IIoT Project

Giovanni Arlotta

September 30, 2023

Contents

1 Introduction

The scope of this project is to realize a simulation of a drone and make it move on Gazebo and QGC, using ROS2 and Python scripts, our goal is also to make it possible to see the packets that will be exchanged during the simulation. We are going to reproduce the following scenarios:

- Drone Backflip
- Offboard control
- Mission for 2 drones in a non-empty world

There are some prerequisites to replicate this project:

- ROS2 Humble
- PX4-Autopilot
- Micro XRCE-DDS Agent
- px4_msgs
- Ubuntu 22.04
- Python 3.10

It's possible to find the code and all the setup steps here:
https://github.com/GioTKD/IIoT_project

2 Workspace

2.1 PX4-Autopilot

The PX4-Autopilot is an open-source autopilot system. Developed by the PX4 Autopilot community. At its core, the PX4 controller combines powerful hardware with a software ecosystem. The hardware typically includes a flight controller board equipped with a microcontroller, sensors such as accelerometers and gyroscopes, and GPS modules. These components work in tandem to stabilize the drone, provide accurate position and orientation data, and enable precise control during flight. Two key features, for this project so much important on the px4 system are:

- Integration with ROS2 Humble
- Integration with QGC(QGroundControl)

To communicate with this component PX4 uses uORB messages for ROS2 and MAVLink for QGC

2.2 ROS2 Humble

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications. From drivers and state-of-the-art algorithms to powerful developer tools.

2.3 uXRCE-DDS

For the communication between ROS2 and PX4, it is mandatory to use a middleware to allow uORB messages to publish and subscribe to topics. This provides a fast and reliable integration between PX4 and ROS 2 and makes it much easier for ROS 2 applications to get vehicle information and send commands. All the set of uORB messages definitions is associated in the px4_msgs branch which as we said in the introduction is mandatory for this project.

Figure 1: Enter Caption

2.4 MAVLink Messaging

MAVLink is a very lightweight messaging protocol designed for the drone ecosystem. In this project, we use it to communicate with QGC but it could

be used also for other ground stations. The protocol defines several standard messages and microservices for exchanging data

2.4.1 Microservices

The MAVLink "microservices" define higher-level protocols that MAVLink systems can adopt to better inter-operate. For example, PX4 uses it to share a common Command Protocol for sending point-to-point messages that require an ACK. The microservices are used to exchange many types of data, including parameters, missions, trajectories, images, and other files. If the data can be far more extensive than 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. So take for example COMMAND_LONG which is the most used in the project:

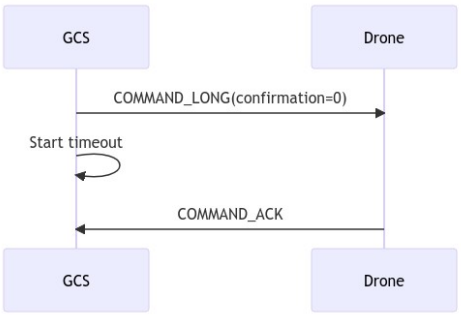

Figure 2: Message Exchange

As we can see we have the GCS(Ground Control Station) which sends COMMAND_LONG to the drone and receives back an ACK, here we have the parameters for COMMAND_LONG:

Field Name	Type	Values	Description
target_system	uint8_t		System which should execute the command
target_component	uint8_t		Component which should execute the command, 0 for all components
command	uint16_t	MAV_CMD	Command ID (of command to send).
confirmation	uint8_t		0: First transmission of this command. 1-255: Confirmation transmissions (e.g. for kill command)
param1	float		Parameter 1 (for the specific command).
param2	float		Parameter 2 (for the specific command).
param3	float		Parameter 3 (for the specific command).
param4	float		Parameter 4 (for the specific command).
param5	float		Parameter 5 (for the specific command).
param6	float		Parameter 6 (for the specific command).
param7	float		Parameter 7 (for the specific command).

Figure 3: Params for COMMAND_LONG

2.5 Gazebo-Classic

As a simulator was used Gazebo-Classic, which is a powerful 3D simulation environment for autonomous robots that is particularly suitable for testing object avoidance and computer vision. This project is used with a SITL(Software In the Loop simulation)simulation, where the flight stack runs on a computer (either the same computer or another computer on the same network), and a single/multi vehicle.

