simply adds the drone to the list of available drones.

4.2.6 Routing

The relay mission's main goal is to employ a drone to send information from a drone performing a specific mission to the GS and vice versa. As a result, having a relay in the proper location is pointless if the drone is not routing any data.

We utilized batman-adv[7] to accomplish this. This library allows us to set up a mesh network connecting the many nodes that make up our system, creating a virtual interface over the wireless USB adapter's physical layer. When a node is correctly configured with these settings and the interface is turned on, the library broadcasts the IP and MAC address to potential neighbors. When it connects to a neighbor, it receives its information and creates a route to that node, regularly sending packets to verify network metrics like latency. When new nodes join the network, they all use a discover protocol to calculate metrics between themselves and determine the best route from one node to the next and whether or not hops are required. This means that when a packet is sent from point A to point B, it may need to be passed by a node in the middle. To the user, everything seems to be a single Local Area Network (LAN)[8].

We did not consider this as a separate module or a component of one of the already existing modules, since we only needed to make a few configurations to set up this network.

Figure 4.12 depicts three examples of how this network behaves.

[7]https://www.open-mesh.org/projects/batman-adv/wiki
[8]https://www.cisco.com/c/en/us/products/switches/what-is-a-lan-local-area-network.html

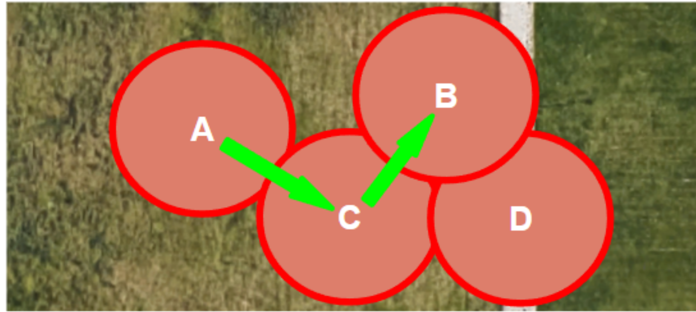

Figure 4.12: Example scenarios using batman-adv to route information.

The circumference of each circle represents the maximum communication range of the node in the system. A batman-adv instance is running on every node. This network will always take the quickest and shortest route to its destination. Consider the situation where we need to communicate data from node A to node B. In the first scenario, these two nodes are within range of each other; hence, even though node C is in the middle, it is unnecessary to route information through this node. The two nodes are also too far apart in the second situation. However, because node C is within range of both, batman-adv will internally identify a path through C. In the last example, A and B are also out of range; thus, C is employed to relay information. It is also feasible to utilize D (A -> C -> D -> B), but the route with the fewest hops would be selected.

4.2.7 Relay summary

The general flow for a relay mission for the first drone, a drone executing any assignment that might require a relay at some point, is shown in Figure 4.13. This first drone sets out on its mission as usual, heading to a certain set of coordinates. It examines the connection with the GS to see if it is good or not. If it is, it proceeds normally; otherwise, it will contact the GS to request a relay drone and proceeding the mission normally, independently from the answer. After the request is granted, it begins monitoring the network to see if, at some point, the relay is no longer necessary.

The flow for the relay drones is a little more complex. In the base station, a relay drone will begin listening for any requests from the GS. After receiving one, it takes off and starts receiving the coordinates of the drone it is intended to follow, using those together with the network signal strength to that drone to calculate the next target location and the appropriate distance to keep from it. At the same time, it keeps an eye on the connection with the Ground Station in case a relay is necessary. These actions are depicted in Figure 4.14.

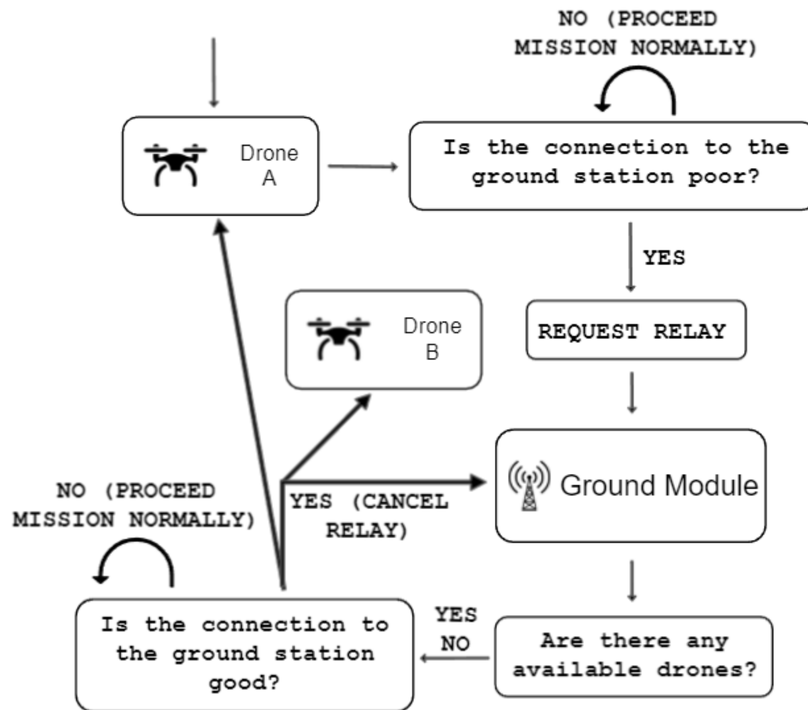

Figure 4.13: Flow for the drone that initiates a mission.

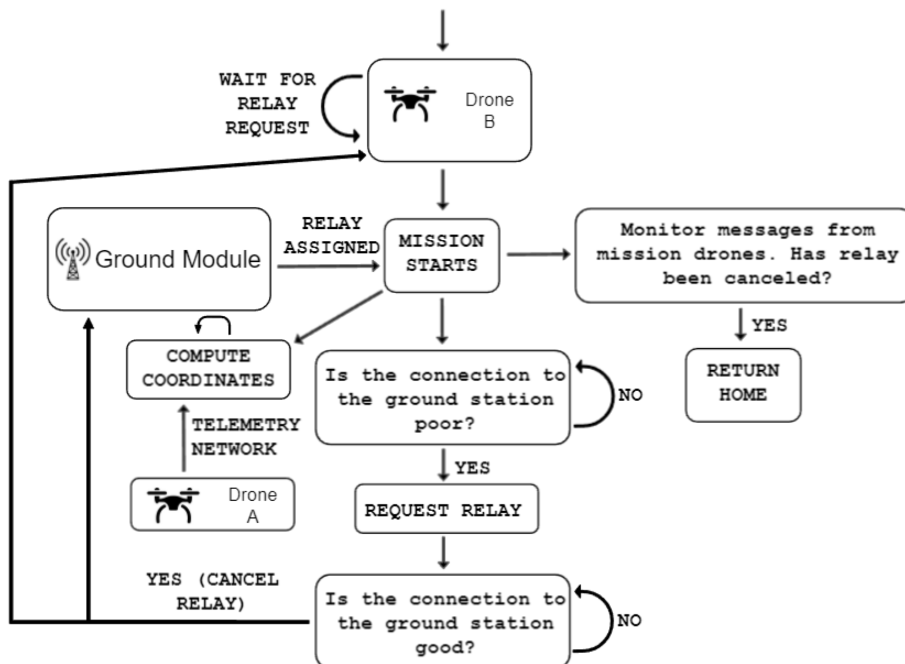

Figure 4.14: Flow for the drone that performs the relay.

4.3 Rescue

Another feature of this system is requesting a drone from the GS using an Android smartphone to come to an individual's encounter, which might be utilized in a rescue mission. The concept is to employ a drone in an emergency circumstance, similar to other scenarios already in existence. An application makes a request and sends GPS coordinates to a drone that has been assigned to the task. The drone is equipped with a camera to transmit visual feedback to the Ground Station to see the individual who performed the request.

4.3.1 Modules integration

All of the designed modules - Drone, Ground Station, Network, and Android - were used to make the rescue mission possible. The method they use to communicate is depicted in Figure 4.15.

