UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Magistrale in
Engineering and Computer Science

Industrial IoT

Project:

Perlustrazione di droni swarm tramite algoritmo greedy

Student:
Miano Alberto

ANNO ACCADEMICO 2024-2025

Contents

Chapter 1

Introduction

This project aims to manage one or more drones autonomously in a simulated environment to explore an urban area (e.g., a neighborhood). The main goal is to ensure that the area is explored efficiently and in a distributed way, using full flight automation and intelligent task management. To reach this goal, the system must be able to:

- Simulate a surveillance mission in a realistic urban environment using PX4 and Gazebo.
- Manage an arbitrary number of drones, each one uniquely identified and configured with specific parameters.
- Automatically divide the waypoints to explore among the available drones.
- Optimize this division using a greedy algorithm, minimizing the total distance traveled by each drone.

Chapter 2

System Architecture

2.1 Simulated Infrastructure

The environment is based on:

- **PX4 SITL** for simulating the drone firmware.
- **Gazebo Classic** for the visual simulation.
- **MAVROS** as the interface between ROS and MAVLink.
- **QGroundControl** for manual monitoring and debugging.

2.1.1 PX4 SITL

PX4 is an open-source firmware for drones that allows flight control in both real and simulated scenarios. It is used by a large community and supports numerous types of air, land and marine vehicles.
`PX4 Software-In-The-Loop (SITL)` is a firmware mode designed to test the entire system in a simulated environment. In this configuration, PX4 runs directly on a computer, simulating the behavior of the real flight controller. Communication with simulators such as Gazebo takes place through MAVLink protocols and allows you to validate flight missions, control logics, and software integration without risking damage to physical hardware.

2.1.2 Gazebo Classic

`Gazebo Classic` is an advanced 3D simulator that allows the visualization of the environment and the drone in real time. It is used to simulate realis-

tic scenarios such as urban neighborhoods, thus allowing visual validation of the mission.

2.1.3 MAVROS

`MAVROS` It is a bridge package between ROS 2 and the MAVLink protocol, the standard communication language used by PX4 drones. This component plays a crucial role in two-way communication between the ROS nodes and the drone firmware. Through MAVROS it is possible, for example, to send mission commands (waypoints), arm or disarm the drone, change flight mode (e.g. `AUTO.MISSION`, `OFFBOARD`, `MANUAL`), read the status of the drone, access telemetry data such as position and speed, and monitor the status of the system. Thanks to its flexibility, MAVROS allows perfect integration between autonomous algorithms developed in ROS 2 and the real/simulated world managed by PX4.

2.1.4 QGC

QGroundControl is a graphical interface for manual control and drone monitoring. It is used to visually verify the behavior of the drone, receive real-time feedback and debug any problems during the simulation.

2.2 File di setup

To simplify the execution of the simulated infrastructure, a bash script called `setupfile.sh` was created. This script allows you to start all the necessary components, with a single command, automatically opening three separate terminals:

- one for PX4 SITL, which starts the simulation with Gazebo in headless mode, or without graphical interface, to save computational resources within the VM, which could otherwise slow down considerably due to the load generated by the visual simulation;
- one for QGroundControl, for visual and diagnostic monitoring;
- one for MAVROS, which acts as a bridge between PX4 and ROS2

Before executing the command to start the script it is necessary to position yourself inside the PX4 folder (e.g. from the PX4-Autopilot home cd) and set the starting coordinates using the environment variables:

```
export PX4_HOME_LAT=<latitudine>
export PX4_HOME_LON=<longitudine>
export PX4_HOME_ALT=<altitudine>
```

This configuration allows the drone to start from a real geographical location. The command to run the bash script is:

```
./setupfile.sh
```

```
#!/bin/bash

gnome-terminal --title="PX4 SITL" -- bash -c "cd ~/PX4-Autopilot &&
    HEADLESS=1 make px4_sitl_default gazebo-classic_iris; exec bash"

gnome-terminal --title="QGC" -- bash -c "./QGroundControl.AppImage;
    exec bash"

gnome-terminal --title="MAVROS" -- bash -c "ros2 launch mavros px4.
    launch fcu_url:=\"udp://:14540@127.0.0.1:14580\"; exec bash"
```

Listing 2.1: Script setupfile.sh

2.3 ROS2 node

The heart of the drone's autonomy is represented by the ROS 2 `swarm.py` node, which performs several crucial functions for mission management. This node has been written to be parametric, that is, it can be executed on different drones by providing dynamic parameters from the command line.

