MAVLink plugin for the Dolphin language

Nuno Gonçalo Neto Martingo
Department of Computer Sciences - 2019/2020
Faculty of Sciences - University of Porto

March 26, 2020

Abstract

Dolphin is a domain-specific language for programming autonomous vehicle networks. A Dolphin script describes an orchestrated execution of tasks that are assigned dynamically to multiple vehicles and that can be composed concurrently, sequentially or in an event-based manner. We developed a Dolphin DSL plugin to support integration with a widely used standard for drone communications, the MAVLink protocol. This report describes the design and implementation of the plugin and associated validation tests. All these tests were conducted using an ArduPilot software-in-the-loop simulation environment.

1 INTRODUCTION

The use of autonomous vehicles has seen a great increase in the last few years due to its incredible versatility and increasing ease of use, as the piloting and control of these vehicles has gotten significantly easier alongside the evolution of the technology driving these vehicles in motor, communication, batteries, etc. Autonomous vehicles are nowadays used for a variety of scientific, military and civilian applications. In addition, several vehicles are increasingly deployed together in a dynamic networked environment that also includes human operators and external sensors in the control loop, motivating the need for effective tools for programming these systems as a whole.

Dolphin [1] is a domain-specific language (DSL) with these concerns in mid, available open-source (https://dolphindsl.github.io). A Dolphin program expresses an orchestrated execution of tasks that are defined for multiple vehicles dynamically available in a network. Dolphin provides operators for composing one-vehicle tasks running in different vehicles for instance according to concurrent, sequential or event-based program flow. Previously, a Dolphin plugin has been developed for an open-source toolchain [2] for autonomous vehicles developed by Laboratório de Sistemas e Tecnologias Subaquáticas (LSTS), which has been tested in real-world operations during the REP-17 exercise co-organised by LSTS and the Portugues Navy [3].

This report documents the addition of a MAVLink [4] plugin to the Dolphin language. MAVLink is an extremely popular and widely used communication protocol for communicating with autonomous vehicles based on the ArduPilot platform [5]. The plugin provides a Dolphin DSL extension to express MAVLink missions. These are one-vehicle tasks that consist of sequences of individual maneuvers (i.e takeoff, land, waypoint tracking, loitering) and commands (e.g. take a photo). To allow these extensions, the plugin also includes all the necessary bindings to deal with the MAVLink protocol and to communicate with MAVLink-enabled vehicles.

The rest of the paper is structured as follows. In Section 2.1 we present Dolphin in terms of the underlying architecture along with some example scenarios, operators and platform bindings. Section 2.2 describes the MAVLink communication protocol, what it is, packet structure and why it was necessary to add support for it to Dolphin. Section 3 describes how to use and presents some examples of actually using the plugin. Section 3.1 provides some examples of this plugin in action. Section 3.2 describes how this plugin was implemented and how it relates with the existing Dolphin API. Section 4 describes the testing setup used throughout the report. Finally, Section 5 ends the report with concluding remarks and discussion of future work.

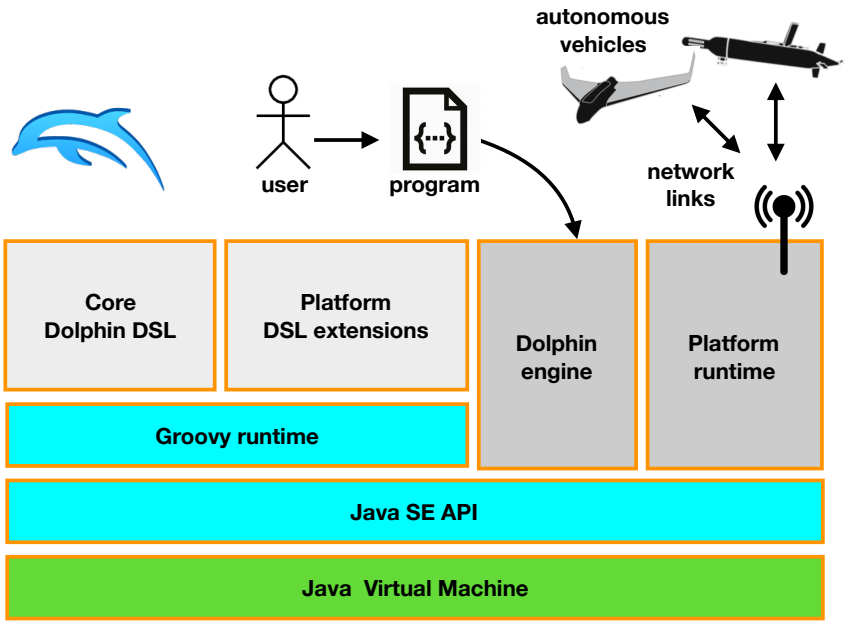


Figure 1: The architecture of Dolphin

2 Background

2.1 Dolphin

The architecture of Dolphin is illustrated in Figure 1. The Dolphin engine runs a centralised program, where a single base station controls and monitors all of the autonomous vehicles that are connected to it over the network. The language itself is embedded in Groovy that provides a number of features for defining the DSL (e.g. operator overloading, meta-class programming or the use of closures [6].

The execution engine is programmed in Java and concrete platform instantiations through plugins may extend the DSL and need to provide a runtime for networked interaction, implementing abstract Java bindings provided by the Dolphin engine.

We can find a simple usage of Dolphin in Listing 1, taken from [3], illustrating in this case the integration with the LSTS toolchain. The program concurrently runs 3 plans in 3 different autonomous vehicles, showing some of the possibilities for task orchestration.

The example provided in Listing 1 is composed of the following:

- We first start by picking three UUVs with the `pick` command. In this pick command we specify the details of the vehicles that we want to choose. In this case we want 3 vehicles (`count 3`) that are of the UUV type (`type 'UUV'` (Unmanned underwater vehicle)) that are situated in the area `Comporta` and that are equipped with a side-scan sonar (`payload 'Sidescan'`). These selected vehicles are then stored in the variable `UUVs`.

- After the selection of the vehicles, we then set the connection timeout at 10 minutes (`setConnectionTimeout 10.minutes`) as the UUVs may be underwater for an extended amount of time and might not be able to communicate very frequently.

- Finally we have the execution of the tasks. This is specified by the `execute` instruction, that tasks the vehicles with three plans, corresponding to ocean surveys in separate areas. These surveys will be executed concurrently (one survey per vehicle), as specified by the use of the "`|`" Dolphin operator.

Listing 1: Example Dolphin Program

```
// Define area of operation.
Comporta = (location 38.43462, -8.86118) ^ 2.km

// Select UUVs.
UUVs = pick {
         count 3
         type 'UUV'
         region Comporta
         payload 'Sidescan'
       }

// Set connection timeout.
setConnectionTimeout 10.minutes

// Execute ocean surveys.
execute UUVs:
          imcPlan['survey1'] |
          imcPlan['survey2'] |
          imcPlan['survey3']

// End!
message 'Done!'
```

To provide a brief idea of other of other compositions supported by Dolphin, Figure 2 depicts a BNF-like grammar for task specifications, (taken from [1]).

```
Task := PlatformTask              // Platform task
     | action '{' Code '}'        // Program-level action
     | condition '{' Cond '}'     // Program-level condition
     | Task '>>' Task             // Sequential composition
     | Task '|' Task              // Concurrent composition
     | Task '[' VSet ']'          // Vehicle set allocation
     | allOf '{' WhenThen+ '}'    // All-of block
     | oneOf '{' WhenThen+ '}'    // One-of block (choice)
     | waitFor '{' Cond '}'       // Execution subject
       then Task                  //   to start condition
     | until '{' Cond '}'         // Execution subject
       run Task                   //   to stop condition
     | idle Time                  // Idle task
     | during Time                // Execution subject
       run Task                   //   to time limit
     | watch Task                 // Error handling
       onError '{' Code '}'
       WhenThen := when '{' Cond '}' then Task
```

Figure 2: Grammar for Dolphin tasks.

2.2 MAVLink

MAVLink is a very lightweight messaging protocol for communicating with autonomous vehicles. It follows a hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as topics while configuration subprotocols are point-to-point with retransmission. It is also extremely versatile as it supports many programming languages and can be run on micro controllers, which is perfect for low power high efficiency autonomous vehicles and is also very reliable as it provides methods for detecting packet drops, corruption and packet authentication.

