Journal of Intelligent & Robotic Systems (2023) 108:2
https://doi.org/10.1007/s10846-023-01820-7

REGULAR PAPER

A Mission Planning Framework for Fleets of Connected UAVs

Margarida Silva[1,2] · André Reis[1,2] · Susana Sargento[1,2]

Received: 8 June 2022 / Accepted: 19 January 2023 / Published online: 2 May 2023

Abstract

Currently, several platforms offer solutions for the management and control of fleets of unmanned aerial vehicles (UAVs), addressing a wide range of scenarios, each with its own particular set of objectives. Some of these solutions are mission planning platforms for broader usage, without aiming to solve a single scenario. However, these often either do not support multi-UAV collaboration or generate a static flight trajectory, which does not facilitate the coordination of a fleet of several UAVs in a mission that may need to adapt to the current environment. Through the development of a domain-specific language (DSL) - EasyMission - for UAV mission definition and control, we introduce a mission planning framework that makes it possible to describe mission plans that also enable the user to define adjustments and constraints that may have to be taken into account in real-time according to sensor readings or other events. This framework enables inexperienced users to design missions with low or moderate complexity levels, while still being a useful tool for advanced users due to its versatility in addressing multiple scenarios through a single platform. We show how the mission framework is able to easily define differentiated mission examples with distinct purposes and scenarios, and with real-time decisions and constraints.

Keywords Autonomous aerial vehicles · Drones · Control platform · User-friendly description language · Missions · Unmanned systems · Multi-drone platform

1 Introduction

Nowadays, the usage of unmanned aerial vehicles (sUAVs)[1] is frequent in several different scenarios, such as monitoring emergency situations and natural disasters, patrolling urban areas to support police forces, and tourist applications, such as the real-time video transmission of points of interest. It is not unusual for the control of the UAVs to be dependent on human intervention in some of these situations, which requires professionals specialized in its control. However, in recent years, several solutions have emerged that enable the autonomous flight of these vehicles, minimizing manual interference. Since the context of these use cases is so heterogeneous, many of the existing solutions for autonomous control are aimed to solve particular scenarios. While there are already several generic mission planning platforms, it is common that they only allow defining missions which consist of a linear set of waypoints to be traversed. This restricts the flexibility of the mission plans, which do not take into account possible unforeseen circumstances.

When designing a mission planning system, more specifically, when defining how the user describes a mission, we have to balance two factors: how difficult it is for the user to describe the desired mission plan successfully, and what level of mission complexity can be achieved with the mission planning. A graphical user interface (GUI) is an intuitive medium for the user to describe several mission steps. However, it is less flexible when there is a need to define restrictions, such as acting upon a sensor reading, or when it is required to coordinate several entities.

Considering that people and organizations that own UAV fleets often come from a technological background, we propose a solution in which the user describes the mission flow through a scripting language developed specifically for this purpose - a DSL to control fleets of UAVs.

[1]UAV will be used whenever the discussion is on a conceptual basis; if the discussion is related to specific types of UAVs, such as drones, in an implementation and experimental basis, we will refer to drones instead.

✉ Margarida Silva
margaridaocs@ua.pt

1 Instituto de Telecomunicações, Aveiro, Portugal

2 DETI, University of Aveiro, Aveiro, Portugal

A mission solely based on following given waypoints can be written without much programming knowledge or even generated using other interfaces, while also enabling the usage of control flow statements for more advanced missions. Many navigation algorithms have already been developed and could be easily implemented using this language, which in turn would make it straightforward to test, comprehend and modify such algorithms. The purpose of this language is to provide flexibility - it should be possible to write simple scripts without a deep understanding of the framework; however, it should not restrict an experienced user from implementing complex logic.

Using a base UAV control platform as groundwork, we seek to develop a mission planning framework, EasyMission, which allows the description and definition of a mission plan by using a DSL. This integration combines the advantages of having a single generic platform that can be used for trivial tasks that require UAVs, with the ability to orchestrate a UAV swarm to address more specialized scenarios. The final platform enables mission plans that control the UAVs, sense the environment and react accordingly, and cooperate between multiple UAVs. This solution requires a much more complex approach than following a mission with a set of waypoints, since decisions must be taken in real-time according to the sensed information.

To evaluate the proposed platform, we defined use cases to validate how the mission planning framework can be used to solve particular scenarios:

- Tracing the perimeter of a fire using a single UAV;
- Using a secondary UAV to continue a mission started by another UAV that required replacement;
- Maintaining connectivity between a main UAV and the ground station by forming a relay bridge of several UAVs.

The results show that it is possible to create a common solution to build these scenarios using the mission planning framework.

The paper is organized as follows. Section 2 presents related work for UAV control and mission planning platforms. The description of the proposed architecture is presented in Section 3, as well as the main components that it comprises. The mission planning framework and description language are described in Section 4. Section 5 presents the use cases that were selected to demonstrate the functionalities of the platform and the validation of the mission description language. With Section 6, we present the final conclusions and aspects that could be improved for future work.

2 Related Work

Recent research efforts present a wide array of UAV control solutions which support mission planning. Some of these are focused on specific use cases, while others provide more generic functionality.

The work in [1, 2] describes a high-level architecture for the design of multi-UAV systems which includes communication and networking, coordination, and sensing modules. The mission planning component takes high-level tasks defined in a user interface, and breaks it down into individual flight routes for each UAV. The path planning control is centralised, which allows replanning the routes and adaptive coordination. The system was demonstrated in several real-world applications, which included assistance during a disaster, documenting progress in a construction site, and participating in a fire service drill.

A system supporting complex mission definition, planning and execution is proposed in [3]. This project provides a platform for infrastructure inspection using UAVs. The mission definition aims at improving planning efficiency by setting waypoints, using high-level mission definition primitives. The user has access to a GUI, where it is possible to use a map tool to select the central target location, and then it is possible to choose the inspection type. However, it provides no support for multi-UAV interaction, and it has a limited communication range using Wi-Fi.

Other works, such as [4], presented methods to generate trajectories and missions for multiple UAVs, satisfying a set of given Signal Temporal Logic requirements. This approach can be used as an offline path planner, and the resulting missions are static, unable to be adjusted at runtime.

In [5], a model for mission planning in outdoor material delivery with UAVs is proposed. Mission plans have to consider several aspects such as different weather conditions, payload capacity, energy capacity, fleet size, and the number of customers visited by a UAV. The proposed solution presents a model that takes the previous considerations into account.

An embedded decision-making module for autonomous UAVs missions is proposed in [6]. This module enables the user to choose a recovery action when there is a failure, generating a new mission plan. According to sensor data, the failures are defined and may lead to changes in trajectory, emergency landings, or decreasing the speed.

The connection of abstract task definition at a mission level with the control functionalities in autonomous missions is addressed in [7]. Since this work also approaches heterogeneous vehicles, a common ground was found by defining parametrized tasks such as *fly-to*, *take-off*, *scan-area*, or *land*. The behaviour of these actions was implemented for each platform.

A language aimed at solving collaboration among robots and communication with humans was proposed in [8], called Situation Information Exchange and Interpretation Language (SIEIL). This language's purpose is to provide a common language specification to command and control robotic forces. SIEIL uses a context-free grammar to describe the possible actions, subjects of those actions, and other variables. Those expressions are transformed into RDF/XML documents to be interpreted as commands by the intervening robots.

The work in [9] targets underwater vehicles instead of UAVs, and it approaches task description languages. The proposed notation provides a hierarchical structure consisting of simple and composite statements. The task description is easily understandable by humans and may contain responses to events, such as finding an object or setting a time limit for the task. However, collaboration was not addressed, with the mission being sent to each vehicle to be executed individually. Similarly, the work in [10] also suggests an approach to mission planning for underwater vehicles by formalising the task description. In this case, the language is based on the GeoJSON standard, which is a format for encoding geographic data structures, such as a point, a line string, or a polygon.

In [11, 12], a platform for communication and multi-UAV control with autonomous mission support was presented. The architecture was composed of the UAV side and the ground station components. It contains a mission planner that enables the user to configure a mission in a web interface, selecting the desired actions. By monitoring the UAV's telemetry, it is possible to verify anomalous situations, such as a low battery level. In that case, the current UAV will be replaced. During a mission, an UAV can request for another one to collaborate, which will cause the remaining waypoints to be distributed among the participating vehicles.

3 Architecture

Before addressing the mission planning framework, we had already designed, implemented and tested a multi-UAV control and monitoring platform. The architecture of this base platform is depicted in Fig. 1 and is published in [13]. With this platform, we are able to remotely send instructions to the UAVs, such as an order to takeoff or to move to a position, as well as monitoring telemetry and sensor data.

The communication between drones, ground station, and sensors relies on Robot Operating System (ROS), more specifically, ROS2. The decision to use ROS2 instead of ROS was due to its ability to provide node discovery without

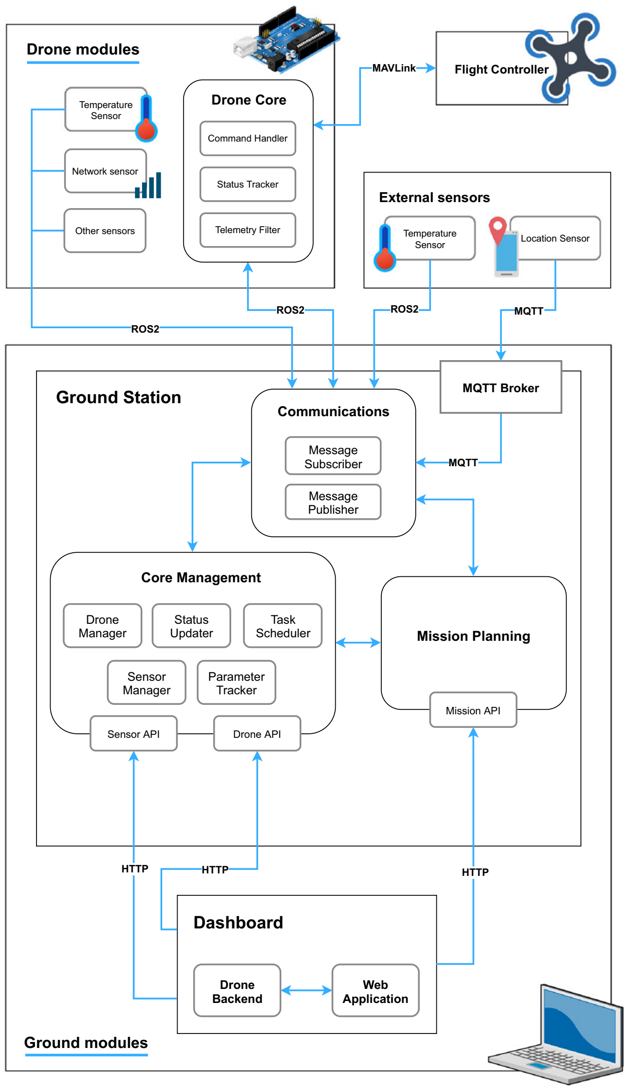


Fig. 1 Multi-UAV mission control platform architecture

a centralized master node, essentially allowing Peer-to-Peer (P2P) communication. The drone modules and ground station both communicate through ROS2 messages. Sensors may either use ROS2 or MQ Telemetry Transport (MQTT), according to the requirements of the devices, but if MQTT is used, only the ground station will be able to receive those messages, since the drone software is not configured to handle them. Alternatively, if an external module is to be integrated with our platform and implements standard ROS instead of ROS2, ROS provides a package to set up a bridge between ROS and ROS2 nodes.