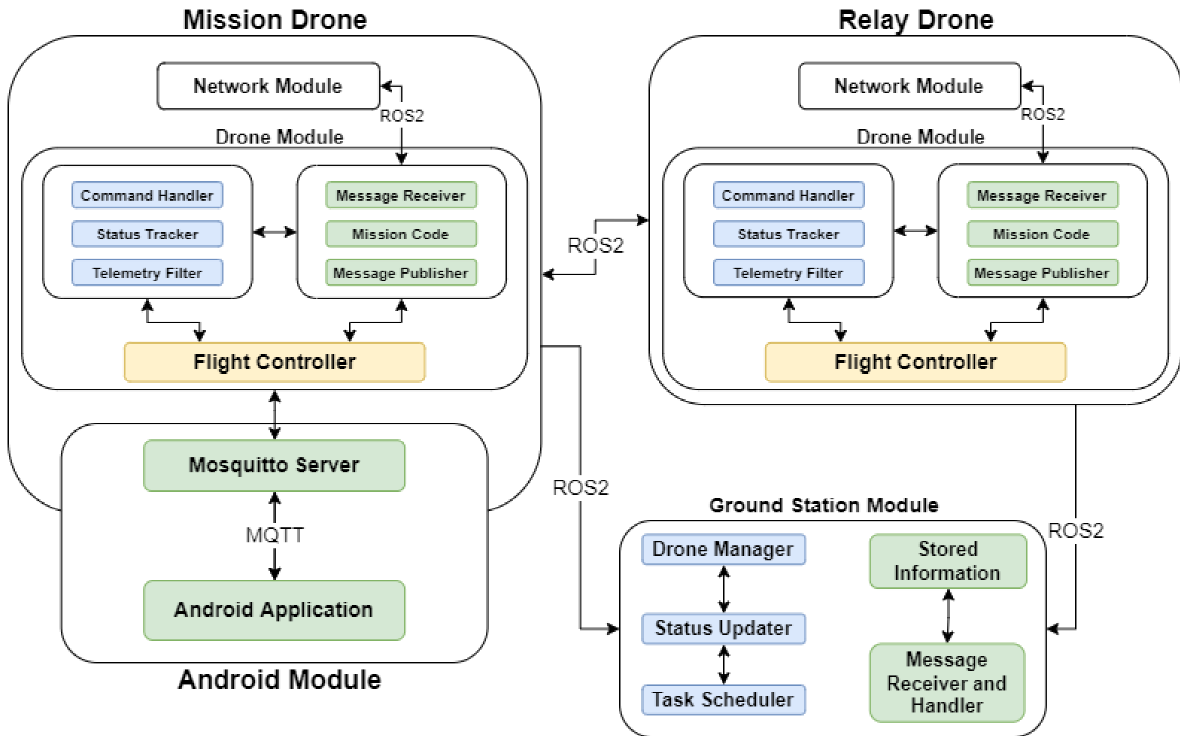

Figure 4.15: Modules connection for a mission with relay and Android integration.

The majority of the modules are connected in the same way as in the relay mission. There is, however, a notable difference. A Mosquitto server is installed on the drone doing the mission, allowing it to interact with the Android app. The android app is not part of the mission drone's module; instead, it is run by a user outside of the mission drone's module. We did not require any complicated computations to test this new functionality. When the drone receives a request, it will simply travel to the coordinates provided by the Android app. The user can cancel the request at any time, and the drone will return

home.

4.3.2 Android application

A user must press the panic button shown in Figure 4.3 to perform a rescue request. Upon pressing it, the app checks to see if the system's GPS is turned on and then checks for its present location.

Because of the way the Android GPS API is designed, anytime an app requests the most recent location from the GPS Service, the Android system obtains coordinates from the OS rather than the GPS itself. Checking the GPS for the present position at a specified time is not possible. This causes a problem because the most recent location saved could be hours/days old and incorrect, which is a problem that can occur from time to time when using older Android versions.

However, there are two options for resolving this:

- Open an app that is more closely linked to the system. The best illustration of this is Google Maps. The OS will save the updated position if you open this program and request the current location;
- Updating the current location. Even if the program is unable to get the first position after clicking the panic button, it activates a listener that is triggered anytime the global position is modified, allowing it to store the new one and communicate it as soon as possible.

Whenever the panic mode is engaged, the application retrieves (if one exists) and communicates the stored coordinate from the system, doing so once every second. If this fails, it will keep retrying once every second until it is able to do so or until the user cancels the request.

Apart from the required global position, the application requires a connection to the same network as the server (same as ROS2) in order to receive and transmit messages to and from the server.

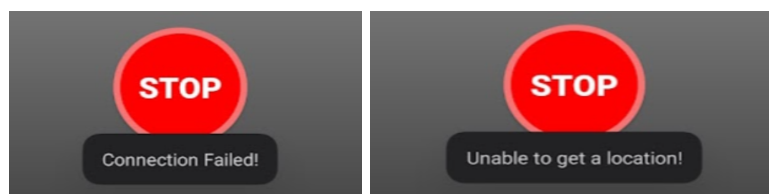

Figure 4.16: The user will see one of these messages if one or both of the prerequisites (connection and GPS, respectively) are not met.

Assuming that everything is in order (connection and GPS), the application will use the **MQTT** node to send the following message to the topic "sensors":

Example Message from Android Application to the server requesting a rescue


```
{
  "sensorId": "phone01",
  "type": "locatorRequest"
}
```


The message contains the request type as well as the phone's ID. After then, the server sends an accept or reject message (depending on whether or not drones are available). The ID of the drone assigned to that task is also included in the affirmative message.

Example Positive response from the server


```
{
  "FROM" : "Server",
  "TO" : "phone01",
  "type" : "chaseAccepted",
  "droneAssigned" : "drone01"
}
```


When there are available drones, the aforementioned message is delivered; otherwise, the following message is sent:

Example Negative response from the server


```
{
  "FROM" : "Server",
  "TO" : "phone01",
  "type" : "chaseDenied"
}
```


The user's rescue request is automatically terminated if the latter is sent (the user can retry at any time).

Feedback on requests will always be provided to the user.

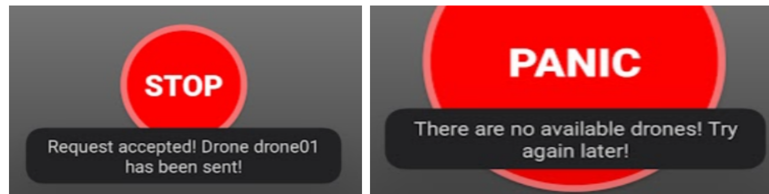

Figure 4.17: The user will receive a response to the request depending on the availability of drones.

Unlike drones, the GPS coordinates on an Android system do not change very often (moving 1 or 2 meters may not be enough to update the position, depending on the GPS model and Android version). Therefore, because sending messages to the server with repeated information at smaller intervals is unnecessary, the application delivers each message at a second interval.

After delivering the first message to request a drone and obtaining a positive reply, the app will communicate each user's current position to the drone in this interval.

Example Message sent from the Android device containing its location

---


```
{
  "sensorId": "phone01",
  "type": "locator",
  "timestamp": 1631554876034,
  "value": {
    "lat": 40.634027873904756,
    "lon": -8.660564121419112,
    "alt": 8.960174560546875
  }
}
```

The phone's ID is always provided together with the coordinates, which the assigned drone needs to make sure it is getting data from the right device.

By pressing the "stop" button, the user can cancel this request at any time. When this occurs, the app cancels the request and sends the following message to the drone:

Example Message sent from the Android canceling the request

```
{
 "sensorId": "phone01",
 "type": "locatorCancel"
}
```

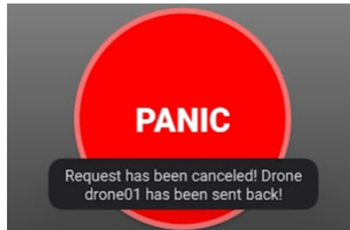

Figure 4.18: Request cancelation.

One additional scenario is possible. The user will receive the following notice and the request will be canceled if the mission has to be canceled by the drone executing the task rather than the user, due to a lack of relay drones.

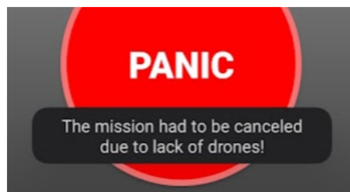

Figure 4.19: The user will receive this message if the drone cancels the mission.

4.3.3 Drone

We need to examine how the drones carry out this task now that we have seen what the application is.

Besides running the server, the drone will run the **chase_with_relay.py** script to complete this task.

We simply run the script with the command:

```
python3 chase_with_relay.py -d drone01
```

The -d parameter represents the drone's ID. When the script starts, the drone will be in its home position and will begin monitoring the **/sensors** topic in **MQTT**, while waiting for a request from a user.

Upon receiving a request, it responds positively or negatively, depending on its availability. If the response is positive, it takes off. After taking off, the drone will listen for coordinates sent by the Android system. And then, the drone will simply dislocate to each coordinate. Another important detail to notice is that this mission type supports relay. Like the relay missions mentioned in the previous Section, the first drone dispatched to assist the user can request a relay drone at any time using the exact same methods.

In addition, the drone executing this flight will monitor the **/sensors** topic for any cancel from the Android device (*locatorCancel*, in case the mission is canceled by the user).

If the drone receives a *locatorCancel* message at any time, it will alert its relay drone (if one exists) and return home. This method is performed for each drone involved in the

mission at that point.

In case there are several involved relays, each one must inform its relay. Assuming drones A (first to be dispatched), B (relay of drone A), C (relay of drone B) and so on, the messages will be transmitted in this order:

A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D

And each one will return home.

4.3.4 Rescue summary

Figure 4.20 depicts the flow of a drone conducting a rescue operation. First, the drone will listen to any requests made by the user through the Android app. When the drone receives one, it will acknowledge the user's request, respond to it and move to the coordinates provided by that same user. In addition, it monitors the network with the Ground Station upon starting its mission because a relay may be required at some point throughout the operation. However, because this part is already fully described in Figure 4.13, it was simplified in Figure 4.20 and is merely displayed as "Relay Procedure".