`Note:` if you add a new custom node (as in this case swarm.py), you must register it in the setup.py file of the ROS 2 package, in the entry_points section, so that ROS 2 recognizes it as an executable.

```
entry_points={
        'console_scripts': [
            "swarm = project_pkg.swarm:main"
            ],
    },
```

Listing 2.2: Adding new node

Node launch command without parameters:

```
cd ros2_ws
colcon build
ros2 run projcet_pkg swarm
```

Node launch command with one or more parameters:

```
cd ros2_ws
colcon build
ros2 run project_pkg swarm --ros-args -p namespace:=drone0 -p drone_id
   :=0
```

2.3.1 Initialization and loading of parameters

In this first phase, the node declares and retrieves the fundamental parameters from the ROS 2 launch, including the drone id, the namespace, the starting location (lat, lon, alt) and the path to the JSON file containing the waypoints. If the file is present and valid, it only loads the waypoints assigned to that drone. If it is not present, upload a default list. The same goes for the other parameters, as they have a default value in case they are not passed during the launch of the node.

```
self.declare_parameter('drone_id', 0)
        self.declare_parameter('namespace', '')
        self.declare_parameter('base_latitude', 38.1838)
        self.declare_parameter('base_longitude', 15.5501)
        self.declare_parameter('base_altitude', 30.0)
        self.declare_parameter('waypoint_file', '')

        self.drone_id = self.get_parameter('drone_id').value
        self.namespace = self.get_parameter('namespace').value
        self.base_lat = self.get_parameter('base_latitude').value
        self.base_lon = self.get_parameter('base_longitude').value
        self.base_alt = self.get_parameter('base_altitude').value
        waypoint_file = self.get_parameter('waypoint_file').value

        self.get_logger().info(f"Drone {self.drone_id} started with
   namespace: '{self.namespace}'")
        self.get_logger().info(f"Base coordinates: {self.base_lat:.6f},
    {self.base_lon:.6f}, {self.base_alt:.1f}m")

        self.flight_altitude = self.base_alt + (self.drone_id * 5.0)

        if waypoint_file and os.path.exists(waypoint_file):
            with open(waypoint_file, 'r') as f:
                all_waypoints = json.load(f)
                if self.drone_id < len(all_waypoints):
                    self.waypoints_gps = all_waypoints[self.drone_id]['
   waypoints']
                else:
                    self.get_logger().error(f"Drone ID {self.drone_id}
   not found in waypoint file")
                    self.waypoints_gps = []
        else:
            self.get_logger().warn("No waypoint file provided or file
   not found. Using default waypoints")
            self.waypoints_gps = [
                {"lat": 38.1833, "lon": 15.5511, "alt": 30.0},
```

```
                {"lat": 38.1826, "lon": 15.5505, "alt": 30.0},
                {"lat": 38.1832, "lon": 15.5496, "alt": 30.0},
                {"lat": 38.1838, "lon": 15.5515, "alt": 30.0},
                {"lat": 38.1842, "lon": 15.5508, "alt": 30.0},
                {"lat": 38.1844, "lon": 15.5502, "alt": 30.0},
                {"lat": 38.1843, "lon": 15.5493, "alt": 30.0},
                {"lat": 38.1836, "lon": 15.5492, "alt": 30.0},
                {"lat": self.base_lat, "lon": self.base_lon, "alt":
    self.flight_altitude},
            ]
```

Listing 2.3: Parameter initialization

2.3.2 Configuration of MAVROS topics and services

To allow the `swarm.py` node to control multiple drones independently, a dynamic configuration logic of the names of the MAVROS topics and services is used. If a `namespace` is specified, all topics and services are prefixed with it; otherwise, standard names are used.