The drone module publishes ROS2 messages with telemetry data (altitude, flight mode...) and status date (taking off, landing...) and receives command messages. The ground station receives these messages and provides a representational state transfer (REST) application programming interface (API) through which it is possible to monitor the fleet's state and issue the commands.

A dashboard is also implemented, which communicates with the ground station and provides a GUI for the previously describe functionalities.

The Mission Planning module featured in this architecture is the main focus of this work. Without this module, the system presents a low level of autonomy, since the user has to provide each action to the UAVs in real-time. The internal architecture of this module is not depicted in this figure since it comprises many components that will be later explained in Section 3.2b.

Ideally, a UAV should autonomously plan the appropriate course of action by observing the current state of the environment (such as sensor or telemetry data from itself or other UAVs), formulating a plan that best suits the current goal and given constraints, and acting accordingly. As an example, the user could provide a task to a UAV, such as monitoring a predetermined area, and the UAV would locally decompose it into several instructions. This could lead to different outcomes in response to events, such as a high temperature reading or detection of bad network conditions, which could trigger additional tasks such as avoiding a particular hot area or requesting a backup UAV. This level of autonomy and collaboration requires a highly sophisticated decentralized decision system.

In the context of the presented work, we propose an intermediate solution in which the description of different tasks and restrictions is possible, but the decision process which may trigger those behaviours is centralized in the ground station machine. There is an improvement on the global autonomy level of the system, since one or more courses of action can be defined in the mission plan without requiring further input from the user after the mission submission. As an example, previously, after sending a UAV to a location, the user had to evaluate in real-time if a relay UAV was necessary to improve the network performance, and then instruct that secondary UAV to takeoff and provide appropriate positions for it to follow. In this iteration, the course of action to take when detecting bad network conditions (for example, assigning a new UAV to the mission which should follow the first UAV) could already be present on the mission plan and be triggered automatically when certain conditions are met - these conditions are evaluated by a centralized control entity.

We propose a mission planning framework that interprets and executes a mission script which defines the possible execution paths that may be followed during the mission. This approach leverages the existing UAV management platform functionalities (sending isolated commands to the UAVs and consulting telemetry and sensor data), and provides an API through which it is possible to interact with the UAV fleet during a mission. We supply a general purpose UAV management platform and mission planning framework that may be used to cover a wide range of highly specific multi-UAV scenarios.

3.1 Mission Planning Requirements

At its core, a UAV mission can be seen as a set of locations that one or more UAVs have to traverse or be placed. However, a random set of waypoints does not fully constitute a mission, since there should be a goal to be achieved - for example, we may want to monitor a forest area, and the UAV will traverse intermediate points that allow full coverage, or a UAV may attempt to trace the perimeter of a fire and will move according to the temperature readings at the previous waypoint. At the lowest complexity level, a mission planning framework has to allow the definition of *where* the UAV should be placed and *how* it should move to reach that location. Complexity arises from the decision process of moving the UAVs to achieve the mission's goals:

- *Which* UAV should be used? Is any particular sensor required in order to perform this mission? Are multiple UAVs required?
- *When* should the UAV move to a new position? Does it need to wait for another UAV to finish a previous task or for a sensor to send new data?
- *Why* should the UAV move to a different position? Is it too close to another UAV? Does a sensor reading indicate that the UAV is in a dangerous situation or that it is currently in a point of interest to explore?

As such, we will formulate concrete requirements that have to be addressed by the mission framework, given that it should support contrasting levels of mission complexity and provide tools for the user to declare the decision process during the mission.

3.1.1 Commands and Telemetry

The base feature that has to be implemented in the mission support framework is the integration of the features developed in the base platform, which includes sending commands to a UAV and accessing telemetry data. Sending a sequence of commands to a UAV and obtaining the UAV's current location has to be a trivial task for the user, as those are the base actions upon which more complex behaviour is built.

3.1.2 UAV Assignment and Revocation

It should be possible to assign a UAV to a running mission at any time - although in many scenarios there will be at least one UAV assigned when a mission starts, it may also be relevant to dispatch additional UAVs afterwards. Similarly, it should be possible to remove a UAV from the mission if it is no longer required, allowing it to be allocated to another mission. The user should be able to have some control over the drones that are assigned to the mission, whether by indicating the specific drone's reference or by providing a set of properties, such as having a sensor or being closer to a certain location. However, the user should also be able to leave that decision entirely to the system, if it is not relevant for the use case.

3.1.3 Multi-UAV and Synchronization

Another important aspect is the support of multi-UAV missions and the coordination/synchronization strategies to enable those. The user should be able to, independently, control several UAVs in a single mission, but also coordinate tasks that require multiple UAVs or that have sequential steps distributed among different UAVs that must be synchronized.

3.1.4 Multiple Execution Paths

As previously mentioned, it should be possible to describe a mission that may follow different execution paths. This means that running the same mission script could yield different results according to the context in which it is being executed. For example, a secondary UAV could be summoned to replace another one in response to low battery. This can be achieved by allowing decision-making statements (if-then-else, switch), looping statements (for, while), and branching statements (break, continue, return) to be part of the scripts.

3.1.5 Sensor Support

One factor that may influence the execution path of a mission is a sensor reading; for example, if the temperature is too high, the UAV's path could be diverted to avoid damage. The framework should allow reading sensor data from within the mission context.

3.1.6 Mission Constraints

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

3.2 Proposal

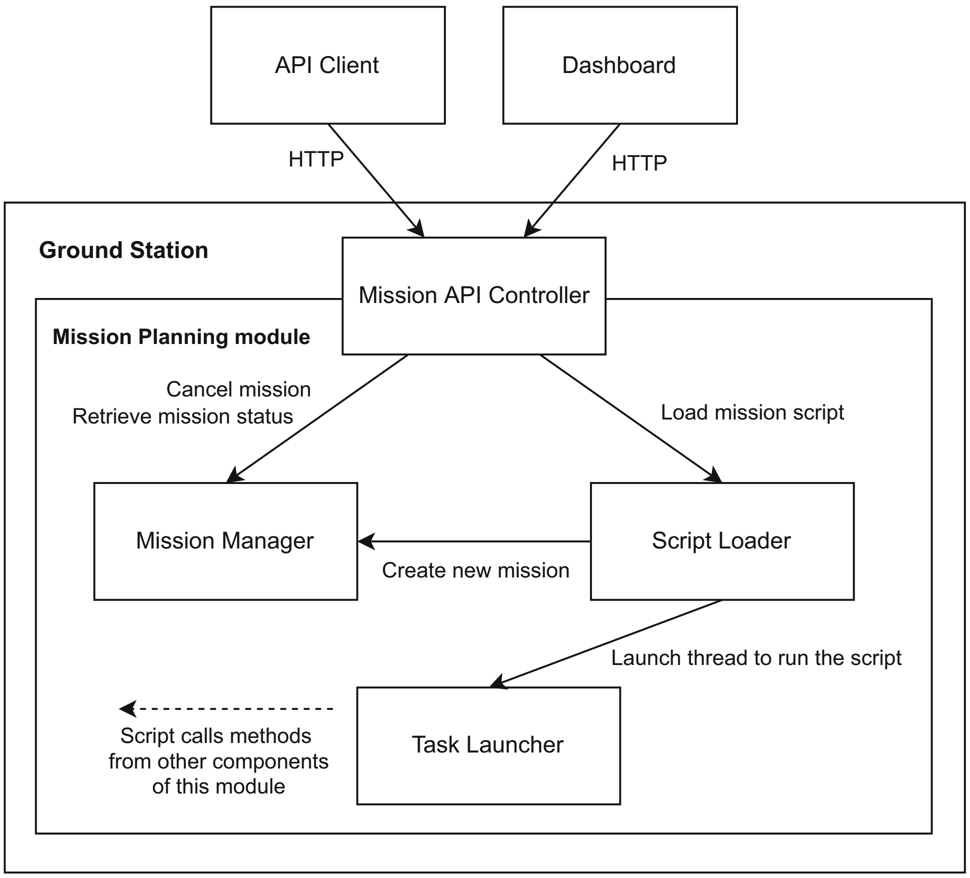
We propose a mission planning framework that lets the user define a mission plan through a script written in a DSL, and that can be integrated into the existing UAV control platform. A DSL provides a fair trade-off between ease of use and adaptability when it comes to designing a mission plan. A GUI could be more intuitive to use, but it is often restrictive on the options that are offered for mission plans that are not linear, and when presenting a more extensive set of tools to fill those needs, it loses the ease of use. These are suitable for presenting higher level tasks, but may not be enough to design mission plans that have several constraints or that may require different courses of action according to the real-time conditions. On the other end, it is possible to develop a solution that does not depend on a GUI, but instead relies on code definition of the mission plan. However, it may be counter-intuitive to leave most of the control to the user, requiring the user to write a fair amount of boilerplate code in order to integrate with the mission planning system. With EasyMission, we aim to fill the gap between these two types of solutions: the user is left with a language that is targeted to this specific use case - which is simpler to read and write - with much of the underlying control logic already handled. This also eases the bridge to building a GUI that can generate a mission plan in this DSL.

As already mentioned, the groundwork for this framework was already established, having developed a working UAV control platform. The global architecture is shown in Fig. 1, and we will focus on the mission planning components, which are the ones developed for providing mission planning capabilities. The components of the mission planning module are depicted in Figs. 2 and 3.

In Fig. 2, we portray the interaction between the user and the system in order to start a mission or retrieve its status. Through a REST API client or the existing dashboard, the user may submit a file containing the mission script. The Mission API controller provides this content to the script loader, which loads all required context, notifies the mission manager of the new mission, and instructs the Task Launcher to launch a thread which will run the mission script. The script can invoke methods that are exposed by other mission planning modules, such as a takeoff method. Further requests may be sent to the Mission API Controller to retrieve the mission status or cancel a running mission.

Figure 3 depicts the internal architecture of the mission planning module, as well as other entities that were already part of the base platform. This diagram excludes the API

Fig. 2 Mission creation and management



controller modules and the *Script Loader*, since those are not required internally after the mission submission.

The components presented in this diagram are colored according to the modules they belong to in the general architecture.

The communication modules, colored in yellow, are those responsible for communicating with the drones, either by publishing or subscribing message topics, and were already present in the original architecture in order to retrieve telemetry data and send commands.