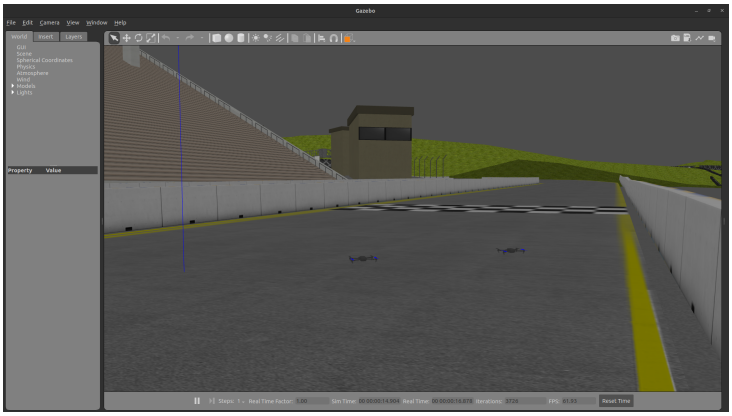

Figure 4: 2 Drones in a custom world

2.6 QGroundControl

QGroundControl could be defined as a GCS(Ground Control Station) that provides full flight control and vehicle for setup PX4.

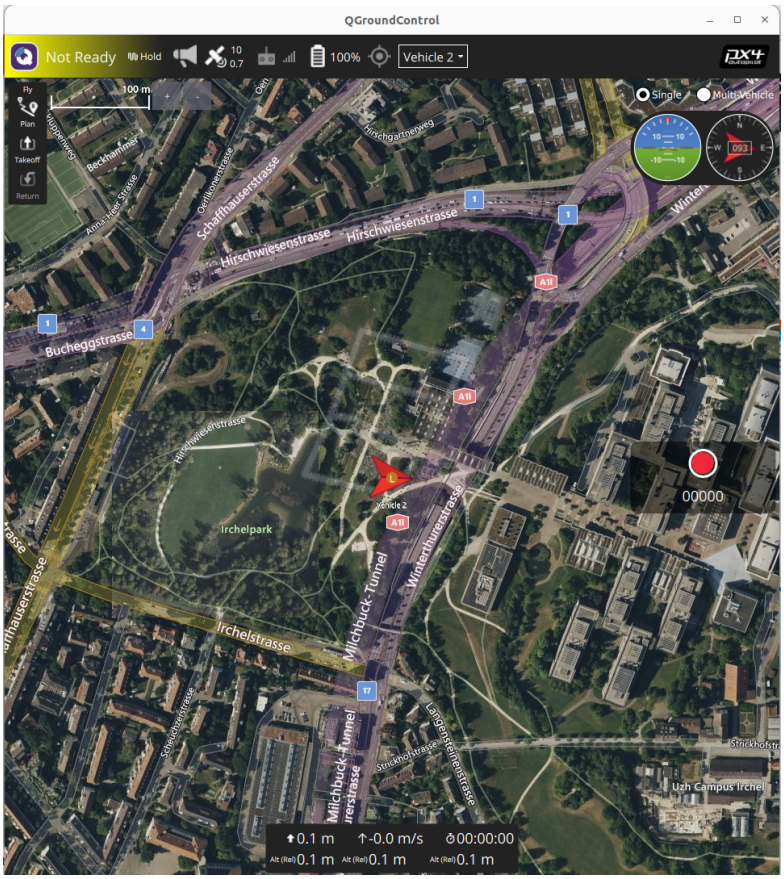

Figure 5: QGC Interface

2.7 Wireshark

Wireshark is maybe the most famous packet sniffer used for network analysis. For this project, a .lua plugin was used to trace the MAVLink packets that were exchanged by PX4 and QGC.