MAVLink uses UDP datagrams to send information between the base station and vehicles and vice-versa. These datagrams (also known as packets) follow a specific format which is dependent on the version that you are using. There are 2 versions of the MAVLink standard, each of which has it's own datagram format. In this report we will only discuss the version 2 of the MAVLink standard as that is what we are using. The format for MAVLink 2 datagrams can be found in Figure 3.

Each of the packet's fields are described in the table found in Table 1.

These datagrams are represented in Java by the class **MAVLinkMessage**, which stores the base parameters for every message like `LEN`, `SEQ`, `SYSID`, `COMPID`, `MSGID`, `CHECKSUM` among

Figure 3: MAVLink 2 packet format

Table 1: Description of each of the MAVLink 2 packet fields

Field:	Description:
`STX`	Protocol specific start of packet marker to indicate when a new packet has started. In MAVLink 2 the value in this field is 0xFD (253)
`LEN`	Specifies the length of the `PAYLOAD` section of the packet.
`INC FLAGS`	Specifies some flags that must be understood for MAVLink compatibility. (The packet is discarded if the implementation does not understand these flags)
`CMP FLAGS`	Specifies some flags that can be ignored if not understood. (The packet is not discarded even if the implementation does not understand these flags)
`SEQ`	This is used to detect packet loss. This field is increased with every packet sent.
`SYSID`	The ID of the system that has sent or that is suposed to receive the message. This is used to differentiate systems on the network. (All UAVs will communicate in the same port).
`COMPID`	The ID of the component sending or receiving the message. This is used to between different systems inside a UAV.
`MSGID`	ID of the message type that is in the payload. (This is what is used to decide what subclass of **MAVLinkMessage** is used to read the information in the packet)
`PAYLOAD`	The data of the message. The contents of this will depend on the message type (Defined in `MSGID`) and the contents of the message.
`CHECKSUM`	This contains an X.25 CRC of the message, to allow for error detection during the sending of the packet. Packets who's `CHECKSUM` does not add up will be discarded.

others. This information is then used to decide which sub class of **MAVLinkMessage** should be picked to read and process the data found in the `PAYLOAD`. Each subclass has it's own different parameters that mean different things, but they all use the same packet base format, as they're all extensions of the **MAVLinkMessage** class.

How does MAVLink use the UDP datagrams?

To send and receive these datagrams both the base station and vehicles initiate a server that listens on a specific port `p` that must be the same on both ends, this port is defined by the user and should be set to a port without a lot of communication as this can interfere with the signal and cause packet drops and data loss. In this project, the base station's server is handled by a class called **MAVLinkCommunications**, more on this in Section 3.2.

The software used for the test scenarios present in this report (ArduPilot [5], QGroundControl [7]) also uses MAVLink as the communication standard, making it easier to test scenarios before actually deploying them out in the real world as both the test simulation and real world vehicles use exactly the same packet format, which provides a seamless transition between simulation and reality.

MAVLink also implements a feature called micro-services [4] which define higher-level protocols that MAVLink systems can adopt in order to better inter-operate. These micro-services are used to exchange many types of data, if the data can't fit into a single message, services will define how

the data is split and re-assembled and how to ensure that any lost data is re-transmitted.

In this report we are going to focus mainly in the *Mission Protocol* service. This service allows a base station to manage the mission (flight plan) of an autonomous vehicle and follows the client/server pattern where operations are initiated by the base station and acknowledged by the vehicle. It also supports re-request of messages that have not been delivered, allowing missions to be reliably transferred over a *lossy* link, as can be seen in Figure 4.

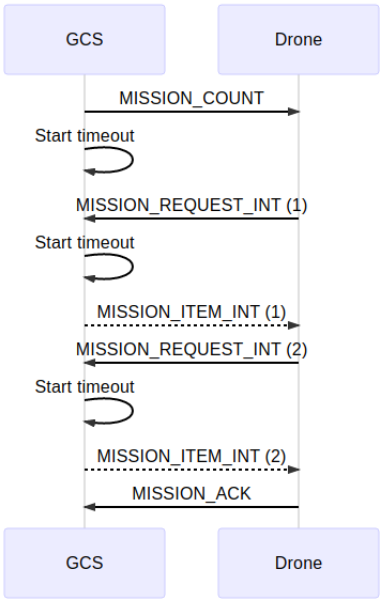


Figure 4: Mission upload protocol. GCS is the base station (Ground Control Station)

The *Mission Protocol* works by splitting a mission into a series of points that each have their own sequence number `seq`, which get executed by the UAV. While executing the mission, the UAV will report back with the base station with information about the mission like the current point it's heading for and when it has successfully reached a point. It does this by sending message to the base station with the `seq` number of the point that the UAV is referring to.

3 The MAVLink Plugin for Dolphin

We will now go into more detail about the plugin. This plugin adds a great list of commands to allow the user to create the missions. All of these commands can be found in Appendix A. Each of the commands represents an extension of the `MissionPoint` or `DroneCommand`, depending on what command it is, more on this in Section 3.2. We will demonstrate how to use these commands and also some of their capabilities and use cases via examples found in Section 3.1.

3.1 Examples

We now present some examples and their corresponding Dolphin program. The scenarios here represented are some possible ways to harness this technology to try to help improve society using in this case UAVs (Unmanned Aerial Vehicles). Each scenarios shows a way this technology can be used ranging from single vehicle, two vehicles performing independent tasks or even an arbitrary number of vehicles that rely on the status of other vehicles on their network to know their mission. In the situations we present here, it would be of great use to have sensors attached to the drones that we are using, to record data on the area that is being analysed. This is possible with Dolphin

Listing 2: Single vehicle Dolphin Program

```
homePos = location(40.444233, -7.375234, 820)
takeOffLocation = location(40.444233, -7.375234, 1200)
simpleMission = mission {
    name "Survey Area"
    home homePos
    takeOff takeOffLocation
    move 100, 200, 100
    moveAndLoiterPos 200, 100, -20, 50
}
returnHome = mission {
    name "Return to base"
    home homePos
    returnHomeAndLand()
}
//Execute the task
while (true) {
    v1 = pick {
        type 'UAV'
        timeout 30.seconds
    }
    execute v1:
        waitFor {battery(v1) >= 0.9}
          then ((simpleMission / {battery(v1) < 0.3})
             >> returnHome)
    //Release the drone for other possible tasks
    release v1
}
```

and MAVLink but is not used here because ArduPilot does not simulate sensors, more on this in Section 5.

3.1.1 Example 1 - Single Vehicle

The example found in Listing 2 shows how Dolphin can be used to program a vehicle for a moving between a few points and then loitering (going in circles around the point) and returning to base when the battery is low. This is a just an extremely simple example of the possibilities that this technology can offer.

In Listing 2 we first start off by defining the base location of the mission `homePos = location (...)` this will be the home location and is assumed to be the location of a charging station for the drones, we also declare the `takeOffLocation` which basically tells the UAV what height it must reach before moving on to it's next mission point. Afterwards we define the main mission for the vehicle with `simpleMission = mission {...}`, we instruct the drone to move 100 meters north, 200 meters east and 100 meters up and following that we tell it to move 200 meters north, 100 meters east and 20 meters down and loiter this position indefinitely. After that we also define the mission for the drone to execute when the battery is low with `returnHome = mission {...}` which just tells the drone to return to the home position and attempt to land.