The code management modules, colored in green, were already implemented for fleet management - keeping track of the drone and sensor data.

The purple and blue modules, in conjunction with the *Script Loader*, are the new development of this work which provide the mission planning features. The mission planning modules interact with the already existent modules, effectively implementing the planning features. The mission planning client interfaces expose methods from the mission planning modules that may be invoked through the mission script. These components provide a layer of abstraction, exposing methods to be used specifically during a mission in a predetermined way, with the internal logic protected by the other mission planning modules.

As an example, this is the flow that happens when providing a mission script with a takeoff command:

- The mission script file is submitted and received by the Mission API Controller.
- The Script Loader launches a thread which will run the mission script.
- At some point, the provided script will invoke the `takeoff` method. This method is exposed by the *Command Parser*, which will perform some validations: can this command be executed by this drone at the moment? If the takeoff altitude is not defined, what value should be used?
- The request is then redirected to the *Command Handler*, which builds the corresponding command message in the right format.
- The message is sent through the *Message Publisher*, which publishes it to other ROS2 nodes.
- The *Command Handler* will request *Synchronization Channel* to wait for the completion of the command, and the thread's execution will be halted until a message is received with the command status.

Many steps had to be taken to ensure the proper execution of the command, but the only method that was exposed at the mission layer is `takeoff`.

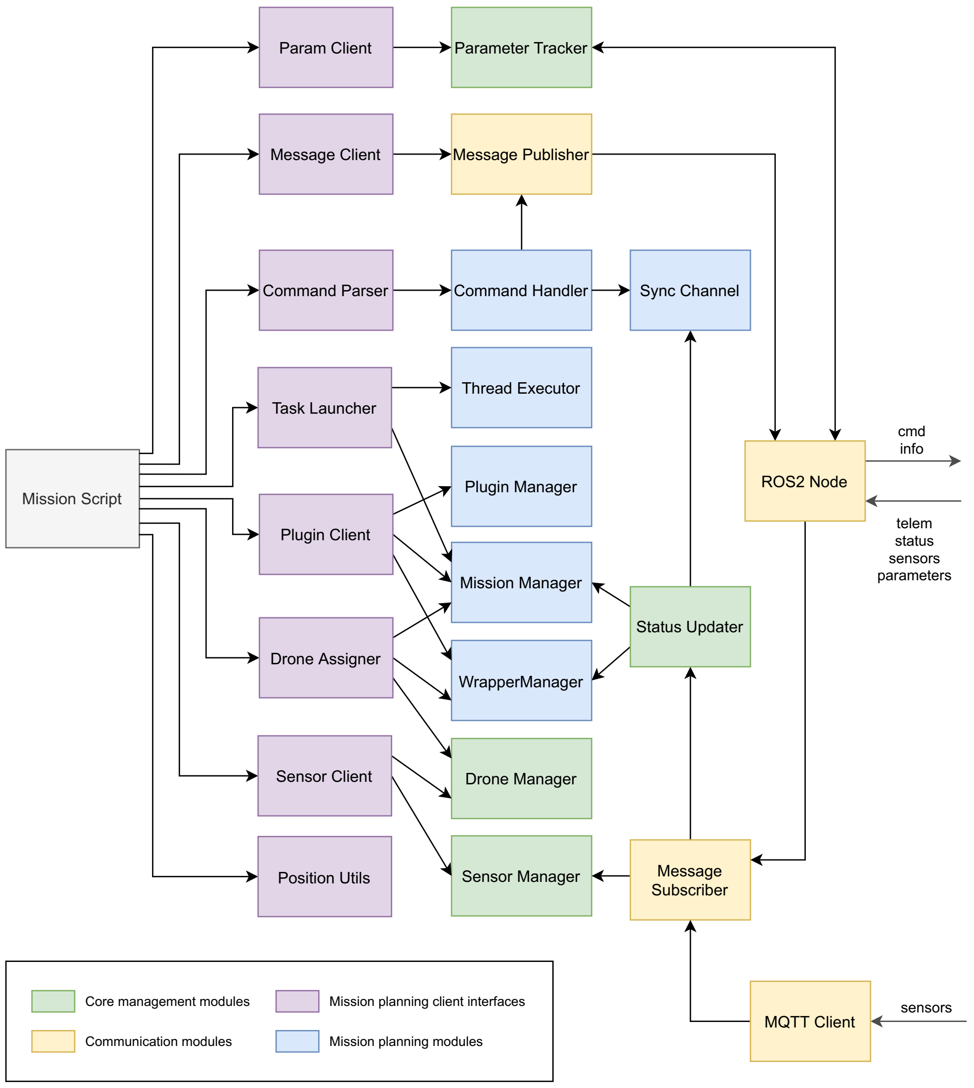


Fig. 3 Multi-UAV mission control platform architecture

Following is the description of the responsibilities of the mission planning components:

- **Script Loader** - Receives a mission file, notifies the *Mission Manager* of the new mission, and launches the thread that runs the main mission script.
- **Mission Manager** - Registers starting missions, keeps track of the mission status, and requests the termination of threads started by the mission when it concludes with an error.
- **Drone Assigner** - Assigns UAVs to a mission according to the requested requirements, and alternatively revokes a UAV or replaces it with another one if that is solicited.
- **Wrapper Manager** - Builds UAV data wrapper models, containing data from currently active UAVs, which includes telemetry fields, running commands, remote parameters, and sensor data. Each UAV's most recent data can be retrieved in the corresponding mission at anytime.
- **Command Parser** - Parses and validates the sequences of commands present in a mission. Through the *Command Handler*, it translates the command request into a message and will signal the *Synchronization Channel* that it wants to wait for the completion of this command. The thread in which the current command is being executed will wait until it is concluded before allowing it to progress to the next action.
- **Synchronization Channel** - Central point for thread synchronization. It provides a mechanism to halt execution until a command is concluded or a new telemetry or sensor message is received.
- **Task Launcher** - Allows the user to start a task in a separate thread from within a mission, which enables the concurrent execution of multiple commands (each handled in a different thread). It also enables consulting task status or waiting for completion.
- **Thread Executor** - Keeps track of the threads for running missions and manages their lifecycle. When a

new task is created, the thread executor will launch a new thread to run it, and when a mission finishes, either successfully or with an error, it will stop all pending threads related to that mission.

- **Plugin Manager** - Mission plugins will allow invoking common behaviour during a mission, without the need of repeating the same code in multiple missions. This component will load and validate mission plugins, and will keep track of the running plugins for each mission. A mission script can enable or disable a plugin through the *Plugin Client*.
- **Parameter Client** - Retrieves the current value of a UAV's parameter, or sets it to a new value. These parameters are relative to the UAV's remote ROS2 node, running in the UAV's companion computer.
- **Sensor Client** - Retrieves the most recent values of a sensor, whether it is or is not attached to a particular UAV.
- **Message client** - Sends any string message through ROS2 to notify external entities (such as a sensor or an auxiliary UAV module) of an event.
- **Position Utils** - Provides methods for some position-related calculations during a mission, such as distance or bearing between coordinates.

Previously in Section 3.1, we defined the requirements of the mission planning framework. After introducing our proposed architecture, we are able to trace how these components address each of the requirements:

- Command handling is directly supported by the *Command Parser* and *Command Handler*, while telemetry data can be retrieved during a mission through the fields provided by the *Wrapper Manager*.
- The *Drone Assigner* assigns and revokes UAVs.
- In order to have multiple UAVs to execute commands simultaneously, the *Task Launcher* provides the mechanisms for launching several simultaneous tasks.
- Multiple execution paths are not explicitly supported by any single component but by the DSL itself, which should include the mentioned control structures. This is extended by telemetry and sensor data retrieval, allowing decisions to be based on the current state of the environment.
- Sensor data can be read in a mission script through the usage of the *Sensor Client* methods.
- Error situations, such as low battery level or bad sensor calibration, which were already automatically detected, can now be used by the *Mission Manager* in order to cancel any mission that is currently using a faulty UAV. More specific mission constraints can be described using the DSL, and stored as a mission plugin which could be enabled during a mission.

3.3 Resulting DSL

EasyMission, the DSL that resulted from this architecture, allows an expressive declaration of mission steps. Algorithm 1 demonstrates one possible way to write a mission script that instructs a drone to arm, takeoff, and follow the path of a 20 meters wide square before landing.

```
drone = assign any

arm drone
takeoff drone, 5.meters
4.times {
    move drone, forward: 20.meters,
     ↪  speed:5.m/s
    turn drone, 90.deg
}
land drone
```

Algorithm 1 Following the path of a square.

4 Mission Framework and Description Language

The literature [8–10] contained some examples of mission description languages for robot systems. Although the presented languages were human readable, they were also verbose, not as straightforward to define by an inexperienced user, and lacked mechanisms for adjusting the course of the mission in response to real time events. Using JSON or XML as the base for the description language, as seen on the previous examples, eases the parsing of the instructions since several libraries exist for that purpose. However, using those formats restricts the fluidity of the language and makes it difficult to express different possible paths of execution within the same mission.

Another possible alternative, since part of our system already uses ROS, would be to rely on the ROS API and using a client library such as the one for the Python language. Since the drone commands and telemetry are already exchanged through ROS messages, with this approach it would be possible to use a Python script as a mission plan by simply reading and sending those messages. This would be simpler and faster to start developing for those already familiar with the ROS ecosystem, which is widely used in this field. We chose to restrict ROS usage to communication between the drones and ground station systems, which enable "lower level" functionality, such as requesting commands and sharing telemetry data. Mission planning is at a higher level, which we preferred to keep agnostic to ROS. With a DSL, anyone can easily type a mission without understanding anything of what is going in the "backstage" - the same cannot be said about a ROS based API, which would require some knowledge about ROS and not only about the context at hand. If at a later stage, it is concluded that ROS no longer meets our needs, we can use a different technology and that will not require any changes on the mission planning framework, provided that we keep the same message format.

Writing a custom language and corresponding interpreter from scratch would require manually implementing basic features and integrating with the existing platform. Using an existing scripting language and building a custom DSL through it is a more efficient approach, leveraging what the base language offers and extending it with the necessary features. With an embedded DSL, we can use the underlying language to provide basic syntax and semantics, such as variable declaration, conditional expressions, and loop constructs.