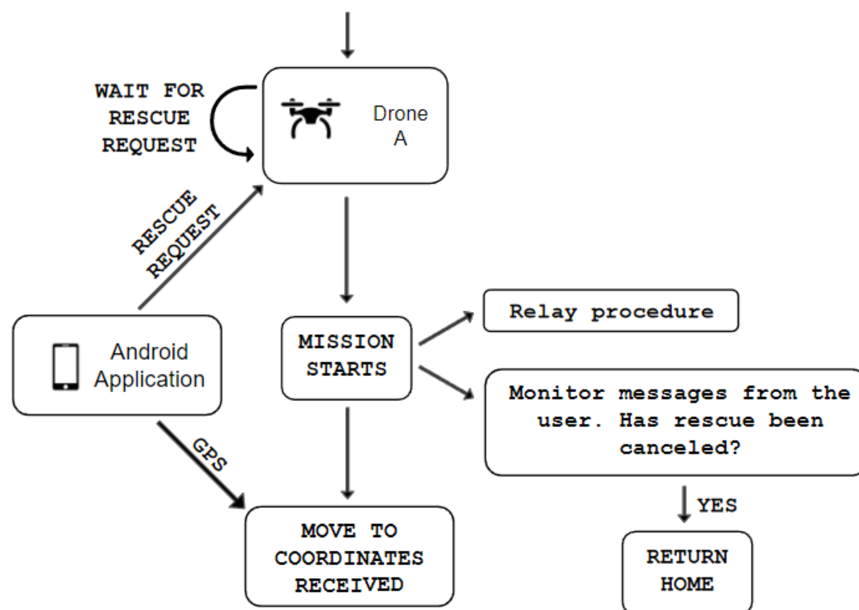

Figure 4.20: Flow for the drone performing a rescue mission.

4.4 Summary

In this chapter, first, we gave a general overview of the several modules created and a description of their purpose. Then, we explained how we implemented the communication between them. Next, we specified how each of the modules was implemented and what was needed to make them functional. Finally, we went over the various scenarios in

which these modules were deployed. We also outlined the role of each one and the role of each node involved in these scenarios. We will look at the individual tests conducted to validate the architecture and implementation outlined in this chapter in the next chapter.

Chapter 5

Experiments and Results

We will discuss the experiments and tests conducted to validate the work presented in this dissertation. The experiments were divided into two categories: simulated and real-world. Simulated experiments used the simulator tools previously developed, as described in Section 3.1.3, whereas real-world testing involved taking real UAVs outside and testing in the approaches. Preliminary validation steps took place in the simulator before going on to a real scenario to ensure the proper operation of the system.

5.1 Simulated experiments

The initial step in creating this Dissertation was to use the features that had already been created. As described in Section 4.1.5, a few functions were created to send the most basic commands to a drone using the RCLPY[1] library. Each function would convert the order to a ROS2 message and publish it, which would be retrieved by the Command Handler in the Drone Module, and this time converted to a low-level command and sent directly to the Flight Controller.

The first approach was to validate the different functions by writing small excerpts of code, such as the one provided in Section 4.1.3. QGroundControl[18] was used to track the missions' progress and check the drones' telemetry to make sure everything was working properly. Following the validation of this part, the next step was to construct a drone-drone communication point using ROS2 nodes. A few simple examples were created to test this communication, using the features in the RCLPY library, publishing the messages from one drone and checking for their receipt in the other. Following this validation, the relay scenario was tested utilizing the message exchange outlined in Section 4.2.2.

[1] https://github.com/ros2/rclpy

5.1.1 Relay mission

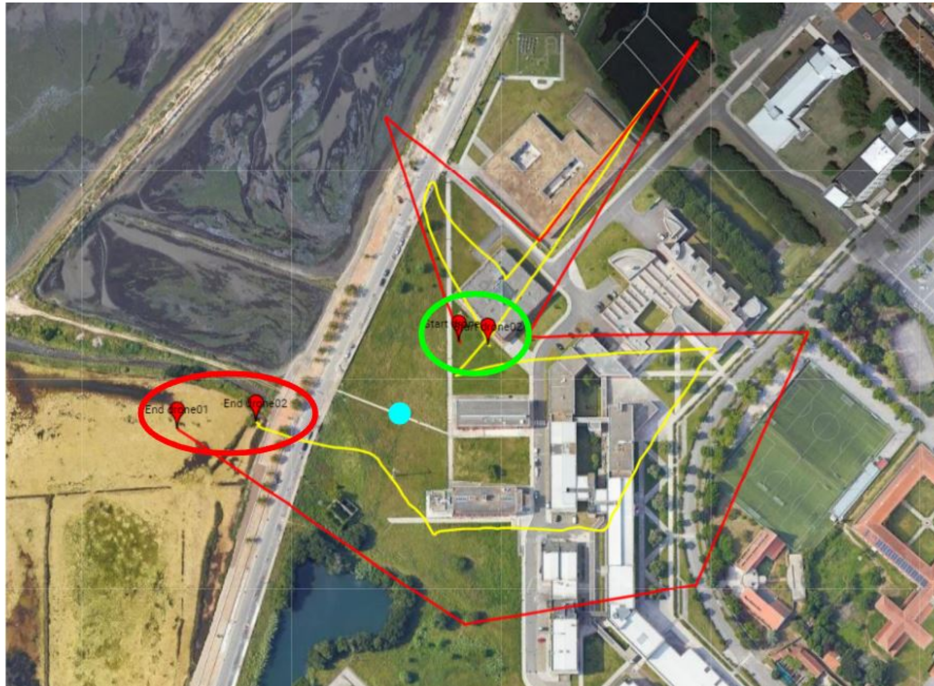

Figure 5.1: Example of a mission with relay, displaying the paths of the drones.

Figure 5.1 depicts the path of two drones (one following a set of coordinates, performing a mission and its relay), marked with colored lines, during a mission. The green circle represents the area where the drones took off, while the red circle represents where the mission came to an end. The little cyan circle represents the Ground Station's location. The first drone follows a set of coordinates around the Ground Station, calling a relay shortly after takeoff, and the second drone continues to follow it until the mission is completed. We can distinguish two comparable forms, one smaller than the other. The larger one, in red, depicts the course of the mission's first drone, while the smaller one, in yellow, depicts the path of a relay drone. Because they are centered around a point, which is the coordinate of the Ground Station, and the relay drone is always in line between the first drone and the GS, the paths for both drones have similar shapes.

This map was obtained by utilizing the rosbags feature to acquire the drone's telemetry, as described in Section 2.4, and then by using the **gmplot**[2] library to convert the data to a plot using Google Maps' features.

[2]`https://pypi.org/project/gmplot/`

Analyzing performance during a relay mission

Figure 5.2: Graph with the distance between the mission drone and its relay.

We have information regarding the distance between the two drones in Figure 5.2. The X-axis shows the number of relay drone coordinates issued throughout the mission, while the Y-axis shows the distance in meters between the relay drone and the drone it is following. The blue line depicts the optimal distance between the drones to optimize the mission, while the orange line displays the actual distance between them.

As we can see, the relay drone is further away in the beginning because it takes a while to take off while the other drone is getting away from the Ground Station. After approaching the first drone, it maintains proximity to the ideal distance. Because the drone's software performs more tasks than just calculating distance, there may be a slight delay, even in the simulator, which is acceptable. This makes it difficult for the drone to maintain the optimal distance all of the time. Thus we must always include in a 6 to 10-meter inaccuracy. When the drone moves away from the Ground Station, it will always be a little further away from the desired distance, such as between 80-140 and 250-310 on the X-axis. However, the opposite behavior happens when it gets close to the Ground Station, between 170-230 on the X-axis.

Overall, the algorithm appears to work effectively, and the relay drone is able to position itself in the correct location.

Performance issues

We discussed the importance of performance in relay missions before. A relay drone has no idea which path the first drone will take and must rely entirely on the data of others to determine the optimum location. If this is a lengthy process for some reason, it may cause additional delays and, in the worst-case situation, the relay may be too far away from the first drone, causing connectivity issues.

The implementation of a slight change to the distance parameter during calculations, as stated in 4.2.4, was one of the ways employed to reduce the delay. This did not make a significant difference, although it was feasible to verify that it assisted the drone in better positioning itself in some occasions.

The change of the QoS settings was the factor that made the most significant difference. QoS is a set of policies that allow users to fine-tune the communication between nodes[3]. Both publishers and subscribers may be affected by these settings. Only the parameters for subscribers were updated for this Dissertation. When a subscriber is set up, it creates a First In, First Out (FIFO)-style[4] structure of a specific size in which the messages received in that issue are automatically stored. The data can then be fetched at any time by the user from that FIFO. Because it is a FIFO, it will always retrieve the oldest message when retrieving a message, which is fine most of the time, but in the case of telemetry, it may lead the drone to use old telemetry even if it has the most recent. The drones' module publishes telemetry every 0.2 seconds, and the FIFO size is set to 10 by default, with the messages having a lifetime duration, which means they are only removed from the FIFO when the user retrieves them. We do not mean lifetime duration in the literal sense because the messages will not last forever, but a lifetime in the context of the mission because they will not expire during the mission. Thus, if two drones are online (one conducting a mission and the other serving as a relay), the FIFO will store five messages from each, with the oldest message being roughly one second old. As a result, the drone will always be one second behind current information, resulting in a second delay. To avoid this, the duration of the messages was modified from lifetime to 0.2 seconds, which is the same as the time interval at which the drones publish telemetry. This ensures that the telemetry messages are always the most recent.