Now that the missions and locations are defined, we must assign them to the UAV. We do this by selecting one UAV from the available vehicles connected to the base station with `pick`. After we've chosen a UAV, we instruct Dolphin to execute the task with `execute v1`, this means we are telling Dolphin to explicitly use `v1` to complete this task. The task then waits for the battery of the UAV to be at 90% or more with `waitFor {...} then ...` and when the battery is at the required level, it then receives the task that is found after the `then ...`. This is a composed task which is comprised of the following:

- First we have the `simpleMission / {battery(v1)< 0.3}` task. The `/` operator means that the UAV will execute the `simpleMission` until the condition within `{...}` is met, in this case `battery(v1)< 0.3`.
- After that condition is met, the first part of the task is completed and we get to the second part of the task, `(...)>> returnHome`. The `task1 >> task2` operator means that `task2`

Listing 3: Multiple vehicles without interaction Dolphin Program

```
//Location of the base
homePos = location(40.444233, -7.375234, 820)
takeOffLocation = location(40.444233, -7.375234, 1200)
surveyNorth = mission {
    name "Survey Area 1"
    home homePos
    takeOff takeOffLocation
    surveyArea 3000, 10000, 500, "NORTH_EAST"
}
surveySouth = mission {
    name "Survey Area 2"
    home homePos
    takeOff takeOffLocation
    surveyArea 3000, 10000, 500, "SOUTH_EAST"
}
returnHome = mission {
    name "Return to base"
    home homePos
    returnHomeAndLand()
}
//Execute the task
while (true) {
    v1 = pick {
        type 'UAV'
        timeout 30.seconds
    }
    v2 = pick {
        type 'UAV'
        timeout 30.seconds
    }
    execute v1:
        waitFor {battery(v1) >= 0.9}
          then ((surveyAreaNorth / {battery(v1) < 0.3})
             >> returnHome),
            v2:
        waitFor {battery(v2) >= 0.9}
          then ((surveyAreaSouth / {battery(v2) < 0.3})
             >> returnHome)
    release v1 + v2
}
```

is executed upon the finishing of `task1` (this is the sequence operator in Dolphin).

- So with this we get the following result: the drone will execute the `simpleMission` until a battery level of 30% is reached after which it will do the `returnHome` task.

All of the code relating to selecting the UAV, assigning it the mission and releasing it is, as can be seen, inside a `while(true){...}` loop. This means that the piece code found within it will be executed until any foreign action is taken, meaning this system (if there are no hardware failures or other outside events that may prevent the correct execution of the program) will run automatically without the need of human input.

3.1.2 Example 2 - Multiple vehicles with concurrent independent missions

The example found in Listing 3 shows Dolphin controlling 2 independent drones that are each running an area survey mission, in which they are supposed to cover a stretch of terrain. A use case of this could be replacing forest guards as a single drone can cover much more ground than a single human person, can work around the clock and besides from a higher initial investment the long term costs of running this setup are far inferior to the long term costs of employing several people to cover the same stretch of terrain.

In the scenario represented in Listing 3, we first start by defining the `homePos` and `takeOffPosition` like we did in the previous example. Following that we define the main missions for the vehicles to execute, with `surveyAreaNorth = mission {...}` and `surveyAreaSouth = mission { ... }`. These missions are composed of a simple `takeOff` command to tell the drone to take off from

the ground followed by a `surveyArea` command. The survey area command basically tells the drone to patrol an area of the dimensions provided, in this case an area that is 3km x 10km in size. Finally we have the actual execution of the missions, which is also very similar to the previous example where we, inside a `while(true){...}` loop, start off by selecting our 2 drones with `pick`. After they have been selected, we assign them to a mission which is comprised of a composed task similar to the task described in Section 3.1.1 with the exception that instead of executing the `simpleMission` we are executing `surveyAreaNorth` and `surveyAreaSouth`, respectively.

3.1.3 Example 3 - Multiple vehicles with coordination between them

In this final test, found in Listing 4, we are going to show some of the more intricate capabilities of Dolphin, specifically the capability to command vehicles based on what's happening in with other vehicles in the fleet. More specifically, with this example we want to show how the planning of a setup that would attempt to always have 2 drones in the air and an arbitrary number of drones (preferably more than 2) at the base, charging.

This scenario starts off very similarly to the previous 2, by defining the `homePos` and the `takeOffPosition`. After this we define the missions that are to be executed with `surveyAreaNorth = mission {...}` and `surveyAreaSouth = mission {...}`. These missions are the same as the ones found in Section 3.1.2, so we will not go into detail about them. Now we can see the first difference between this example and the one found in Section 3.1.2. After defining the missions, we then have to select the UAVs to execute them with `pick`. This, as can be seen, is not inside the `while (true){...}` loop, unlike the first two examples. This is due to the fact that because we want to always maintain 2 drones in the air, we can't pick drones that have just returned from a mission and still need to charge, as this would cause the base station to wait for these drones to charge to 90% before actually deploying them, leading to possibly having UAVs just sitting in the charging station. To prevent this the following is done:

- We start by picking 2 drones regularly, outside of the while loop. We can't pick them inside of the loop because then we wouldn't be able to distinguish the drones that have just returned from a mission from the drones that have been at the base charging and are waiting to be deployed.
- After picking the drones, we send them on the mission like in the previous examples. We will not go into detail as further explanation can be found in Section 3.1.2.
- When the drones have finished their assigned mission we start selecting the next UAV's to perform the missions. Note that this selection is done before releasing the drones that have been on mission and are just now returning to the base. Doing this selection before releasing them means that there is no possibility of the base station choosing the drones that have just returned as they are not marked as *available*. We save these selected drones into the `v1Temp` and `v2Temp` variables.
- After we have chosen the two drones new that will be executing the missions, we can release the drones that were out on mission and move the new drones into the `v1` and `v2` variables, respectively.
- At last the loop will run again, with the `v1` and `v2` variables containing the correct drones to execute the mission.

3.2 Implementation

This project is an extension to the Dolphin language and introduces a new way to communicate to autonomous vehicles. This extension operates in the Platform Runtime of the architecture that can be seen in Figure 1. With the introduction of MAVLink a new concept is introduced: **Missions**. These missions are a protocol that tell the autonomous vehicle what to do and they must follow the MAVLink specification. To implement these **Missions** while retaining the full

Listing 4: Multiple vehicles with interaction Dolphin Program

```
//Location of the base
homePos = location(40.444233, -7.375234, 820)
takeOffLocation = location(40.444233, -7.375234, 1200)
surveyNorth = mission {
    name "Survey Area 1"
    home homePos
    takeOff takeOffLocation
    surveyArea 3000, 10000, 500, "NORTH_EAST"
}
surveySouth = mission {
    name "Survey Area 2"
    home homePos
    takeOff takeOffLocation
    surveyArea 3000, 10000, 500, "SOUTH_EAST"
}
returnHome = mission {
    name "Return to base"
    home homePos
    returnHomeAndLand()
}
v1 = pick {
    type 'UAV'
    timeout 30.seconds
}
v2 = pick {
    type 'UAV'
    timeout 30.seconds
}
//Execute the task
while (true) {
    execute
    v1:
     waitFor {battery(v1) >= 0.9}
      then ((surveyAreaNorth /
             {battery(v1) < 0.3})
        >> returnHome),
    v2:
     waitFor {battery(v2) >= 0.9}
      then ((surveyAreaSouth /
             {battery(v2) < 0.3})
        >> returnHome)
    v1Temp = pick {
        type 'UAV'
        timeout 30.seconds
    }
    v2Temp = pick {
        type 'UAV'
        timeout 30.seconds
    }
    release v1 + v2
    v1 = v1Temp
    v2 = v2Temp
}
```

Listing 5: Mission Class structure

```
public class Mission extends PlatformTask {
    private List<DroneCommand> droneCommands;
    public List<DroneCommands> getCommandList() {...}
    public void setDroneCommands(List<DroneCommand> commands) { ... }
    public int missionPoints() {...}
    public List<MAVLinkMessage> toMissionMessages(MAVLinkNode dest) {...}
    public Map<Integer, List<MAVLinkMessage>> droneCommandsToMissionItem(
        MAVLinkNode dest) {...}
    public List<NodeFilter> getRequirements() {...}
    public TaskExecutor getExecutor() {}
}
```

functionality of Dolphin, we must use the already existing extendable structure which can be found in Figure 5.

We extended this and made use of the **Mission** (an extension of **PlatformTask**) for the definition of the missions to be assigned to vehicles, **MissionExecutor** (an extension of **PlatformTaskExecutor**) to define the current executing state of a certain vehicle performing a Task, **DroneCommand** to define the points that define the commands that constitute the **Mission** itself, **MAVLinkNode** (an extension of **AbstractNode**) which interprets the messages sent by the UAV and then serves the UAV's information using the already defined methods in Dolphin (e.g. Battery capacity). This class also contains the **MissionUploadProtocol** and **MissionDownloadProtocol**, which are responsible for handling the *Mission Protocol* uploading and downloading, respectively. Finally we have the **MAVLinkCommunications** class that manages the base station's UDP server and is responsible by sending and receiving information to and from the UAVs.