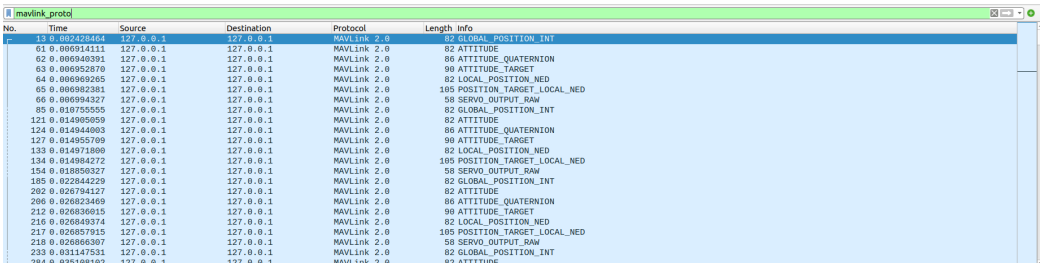

Figure 6: Wireshark

2.8 MAVSdk

MAVSdk is a collection of libraries for various programming languages to interface with MAVLink systems, in this project the language used is Python. Using this was possible to send MAVLink messages from PX4 to QGC to arm, take, set waypoints, and plan a mission ... for the drones.

```
import asyncio
from mavsdk import System
from mavsdk.mission import (MissionItem, MissionPlan)

async def run():
    drone = System(sysid=1)
    await drone.connect(system_address="udp://:14541")
    sysid = drone._sysid
    compid = drone._compid
    print("sysid: {sysid}")
    print("compid: {compid}")

    await drone.action.arm()

    mission_items = []

    mission_items.append(MissionItem(38.1608227, -122.4535923, 15, 5,
     True, float('nan'), float('nan'), MissionItem.CameraAction.NONE,
     float('nan'),
     float('nan'),float('nan'),float('nan'),float('nan')))

    mission_items.append(MissionItem(38.1603321, -122.4531431, 15, 5,
```

```
        True, float('nan'), float('nan'), MissionItem.CameraAction.NONE,
        float('nan'),
        float('nan'),float('nan'),float('nan'),float('nan')))

    mission_items.append(MissionItem(38.1586948, -122.4524834, 15, 5,
        True, float('nan'), float('nan'), MissionItem.CameraAction.NONE,
        float('nan'),
        float('nan'),float('nan'),float('nan'),float('nan')))

    mission_items.append(MissionItem(38.1588013, -122.4518987, 15, 5,
        True, float('nan'), float('nan'), MissionItem.CameraAction.NONE,
        float('nan'),
        float('nan'),float('nan'),float('nan'),float('nan')))

    mission_items.append(MissionItem(38.1614728, -122.4528851, 15, 5,
        True, float('nan'), float('nan'), MissionItem.CameraAction.NONE,
        float('nan'),
        float('nan'),float('nan'),float('nan'),float('nan')))
    # ... aggiungi altri waypoint come necessario

    mission_plan = MissionPlan(mission_items)
    print(drone)
    await drone.mission.upload_mission(mission_plan)
    await drone.mission.start_mission()

asyncio.get_event_loop().run_until_complete(run())
```

Take as an example our "mission.py" file. In this script, we could see: first of all the import, which are asyncio, and mavsdk system and from mission MissionItem and MissionPlan. The goal of this script is to plan a mission for drone 1, the most important part to understand how it works, is the MissionItem. If we take this from the documentation we can see the structure:

class mavsdk.mission.**MissionItem**(*latitude_deg*, *longitude_deg*, *relative_altitude_m*, *speed_m_s*, *is_fly_through*, *gimbal_pitch_deg*, *gimbal_yaw_deg*, *camera_action*, *loiter_time_s*, *camera_photo_interval_s*, *acceptance_radius_m*, *yaw_deg*, *camera_photo_distance_m*)

Bases: **object**

Type representing a mission item.

A MissionItem can contain a position and/or actions. Mission items are building blocks to assemble a mission, which can be sent to (or received from) a system. They cannot be used independently.

Parameters:
- **latitude_deg** (*double*) – Latitude in degrees (range: -90 to +90)
- **longitude_deg** (*double*) – Longitude in degrees (range: -180 to +180)
- **relative_altitude_m** (*float*) – Altitude relative to takeoff altitude in metres
- **speed_m_s** (*float*) – Speed to use after this mission item (in metres/second)
- **is_fly_through** (*bool*) – True will make the drone fly through without stopping, while false will make the drone stop on the waypoint
- **gimbal_pitch_deg** (*float*) – Gimbal pitch (in degrees)
- **gimbal_yaw_deg** (*float*) – Gimbal yaw (in degrees)
- **camera_action** (*CameraAction*) – Camera action to trigger at this mission item
- **loiter_time_s** (*float*) – Loiter time (in seconds)
- **camera_photo_interval_s** (*double*) – Camera photo interval to use after this mission item (in seconds)
- **acceptance_radius_m** (*float*) – Radius for completing a mission item (in metres)
- **yaw_deg** (*float*) – Absolute yaw angle (in degrees)
- **camera_photo_distance_m** (*float*) – Camera photo distance to use after this mission item (in meters)

Figure 7: MissionItem Structure

So to plan a mission we have to pass these values, and after this is possible to store it in a list and run the script until complete.