[3] https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html
[4] https://www.investopedia.com/terms/f/fifo.asp

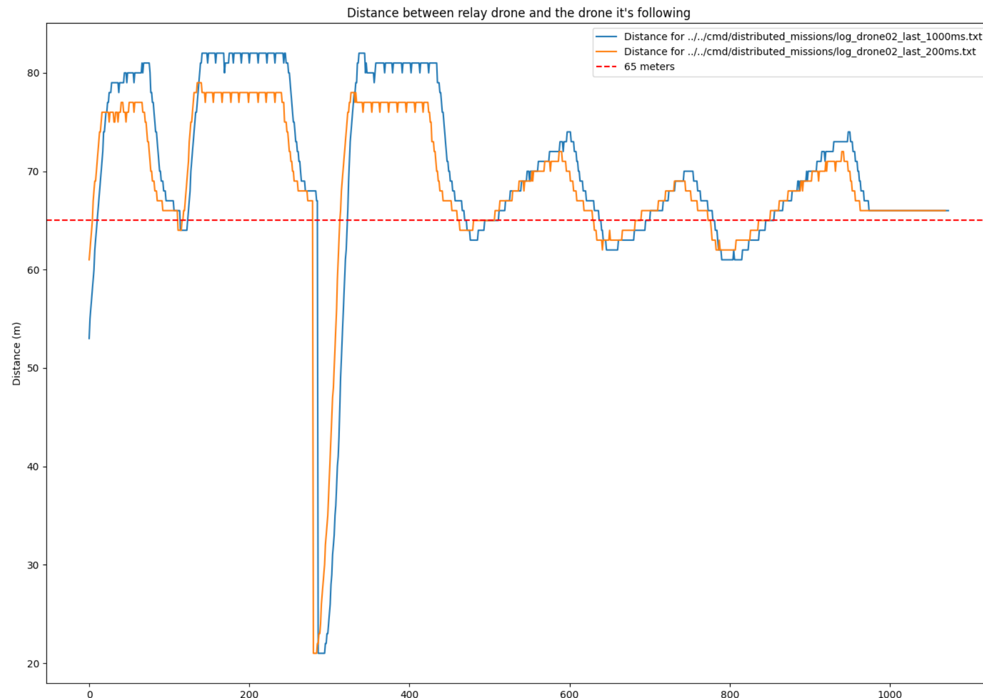

Figure 5.3: Graph displaying the improvement using different QoS policies.

Figure 5.3 depicts the outcome of a test to examine how the telemetry delay impacts the relay mission's performance. We kept a constant distance of 65 meters between the drones for this test, marked by the red dotted line. Thus, there was no network integration. The missions followed the same route as the one illustrated in the Figure 5.2 but with two different sets of QoS settings.

The blue line depicts the distance when using default QoS policies (resulting in a one-second delay), whereas the orange line displays the distance when using customized policies (message lifetime adjusted to 0.2 seconds). We can easily compare the two lines and observe that the drone can keep itself significantly closer to the front drone with this change, making the strategy the best.

Network probes

Because the **Network.py** script does not work in a simulated environment, testing the integration of network probes is a little different from a real scenario. This happens since measuring a network between two different nodes is only possible when they are running on different devices. Unfortunately, everything in a simulated environment runs on the same device, making it impossible. However, it is possible to emulate this behavior by creating a script that periodically sends a message to the topic containing network information, with the same format as the message in **Network.py**.

The probes integration was already included in the tests provided in 5.1.1, ensuring its usability.

5.1.2 Android integration

A Mosquitto server, as described in 4.1.4, was used to evaluate Android device interoperability. The server ran in a simulated environment on localhost on the same device as the other modules. The application's usefulness was evaluated by requesting and canceling a drone, then analyzing the drones' behavior with QGroundControl and using a relay in this type of mission with network integration. The laptop running the Mosquitto server also had an Access Point that the smartphone connected to allow communication with this device. However, throughout the simulation, everything was connected to the same network, allowing the drones to send and receive data directly from the Android device. This would not be possible in a real-world scenario because the drones only have an ad-hoc connection to establish point-to-point contact with other nodes (other drones and the Ground Station). In order to connect to the Android system, a drone requires a wireless connection. One possible solution is for the drone to have two connections: an ad-hoc connection that is already in use to connect to other drones and the GS, and a wireless connection that serves as an access point for the Android system. As a result, the Access Point would be on a drone rather than in the GS. At the time of writing this Dissertation, due to the limitations mentioned above, no real hardware tests were conducted to test integration with Android devices. Instead, we recorded a video[5] demonstrating the validity of the Android Module using the simulator and the application.

5.2 Experiments in a real environment

We tested the software on real hardware after the validation in the simulated environment has been completed. This was accomplished with the usage of real drones. A USB connection was used to connect the Flight Controller to the onboard computer. Each drone featured a wireless USB adapter with a specific connection to an AD-HOC network to link to other drones and the Ground Station. Even though the drones were using autopilot software, someone with a RC was always ready to take manual control of a drone if necessary.

5.2.1 Hardware

The drones' complete specifications are shown in table 5.1.

Table 5.1: Drones specifications.

Component	**Model**
Flight controller	Pixhawk 4 with NEO-M8N GPS
Frame	DJI F550
Radio Control	FLYSKY FS-i6 and FLYSKY FS-iA6B
Onboard Computer	NVIDIA® Jetson Nano™
Operating System	Linux4Tegra
Wireless USB Adapter	TP-Link TL-WN722N

[5] `https://youtu.be/9HcQscgYTRk`

Figure 5.4: One of the drones used in the experiments.

A laptop was used as Ground Station. To connect to the fleet of drones, this device, like the drones, used a wireless USB adapter. The complete specifications are listed in table 5.2.

Table 5.2: Ground Station specifications.

Component	**Model**
CPU	Intel Core i5-8300H 2.3-4.0GHz
Memory	16GB
Operating System	Ubuntu 18.04 LTS
Wireless USB Adapter	TP-Link TL-WN722N

Only the onboard computer was used in the initial experiments with real hardware. We needed to make sure that communication was flowing through every node in the platform before sending the drones out into the field, so we tested it in the lab. Following that, some tests were performed solely to test the placement of the relay drone in order to check if it could respond swiftly and avoid the delay discussed in 5.1.1. This was accomplished by simply flying a drone in manual mode, assigning a second drone to it and hard coding the relay drone to follow the first drone without requiring the first drone to request a relay from the Ground Station. Following the placement validation, the communication drone-drone and drone-GS was tested on the field, and in the end, we attached the network probes.

Having validated every feature individually, we advanced to a full scale test.

5.2.2 Testing relay with network integration

The main goal of this test was to ensure that all components worked together in a real-world scenario using real hardware. However, for this test, we did not integrate the Android module.

The hardware used was a laptop that served as the Ground Station and had the specifications listed in table 5.2, and two drones using the components listed in table 5.1. The

modules’ interfaces and algorithms used were already described (Section 4.2).

Goals

The goal of this test was to validate the following functionalities:

- **Drone-Drone communication** - Drones can communicate with one another by sending and receiving messages.
- **Drone-GS communication** - Drones can send requests to the GS, and the GS can reply.
- **GS management** - The GS is capable of adequately handling requests and assigning relay drones missions.
- **Drone placement** - The relay drone can position itself between the mission drone and the Ground Station.
- **Network probes integration** - Nodes can use the network probes to monitor their connections with other nodes appropriately.
- **Relay drone distance** - The relay drone can use the network probes to find the optimal distance to keep from the mission drone, as mentioned in Section 4.2.4.
- **Routing** - The relay can efficiently perform its primary function by forwarding information from the mission drone, such as telemetry, to the Ground Station and vice-versa.

Software and scripts

Each node was executing a collection of programs that enabled them to carry out their mission duties:

- Mission drone (ID: drone01) and relay drone (ID: drone02)
  - **Telemetry Filter**, **Command Handler**, and **Status Tracker** - It will be able to receive commands and send telemetry and status updates.
  - **Rosbags** - Record every ROS2 topic to obtain logs.
- Mission drone (ID: drone01)
  - **Flight mode** - The drone will be operated by hand using an RC.
  - **Network probes** - It is constantly monitoring the network with Ground Station.
  - **Mission code** - The code (**goto_with_relay.py**) analyzes the probe’s output to determine when a relay is required.
- Relay drone (ID: drone02)

- **Flight mode** - Autonomously flight using commands sent by the mission code.
- **Network probes** - Both the mission drone and the Ground Station's network are monitored.
- **Mission code** - The code (**relay_drone.py**) places the relay in the proper location using the probe's data and telemetry from the mission drone.

- Ground Station
  - **Drone Manager**, **Status Updater** and **Task Scheduler** - Monitor the drones' telemetry and other relevant information.
  - **Rosbags** - Record every ROS2 topic to obtain logs.
  - **Network probes** - Monitor the network with both drones.