An updated diagram of Dolphin with these extensions can be visualized in Figure 6. Not all the classes and methods are represented in this Figure as they are not essential to understanding how the Dolphin language was extended to handle MAVLink.

3.2.1 Mission

The **Mission** class' structure is defined as can be found in Listing 5.

As we can see in Listing 5, each **Mission** contains a `List<DroneCommand>` that stores all of the commands that compose the mission, note that not all of these commands are part of the actual **Mission** protocol of MAVLink, some of the commands stored are there to add extra functionality and are completely handled by this extension and not MAVLink or Dolphin.

The method `getCommandList` is a simple *getter* for these commands.

The method `setDroneCommands` sets the commands for this mission but also does some extra work to make them compatible with the MAVLink protocol. The `List<DroneCommand>` that is accepted by it is the simple list of commands that comes from the Dolphin DSL using the methods found in Table 2. This simple list needs some additions before it is possible to send it to the UAV that are done in the `setDroneCommands` like adding the *Mission Item Count* (the first message to be sent to the UAV, as can be seen in Figure 4) and adding the actual *start* command to the mission (setting the UAVs mode to *AUTO*), amongst others.

The `missionPoints` method is a simple *getter* for the amount of items in this mission.

The method `toMissionMessages` converts all of the points that extend **MissionPoint** into **MAVLinkMessage** instances that can actually be sent to the drone. This method does not convert every **DroneCommand** because not all of them actually belong to the **Mission**, instead some of the **DroneCommand** found here are a part of extra functionality added by this extension and need to be handled specially, by `droneCommandsToMissionItem`.

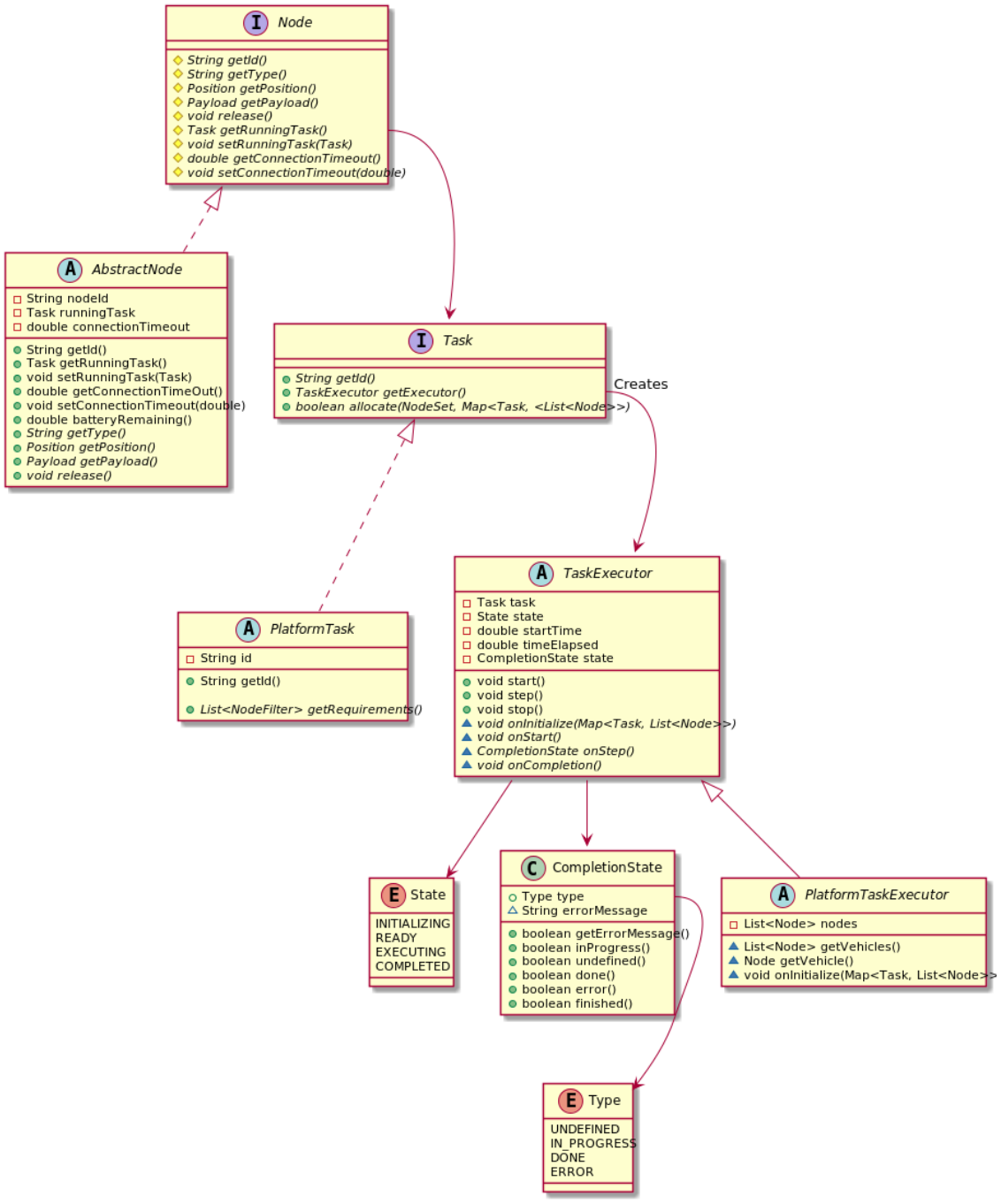


Figure 5: Dolphin Class Structure (not all Dolphin related classes are represented as they are not necessary to illustrate how this extension operates.)