The language we selected to build the DSL is the Apache Groovy programming language. Many features present in Groovy make it easy to hide complex logic behind a more comprehensible and domain-specific API - the *Groovy in action* document [14] details how to write a custom DSL using Groovy, with the most relevant features it provides being the following:

- **Command chaining** - Groovy allows omitting parentheses around the arguments of a method call and chaining consecutive calls, which means `turn(drone).by(45)` can also be written as `turn drone by 45`.
- **Named arguments** - The arguments of a method may be named instead of relying on the order in which they are provided, which makes it more readable. Instead of writing `move drone to 40.6, 8.6, 10, 5`, we can opt for a named approach of `move drone to lat: 40.6, lon: 8.6, alt: 10, speed: 5`.
- **Categories** - We may extend the functionality of a class by implementing a *category*, which allows creating additional methods for that class without modifying existing behaviour, while also limiting the introduced changes to the scope in which it is applied. This feature can be useful to add new methods to numbers, which enables a statement such as `10.centimeters`, in which we are able to easily declare which unit we are using and convert to different units.
- **Binding and delegation** - Variables can be created outside of a Groovy script and passed into it through a *Binding* when it is loaded. This can be used to pass initialized instances of classes to be referenced from within the user-provided script without explicitly declaring those variables. It is also possible do delegate all public methods of a class instance to a script, allowing those methods to be directly called through a groovy script without specifying the instance. As an example, a script would not require a *Command Parser* instance to be initialized followed by arming the drone as `commandParser.arm(drone)`; instead, directly invoking `arm(drone)` would have the same effect given that a *Command Parser* instance is passed to the script binding and its methods are delegated to the script.

The usage of these features improves the expressiveness of the DSL, making it easier to understand, even by inexperienced users. By passing instances of the mission planning client interface classes to the mission script binding and delegating its methods, we can access these new functionalities through the mission script in a clear way. All of these methods are handled internally, meaning that the user only has to provide a script containing mission logic. Combining this with the existing control flow structures enables the user to design missions with diverse complexity levels, with a layer of abstraction that hides the UAV control and communication logic.

4.1 Describing Mission Logic

In order to better demonstrate how it is possible to write a mission using EasyMission, we will present how to declare some base actions and simple examples.

4.1.1 UAV Assignment and Revocation

A UAV/drone has to be assigned to a mission before it can be controlled. The drone can be picked according to its drone ID, the available sensors or its proximity to a coordinate. It is also possible to request a drone without specifying any requirements.

```
drone1 = assign 'drone01'
drone2 = assign temperature
drone3 = assign lat: 40.6339,
lon: -8.6605
drone4 = assign any
```

If a drone meeting the requirements is currently available, it will be assigned to the mission, and the `drone` variable will reference a drone wrapper object. This object can be used subsequently to send commands to the drone or monitor telemetry data.

Besides handling drone assignment, the *Drone Assigner* can also be used to replace and revoke a drone from a mission. Once the drone is revoked, it is no longer possible to send a command or read its telemetry within the mission, unless it is reassigned. Both tasks can be executed by calling either `replace drone` or `revoke drone`, given that `drone` is the variable the drone was assigned to.

4.1.2 Reading Telemetry and Sensor Data

With the previously developed platform, we can retrieve the most recent telemetry data for all available drones. Not all telemetry data is relevant in the context of a mission. To better suit the mission scope, we created a drone wrapper model that contains the most important values, as well as sensor readings from available sensors. The data can be easily accessed during a mission by accessing the attribute in the corresponding drone's object, such as `drone.id`, `drone.cmd`, or `drone.position.alt`. The *Wrapper Manager* manages this data, updating the values after a new telemetry message is received for the drone. If the drone is revoked from the mission before it ends, these values will no longer be accessible - the only available data is from drones within the mission scope.

The sensor readings are also updated when the sensor sends a new value. These can be retrieved on a mission script using, for example, `sensor.drone01.temperature`, while `sensor.drone01` would simply return the latest values of all sensors connected to the drone with that ID.

During a mission, the user may want to wait for a new telemetry or sensor message before evaluating the next step. Through the `wait` command, it is possible to do so. This will prevent the current thread from progressing until a new message is received and the values are updated.

```
wait drone: drone1
wait sensor: [type: 'temp', drone:
'drone01']
```

4.1.3 Command Execution

The most crucial feature of the framework is the ability to send commands to the drone. All commands follow a similar format, comprised of `[command] [droneId] [options]`, with the main difference being the possible additional parameters and how those can be provided. If the command is provided with non-existent parameters or parameters with an invalid value, the mission will fail; otherwise it is then sent to the drone. If the drone-side execution of the command fails, the mission also fails.

A simple mission script will include the assignment of a drone followed by a sequence of commands that should be executed consecutively. However, if the script execution progressed immediately after sending the first command without waiting for confirmation that it successfully finished, it would be cancelled by the following command, which would be immediately sent. To avoid requiring to manually polling if the command has completed, command methods are synchronous: each command method call results in halting the mission's thread execution until the ground station is notified that the command has concluded. After sending the command, the *Command Parser* will lock the current thread until it is notified that the command has finished, which happens after the *Status Updater* receives the corresponding status message from the drone.

The `arm`, `disarm`, `land`, and the `cancel` commands do not take additional arguments. As such, they can be executed by simply calling them in the `[command] [droneId]` format, such as:

```
drone = assign any
      arm drone
      disarm drone
```

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

```
  takeoff drone, 10.meters
          home drone
```

The `turn` command can be used to rotate the drone, either by a certain angle or to face a certain direction, drone, or coordinate. This means that turning the drone *by* x degrees will rotate the yaw angle by that amount, while turning it *to* x degrees will rotate it to that angle, relative to North.

```
turn drone1 by 90.deg
turn drone1 to 90.deg
turn drone1 to drone2
```

In order to instruct the drone to move to a certain position, the `move` command is available. The `move` command can either define a latitude/longitude coordinate or a location relative to the current position. The latitude and longitude are represented by the `lat` and `lon` arguments, while relative coordinates can be defined using `up, down, left, right, forward, backward`. The `alt`, `yaw` and `speed` parameters may be provided but are optional. However, to improve the readability, these commands can be issued in an alternate form. GPS coordinates can be provided after a "to" keyword. When sending a command with coordinates relative to the current position, the first direction has to be followed by the keyword "by" (such as "down by:"), while the remaining ones do not require the "by".

```
move drone1 at 5./s to lat: 40.6342,
lon:
 ↪  -8.6614, alt: 17.4.meters
move drone1 at 10.m/s forward by: 5.m,
right:
 ↪  10.m
```

The drone-side movement can be interpreted either as GPS coordinates or by North-East-Down (NED) coordinates, as this is what the library that we used for drone control (MAVSDK) exposed. When the drone receives a move command with GPS coordinate, it will move to that GPS coordinate. The drone can be configured to execute commands specified in relative coordinates differently, it will either calculate the GPS coordinates or the NED coordinates of the target position.

4.1.4 Controlling Multiple Drones

As previously explained, a running command will halt the thread progress until it has completed. A consequence of this behaviour is that, if the script contains two consecutive commands targeting different drones, the second drone's command will only start after the first drone's command has finished. This restriction is not appropriate for multi-drone missions, as sending commands to multiple drones in parallel may be a requirement.

To solve this, we have to allow the user to define different tasks that can be executed concurrently. A task is a set of instructions, such as those that have already been described previously, but that are designated to be executed in a separate thread instead of running in the main loop. Any failure that happens during its execution will also lead to the failure of the mission. After launching a task, it is possible to stop it, wait for its conclusion, or verify if it is still running.

```
task1 = run  takeoff drone1
task2 = run  takeoff drone2
if (task1.running)
     stop task2
wait task1
```

When the user requests for a task to be launched during a mission, the *Task Launcher* will generate a task ID and send it to the *Thread Executor*, which will assign a thread to execute it. The task ID is returned when it is requested to run the task, which can be used later for further operations.

4.2 Mission Examples

This section presents examples demonstrating how to write a coherent mission script compatible with EasyMission.

4.2.1 Mapping Mission

A task that is commonly useful is the mapping of an area. Algorithm 2 presents an algorithm that instructs the drone to traverse the area to the right and ahead of itself, according to the initial heading after takeoff.

The `map_area()` method is responsible for the movement logic. In this example, the method call contains parentheses around the arguments, as it is valid, but it is not

```
drone = assign any
arm drone
takeoff drone
map_area(drone, 100.m, 80.m, 2, 4)
land drone1


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

Algorithm 2 Mapping.

a requirement. Due to Groovy's command chaining, those can be omitted when it does not cause ambiguity, which is the case of the remaining examples.

Groovy provides different looping strategies out of the box, which are made possible by extending the *Integer* class with other methods. One example is the `times` method, which executes a code block the number of times upon which it is called. This type of code blocks in Groovy is called Closure, which is an anonymous code block which receives arguments, returns values and may be assigned to a variable. Closures follow the syntax of { `argument -> statements`}. In this script, since the provided value for `steps_y` was 4, the closure that follows will be executed 5 times, and the `traversal` variable will iterate over the numbers 0 to 4, representing the number of the current traversal of the area width.

Figure 4 depicts the resulting path after concluding this mission. The drone covered an area of 100 by 80 meters, with the next coordinate being calculated in real time, after reaching the previous waypoint.

4.2.2 Drone Replacement

During mission execution, it is possible that the drone reaches a low battery level before its conclusion. One simple approach to this problem is to verify the drone's battery level before executing a command, and replacing it with a different drone if it is too low.

Adapting the previous example, we have Algorithm 3 featuring a `move_or_replace` method which is invoked instead of calling the `move` command directly. If the battery is lower than 40% before instructing the drone to move, it is

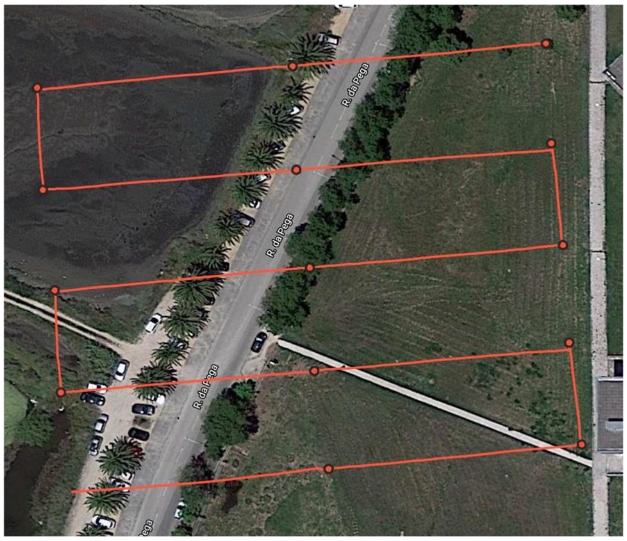


Fig. 4 Drone path of the mapping mission

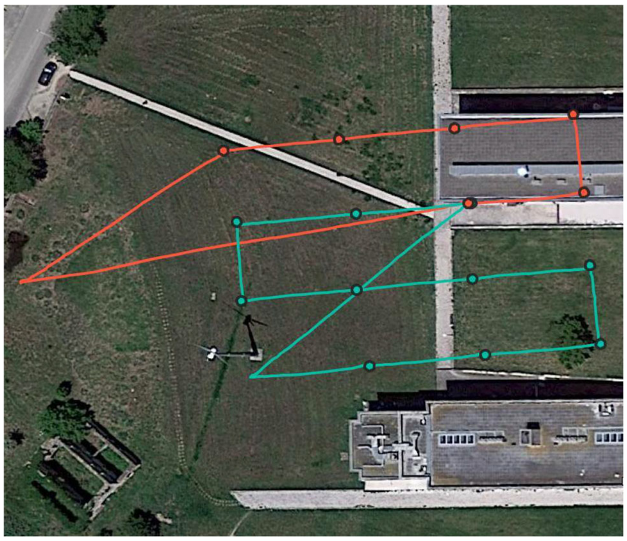


Fig. 5 Drone path of the replacement mission