3 Simulation

As said in the introduction, in this project were reproduced 3 scenarios:

- Drone Backflip
- Offboard control
- Mission for 2 drones in a non-empty world

Let's analyze each scenario, but first, we must see what happened at the launch of the project: it will open some terminals:

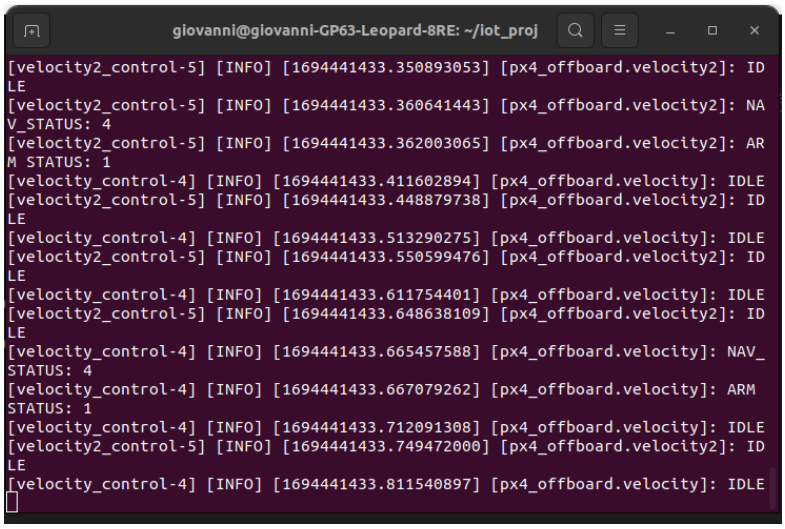

Figure 8: Velocity

Here we can see the status of the drone, in this example, both drones are landed so their status is "IDLE"

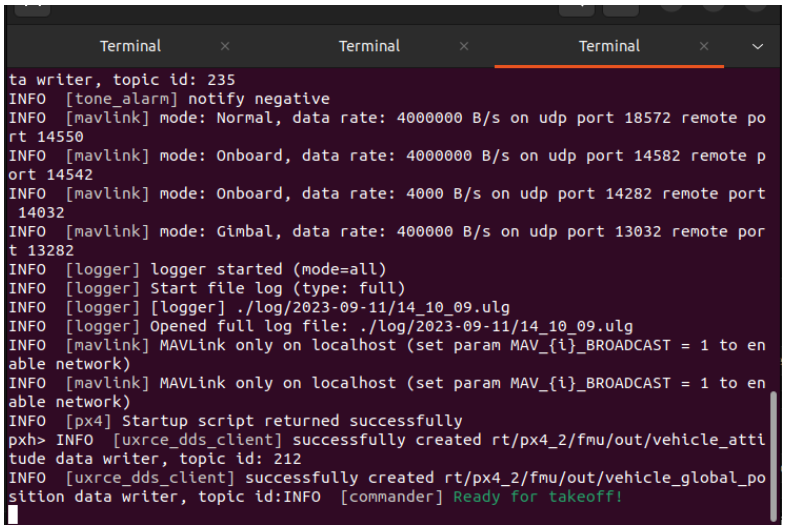

Figure 9: Gazebo instance

Here we can see the gazebo instances, if we take a look at the last lines, we can see the creation of global_position topic and the command that says

"Ready for takeoff" It means that the drone is ready to perform actions, of course, we have two different of this, for the two different drones.

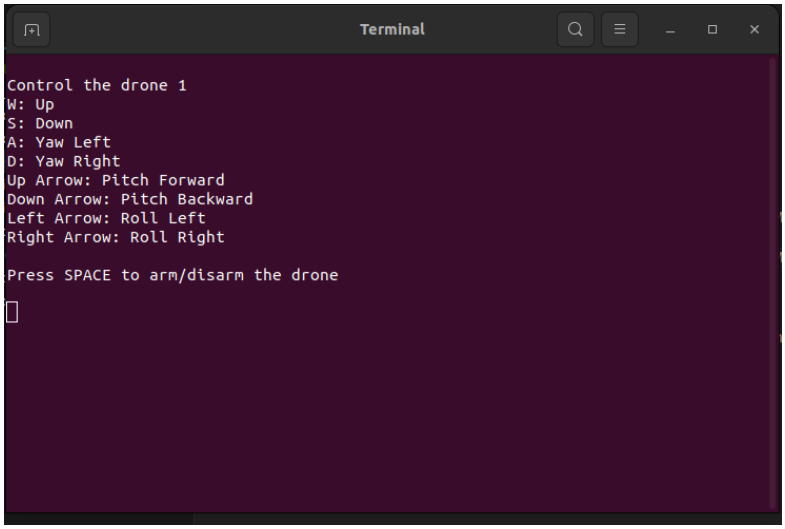

Figure 10: Control panel

Here we have the control panel to perform the offboard actions, so with a keybinding, we can move the drone with our keyboard

Figure 11: uXRCE-DDS

Here (as we already seen in the previous chapter) is the middleware to publish/subscribe to topics

3.1 Offboard Control

For the Offboard Control scenario, we move the drone by the control panel seen before.