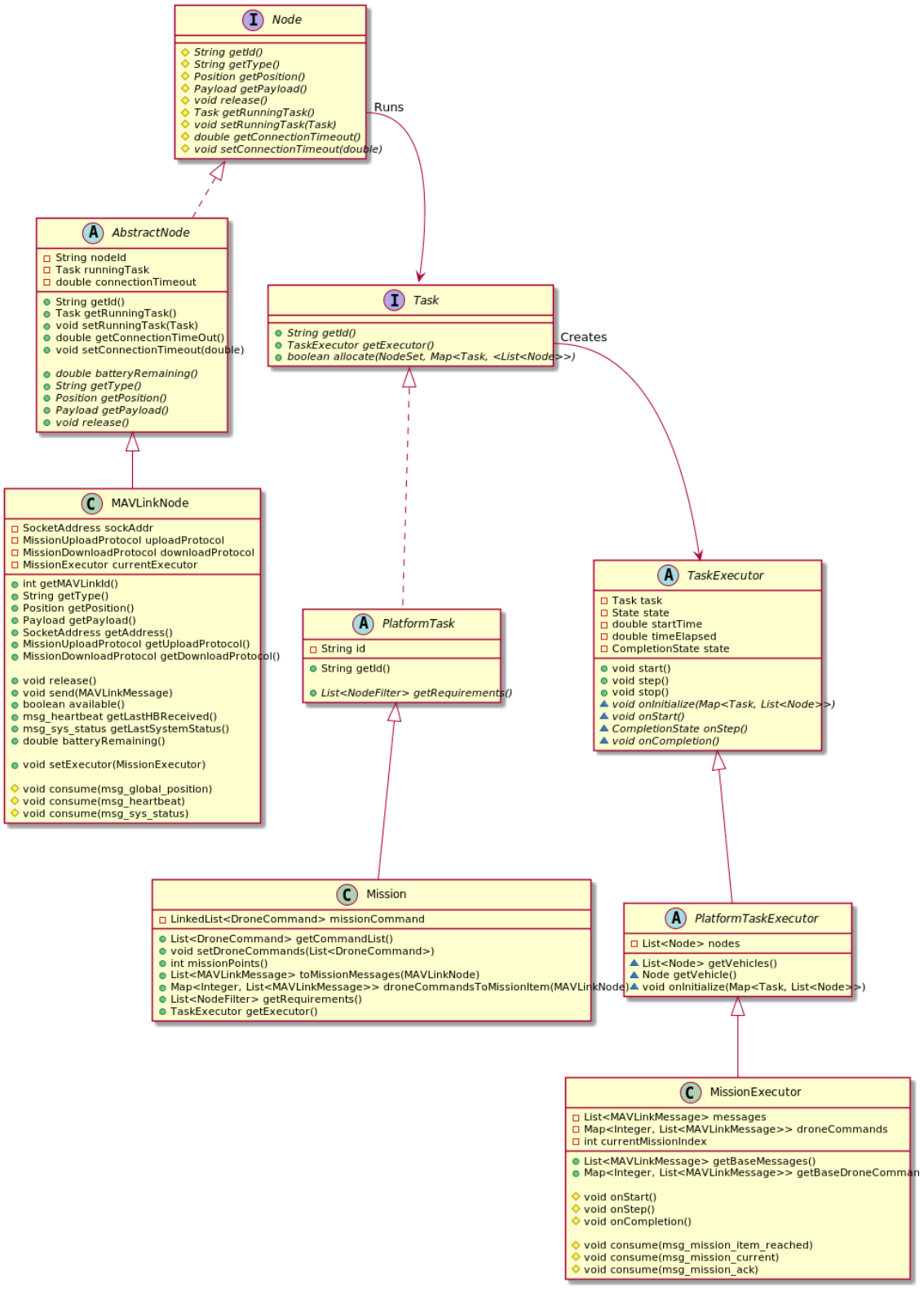


Figure 6: The Dolphin base structure with the MAVLink extensions added to them.

Listing 6: DroneCommand Class structure

```
public abstract class DroneCommand {
    public abstract Collection<MAVLinkMessage> toMavLinkMessage(MAVLinkNode
        dest);
}
```

Listing 7: MissionPoint Class structure

```
public abstract class MissionPoint extends DroneCommand {
    Position position;
    MissionPoint(Position position) {...}
    public Collection<MAVLinkMessage> toMavLinkMessage(MAVLinkNode dest) {
        return toMavLinkMessage(dest, 0);
    }
    public abstract Collection<MAVLinkMessage> toMavLinkMessage(MAVLinkNode
        dest, int seq);
    public abstract int messageCount();
}
```

The method `droneCommandsToMissionItem` translates all the messages that don't belong to the *Mission Protocol* of MAVLink and creates a `Map<Integer, List<MAVLinkMessage>>` with them. These messages must be sent at the correct time, which is done with the key of the map. The key represents the `seq` number at which the corresponding value (`List<MAVLinkMessage>`) must be sent, so the base station waits for the UAV to report when it has reached a certain point `seq`, and then consults the Map to see if there are any messages to send at that point and if so what those messages are. Any message with the key `seq = 0` is sent before the mission even gets uploaded (e.g. the `Mission Count` message that informs the drone of how many items to expect needs to be sent before starting the upload of the mission and it isn't actually a mission item and therefore is not contained in the messages returned by `toMissionMessages`).

The `getRequirements` method returns any possible extra requirements that the mission might have in the selection of vehicles to fulfill it (e.g. A mission that want's to capture photos will require a vehicle equipped with a camera).

The `getExecutor` method creates and returns a new `MissionExecutor` connected to this mission. This executor can then be assigned a `MAVLinkNode` to complete it.

3.2.2 DroneCommand

The `DroneCommand` class' structure is defined as can be found in Listing 6.

The method `toMavLinkMessage` returns the corresponding list of **MAVLinkMessages** that should be sent to the UAV.

This class represents the base of any sort of message that can be sent to a UAV and all the possible Mission commands extends this class.

3.2.3 MissionPoint

The `MissionPoint` class' structure is defined as can be found in Listing 7.

This class extends the `DroneCommand` class and defines the subset of messages that belong to the *Mission Protocol* of MAVLink (all subclasses of this class originate a collection of **MAVLinkMessage**'s sub-class `msg_mission_item`, and uses message id's to distinguish between each different type of command).

Listing 8: MissionExecutor Class structure

```
public class MissionExecutor extends PlatformTaskExecutor {
    protected void onStart() {...}
    protected CompletionState onStep() {...}
    protected void onCompletion() {...}
    public List<MAVLinkMessage> getBaseMessages() {...}
    public Map<Integer, List<MAVLinkMessage>> getBaseDroneCommands() {...}
    public void consume(msg_mission_item_reached) { ... }
    public void consume(msg_mission_ack mission_received) { ... }
}
```

Because most of the MAVLink mission items are based on the UAV moving to or performing some task in a given location, we store the `Position` of the `MissionPoint` and require it to instantiate any mission point. Some commands however do not require a position (e.g. the `speed` command). In these special cases, the position can be `null` and since the transformation of the `MissionPoint` into **MAVLinkMessage** is performed by the extensions of this class, each of these extensions can handle this `null` in it's own way.

The `toMavLinkMessage(MAVLinkNode dest)` method that is inherited from the `DroneCommand` class is overridden with a call to the newly defined `toMavLinkMessage(MAVLinkNode dest, int seq)`. This is because on the contrary of the `DroneCommand`, the `MissionPoint` requires the `seq` number(it's number in the execution order of the mission) for it to be able to instantiate the `msg_mission_item` correctly.

The `messageCount` method returns the amount of **MAVLinkMessage** that this `MissionPoint` needs to encode it's objective (e.g. the `surveyArea` command requires more than 1 message to define it's complete area of search). This is needed to calculate the `seq` numbers of following `MissionPoint`.

This class is extended by most of the possible commands found in Table 2 (not all as some of them have special features that are handled by this extension), where they each have their own sub class with their individual values, parameters and different ways of creating the corresponding **MAVLinkMessage**.

3.2.4 MissionExecutor

The `MissionExecutor` class is defined as can be found in Listing 8

This class is an extension of the **PlatformTaskExecutor** and therefore inherits it's responsibilities, so it is responsible for maintaining the correct execution of a **Mission** by a UAV.

The `onStart` method, inherited from **PlatformTaskExecutor** serves the purpose of beginning the execution of the given task with the given UAV. Before the UAV can start the mission though, the mission needs to be uploaded and some basic checks need to be performed to assert that the UAV is ready for the mission. First, we start by checking if the UAV is already on another mission and if so, we must stop it's current mission and remove it from the drone so we can upload the new one. After this, we must also check if the UAV is *armed* or, in other words, ready to be deployed. Finally, we call the `upload(MissionExecutor)` method in **MissionUploadProtocol** which handles actually sending all the messages to the drone (the mission is sent using the protocol found in Figure 4).

The `onStep` method, also inherited from **PlatformTaskExecutor** is used to check in on the progress of the mission that is being executed. It is called periodically by Dolphin and returns the current **CompletionState** of the execution.

The `onCompletion` method, also inherited from **PlatformTaskExecutor** gets called when the **CompletionState** is `DONE` or when Dolphin wants to terminate the mission sooner than anticipated (if for example, the UAV gets low on battery). This method sets the UAV back into

Listing 9: MAVLinkNode Class structure

```
public class MAVLinkNode extends AbstractNode {
    public MAVLinkNode(int sysId, SocketAddress addr) {...}
    public int getMavLinkId() {...}
    public Position getPosition() { ... }
    public Payload getPayload() { ... }
    public void release() { ... }
    public void send(MAVLinkMessage message) { ... }
    public msg_heartbeat getLastHBReceived() { ... }
    public msg_sys_status getLastSystemStatus() { ... }
    public double batteryRemaining() { ... }
    public MissionDownloadProtocol getDownloadProtocol() { ... }
    public MissionUploadProtocol getUploadProtocol() { ... }
    public void setExecutor(MissionExecutor executor) { ... }
    void consume(msg_global_position_int msg) { ... }
    void consume(msg_heartbeat msg) { ... }
    void consume(msg_sys_status status) { ... }
}
```

a "Wait for further instructions" mode, preparing it to receive the next mission.

The `getBaseMessages` method returns the messages created by `toMissionMessages` of the corresponding **Mission** class.

The `getBaseDroneCommands` method returns the messages created by `droneCommandsToMissionItems` of the corresponding **Mission** class.

The `consume` method accepts a `msg_mission_item_reached`, which is a sub class of the **MAVLinkMessage** class and represents the message sent by the UAV to report that it has reached a certain point in the mission. The **MissionExecutor** must monitor these messages to keep track of what `seq` the drone is currently executing, not only to know it's progress but also to know when to send the **DroneCommand** found in `getBaseDroneCommands`, which have to be sent at a specific point in the mission.