The logic implemented is as follows:

```
if self.namespace:
    local_pos_topic = f'/{self.namespace}/mavros/local_position/pose'
    state_topic = f'/{self.namespace}/mavros/state'
    waypoint_push_service = f'/{self.namespace}/mavros/mission/push'
    set_mode_service = f'/{self.namespace}/mavros/set_mode'
    arming_service = f'/{self.namespace}/mavros/cmd/arming'
else:
    local_pos_topic = '/mavros/local_position/pose'
    state_topic = '/mavros/state'
    waypoint_push_service = '/mavros/mission/push'
    set_mode_service = '/mavros/set_mode'
    arming_service = '/mavros/cmd/arming'
```

Listing 2.4: MAVROS topics and services

This configuration allows you to isolate the interface of each drone, avoiding conflicts and allowing simultaneous execution.

- `/mavros/local_position/pose`: Topic of type `PoseStamped`, published by MAVROS, which provides the current position of the drone in local space (coordinates x, y, z relative to the take-off point). It is used to monitor the movement of the drone in real time.
- `/mavros/state`: Topic of type `State`, which transmits the current state of the drone. It includes essential information such as whether the drone is armed, the active flight mode (e.g. `MANUAL`, `AUTO.MISSION`, `OFFBOARD`) and whether it is connected to the flight controller. The

ROS node uses this information to know when to start a mission or when to wait.

- `/mavros/mission/push`: Service of type `WaypointPush`, which allows you to send a list of waypoints (GPS coordinates) to the drone. The drone will follow these points in sequence during the `AUTO.MISSION` mode.
- `/mavros/set_mode`: `SetMode` type service, used to change the drone's flight mode. In the project it is used to set the `AUTO.MISSION` mode, in which the drone automatically follows the loaded mission.
- `/mavros/cmd/arming`: Service of type `CommandBool`, which allows you to arm or disarm the drone. The drone must be armed to be able to fly.

2.3.3 Subscriptions and ROS2 clients

This part of the ROS node defines the communication channels with the drone through **topic** and **services** provided by the MAVROS package. Communication takes place according to the `publish/subscribe` paradigm for real-time data and via *service call* for specific commands.

The following code creates the necessary `subscriber` and `client`:

```
self.local_pos_sub = self.create_subscription(PoseStamped,
    local_pos_topic, self.local_pos_callback, qos_profile_sensor_data)
self.state_sub = self.create_subscription(State, state_topic, self.
    state_callback, 10)

self.wp_push_client = self.create_client(WaypointPush,
    waypoint_push_service)
self.set_mode_client = self.create_client(SetMode, set_mode_service)
self.arming_client = self.create_client(CommandBool, arming_service)
```

Listing 2.5: Creation of MAVROS subscriber and client

The code logic can be divided into:

Subscriber

- **Local position of the drone** (`/mavros/local_position/pose`)
  The subscriber `self.local_pos_sub` receives messages of type `PoseStamped` that contain the current position of the drone in local coordinates (x,

y, z). The data is processed by the `self.local_pos_callback` function and is saved for use at other points in the node (e.g. debugging, logging or autonomous decisions).

- **Drone status** (`/mavros/state`)
  The subscriber `self.state_sub` receives messages of type `State`, which provide information about the state of the drone. The callback `self.state_callb` updates an internal variable that maintains the current state of the drone, used to verify that the system is ready before starting a mission.

Service Client

- **Loading mission** (`/mavros/mission/push`)
  self.wp_push_client creates a service client that allows you to send a list of waypoints (mission) to the drone. It is used in the `send_mission()` function to load the path to be executed.

- **Change of flight mode** (`/mavros/set_mode`)
  self.set_mode_client is the client for the service that sets the drone's flight mode, for example by switching to `AUTO.MISSION` to start the mission automatically. This service is called after `arming`.