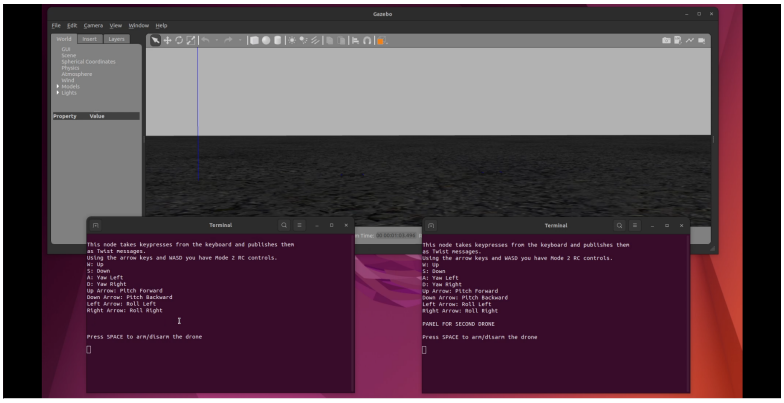

Figure 12: Offboard Control

3.2 Backflip

For the backflip scenario, we need to arm the drone, send a takeoff command from QGC to the altitude of 10-11 meters, and then launch the script flip.py (look for it in the Github repo)

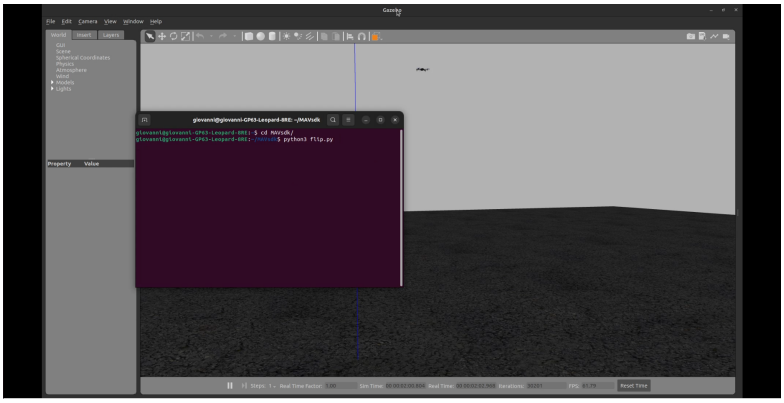

Figure 13: Backflip

3.3 Mission Plan for 2 drones

For this scenario, we have 2 drones on the world sonoma_raceway. Using a Python script that allows us to plan a mission, we perform a little flight over the track.

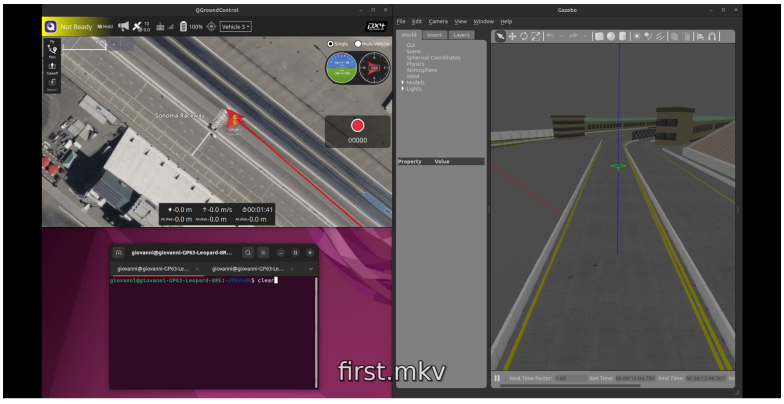

Figure 14: Mission Plan

4 Conclusions

In conclusion, the IIoT (Industrial Internet of Things) project presented in this document demonstrates the integration of various technologies and tools to simulate and control drones. Through the use of ROS2, PX4-Autopilot,

Gazebo, QGroundControl, uXRCE-DDS, MAVLink messaging, Wireshark, and MAVSdk, we have successfully achieved several key scenarios, including drone backflips, offboard control, and mission planning for multiple drones in a dynamic environment.