To work as a single network, every node on the platform ran the batman-adv software, as detailed in 4.2.6. This allowed the relay drone to forward data from the mission drone to the GS and vice versa.

Parameters used

For this test, the following parameters were used:

- **Mission drone**
  - **relayNetworkThreshold** - 10. When the drone has ten network measurements in a row with MEDIUM or LOW quality, it requests a relay, and when it has ten network measurements in a row with HIGH quality, it will cancel the relay.
- **Relay drone**
  - **targetDistanceDefault** - 12 meters. The relay drone will stay 12 meters away from the mission drone when the mission begins.
  - **distanceAdjustment** - 0.1 meters. The relay drone will adjust the distance to maintain from the first drone with this amount each time a new coordinate is calculated, adding or subtracting from the current distance (see Section 4.2.4).
  - **relaySignalToKeep** - 60. This is the relay drone's signal value to keep from the mission drone. We chose the value 60 since it is in the middle range (range is 30 to 90).

Video emulation

One of the goals of this scenario was to use a camera in the mission drone to broadcast video to the Ground Station. We could not attach a camera to the drone due to hardware limitations. We emulated this transmission since it is essential to see how

video transmission affects network quality. We used iperf[6] for this, simulating the load of the video in the network, sending packets from the mission drone to the Ground Station at a set rate.

The Ground Station was running the following command:

```
iperf -s -u -i 1
```

It ran as a server (receiving packets) because of the -s parameter. The -u parameter tells the server to use User Datagram Protocol (UDP) to accept packets. The -i specifies the interval between periodic bandwidth reports (this only changes the rate that iperf notifies in the terminal of the packets receives in that interval).

The mission drone, on the other hand, was running the following configurations to run as a client:

```
iperf -c X.X.X.X -u -b 1m -t 600
```

The -c argument provide the IP address to which the packets should be sent (in this case, the Ground Station). The -u option instructs the client to send packets using UDP. We sent data at a pace of 1MB/s, an expected rate for video transmission, determined by the -b parameter. Finally, the -t parameter specifies the duration of the command. We set the timer to 600 seconds, the average length of time for a mission.

Mission plan

Figure 5.5 shows the mission path that the mission drone was planned to follow.

Figure 5.5: Path planned for the mission drone.

[6] `https://iperf.fr/`

The location of the Ground Station is indicated by the letters GS. The mission drone was controlled by an RC and flew in the direction of the arrows. The green arrow depicts the predicted zone where the drone should have a good connection with the GS. The yellow arrow denotes the point at which the network begins to deteriorate, resulting in a relay request from this drone. Finally, the red zone is where the relay should be within range of the first drone to perform its duty.

The first part of the mission entails maneuvering the first drone slowly via the arrow-marked path. The drone's path does not have to be equal to the one illustrated in Figure 5.5, but the goal is to move the drone perpendicularly from the GS to determine if the relay drone is correctly aligned with the mission drone and the GS. The second stage entails examining the recovered network data and determining whether the relay could correctly calculate a distance based on signal strength using the parameters defined in the previous Section.

Results

Using the telemetry data acquired, the first thing we will look at is the route taken by the two drones. We will filter part of the data and recreate the path from the raw data, because placing all of the data together in a map could be confusing to examine.

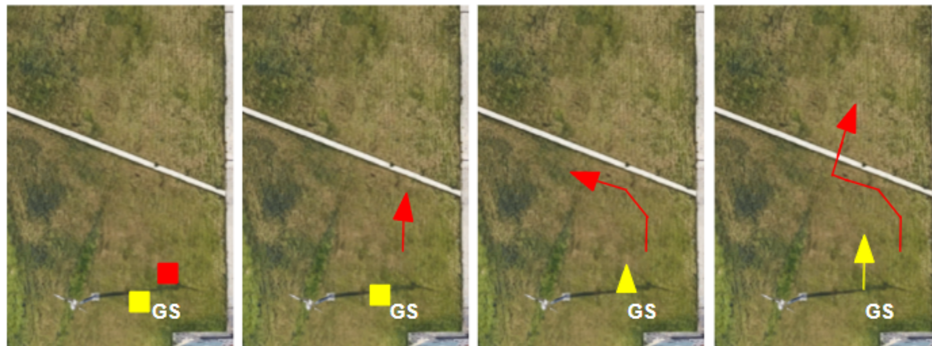

Figure 5.6: Relay request performed by the first drone.

Figure 5.6 shows the moment the relay request was performed by the mission drone. The red color represents the mission drone, while the yellow color represents the relay drone. Both drones are marked with squares while on the ground. The mission started when the first drone took off and followed its path. Within a distance of more than 10 meters from the Ground Station, the network began to lose quality, prompting the first drone to request the relay only seconds after it started moving forward. The third image in Figure 5.6 illustrates the relay request being made, indicating that the relay drone (yellow triangle) took off. The latter was already doing its duty in the fourth image.

The remainder of the drones' itinerary is shown in Figure 5.7. The first drone was piloted manually and followed an "S"-shaped trajectory, while the relay drone autonomously gathered telemetry from the first drone and calculated the ideal position to land. The pilot would stop the drone for roughly 1 second when turning on this "S"-shaped path. Then, it accelerates until reaching the next corner, then coming to a complete stop be-

fore turning and proceeding to the next destination doing this for the whole course.

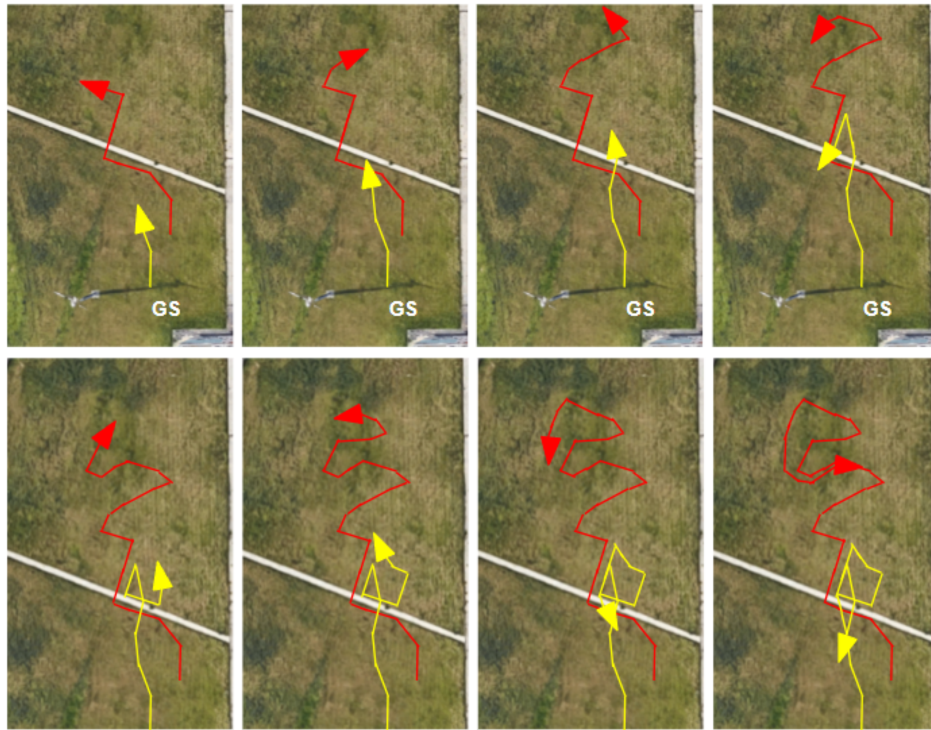

Figure 5.7: Path followed by the drones.

When the drone approached the GS, the relay did the same, and when turning left or right, the relay would also do the same.

At the end of the mission, the drone approached the GS and communicated directly with it without needing the relay, so it canceled it. The relay drone returned home and landed after receiving the canceling message.

We can conclude from the images that the relay remained inline between the mission drone and the Ground Station. Of course, this alignment was not perfect; the drone may depart from the "ideal line" occasionally, but minor deviations are acceptable. This validates one of the features of the relay mission's implementation: the drone's positioning.

Mission drone

Now we will address how the network's quality affected the mission. In this test, a drone will request a relay if the network quality to the Ground Station begins to deteriorate.

Figure 5.8 depicts the history of network quality between drone01 and the GS throughout the mission. The X-axis indicates the number of seconds since the mission began, while the Y-axis represents the network's quality. A value of 1 indicates low quality, a value of 2 indicates medium quality, and a value of 3 indicates high quality.