```
...
    move_or_replace right: traversal.even ?
     ↪  step_sz_y : -step_sz_y
...
    move_or_replace forward: step_sz_x
...
def move_or_replace(movement) {
   if (drone.battery < 0.4) {
      def curr_pos = [lat: drone.position.lat,
                      lon: drone.position.lon,
                      alt: drone.position.alt,
                      yaw: drone.heading]
      run {home drone}
      revoke drone
      drone = assign any
      arm drone
      takeoff drone
      move drone, curr_pos
   }
   move drone, movement
}
```

Algorithm 3 Replacement during mapping.

revoked from the mission, and another one is assigned. This could be achieved by running `replace drone`, but for demonstrative purposes, each intermediate step was written for this mission.

Figure 5 shows the path of both drones during the mission, the green path belonging to the drone that started the mission, and the red path corresponding to the drone that executed the replacement and concluded the mission. The secondary drone continued the mission where the previous drone stopped.

4.3 Extending and Reusing Mission Functionalities

Most of the mission framework requirements have been addressed so far, but the framework still lacks support for mission constraints. Those can be solved in the mission script itself, such as the drone replacement in Algorithm 3; however, it is not an ideal solution. One consequence is that it demands the repetition of the same algorithms through different missions scripts. Besides, an inexperienced user may need to solve a constraint, such as keeping network connectivity throughout the mission, but the required algorithm could be too complex to be implemented.

To reduce the amount of code that is repeated across multiple missions and to allow verifying certain conditions in the background, we introduced the concept of mission plugins. Plugins are mission scripts that can be enabled from within other mission scripts to provide additional functionality.

Several types of plugins can be implemented: (1) plugins that expose methods, such as mapping, to be used during the mission; (2) plugins that monitor each drone's status and act upon it; (3) plugins that transform a command into subcommands (such as to implement obstacle avoidance); (4) plugins that re-transmit telemetry and drone status to external components in different formats; (5) plugins that process and transform sensor data, and many other functionalities.

At the current state of the platform, monitoring plugins have been implemented, which allow demonstrating how to integrate plugins into the framework, and at the same time, support mission constraints. Monitoring plugins will analyse each drone's data after a telemetry update, which may lead to additional actions, such as assigning another drone to the mission or revoking the current drone.

In order to communicate how the user should launch the plugin, such as what parameters have to be present, each plugin file should provide a plugin configuration. These are the available parameters:

- **id** - plugin ID used to identify the plugin;
- **input** - some plugins may require that the user provides input parameters; if so, the parameter data type has to be declared, as well as any default value if applicable;
- **vars** - initialises internal variables, required by the plugin, that do not have to be provided by the user;
- **callback** - in the case of a monitoring plugin, the callback indicates the closure that is called upon receiving a new telemetry message for a drone;
- **init** - closure to be called once at startup when a mission enables this plugin, aimed at initializing internal vars according to the received input.

4.3.1 Replacement Plugin

In the previous mission example, drone replacement was conducted by checking at each waypoint if the drone still had enough battery to proceed. A better approach is to monitor the drone's battery level as a background task, interrupting the drone if it is decided that it should be replaced. Algorithm 4 demonstrates how such a script could be developed, although with a naive approach in regards to the necessary battery calculation. A file with the contents of this script can be placed on the ground station's plugin directory, which will load it when starting the software. This plugin can be later enabled on any mission script, as in the example provided in Algorithm 5, which demonstrates how to enable the drone replacement plugin during a regular mission.

Since this mission enabled a plugin of type *monitor*, the closure defined in the `main` variable in the plugin will be called for every drone from this mission when it receives a telemetry update.

4.4 Mission Extensions and Considerations

Plugins were developed as a starting point for implementing safety constraints in missions. The replacement plugin is one such constraint, but many more could be useful to provide out of the box - such as collision avoidance. However, this only accounts for conditions that may unravel at run time, leaving it up to the user to validate if the written mission plan is coherent. For example, a user could write a valid mission plan that moves the drone forward until a sensor alerts that there is an obstacle 1 meter ahead. This mission plan could theoretically be endless and there would be no way to predict this beforehand.

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

Algorithm 4 Replacement plugin.