The `consume` method accepts a `msg_mission_ack_mission_received`, which is a sub class of the **MAVLinkMessage** class and represents the message sent by the UAV to report that it has correctly received the mission and is ready to start (this message is sent at the end of the *MissionProtocol* mission upload, as can be seen in Figure 4).

3.2.5 MAVLinkNode

The `MAVLinkNode` class' structure is defined as can be found in Listing 9.

This class is an extension of the **AbstractNode** class and represents a UAV that's in the network.

To initialize an instance of this class, we must provide the `sysId` of the UAV (this is provided by the UAV as a part of any **MAVLinkMessage**) and the `addr` (the address to send the datagrams meant for that UAV, also known as the UAV's IP address).

The methods `getMavLinkId`, `getPosition`, `getPayload` are simple getters for the `sysId` of the UAV, the current position and the UAV's Payload which is the equipment that it is carrying (e.g. cameras, sensors, etc).

The method `release` releases the UAV from it's current mission, should it be executing any and marks it as available to be picked for any mission.

The method `send` sends the provided **MAVLinkMessage** to the UAV. It does this by calling the `send(MAVLinkMessage, MAVLinkNode)` method of the **MAVLinkCommunications** class which is described in Section 3.2.6.

Listing 10: MAVLinkCommunications Class structure

```
public class MAVLinkCommunications extends Thread {
    private final MessageHandler<MAVLinkNode, MAVLinkMessage> msgHandler;
    public static MAVLinkCommunications getInstance() { ... }
    private MAVLinkCommunications() { ... }
    public void run() { ... }
    private void handleIncomingMessages() { ... }
    public void send(MAVLinkMessage msg, MAVLinkNode node) { ... }
    public void terminate() { ... }
}
```

The methods `getLastHBReceived` and `getLastSystemStatus` returns the last `msg_heartbeat` and `msg_sys_status` that the base station has received from the UAV, respectively.

The method `batteryRemaining` returns the current battery remaining on the UAV, in a scale of 0-1 (0 being 0% and 1 being 100%).

The methods `getDownloadProtocol` and `getUploadProtocol` return the handlers for MAVLink's *Mission Protocol* download and upload, respectively. We have not discussed the download aspect of the *Mission Protocol* as it wasn't very used in this project, but it is possible for the base station to download the mission that the drone is currently executing using the **MissionDownloadProtocol**.

The method `setExecutor` sets the current MissionExecutor that the UAV is executing.

The various implementations of the `consume` methods exist to receive all of the important information that the UAV is sending to the base station and handle it accordingly. The received packets are the following: `msg_global_position_int`, `msg_heartbeat` and `msg_sys_status`, which provide information about the UAV's current position, it's current vital information (what mode it is on, the type of vehicle it is, etc) and the status of the UAV (battery, communication status, on board controllers, etc), respectively.

3.2.6 MAVLinkCommunications

The MAVLinkCommunications class' structure is defined as can be found in Listing 10.

This class is responsible for all communication to and from the vehicles and redirecting the received information to the corresponding classes that will actually read and analyse this information. This redirection is achieved by using the `MessageHandler<MAVLinkNode, MAVLinkMessage>`, a class that is provided by Dolphin and it's purpose is to redirect the correct **MAVLinkMessage** to the correct **MAVLinkNode**.

This class follows the Singleton pattern and the only way to instance it is by calling `getInstance()`, which will create a new instance of the class if it doesn't exist yet or return the already existing instance of it if it does exist. The singleton pattern means that there can only be one instance of this class living at any point of execution in the program.

In the constructor of the class, the UDP server is initiated with a `new DatagramSocket` and the all of the redirections for each of the messages that are of interest are registered into the `MessageHandler`.

The method `run` is inherited from the **Thread** class and is where the `DatagramSocket` created in the constructor is put to use, as we create an infinite loop that is constantly listening to any incoming messages, using the method `handleIncomingMessages`.

The method `handleIncomingMessages` is what actually takes all of the information received via the `DatagramSocket` and transforms it into the correct **MAVLinkMessage**, which is then passed on to the `MessageHandler` to be delivered to the correct instance of **MAVLinkNode**.

The method `send` is responsible for sending the **MAVLinkMessage** to it's due destination. If first starts by transforming this class into an instance of **MAVLinkPacket**, which contains all

of the information in a `byte[]` ready to be sent via the `DatagramSocket`.

The method `terminate` stops the infinite while loop that is started inside `run` and closes the `DatagramSocket` meaning that after executing this method the base station can no longer send or receive any information.

4 Tests

The examples shown is this report were created with the test setup illustrated in Figure 7.

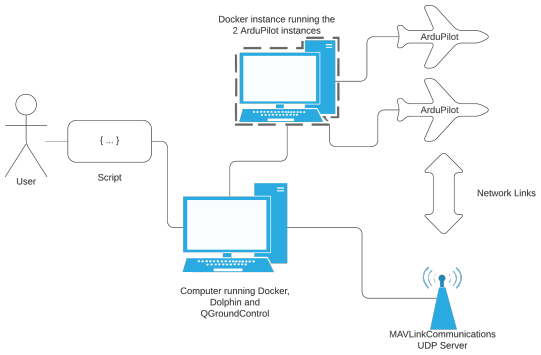


Figure 7: Representation of the testing setup

- The test location chosen was Serra da Estrela, a location where this technology could be of great use, as the terrain is very mountainous and therefore very hard for humans to navigate. Using a UAV in this scenario not only has the inherent advantage of being able to cover a lot more ground but also means that we do not need a human to traverse this hard terrain. It is also a very sparsely populated area with a great forest surrounding it, making it a great test location for our use case (early detection of fires). This is where our charging dock and base station running Dolphin would be found. A picture of this location can be found in Figure 8.
- An instance of QGroundControl [7], our chosen flight control software, to monitor all of the running UAVs, their current position, mission and general status in a GUI instead of just from the command line. There are many other flight controllers that are compatible with MAVLink but QGroundControl presents a simple, easy to understand and use interface and native compatibility with ArduPilot. An image of this software running without any connected vehicles can be found in Figure 8.
- A Docker container [8] that is running 2 instances of our auto pilot of choice, ArduPilot in SITL mode [9]. This was our choice as it provides great support for MAVLink and is the recommended flight simulator by QGroundControl. An image of the Docker container running ArduPilot can be found in Figure 9.
- Finally we have an instance of Dolphin with our MAVLink extension running in the base station, controlling the UAVs through UDP. An image of Dolphin running can be seen in Figure 10.

4.1 Results of the test

We recorded a run of the example found in Section 3.1.2 which can be found in `https://youtu.be/PUhr133ClN4`. We made some changes to the example as it would take too long if we waited for the battery to reach 30% so we raised the threshold to 70%. So what we observed was:

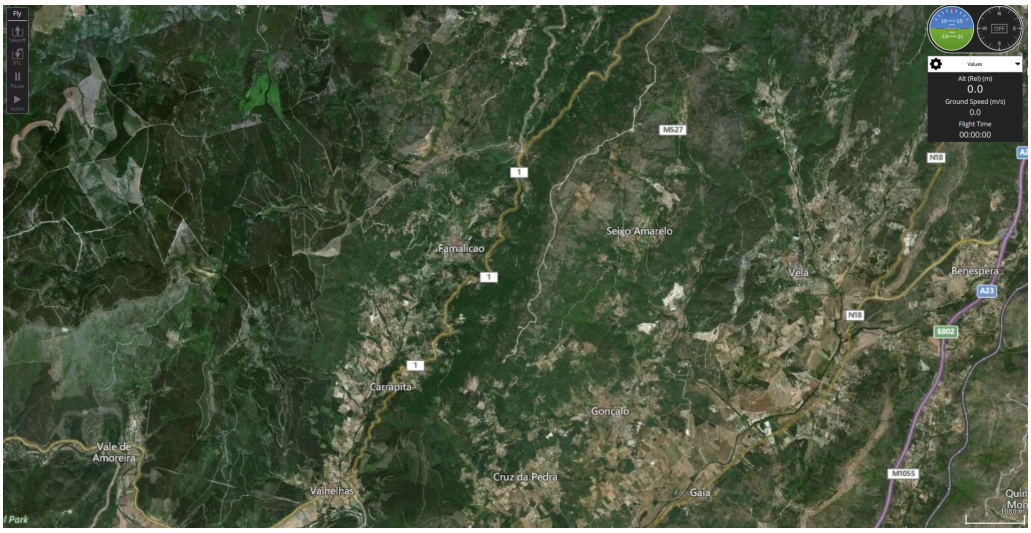
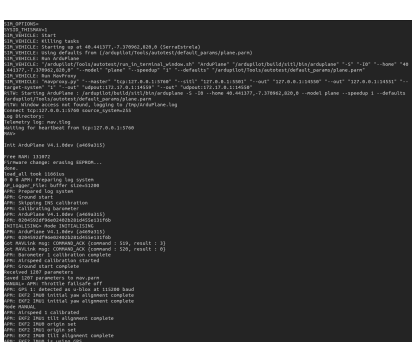
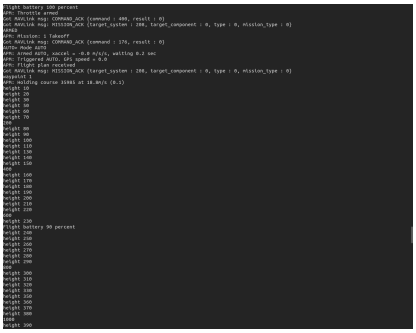