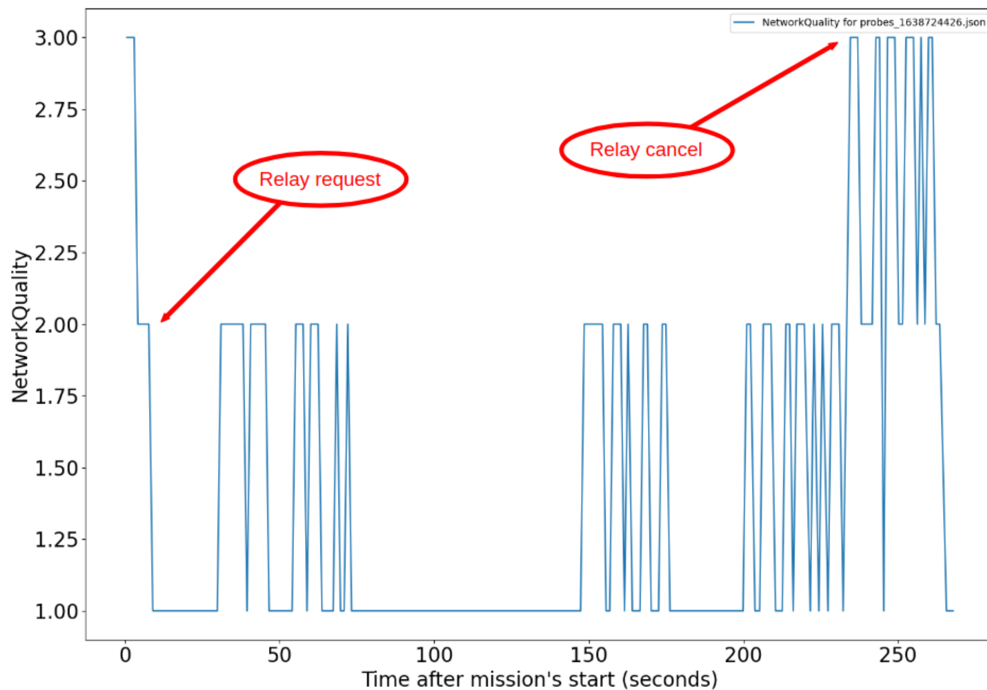

Figure 5.8: Network quality between drone01 and the Ground Station during the mission.

As we can see, within a few seconds after the mission began, the network quality dropped below high, prompting the drone to request a relay. Throughout the rest of the mission, the quality remained low, for the most part, occasionally rising to medium. This occurred because, despite the relay being already in place, the information in the graph above is computed using a direct connection from drone01 to the GS. As a result, while the mission drone could still send telemetry to the GS via the relay drone, it could not do so directly, leading to low quality in the connection (or no connection at all). Around 230 seconds after the mission's beginning, the first drone approached the GS, restoring the quality to high and allowing direct communication between these two nodes. As a result, the drone canceled the relay, and the relay drone returned home.

This concludes the information about the mission drone. It was able to carry out its duties correctly, requesting and canceling the relay at the appropriate times using the network probes' information.

Relay drone

We will now look at the second drone, the relay. This one is a little more complicated because it uses probes to measure distance and check for connection with the GS if a relay is required. However, we only had two drones for the mission, so we had to disable the latter. Figure 5.9 depicts a part of the data retrieved by the relay drone during the mission. The drone used telemetry to determine the optimal and actual distance between the two drones each time it issued a new coordinate.

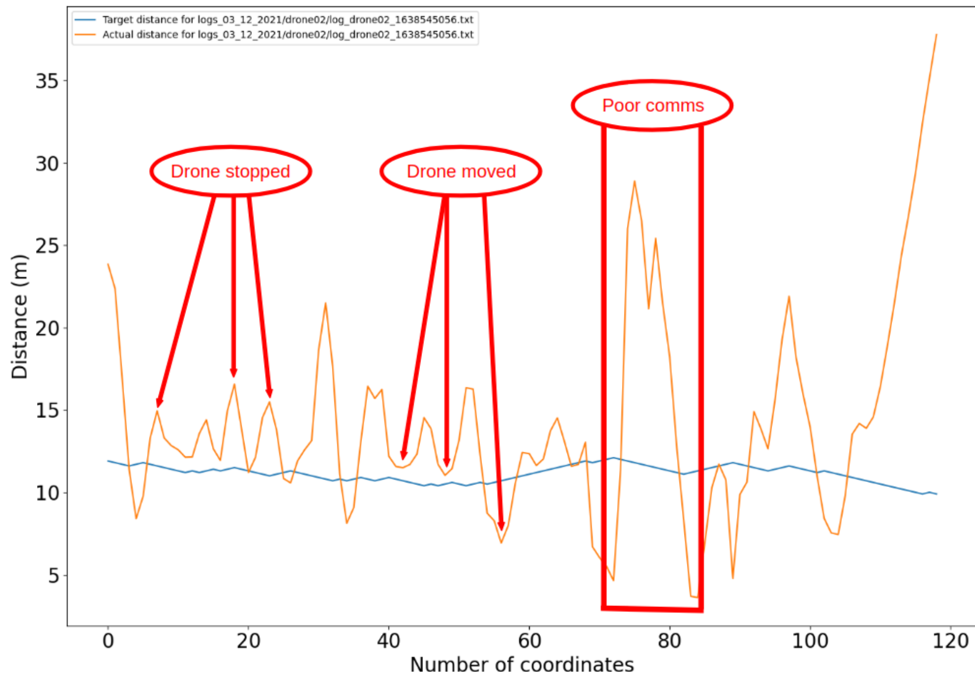

Figure 5.9: Optimal and real distance between the two drones throughout the mission.

The blue line denotes the distance the relay should maintain from the first drone, while the orange line denotes the actual distance. As we can see, the optimal distance began at 12 meters, specified in the **targetDistanceDefault** parameter. However, contrary to our expectations, the distance did not increase above this value. Preliminary tests with the drones on the ground showed that the **Signal**'s value would only go above 60 when the drones were 20-25 meters apart. Instead, we can observe that the value fluctuated continuously around 12 meters, even dropping to 10 meters near the end. This is due to the **Signal**'s value oscillating around 60 throughout the mission, causing the ideal distance to change quite frequently and could be caused by the drone's vibrations during the flight. The orange line also oscillates a lot, but this is to be expected. As previously stated, when the pilot was flying the mission drone, it stopped the drone while turning for a brief moment before continuing. When the first drone goes forward, the relay drone will need a fraction of a second to pick up the telemetry, calculate, and issue a new coordinate when moving forward. Thus there will be a slight delay. As a result, the distance increases. Similarly, when the drone comes to a halt, the relay can approach it, resulting in a reduction in distance and a close approach to the optimal distance, the blue line. This happened in a cycle throughout the mission, so the orange line is made up of rising and falling lines. Figure 5.9 shows a few of these maximum and minimum peaks. We can also see a time when the relay lost connection totally for a brief while, as indicated by the message "Poor comms", resulting in a higher climb in the distance than the rest of the mission. This happened because the mission drone moved forward too quickly. Because the connectivity was deteriorating at more than 12 meters, the drone lost contact with the first drone, which it only regained when we brought the first

drone closer. The distance increased near the end because the first drone had already canceled the relay, and the relay was returning home. The same happened at the start when the drone had just taken flight and was further away. Overall, the relay stayed between 14 and 15 meters away from the first drone. This results in an acceptable average inaccuracy of 4-5 meters because the ideal distance is always around 10-11 meters.

Figure 5.10 depicts the Signal's value throughout the mission between drone02 and drone01 to verify the oscillation in this parameter.

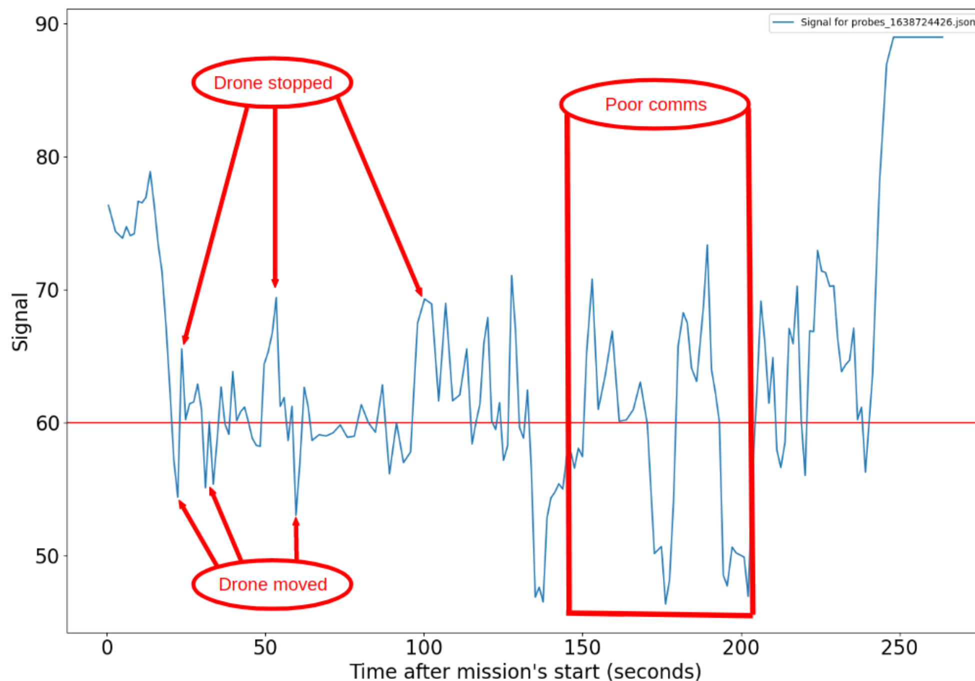

Figure 5.10: Signal between the two drones throughout the mission.