```
enable 'replacement', min_battery: 0.3
drone01 = assign any
arm drone01
takeoff drone01
move drone01 forward by: 200.m
home drone01
```

Algorithm 5 Drone replacement using plugin.

A path that was not explored is the pre-processing of a mission plan. The mission script is only interpreted by Groovy at the moment the Script Loader launches a thread to run it. There could be a previous step in which a different context was used to execute the mission (without actually running any command), opening the possibility to an early error detection.

4.5 Viability in Other Contexts

This work was developed solely with UAVs in mind, and those are the use cases we target. However, our mission planning framework could be applied to different contexts, such as Unmanned Water Vehicles or ground robots. This could be achieved by creating an appropriate robot-side module and introducing a few changes on the fleet management side.

An equivalent module to the drone module would have to be developed, but compatible with that specific robot. This would mean retrieving the appropriate telemetry data and exposing the available commands, as well as implementing how those commands are issued. Our implementation of the drone module makes use of the MAVSDK library, which provides an API for MAVLink; robots which are not UAVs would need an equivalent library. As long as the message format is the same as the one currently in use, it would be easily integrated with the ground station system.

On the ground side, equivalent changes would need to be applied in order to be aware of the new commands and telemetry data fields. Some functionality would have to be locked according to the vehicle type (it would not make sense for a ground robot to takeoff, for example).

5 Use Cases and Evaluation

To validate that this mission planning framework can be useful to address concrete problems, we chose three different use cases. Since these use cases address unrelated issues, we can use those to demonstrate how versatile the platform is, and to prove that it is also possible to declare missions with a moderate level of complexity.

The experimental setup used for evaluating the performance of the missions is delineated and the results of these experiments are presented.

5.1 Use Cases

5.1.1 Fire Perimeter Tracing

Forest fires are usually monitored by humans, which may take some time to be identified when occurring in a secluded area. UAVs could be dispatched to monitor remote forest areas that are at risk of a fire, equipped with temperature sensors to detect any anomalies. Once a certain temperature threshold is surpassed, the UAV should stop traversing the area and start tracing the perimeter of the potential fire. This data could be useful for firefighters to assess the fire dimensions and the required resources to extinguish it.

We designed a mission script in which a drone, equipped with a temperature sensor, maps a given area. Simultaneously, the sensor values are monitored, and as soon as they reach a predefined value which could represent danger, the drone will begin to trace the area around that temperature level. The perimeter tracking algorithm uses (PID) control [15].

5.1.2 UAV Replacement

During the execution of a regular mission, it may occur that the UAV's battery reaches a critical level in which it would not be able to complete its task and return to the home position safely. This may happen either because the UAV started the mission without enough battery for the mission, due to a faulty battery, adverse wind conditions required more effort to move the UAV causing the battery to drain faster, or the mission may take too long to complete without a battery replacement. In some cases, instead of restarting the mission, it may be feasible to simply replace the current UAV with another UAV with a higher battery level, which will simply takeover the mission at the point at which it stopped.

When the user assesses that the mission could be safely taken over by a different UAV during execution, they can enable the replacement plugin previously depicted in Algorithm 4, as demonstrated on Algorithm 5.

5.1.3 Network Relay

The scope of some missions may require a UAV to move to a location that is far from the ground station computer. Under traditional circumstances, it would no longer be possible to monitor the UAV's behaviour once it got out of reach. This is not ideal, since it would not be even possible to retrieve the UAV remotely in case of an emergency. A solution is to issue a chain of UAVs to follow the main UAV during the mission, forming a bridge of UAVs with the purpose of acting as a relay for the network, effectively maintaining connectivity throughout the mission.

To address this, and since this is a common mission constraint that has to be solved, a relay mission plugin can be created. Given the groundstation coordinates, it should monitor the telemetry data of each mission UAV and, according to where it is moving, a relay UAV will be dispatched to prevent loss of connectivity. Since the relay UAVs are assigned to the mission, this process is recursive. Once the relay UAV is ready, it should follow the UAV ahead of it and position itself accordingly.

5.2 Experimental Setup

For each of the use cases, one or more mission scripts are written using this mission planning framework. Some of the examples can be tested in a real environment, while others are only possible to experiment in a simulated setting.

Since ROS2 is used for sending telemetry updates, the rosbag package, the official tool for storing ROS data, is used to record data during the missions for later analysis.

5.2.1 Simulation Setup

During simulated experiments, the computer that hosted the ground station software was simultaneously running instances of the jMAVSim simulator and of the drone control software.

Fire perimeter tracing cannot be properly tested in a real setting; as such, the results are only demonstrated in simulation. A simple simulated temperature sensor was developed for this purpose, which publishes the current

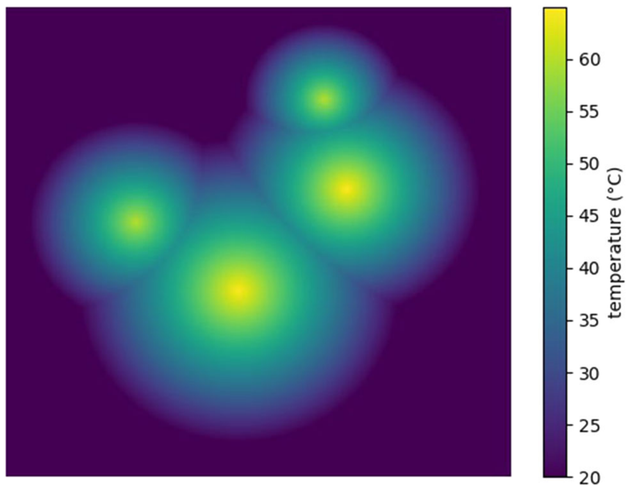


Fig. 6 Topology of simulated heat sources

temperature reading according to the drone's position and a user-provided heat topology configuration.

The chosen topology and heatmap for these experiments are both depicted in Fig. 6, which displays a complex shape and temperatures ranging from 20°C and 65°C. In the algorithm, the temperature threshold at which the drone should begin tracing the heat source is fixed at 37°C, with the target temperature level to trace at 40°C.

The relay bridge algorithm can be used in a real setting, but since we are restricted by the number of available drones, we will also test this scenario in simulation to observe how it behaves with a larger fleet. This required launching four instances of the UAV simulator. The maximum distance to maintain between drones and ground station is recommended to be lower than 70 meters, as determined experimentally.

5.2.2 Field Setup

The tests conducted on the field are performed with drones with FPV S550 and DJI Flamewheel F550 Hexacopter frames. An NVIDIA Jetson Nano (SBC) is coupled to the frame of each drone, in which the drone-side software modules were executed. This device is connected through a serial port to the Pixhawk 4 flight controller. A USB Wi-Fi adapter is connected to the SBC, which enables connectivity to the ground station computer and the other drone's SBCs through an ad-hoc network. The ground station machine is a laptop with a Linux distribution, running the ground station software. Similarly to the SBCs, the laptop has a USB Wi-Fi adapter connected to the ad-hoc network.

These experiments have been performed at the campus of the University of Aveiro, Portugal. The mission scripts were prepared and tested on a simulated environment beforehand to prevent possible dangerous situations, such as a drone crashing into a nearby wall. Before starting a mission, the drones were placed in an appropriate initial position, after which the mission script was submitted. Telemetry data was monitored while the mission was running.

The drone replacement mission was modified to be executed in a real scenario. We chose to trigger the replacement with a timeout instead of checking the battery level, because it is not easy to predict how fast it will deplete in a physical drone. In this case, 40 seconds after the initial drone finishes taking off, the second drone will start the replacement and then conclude the mission.