- **Arming** (`/mavros/cmd/arming`)
  self.arming_client is the service client that allows you to arm or disarm the drone's engines. It is essential to authorize the take-off and execution of the mission.

2.3.4 Status and position callback

This section defines the callback functions associated with drone subscribers. Callbacks are automatically activated every time a new message is received on the relative MAVROS topics. In particular, these functions allow you to monitor in real time the status of the drone and its local position with respect to the starting point.

```
1 def state_callback(self, msg):
2     """Handle state updates from MAVROS"""
3     if msg.mode != self.current_state.mode:
4         self.get_logger().info(f"Drone {self.drone_id} - Mode changed:
  {msg.mode}")
```

```
5
6       if msg.connected != self.current_state.connected:
7           self.get_logger().info(f"Drone {self.drone_id} - Connection: {
    msg.connected}")
8
9       if msg.armed != self.current_state.armed:
10          self.get_logger().info(f"Drone {self.drone_id} - Armed: {msg.
    armed}")
11
12      self.current_state = msg
13
14  def local_pos_callback(self, msg):
15      """Handle local position updates"""
16      self.current_position = msg.pose.position
```

Listing 2.6: Callback for status and location updates

The two functions perform the following operations:

- **state_callback function**
  This function manages the messages received on the topic `/mavros/state`, which provide critical information about the state of the drone. In particular:
  - Detects any flight mode changes (`msg.mode`) compared to the previous state and prints them on the log;
  - Notification of variations in the connection with MAVROS (`msg.connected`);
  - Record every change in the drone's arming state (`msg.armed`).

  When finished, the entire received message is saved in the variable `self.current_state` to be accessible in the rest of the node.

- `local_pos_callback` **function**
  Manages messages on the topic `/mavros/local_position/pose`, containing the current local position of the drone in Cartesian coordinates. The internal variable `self.current_position` is updated, used to monitor the movement of the drone in real time or for debugging purposes.