The X-axis shows the number of seconds since the mission began, while the Y-axis shows the **Signal** value. The blue line reflects the value of **Signal** collected by drone02 throughout the mission, and the red line indicates the value 60 we provided to **relaySignalToKeep**. We can observe from Figure 5.10 that the **Signal** was exceedingly unreliable during the mission. It is important to remember that a lower number indicates a greater connection and vice versa. As previously stated, when the first drone comes to a halt, the connection's strength should grow, lowering the value as the relay can get closer. Therefore, the graph will show maximum peaks, marked with "Drone Stopped." When the first drone resumes flight after a halt, it increases its distance from the relay, causing the connection to deteriorate, resulting in a greater value and a minimal peak, labeled "Drone Stopped." We expected the **Signal** to climb to 90 in the mission phase when contact had stopped ("Poor comms"), indicating no connection. However, due to the instability we discussed before, the **Signal** just bounced quickly between 50 and 75. Similar to Figure 5.9, the value was higher at the start of the mission because the relay had just taken off and was further away, and in the end, when it was returning home.

We can now compare Figures 5.9 and 5.10 using this information. According to the

implementation explained before, when the Signal's value falls below 60, indicating a good connection, the optimal distance (blue line in Figure 5.9) should increase and vice-versa. Looking at Figure 5.9, we can see that the optimal distance oscillates many times around 12 meters in the first part of the mission, whereas the blue line rises and falls more linearly in the second half. This happens because, in Figure 5.10, we can see that, while the Signal's value tends to bounce around 60 through the whole mission, this oscillation occurs far more frequently in the first half, while the value stays longer above or below the red line in the second half. Analyzing the second half of the mission, we can see that the Signal remains above 60 most of the time. Similarly, as shown in Figure 5.9, the ideal distance tended to reduce during this period. As a result, both graphs have an observable correlation, as intended.

Routing

We need to validate the routing to complete the analysis of the data collected during this mission. We will look at the raw telemetry data collected by the various nodes to verify the routing, shown in Figure 5.11. Every message published over ROS2 was individually recorded by each of the drones and the Ground Station.

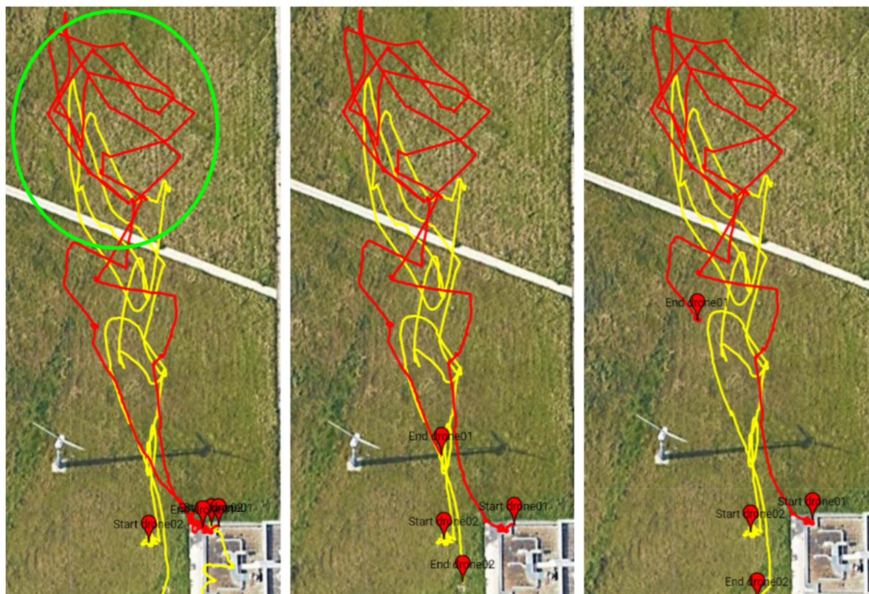

Figure 5.11: Telemetry gathered by the different nodes during the mission.

The telemetry data acquired by the Ground Station is shown on the left, the telemetry gathered by drone01 is shown in the middle, and the telemetry data collected by drone02 is shown on the right. The path taken by drone01 is shown in red, whereas the route taken by drone02 is shown in yellow.

By comparing the three images in Figure 5.11, we can observe that the start and end positions of the drones' path change slightly, as represented by the red markers. This happened because the data started and stopped being recorded in the different nodes at different times, which differed by a few seconds. However, the routes appear to be

equal, and the information obtained by the three nodes shows no significant variances. Both drones have the same information because if one drone can receive telemetry from the other, the opposite is also true. The case is slightly different with the Ground Station, since it will eventually lose communication with the first drone during the mission, resulting in telemetry loss, unless, of course, the relay is forwarding such data. The fact that the GS contains the same information as the drones demonstrates that it was getting this data via the relay throughout the flight. Otherwise, we would not have the telemetry in the green circle area because the drone was too far away. This is because the green circle symbolizes a zone where the mission drone would be too far away from the GS and unable to send telemetry if the relay was not present.

Discussion

We completed the mission successfully in the end. We were able to verify the features we wanted and meet our objectives. Drones can send requests to the Ground Station, which the latter can answer. A relay can receive telemetry from another drone and utilize it, along with the data gathered by the network probes, to compute an appropriate location between that drone and the GS. There are, however, a few things that need improvement. The obtained Signal was extremely unstable, resulting in a significant oscillation in the optimal distance value. This did not happen as much in the lab, and it could be due to the drone's movements during flight. Another factor that hampered our testing was the wireless USB adapter's maximum range, which was only about 12-15 meters in the field but at least 20 meters in the lab. Because the relay would not react quickly enough before losing the connection if we moved the mission drone too fast, we could only make minimal motions with this drone. The goal would be to obtain a distance graph comparable to Figure 5.2, which was not achieved under these circumstances. Aside from the hardware, we must improve the network metrics. It is unreliable to utilize merely the current Signal's value; alternatively, we could use an average Signal with an average weight based on the previous seconds. This would result in a more consistent value sequence for this parameter, and hence a more consistent value for the ideal distance.

5.3 Summary

We discussed the strategies used to debug and validate the software developed in this Dissertation in this chapter. Initially, everything was tested in a virtual environment, with both the Ground Station and the drones running on the same device. The testing in real hardware was carried out only after ensuring that everything was working well in the simulator and was carried out in incremental steps. First, the primary communications were established, followed by the positioning of the relay drone, followed by the relay in combination with full drone-drone and drone-GS connection, and last, the network probes were integrated and validated.

The relay was tested with real hardware, and while there is a need for improvement in a few aspects, such as network metrics, the majority of the results were favorable.

A few constraints with the Android device integration, as stated in Section 5.1.2, rendered it impossible to test in real conditions at the time of writing this Dissertation, but this will be addressed in the near future.

Chapter 6

Conclusion

We conclude this Dissertation with a brief overview of the work done so far and some recommendations to improve the platform.

6.1 Final observations

This Dissertation established a foundation for a new type of architecture. We based on past work from other Dissertations and attempted to address the most significant drawbacks. The earlier architecture was more centralized, which meant that the drones were always controlled by the Ground Station, causing issues when the drones were too far away and leading to communication losses. We began by establishing a direct communication channel between drones, allowing them to contact one another. The relay scenario was then chosen as a starting point because it involves communication between drones as well as between drones and the Ground Station. The network probes, which measure the network's quality, signal, and other essential information, were then added to the platform as a new feature. The drones utilize this information to determine when a relay is required, allowing them to make a relay request. The ability to use Android devices within the rest of the platform was another new feature. A user uses an application to request a drone by providing their coordinates to the drone.

The work was accomplished by improving the two main modules: the Drone Module and the Ground Station Module. The drone module runs the software from the previous Dissertation, which allows the drone to publish telemetry and receive commands. The software from this Dissertation executes the new mission code and monitors the network's quality. The Ground Station Module also runs the previously developed software, which acts as a fleet manager by receiving telemetry from the drones, monitoring which ones are online, and reporting any errors that may occur during missions. In addition, the new software handles requests from the drones and keeps track of which drones are available to perform tasks, such as the relay mission.

There are two new modules - Network and Android. The first module allows a node to keep track of the network's conditions to other nodes. This information may be used for logging purposes or, as we have seen in the relay scenario, may have a more prominent role and affect some conditions during the mission's execution. The Android module is

used to allow the connection of Android systems to the rest of the platform. For example, a user could use an application to request a drone to its location in a rescue situation. This drone would provide video feedback to the Ground Station, so whoever is watching could help assess the situation.

The work produced throughout this Dissertation was rigorously evaluated. Everything was tested in stages, which meant that every time a new feature was added, it was debugged before moving on to the next one. Thanks to the simulation tool, we were able to test the software locally, on the same device, before exporting it to real hardware. The drone-drone communication was the first thing to be tested, followed by the relay placement algorithm and then the relay mission, which included network monitoring. After validating these features using the simulator, we proceeded to field tests with the software produced proving functional. However, some components need improvement.