Besides testing the mission performance, we also executed an experiment to retrieve metrics for the relay algorithm. This experiment is relevant to calculate an estimate of the distance that the drone could have from each other or the ground station while still maintaining a reliable network connection. These metrics can be obtained by monitoring the network performance while moving a drone further away from the ground station. We opted for using the iPerf3[2] tool for monitoring the network performance.

The relay bridge is enabled while the main drone traverses predefined waypoints. The target distance to keep between relay drones is chosen based on the reliable distance discovered through the previous experiment.

5.3 Results

After executing the experiments, the drone's recorded data was used to generate several plots and maps for visualization of the mission. We will present and discuss the results concerning each of the experiments.

5.3.1 Fire Tracing Simulation

The heatmap previously presented in Fig. 6 was used by the simulated temperature sensor during this mission. The resulting drone path, after executing this mission, is shown in Fig. 7. The drone is initially mapping the area until the temperature surpassed the threshold, and we can observe that the path roughly resembles the simulated fire topology.

Although there is some accentuated error around the edges between heat sources, it is still possible to discern the approximate perimeter of the fire. This can be a result of the simplified temperature simulation, and not of the perimeter tracing algorithm itself.

Figure 8 contains the temperature of the simulated sensor through time. The registered temperature is also coherent with what is expected from this algorithm, surpassing the limit when the abnormal temperature is first detected, and then occurring a few times after that, at each edge, before it corrects its own path. This behaviour is expected, since the

[2]https://iperf.fr/

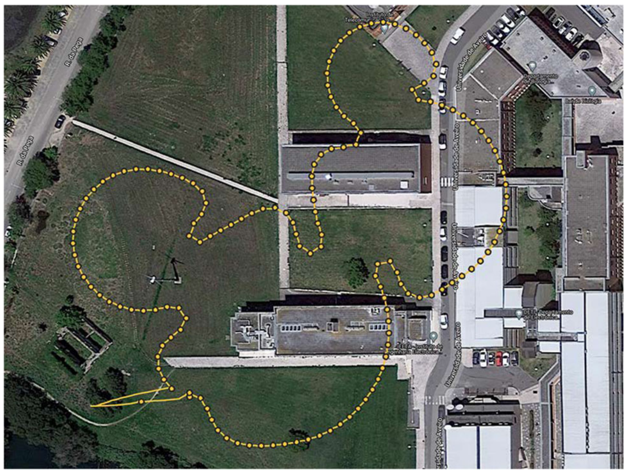


Fig. 7 Drone path during fire tracing

algorithm is based on a PID controller, which attempts to follow a setpoint value (in this case, the defined temperature threshold) and applies constant corrections to the angle of movement in order to keep it. Overall, it is possible to notice that there was an attempt to keep the trajectory close to the 40°C perimeter.

5.3.2 Drone Replacement Experiment

In a real setting, we use two drones to demonstrate the replacement mission, which is supposed to take place approximately 40 seconds after the initial drone finishes taking off. The resulting path of both drones is depicted in Fig. 9. The first drone, portrayed in yellow, returned to the launch position once it reached a waypoint after being on air for 40 seconds. The second drone moves to the position where the previous one stopped and finishes mapping the area successfully.

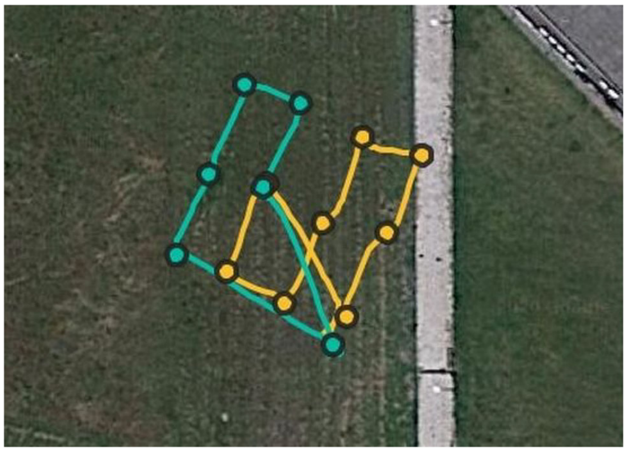


Fig. 9 Path of two drones during mapping mission with replacement

Figure 10 displays the registered altitude of both drones during the mission. By observing the altitude variation, we can notice that the second drone only started taking off after the first one was landing.

5.3.3 Relay Bridge Experiment

In order to conduct the relay bridge experiment, it was necessary to estimate a reasonable distance to maintain between drones at which they were still able to stay connected. We executed a separation-value test to reach a conclusion experimentally.

We analysed network performance while increasing the distance between the drone and the ground station. Figure 11 depicts the registered distance between the drone and the ground station through time, and Fig. 12 depicts the bitrate it was possible to achieve. During some periods, telemetry messages were not received; thus, we implicitly

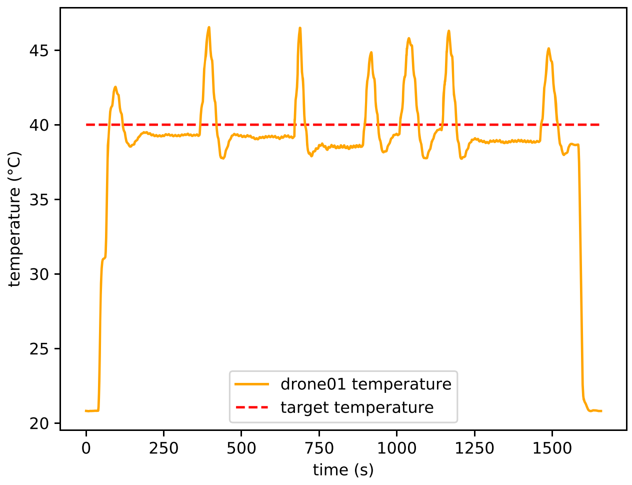


Fig. 8 Sensed temperature

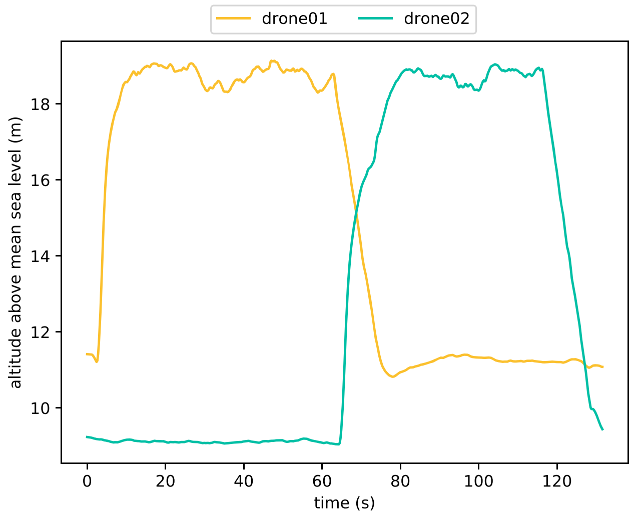


Fig. 10 Drone altitude

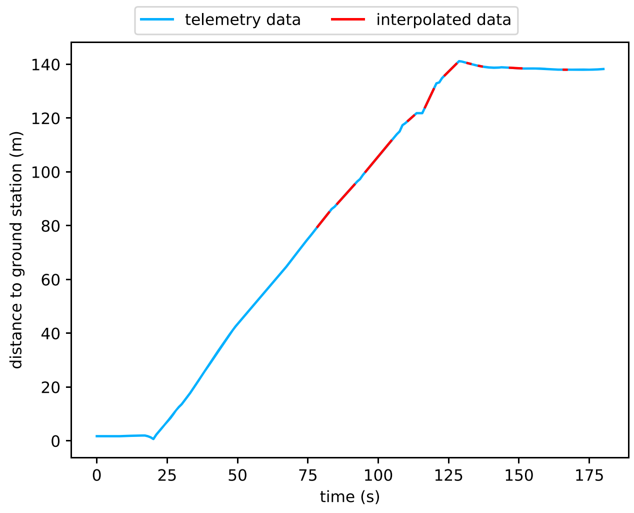


Fig. 11 Distance between the drone and the ground station

determined the distance based on the previous and next available data.

At around 75 seconds, after some instability, the network stopped being able to handle the requested bitrate, as shown in Fig. 12, which happened when the drone was located around 70 meters from the ground station. Since the telemetry data was recorded in the ground station, we can also observe that some ROS messages were lost. Fifty seconds later, although there was an increase in the bitrate, it remained considerably unstable, and ROS messages were still occasionally lost. With this information, we can adjust future missions to be compatible with this limitation. We concluded that the distance between drones in this setting has to be less than 70 meters.

By enabling the relay plugin, we are able to maintain connectivity with the ground station even if the drones move far away from it. To test the algorithm in simulation, a mission script is used in which the main drone is only concerned with mapping a remote area, while the distance to the ground station is continuously monitored by the plugin to dispatch a relay drone when needed.

The traversed path of each drone is shown in Fig. 13, in which the white circle represents the ground station location. The path of the yellow drone shows the mapping algorithm, and it is possible to observe that each relay drone followed the previous one, drawing a path that is roughly similar to the first one's. The placement of the different drones throughout the mission is explained in Fig. 14.

In Fig. 15, we can observe that, although the distance between drones surpassed 50 meters, which was the target distance, it did not reach the 70 meters mark, which was determined by the separation-value test as the maximum distance. The target distance should not be seen as a hard limit that if surpassed would cause the drones to lose connectivity, but as an approximate distance that the drones should keep between each other. Regarding the distance from the ground station, as seen in Fig. 16, it is apparent that

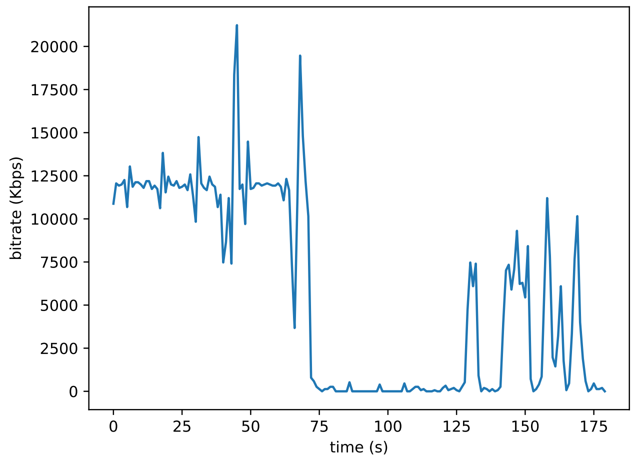


Fig. 12 Achieved bitrate

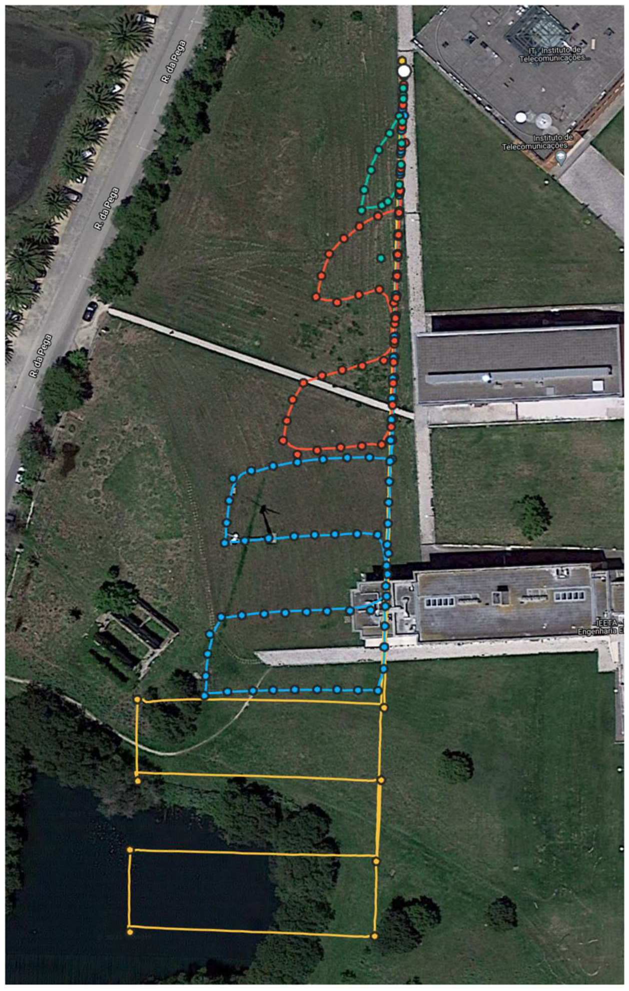
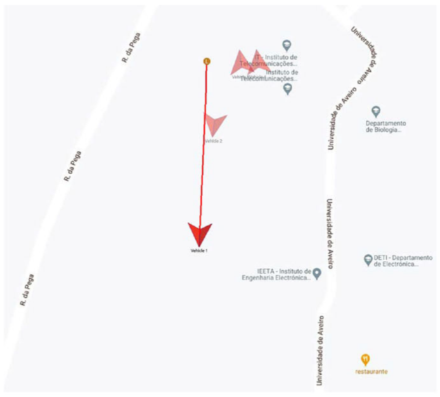
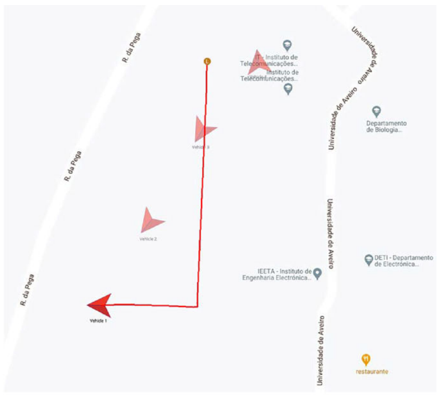
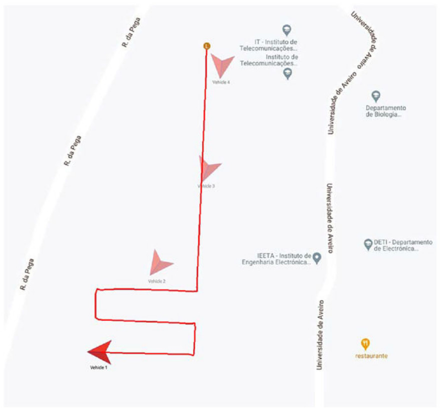

R. da Pega
IT - Instituto de Telecomunicações
Instituto de Telecomunicações
CEETA Engenharia

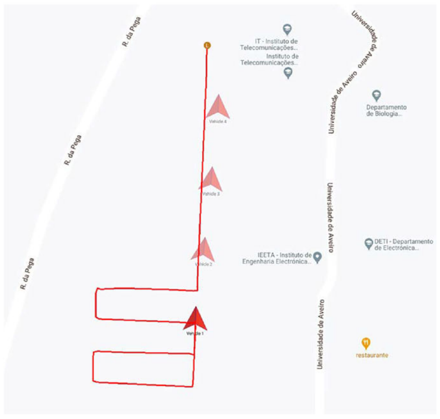
Fig. 13 Path of multiple drones with three relay drones on simulation

Fig. 14 Progress of a simulated mission in which one drone is mapping an area and three drones serve as relay


(a) The main drone is moving and a relay drone is dispatched


(b) The main drone starts mapping the area and another relay drone follows the previous one


(c) The last relay drone is dispatched


(d) The main drone finishes mapping and returns home; the relay drones move towards the ground station

the distance to the relay drones varies proportionally to the main drone's movement.

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

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

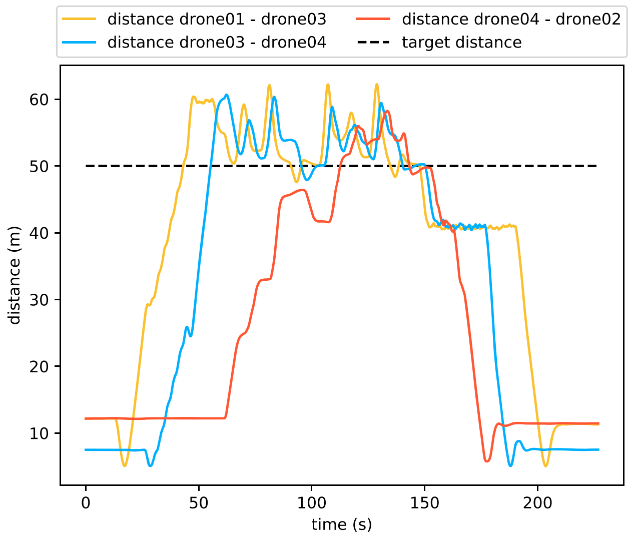


Fig. 15 Distance between drones on simulation

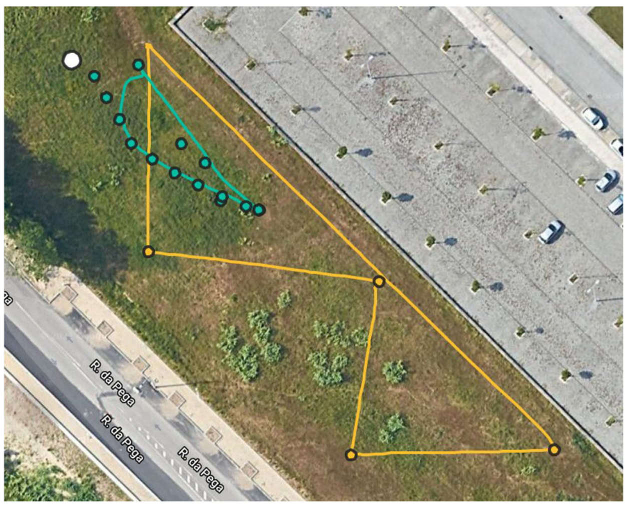


Fig. 17 Mission path in a real setting with one relay drone

Even though some aspects could be refined, this algorithm provides a good starting point towards guaranteeing connectivity in a mission. Overall, this was successful, as we could observe that it was possible to place the relay drones where they could help maintain connectivity.

Finally, we conducted the relay bridge experiments on the field. We increased the target distance a bit in relation to what was used in the simulation, setting it at 60 meters. This enabled us to cover a larger area with less drones, since either two or three were available at the time of the experiments. We started with a relay mission with two drones and progressed to a mission with a bridge of three drones.

For the relay mission with two drones, the path of both drones is depicted in Fig. 17. In this experiment, the yellow path is the one of the main drone, the one performing the task, which travelled to predefined waypoints; the green path is the one of the relay drone.

In Fig. 18, the results show an equivalent behaviour to what we obtained in the previous simulated relay experiment, although with only one relay drone. Overall, the distance between drones was close to the target distance, with some slight fluctuations above that mark, which, given the margin, is very close to the objective.

Although a 3-drones' mission is more complex to evaluate in a real scenario, we also performed this mission experiment. In this mission, the path of the three drones is depicted in Fig. 19. The yellow path is the one of the main drone, the blue path is the one of the second drone, and the green path is the one of the drone connected to the ground station.