2.3.5 Mission connection and activation

This section describes the fundamental functions for the operational start of the drone. They manage the connection with MAVROS, the availability of services, the sending of mission waypoints, the armament of the vehicle and the setting of the automatic flight mode.

```
def wait_for_mavros_connection(self):
    self.get_logger().info(f"Drone {self.drone_id}: Waiting for MAVROS connection...")

    start_time = time.time()
    while not self.current_state.connected and (time.time() - start_time) < 30.0:
        rclpy.spin_once(self, timeout_sec=0.1)
        if not self.current_state.connected:
            self.get_logger().info(f"Drone {self.drone_id}: Waiting for MAVROS connection... ({time.time() - start_time:.1f}s)")

    if not self.current_state.connected:
        self.get_logger().error(f"Drone {self.drone_id}: Failed to connect to MAVROS after 30 seconds")
        return False

    self.get_logger().info(f"Drone {self.drone_id}: MAVROS connected successfully")
    return True

def wait_for_services(self):
    self.get_logger().info(f"Drone {self.drone_id}: Waiting for services...")

    services = [
        (self.wp_push_client, "waypoint push"),
        (self.set_mode_client, "set mode"),
        (self.arming_client, "arming")
    ]

    for client, service_name in services:
        timeout = 10.0
        if not client.wait_for_service(timeout_sec=timeout):
            self.get_logger().error(f"Drone {self.drone_id}: Service {service_name} not available after {timeout}s")
            return False
        self.get_logger().info(f"Drone {self.drone_id}: Service {service_name} available")

    return True

def send_mission(self):
    self.get_logger().info(f"Drone {self.drone_id}: Sending mission with {len(self.waypoints_gps)} waypoints...")

    waypoints = []
    for i, wp_gps in enumerate(self.waypoints_gps):
        wp = Waypoint()
        wp.frame = Waypoint.FRAME_GLOBAL_REL_ALT
        wp.command = 16  # NAV_WAYPOINT
        wp.is_current = (i == 0)
        wp.autocontinue = True
        wp.param1 = 0.0
        wp.param2 = 0.0
        wp.param3 = 0.0
        wp.param4 = 0.0
        wp.x_lat = wp_gps["lat"]
```

```
50          wp.y_long = wp_gps["lon"]
51          wp.z_alt = wp_gps["alt"]
52          waypoints.append(wp)
53          self.get_logger().info(f"Drone {self.drone_id}: Waypoint {i+1}:
        ({wp.x_lat:.6f}, {wp.y_long:.6f}, {wp.z_alt:.1f}m)")
54
55      req = WaypointPush.Request()
56      req.start_index = 0
57      req.waypoints = waypoints
58
59      future = self.wp_push_client.call_async(req)
60      rclpy.spin_until_future_complete(self, future, timeout_sec=10.0)
61
62      if future.result() and future.result().success:
63          self.get_logger().info(f"Drone {self.drone_id}: Mission pushed
       successfully!")
64          self.mission_sent = True
65          return True
66      else:
67          self.get_logger().error(f"Drone {self.drone_id}: Error in
       pushing mission")
68          return False
69
70  def arm_drone(self):
71      self.get_logger().info(f"Drone {self.drone_id}: Attempting to arm
       ...")
72
73      req = CommandBool.Request()
74      req.value = True
75      future = self.arming_client.call_async(req)
76      rclpy.spin_until_future_complete(self, future, timeout_sec=10.0)
77
78      if future.result() and future.result().success:
79          self.get_logger().info(f'Drone {self.drone_id}: Armed
       successfully')
80          return True
81      else:
82          self.get_logger().error(f'Drone {self.drone_id}: Unable to arm'
       )
83          return False
84
85  def set_auto_mission_mode(self):
86      self.get_logger().info(f"Drone {self.drone_id}: Setting AUTO.
       MISSION mode...")
87
88      req = SetMode.Request()
89      req.custom_mode = "AUTO.MISSION"
90      future = self.set_mode_client.call_async(req)
91      rclpy.spin_until_future_complete(self, future, timeout_sec=10.0)
92
93      if future.result() and future.result().mode_sent:
94          self.get_logger().info(f'Drone {self.drone_id}: AUTO.MISSION
       mode successfully set!')
95          return True
96      else:
97          self.get_logger().error(f'Drone {self.drone_id}: Error setting
       AUTO.MISSION mode.')
```

```
98          return False
```

Listing 2.7: Connection and start of the autonomous mission

- `wait_for_mavros_connection` **function**: Waits for MAVROS to establish the connection with PX4. Use the `connected` field of the status to verify the connection and provide a 30-second timeout to avoid endless waiting.

- `wait_for_services` **function**: Check the availability of essential services (mission loading, arming, mode change). Each service is waited up to a maximum of 10 seconds.

- `send_mission` **function**: Create the list of GPS waypoints for the drone and send them through the service `/mavros/mission/push`. Each waypoint is encoded according to the frame `GLOBAL_REL_ALT` with the command `NAV_WAYPOINT`.

  `GLOBAL_REL_ALT` indicates that the coordinates of the waypoint are expressed in geographical latitude and longitude (global coordinates) and the altitude is relative to the ground level. This frame is commonly used for missions in an outdoor environment, where it is important to specify the altitude of the drone with respect to the ground rather than the absolute altitude above sea level.

  `NAV_WAYPOINT` is a MAVLink command that tells the drone to navigate to a given waypoint. When the drone receives this command, it moves autonomously to the position indicated by the waypoint, following the trajectory calculated internally by the flight controller.

- `arm_drone` **function**: Invokes the `/mavros/cmd/arming` service to arm the drone and enable its flight.

- `set_auto_mission_mode` **function**: Change the drone's flight mode to `AUTO.MISSION`, a necessary condition for the drone to perform the mission automatically following the waypoints.

These functions constitute the operational core of the ROS 2 node and guarantee the autonomous and safe execution of the mission for each individual drone.