Some difficulties had to be overcome during the development of this Dissertation. The implementation used the ROS Client Library for Python (RCLPY) and required a specific approach for our work, which did not appear to be widely used because there were few examples of how we could implement it. Most of the examples found were straightforward, making it a little challenging to work with initially. The $paho - mqtt$ library, which we used to create the Android app, presented the same issue. Unfortunately, most of the available examples only covered the essentials. Because our application was a little more complex to implement, we needed to take a few more steps to make it work, which required a lot of trial and error. The tests on real hardware were difficult at times, since the drone's hardware can have small problems, such as displaying errors that prevent the drone from taking off and needing several restarting of the drone's Flight Controller. As a result, having multiple drones working simultaneously proved complicated sometimes when more than one drone was required.

Finally, we believe we addressed the key points. The main goal of creating an architecture more independent from the Ground Station was achieved. We were able to deploy relay drones and make them work even when the connection with the GS was momentaneously lost. The communication was successfully performed, and the drones could send messages to each other. Network probes are now integrated with the rest of the platform, allowing a drone to monitor the connection with other nodes. The Android module needs testing since it was not validated in a real scenario, only in a simulated environment.

Although only a few use case scenarios were explored, the work produced here served as a good starting point for a more decentralized architecture. Now is the time to continue the work that was started here, refining existing features and introducing new ones, and expanding the variety of scenarios in which they can be employed.

6.2 Future work

This Dissertation supported the development of drones that are less reliant on the Ground Station and added new modules. However, many features can be enhanced or added, such as the following:

- **Improve the relay placement algorithm** - The developed placement algorithm is

very straightforward. However, it is possible to improve and optimize it in the future by employing more advanced algorithms to find the ideal location for the drone, for example, by estimating the first drone's future position based on its direction and speed.

- **Improve the network metrics** - Using only the actual Signal value to verify the network's conditions did not show to be the most effective way. We should employ more complex methods in the future, such as generating an average network signal based on several past measurements and combining it with specialized software like iperf.

- **Add support for several mission drones to have a single relay drone** - For the time being, a relay drone can operate as a relay for a single mission drone. If two or more mission drones are too far away from the Ground Station but close enough to each other, a single relay drone can relay to all of them. We could reduce the number of drones required in real-world circumstances by developing this type of support.

- **Extend the number of use case scenarios** - Since this was a starting point, the relay and the Android scenario were the only ones tested with the new decentralized strategy, with drones solely using telemetry and network data to progress the mission. More use cases should be explored, such as when a drone uses a specific type of sensor, such as a temperature sensor, and the data collected from that sensor influences the flight path of other drones in the fleet.

- **Test the Android support in real scenarios** - The Android implementation was evaluated in simulation, but we could not test it in real-world circumstances due to the limitations stated in Section 5.1.2. To implement this in a real scenario, we need a method to allow the drone that is performing the rescue mission to have a connection through ad-hoc (for other drones and the GS) and a second connection to Android devices.

References

[1] Ana Silva. “A mission planning framework for fleets of connected drones.” Master’s thesis. Integrated Masters in Computer and Telematics Engineering - University of Aveiro, 2021. Available at: http://hdl.handle.net/10773/31385.

[2] Junyan Hu and Alexander Lanzon. “An innovative tri-rotor drone and associated distributed aerial drone swarm control.” In: *Robotics and Autonomous Systems* 103 (2018), pages 162–174. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2018.02.019. Available at: https://www.sciencedirect.com/science/article/pii/S0921889017308163.

[3] David Donald. *Encyclopedia of World Aircraft*. Prospero Books, 1997, page 854.

[4] Richard J Evans. *The Third Reich at War*. Penguin Press, 2009.

[5] Priscilla Mary Roberts Spencer C. Tucker. *The Encyclopedia of the Arab-Israeli Conflict: A Political, Social, and Military History: A Political, Social, and Military History*. 2008, pages 1054–55.

[6] Hendrik Bodecker Kay Wackwitz Lukas Schroth. *Drone Application Report 2021*. Drone Industry Insights, 2021.

[7] Paolo Tripicchio et al. “Towards Smart Farming and Sustainable Agriculture with Drones.” In: *2015 International Conference on Intelligent Environments*. 2015, pages 140–143. DOI: 10.1109/IE.2015.29.

[8] Kang Yang, Guang You Yang, and S Isi Huang Fu. “Research of Control System for Plant Protection UAV Based on Pixhawk.” In: *Procedia Computer Science* 166 (2020), pages 371–375. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.02.082.

[9] Khoa Dang Nguyen, Cheolkeun Ha, and Jong Tai Jang. “Development of a New Hybrid Drone and Software-in-the-Loop Simulation Using PX4 Code.” In: *Intelligent Computing Theories and Application*. Edited by De-Shuang Huang et al. Cham: Springer International Publishing, 2018, pages 84–93. ISBN: 978-3-319-95930-6. Available at: https://link.springer.com/chapter/10.1007/978-3-319-95930-6_9.

[10] Chien-Ming Tseng et al. “Flight Tour Planning with Recharging Optimization for Battery-operated Autonomous Drones.” In: *CoRR* abs/1703.10049 (2017). arXiv: 1703.10049. Available at: http://arxiv.org/abs/1703.10049.

[11] Fendy Santoso et al. “Robust Hybrid Nonlinear Control Systems for the Dynamics of a Quadcopter Drone.” In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 50.8 (2020), pages 3059–3071. DOI: 10.1109/TSMC.2018.2836922.

[12] Felipe Matheus Mota Sousa, Vitória Matos Barbosa Amarante, and Rodolpho Rodrigues Fonseca. “Adaptive Fuzzy Feedforward-Feedback Controller Applied to Level Control in an Experimental Prototype.” In: *IFAC-PapersOnLine* 52.1 (2019). 12th IFAC Symposium on Dynamics and Control of Process Systems, including Biosystems DYCOPS 2019, pages 219–224. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2019.06.065. Available at: https://www.sciencedirect.com/science/article/pii/S2405896319301508.

[13] Chao-Yang Lee. “Cooperative Drone Positioning Measuring in Internet-of-Drones.” In: *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*. 2020, pages 1–3. DOI: 10.1109/CCNC46108.2020.9045111.

[14] Nuno Paula. “Multi-drone control with autonomous mission support.” Master’s thesis. Integrated Masters in Computer and Telematics Engineering - University of Aveiro, 2018. Available at: http://hdl.handle.net/10773/27846.

[15] Nuno Paula et al. “Multi-drone Control with Autonomous Mission Support.” In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pages 918–923. DOI: 10.1109/PERCOMW.2019.8730844.

[16] Juan A. Besada et al. “Drones-as-a-service: A management architecture to provide mission planning, resource brokerage and operation support for fleets of drones.” In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pages 931–936. DOI: 10.1109/PERCOMW.2019.8730838.

[17] Shafkat Islam and Abolfazl Razi. “A Path Planning Algorithm for Collective Monitoring Using Autonomous Drones.” In: *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*. 2019, pages 1–6. DOI: 10.1109/CISS.2019.8693023.

[18] Albert Y. Chung, Joon Yeop Lee, and Hwangnam Kim. “Autonomous mission completion system for disconnected delivery drones in urban area.” In: *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2017, pages 56–61. DOI: 10.1109/ROBIO.2017.8324394.

[19] Roberto Pinto et al. “A network design model for a meal delivery service using drones.” In: *International Journal of Logistics Research and Applications* 23.4 (2020), pages 354–374. DOI: 10.1080/13675567.2019.1696290. eprint: https://doi.org/10.1080/13675567.2019.1696290. Available at: https://doi.org/10.1080/13675567.2019.1696290.

[20] Theodore Zahariadis et al. “Preventive maintenance of critical infrastructures using 5G networks amp; drones.” In: *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. 2017, pages 1–4. DOI: 10.1109/AVSS.2017.8078465.

[21] Pietro Tosato et al. “An Autonomous Swarm of Drones for Industrial Gas Sensing Applications.” In: *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. 2019, pages 1–6. DOI: 10.1109/WoWMoM.2019.8793043.

[22] Sven Mayer, Lars Lischke, and Pawel W. Woźniak. “Drones for Search and Rescue.” In: *1st International Workshop on Human-Drone Interaction*. Ecole Nationale de l’Aviation Civile [ENAC]. Glasgow, United Kingdom, May 2019. Available at: https://hal.archives-ouvertes.fr/hal-02128385.

[23] Jiyeon Lee et al. “Constructing a reliable and fast recoverable network for drones.” In: *2016 IEEE International Conference on Communications (ICC)*. 2016, pages 1–6. DOI: 10.1109/ICC.2016.7511317.

[24] Luis Ramos Pinto, Luis Almeida, and Anthony Rowe. “Balancing Packet Delivery to Improve End-to-End Multi-hop Aerial Video Streaming.” In: *Iberian Robotics conference*. Springer. 2017, pages 807–819.

[25] Luis Ramos Pinto, Luis Almeida, and Anthony Rowe. “Demo abstract: Video streaming in multi-hop aerial networks.” In: *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE. 2017, pages 283–284.

[26] Luis Ramos Pinto and Luis Almeida. “Optimal Relay Network for Aerial Remote Inspections.” In: (2021). DOI: 10.20944/preprints202112.0037.v1.