Figure 8: The location of the tests



(a) ArduPilot for Vehicle 1, initiating



(b) ArduPilot for Vehicle 2, running a mission

Figure 9: ArduPilot Examples

- Before we initialized Dolphin, both UAVs were in their home location with no assigned mission, as expected (00:00 in the video).
- After we initialized Dolphin, both UAVs were found and successfully took off and were each assigned their own separate missions, which they ran successfully. This can be seen in Figure 11 (00:15 in the video).
- When the Dolphin base station detects that the UAVs battery have hit the 70% threshold, the UAV is removed from his current mission and receives a new one, instructing it to return to his home location and attempt to land. This can be seen in Figure 12 where the drone only has the mission point `1 - Land` (02:00 in the video).

5 Conclusion and Future work

We presented an implementation of the MAVLink communication protocol to the Dolphin framework, how it was integrated into the already existing Dolphin API and how it's versatility can be used to solve various types of problems in our society. This extension adds an extensive list of supported vehicles as the MAVLink protocol is one of if not the most used communication

Figure 10: Dolphin instance running

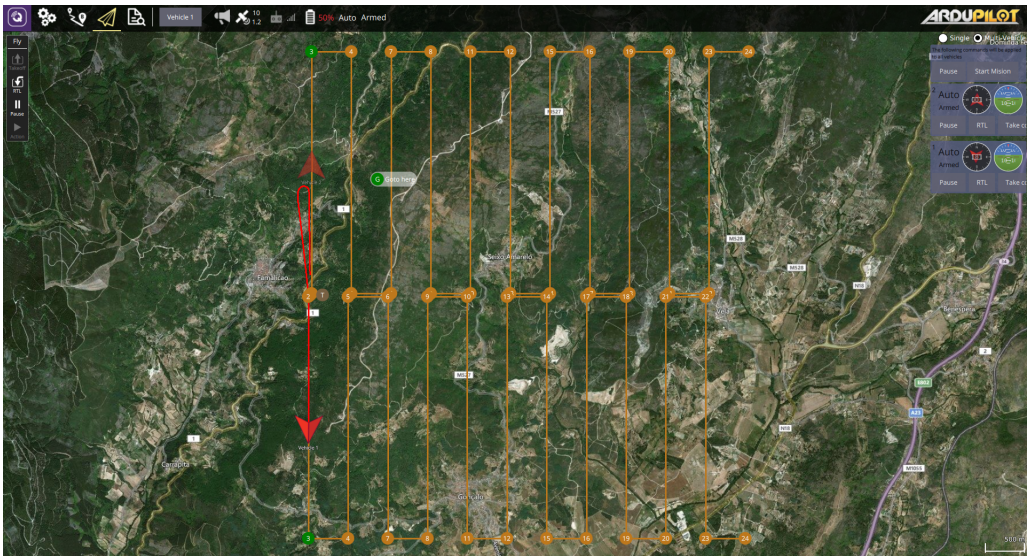


Figure 11: Screenshot of QGroundControl with two UAV's running their own separate missions

protocol in autonomous vehicles. The extensive list of DSL mission commands that are present in this extension combined with the very powerful operators that are present in Dolphin generates an extremely broad list of possibilities.

As future work, we would like to test this extension on real vehicles which would also allow us to test the existing camera features that could not be used as we could only use simulations. We would also like to add add more functionality related to the gathering of information and use of external sensors for example if we could possibly use a smoke detector to check if the region the drone is flying over is on on fire. We would also like to work on a extensible sensor API that would allow each end user to add support for their own specialized sensors, which would broaden the use cases of this technology even more. To implement this broader sensor support, we would have to handle the MAVLink micro-services that allow the base station to communicate with the sensors on the drone.

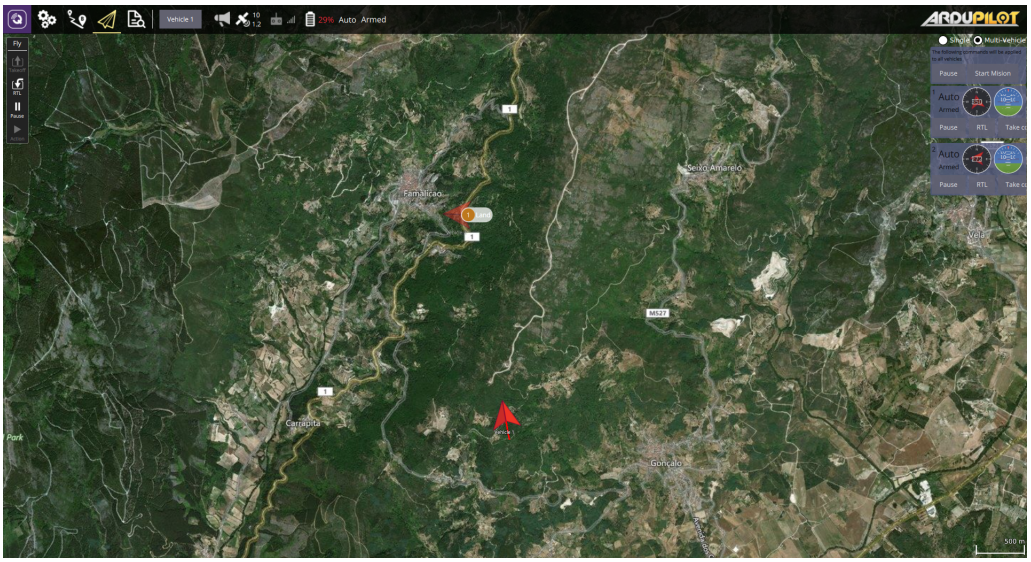


Figure 12: Screenshot of QGroundControl with a UAV that has reached 29% battery

References

[1] K. Lima, E. R. B. Marques, J. Pinto, and J. B. Sousa, "Dolphin: a task orchestration language for autonomous vehicle networks," in *Proc. 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'18)*. IEEE, 2018, pp. 8294–8301.

[2] J. Pinto, P. Dias, R. Martins, J. Fortuna, E. Marques, and J. Sousa, "The LSTS toolchain for networked vehicle systems," in *Proc. 2013 MTS/IEEE OCEANS*, ser. OCEANS'13. IEEE, 2013, pp. 1–9.

[3] K. Lima, E. R. B. Marques, J. Pinto, and J. B. Sousa, "Programming networked vehicle systems using Dolphin – field tests at REP'17," in *Proc. IEEE/MTS Oceans'18*. IEEE, 2018.

[4] "Mavlink: Micro air vehicle communication protocol," https://mavlink.io/en/.

[5] "Ardupilot: Fully featured reliable open source autopilot software." https://ardupilot.org/.

[6] F. Dearle, *Groovy for Domain-Specific Languages*. Packt Publishing, 2015.

[7] "Qgroundcontrol: Intuitive and powerful ground control station for the mavlink protocol." http://qgroundcontrol.com/.