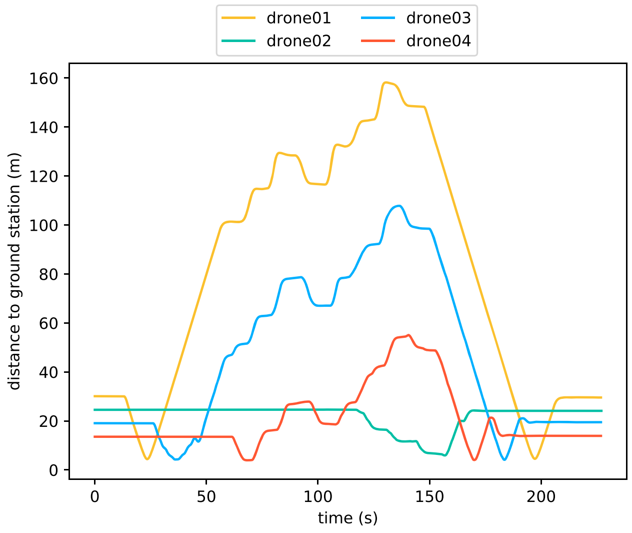


Fig. 16 Distance between drones and ground station on simulation

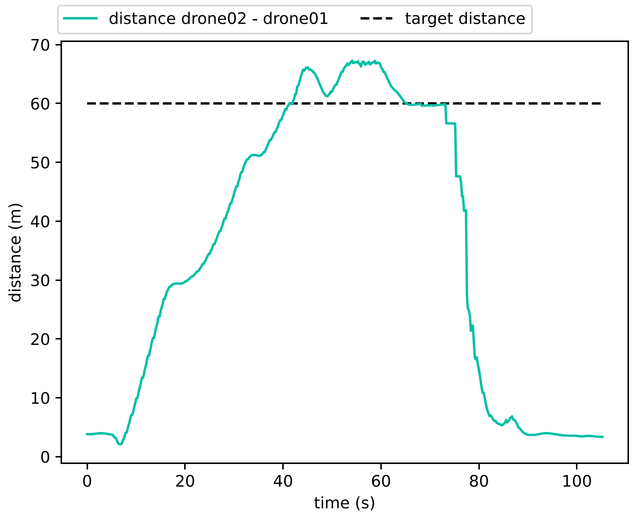


Fig. 18 Distance between drones on the real setting

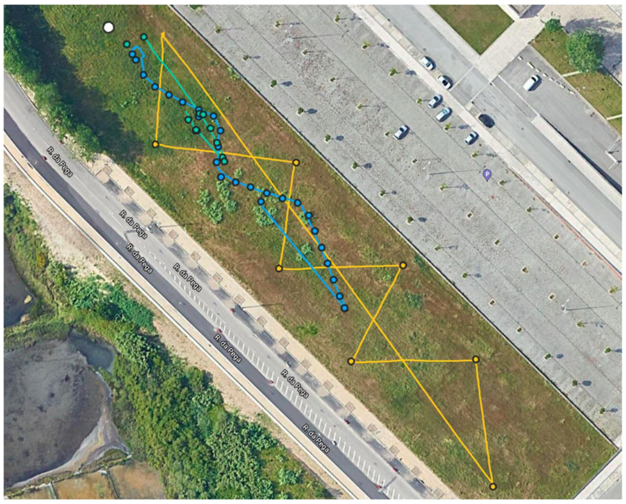


Fig. 19 Mission path in a real setting with two relay drones

Figure 20 shows the distance between drones. Again, the distance between drones is close to the target distance, and we can conclude that the relay mission is deployed and performed as expected.

6 Conclusions

This paper proposed a novel mission planning framework for autonomous fleets of aerial UAVs. While most available solutions offer restricted mission planning support, often comprised of a linear set of waypoints, or alternatively target a very specific scenario without providing support for generically describing custom mission plans, this framework offers flexible mission planning, with non-linear missions that can change the course of a UAV at any time during its sequence of actions, while being able to request data from its own sensors and reacting to that data. Mission plans can be described using EasyMission, providing a layer of abstraction which allows an inexperienced user to plan, execute and monitor complex missions with multiple UAVs. Using a single platform, it is possible to address multiple different scenarios and rapidly develop and test new algorithms.

Through the completion of a set of simulated and real-life experiments, it was possible to demonstrate that this framework can effectively be used to describe mission plans of varying complexity and context.

As future work, we aim to transition the current platform to a decentralized system that does not constantly rely on the ground station for the decision process, granting a higher level of autonomy to the UAVs.

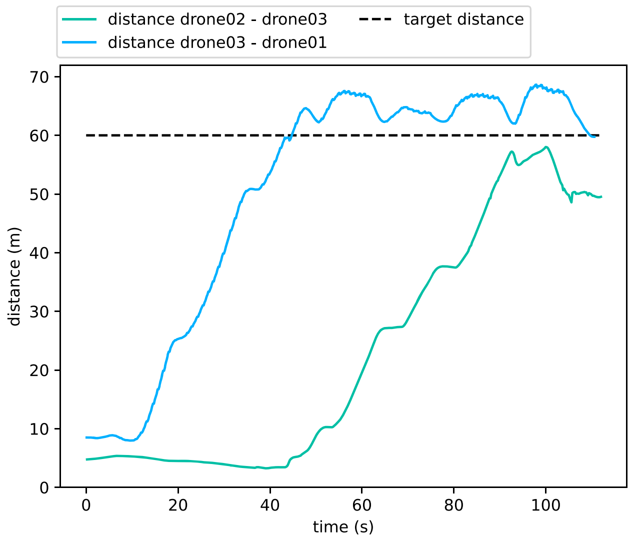


Fig. 20 Distance between drones on the real setting

Author Contributions Conceptualization, M.S, A.R. and S.S.; methodology, M.S. and A.R.; software, M.S.; validation, M.S.; investigation, M.S, A.R, S.S; writing—original draft preparation, M.S.; writing—review and editing, S.S. and A.R.; supervision, S.S. and A.R.; project administration, S.S. All authors have read and agreed to the published version of the manuscript.

Funding Open access funding provided by FCT—FCCN (b-on). This work is funded by FCT/MEC through national funds under the project PTDC/EEI-ROB/28799/2017 FRIENDS: Fleet of dRones for radIological inspEction, commuNication anD reScue.

Code or data availability Not applicable.

Declarations

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent to publish Not applicable.

Conflict of Interests On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Yanmaz, E., Quaritsch, M., Yahyanejad, S., Rinner, B., Hellwagner, H., Bettstetter, C.: Communication and coordination for drone

networks. In: Ad Hoc Networks, vol. 12, pp. 79–91. Springer International Publishing (2017)
2. Yanmaz, E., Yahyanejad, S., Rinner, B., Hellwagner, H., Bettstetter, C.: Drone networks: Communications, coordination, and sensing. Ad Hoc Netw. **68**, 1–15 (2018). advances in Wireless Communication and Networking for Cooperating Autonomous Systems
3. Besada, J., Bergesio, L., Campaña, I., Vaquero-Melchor, D., López-Araquistain, J., Bernardos, A., Casar, J.: Drone mission definition and implementation for automated infrastructure inspection using airborne sensors. Sensors **18**(4), 1170 (2018)
4. Pant, Y.V., Abbas, H., Quaye, R.A., Mangharam, R.: Fly-by-logic: control of multi-drone fleets with temporal logic objectives. In: 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), pp. 186–197 (2018)
5. Thibbotuwawa, A., Bocewicz, G., Banaszak, Z., Nielsen, P.: A solution approach for uav fleet mission planning in changing weather conditions. Appl. Sci. **9**, 09 (2019)
6. Zermani, S., Dezan, C., Euler, R.: Embedded decision making for uav missions. In: 2017 6th Mediterranean Conference on Embedded Computing (MECO), pp. 1–4 (2017)
7. Rudol, P., Doherty, P.: Bridging reactive and control architectural layers for cooperative missions using vtol platforms. In: 2017 25th International Conference on Systems Engineering (ICSEng), pp. 21–32 (2017)
8. Wu, Y., Bao, G., Xu, X., Lei, Z.: Research on a language for commanding and controlling multi-robot systems. In: 2014 10th International Conference on Natural Computation (ICNC), pp. 1077–1081 (2014)
9. Bagnitckii, A., Inzartsev, A., Senin, R.: Facilities of auv search missions planning. In: OCEANS'11 MTS/IEEE KONA, pp. 1–7 (2011)
10. Pavin, A., Inzartsev, A.: A geojson-based mission planning language for auv (auvgeojson language). In: OCEANS 2018 MTS/IEEE Charleston, pp. 1–5 (2018)
11. Paula, N.: Multi-drone control with autonomous mission support, Master's Thesis, Universidade de Aveiro. [Online]. Available: http://hdl.handle.net/10773/27846 (2018)
12. Areias, B., Martins, A., Paula, N., Reis, A., Sargento, S.: A control and communications platform for procedural mission planning with multiple aerial drones. Pers. Ubiquit. Comput. 03 (2020)
13. Silva, M., Mourato, A., Marques, G., Sargento, S., Reis, A.: A platform for autonomous swarms of uavs. In: 2022 IEEE Symposium on Computers and Communications (ISCC) (2022)
14. König, D., King, P., Laforge, G., D'Arcy, H., Champeau, C., Pragt, E., Skeet, J. Groovy in Action, 2nd edn. Manning Publications Co, USA (2015)
15. Saldaña, D., Ovalle, D., Montoya, A.: Improved algorithm for perimeter tracking in robotic sensor networks. In: 38th Latin America Conference on Informatics, CLEI 2012 - Conference Proceedings, vol. 10, pp. 1–7 (2012)

Margarida Silva earned her Master's Degree in Computer and Telematics Engineering from Universidade de Aveiro in 2021. During her studies, she was also a student researcher at the Instituto de Telecomunicações, specifically in the Network Architectures and Protocols group. Her contributions during this time were focused on the field of Unmanned Aerial Vehicles, namely participating in the FRIENDS (Fleet of dRones for radIological inspEction, commuNication anD reScue) project. In the scope of this project, she designed and implemented a drone control and management system. She continues her work in the field of computer engineering.

André Braga Reis received the B.S. and M.Sc. degrees in electronics and telecommunications engineering from the University of Aveiro, Portugal, in 2009, in collaboration with the Eindhoven University of Technology, The Netherlands, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA. His research interests are in vehicular, ad-hoc, and mesh networks.

Susana Sargento (Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Aveiro, in 2003. She was a Visiting Student at Rice University, from 2000 to 2001. She was the Founder of Veniam (https://www.veniam.com), which builds a seamless low-cost vehicle-based internet infrastructure. She is currently a Full Professor with the University of Aveiro and a Senior Researcher with the Institute of Telecommunications, where she is leading the Network Architectures and Protocols (NAP) Group (https://www.it.pt/Groups/Index/62). Since 2002, she has been leading many national and international projects, and worked closely with telecom operators and OEMs. She has been involved in several FP7 projects (4WARD, Euro-NF, C-Cast, WIP, Daidalos, and C-Mobile), EU Coordinated Support Action 2012-316296 "FUTURE-CITIES," EU Horizon 2020 5GinFire, EU UIA Steam City, CMU-Portugal projects (S2MovingCity and DRIVE-IN with the Carnegie Melon University), and MIT-Portugal Snob5G Project. She regularly acts as an Expert of the European Research Program. She was also the Co-coordinator of the national initiative of digital competences in the research axis (INCoDe.2030, https://www.incode2030.gov.pt/), belonged to the evaluation committee of the Fundo200M (https://www.200m.pt) government coinvestment and funding. She is one of the Scientific Directors of CMU-Portugal Program (https://www.cmuportugal.org/). Her main research interests include self-organized networks and ad-hoc and vehicular network mechanisms and protocols, such as routing, mobility, security and delay-tolerant mechanisms, resource management, and content distribution networks. She was the Winner of the 2016 EU Prize for Women Innovators. She has been the TPC-Chair and organized several international conferences and workshops, such as ACM MobiCom, IEEE GLOBECOM, and IEEE ICC. She has been a Reviewer of numerous international conferences and journals, such as IEEE Wireless Communications, IEEE Networks, and IEEE Communications Magazine.