[8] "Docker container," https://github.com/edrdo/ardupilot-sitl-docker.

[9] "Ardupilot: Sitl simulator (software in the loop)," https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

A DSL Commands

Table 2: All supported MAVLink mission commands.

DSL Commands	
Commands:	Description:
`name(String name)`	Set the name of the mission to `name`
See more in the next page	

Continuation of DSL Commands	
Commands:	Description:
`home(double lat, double lon, double hae)`	Set the home position to the position with latitude `lat`, longitude `lon` and height `hae`
`goPos(double lat, double lon, double hae)`	Add a go-to point to the mission, that tells the vehicle to go to the position with latitude `lat`, longitude `lon` and height `hae`
`takeOff(Position p, float pitch)`	Take off from a position `p` with a pitch of `pitch`. The `pitch` argument is optional and defaults to 15 degrees.
`move(double north, double east, double up)`	Move the drone relative to it's position by `north` meters north, `east` meters east, `up` meters up.
`moveFromHome(double north, double east, double up)`	Move the drone relative to it's home position by `north` meters north, `east` meters east, `up` meters up.
`moveAndLoiterPos(double north, double east, double up, double radius)`	Move the drone relative to it's last position by `north` meters north, `east` meters east, `up` meters up. When the drone arrives at this position, it will start to go in circles with radius of `radius` around it (Loiter) forever. The `radius` argument is optional and defaults to 15m.
`moveAndLoiterTurns(double north, double east, double up, int turns, double radius)`	Move the drone relative to it's last position by `north` meters north, `east` meters east, `up` meters up. When the drone arrives at this position, it will do `turns` circles around the position with radius `radius`. The `radius` argument is optional and defaults to 15m.
`moveAndLoiterTime(double north, double east, double up, int time, double radius)`	Move the drone relative to it's last position by `north` meters north, `east` meters east, `up` meters up. When the drone arrives at this position, it will circle the position for the specified `time` (in seconds) with the given `radius`. The `radius` argument is optional and defaults to 15m.
`moveAndLoiterFromHome(double north, double east, double up, double radius )`	Move the drone to it's home position by `north` meters north, `east` meters east, `up` meters up. When the drone arrives at this position, it will start to go in circles with radius of `radius` around it (Loiter) forever. The `radius` argument is optional and defaults to 15m.
`moveAndLoiterTurnsFromHome(double north, double east, double up, int turns, double radius)`	Move the drone relative to it's home position by `north` meters north, `east` meters east, `up` meters up. When the drone arrives at this position, it will do `turns` circles around the position with radius `radius`. The `radius` argument is optional and defaults to 15m.
`moveAndLoiterTimeFromHome(double north, double east, double up, int , double radius)`	Move the drone relative to it's home position by `north` meters north, `east` meters east, `up` meters up. When the drone arrives at this position, it will circle the position for the specified `time` (in seconds) with the given `radius`. The `radius` argument is optional and defaults to 15m.
See more in the next page	

Continuation of DSL Commands	
Commands:	Description:
`loiterPos(Position position, double radius)`	Move the drone to the specified `position` and when it arrives, circle it with the given `radius`. The `radius` argument is optional and defaults to 15m.
`loiterPos(double latitude, double longitude, double height, double radius`	Move the drone to the specified `latitude`, `longitude` and `height` and when it arrives, circle it with the given `radius`. The `radius` argument is optional and defaults to 15m.
`loiterTurns(double latitude, double longitude, double height, int turns, double radius)`	Move the drone to the specified `latitude`, `longitude` and `height` and when it arrives, circle it `turns` times with the given `radius`. The `radius` argument is optional and defaults to 15m.
`loiterTime(double latitude, double longitude, double height, int time, double radius)`	Move the drone to the specified `latitude`, `longitude` and `height` and when it arrives, circle it for `time` seconds with radius of `radius`. The `radius` argument is optional and defaults to 15m.
`landingPoint(double latitude, double longitude, double height, float yaw, int landMode)`	Instruct the drone to attempt to land at the given `latitude`, `longitude` and `height` at the given `yaw` angle. The `yaw` argument is optional and defaults to `NaN`, which lets the drone choose the yaw to land with. The `landMode` argument is also optional and can take the values of the **PRECISION_LAND_MODE** class, which are `PRECISION_LAND_MODE_DISABLED (= 0), PRECISION_LAND_MODE_OPPORTUNISTIC (= 1), PRECISION_LAND_MODE_REQUIRED (= 2)` the default value is `PRECISION_LAND_MODE.PRECISION_LAND_MODE_DISABLED`.
`landingPoint(Position position, float yaw, int landMode)`	Instruct the drone to attempt to land at the given `position` at the given `yaw` angle. The `yaw` argument is optional and defaults to `NaN`, which lets the drone choose the yaw to land with. The `landMode` argument is also optional and can take the values of the **PRECISION_LAND_MODE** class, which are `PRECISION_LAND_MODE_DISABLED (= 0), PRECISION_LAND_MODE_OPPORTUNISTIC (= 1), PRECISION_LAND_MODE_REQUIRED (= 2)` the default value is `PRECISION_LAND_MODE.PRECISION_LAND_MODE_DISABLED`.
`returnHome()`	Instruct the drone to move to it's home position.
`returnHomeAndLoiter(double radius)`	Instruct the drone to move to it's home position and go in circles around it with the `radius`. The `radius` argument is optional and defaults to 15m.
See more in the next page	

Continuation of DSL Commands	
Commands:	Description:
`returnHomeAndLand(float yaw, int landMode)`	Instruct the drone to return to it's home position and land at the given `yaw` angle. The `yaw` argument is optional and defaults to `NaN`, which lets the drone choose the yaw to land with. The `landMode` argument is also optional and can take the values of the **PRECISION_LAND_MODE** class, which are `PRECISION_LAND_MODE_DISABLED (= 0)`, `PRECISION_LAND_MODE_OPPORTUNISTIC (= 1)`, `PRECISION_LAND_MODE_REQUIRED (= 2)` the default value is `PRECISION_LAND_MODE.PRECISION_LAND_MODE_DISABLED`.
`speed(double newSpeed, int speedType)`	Instruct the drone to travel at the given `newSpeed`. `speedType` is optional and allows the user the speed that he is refering to. 0 corresponds to Air-Speed and is the default, 1 corresponds to Ground Speed, 2 to Climb Speed and 3 to Descent Speed. The last 2 should be used when changing the altitude of the drone.
`altitude(double newAltitude)`	Instruct the drone to climb or descend to the `newAltitude` (The altitude should be in meters and should be from the sea level).
`delay(int seconds)`	Delay the mission for `seconds` second.
`surveyArea(double length, double width, double direction_change, String direction)`	Instructs the drone to survey an area (The base position of which is the last position of the drone) `direction` of the base point that is `length` meters long, `width` meters wide. The argument `direction_change` is optional and defaults to 50 meters and is used to set the space between each vertical pass. The argument `direction` is optional and has the default value of `"NORTH_WEST"`. It can take the values `"NORTH_WEST"`, `"NORTH_EAST"`, `"SOUTH_WEST"`, `"SOUTH_EAST"`.
`surveyArea(Position p, double length, double width, double direction_change, String direction)`	Instructs the drone to survey an area with the base position `p` to the `direction` that is `length` meters long, `width` meters wide. The argument `direction_change` is optional and defaults to 50 meters and is used to set the space between each vertical pass.. The argument `direction` is optional and has the default value of `"NORTH_WEST"`. It can take the values `"NORTH_WEST"`, `"NORTH_EAST"`, `"SOUTH_WEST"`, `"SOUTH_EAST"`.
`capturePhoto()`	Instruct the drone to take a picture. (This feature is not available in simulations).
`captureSeveralPhotos(int photoCount, int interval)`	Instruct the drone to `photoCount` pictures at an interval of `interval`. The `interval` is optional and defaults to 2 seconds.
`captureUntilStop(int interval)`	Instruct the drone to take pictures at an interval of `interval` seconds. The drone will keep taking pictures until it is told to stop. The argument `interval` is optional and defaults to 2 seconds.
See more in the next page	

Continuation of DSL Commands	
Commands:	Description:
`stopCapturing()`	Instruct a drone that was taking pictures in the `captureUntilStop` mode to stop capturing.
End of DSL Commands	