2.4 Multi-drone configuration

To extend control from a single drone to a multi-drone scenario, you need to create a custom launch.py file, which allows the simultaneous start of multiple instances of MAVROS and the ROS 2 control node. This launch file receives as an argument the number of drones to start (num_drones) and, for each of them, performs a separate configuration with dedicated namespaces and other parameters, thanks to the parameterized configuration of the node, avoiding conflicts between the different nodes and topics.

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, OpaqueFunction,
    TimerAction
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def launch_setup(context, *args, **kwargs):
    num_drones = int(LaunchConfiguration('num_drones').perform(context)
    )
    nodes = []

    for i in range(num_drones):
        namespace = f"drone{i}" if num_drones>1 else ""
        fcu_url = f"udp://:{14540 + i}@127.0.0.1:{14580 + i}"
        target_system = i + 1

        # MAVROS Node
        mavros_node = Node(
            package='mavros',
            executable='mavros_node',
            namespace=namespace,
            name=f'mavros_{i}',
            output='screen',
            parameters=[
                {'fcu_url': fcu_url},
                {'target_system_id': target_system},
                {'target_component_id': 1},
                {'system_id': target_system},
                {'component_id': 1},
                {'use_sim_time': True},
            ]
        )
        nodes.append(mavros_node)

        drone_node_delayed = TimerAction(
            period=10.0,
          actions=[
          Node(
              package='project_pkg',
              executable='swarm',
              namespace=namespace,
              name=f'drone_controller_{i}',
              output='screen',
              parameters=[
```

```
                {'drone_id': i},
                {'namespace': namespace},
                {'base_latitude': 38.1838},
                {'base_longitude': 15.5501},
                {'base_altitude': 30.0},
                {'waypoint_file': '/home/user/ros2_ws/src/Project_PX4_IIoT/project_pkg/waypoint_file.json'},  # Opzionale
            ]
        )
      ]
    )
      nodes.append(drone_node_delayed)

  return nodes

def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument('num_drones', default_value='1'),
        OpaqueFunction(function=launch_setup)
    ])
```

Listing 2.8: Launch file drone*launch.py*

The execution of the command:

```
ros2 launch project_pkg drone_launch.py num_drones:=2
```

allows you to automatically launch both MAVROS and the ROS 2 node by starting multiple independent instances in parallel, each associated with a virtual drone in the simulator.

Chapter 3

Multi-drone mission planning system

3.1 Flask server and API

To reduce VM load, a Flask server was created on an external computer. This server exposes a REST API that:

- Receives as input via POST the total waypoints and the number of drones.

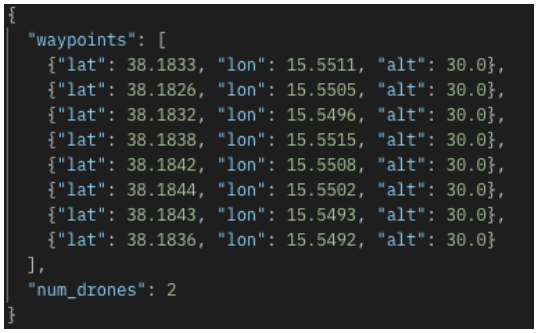

Figure 3.1: Example body API

- Apply a greedy algorithm to divide the waypoints between the drones, minimizing the distance traveled.
- Return a structured JSON with the waypoints divided by drone_id

```
from flask import Flask, request, jsonify
from greedy import greedy_assignment

app = Flask(__name__)

HOME_POSITION = {
    "lat": 38.1838,
    "lon": 15.5501,
    "alt": 30.0
}
```

```python

@app.route("/waypoints", methods=["POST"])
def waypoints():
    data = request.get_json()
    if not data or "waypoints" not in data or "num_drones" not in data:
        return jsonify({"error": "Missing 'waypoints' or 'num_drones'"
    }), 400

    waypoints = data["waypoints"]
    num_drones = data["num_drones"]

    try:
        result = greedy_assignment(waypoints, num_drones, HOME_POSITION
    )
        return jsonify(result)
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5050, debug=True)
```

Listing 3.1: Flask server

To run the Flask server, having already positioned ourselves on the folder where the server is present, just type the following command:

```
python3 server_flask.py
```

If everything is going well, the output should be as follows:

```
 * Serving Flask app 'server_flask'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5050
 * Running on http://192.168.0.179:5050
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 102-109-033
```

Figure 3.2: Output Server

3.2 Greedy algorithm for waypoint assignment

A greedy algorithm is an optimization strategy that makes optimal local decisions step by step, in the hope that these will lead to an optimal (or acceptable) overall solution. These algorithms are often used when it is important to have quick rather than perfect solutions, making them ideal for problems that need to be solved in real time or with limited computational resources.

In the context of this project, the greedy algorithm was used to divide the waypoints between the various drones available. The goal is to minimize

the distance traveled by each drone, starting from a common initial position called "HOME POSITION" and trying to assign each waypoint to the closest drone available.
The greedy algorithm works according to the following logic:

1. Each drone starts from a common home position.
2. As long as there are waypoints to assign:
   - It is calculated which drone is closest to each available waypoint.
   - The nearest waypoint is assigned to the corresponding drone.
3. At the end, the home position is added at the bottom of the list of each drone.

```
from typing import List, Dict
from geopy.distance import geodesic

def greedy_assignment(waypoints: List[Dict], num_drones: int,
    home_position: Dict) -> List[Dict]:
     drone_paths = [[] for _ in range(num_drones)]
     drone_totals = [0.0] * num_drones
     for wp in waypoints:
         distances = []
         for i in range(num_drones):
             last = drone_paths[i][-1] if drone_paths[i] else
    home_position
             dist = geodesic((last["lat"], last["lon"]), (wp["lat"], wp[
    "lon"])).meters
             distances.append(dist)

         chosen = distances.index(min(distances))
         drone_paths[chosen].append(wp)
         drone_totals[chosen] += distances[chosen]

     for i in range(num_drones):
         drone_paths[i].append(home_position)

     result = [
         {
             "drone_id": i,
             "waypoints": drone_paths[i]
         }
         for i in range(num_drones)
     ]

     return result
```

Listing 3.2: Greedy algorithm

3.3 API client in the VM

In the VM there is a Python script that:

- Send a request to the Flask API with the waypoints and number of drones.
- Receives the response containing the split waypoints.
- Writes or overwrites the waypoint_file.json file in a default directory.

```python
import requests
import json
import os

API_URL = "http://192.168.0.179:5050/waypoints"
OUTPUT_FILE = "waypoint_file.json"

try:
    num_drones = int(input("Inserisci il numero di droni: "))
    if num_drones <= 0:
        raise ValueError("Il numero di droni deve essere maggiore di 0.")
except ValueError as ve:
    print(f"[ERRORE] Input non valido: {ve}")
    exit(1)

waypoints = [
    {"lat": 38.1833, "lon": 15.5511, "alt": 30.0},
    {"lat": 38.1826, "lon": 15.5505, "alt": 30.0},
    {"lat": 38.1832, "lon": 15.5496, "alt": 30.0},
    {"lat": 38.1838, "lon": 15.5515, "alt": 30.0},
    {"lat": 38.1842, "lon": 15.5508, "alt": 30.0},
    {"lat": 38.1844, "lon": 15.5502, "alt": 30.0},
    {"lat": 38.1843, "lon": 15.5493, "alt": 30.0},
    {"lat": 38.1836, "lon": 15.5492, "alt": 30.0}
  ],

body = {
    "waypoints": waypoints,
    "num_drones": num_drones
}

try:
    response = requests.post(API_URL, json=body)
    response.raise_for_status()  # Solleva eccezione se HTTP != 200

    result = response.json()

    with open(OUTPUT_FILE, "w") as f:
        json.dump(result, f, indent=2)
    print(f"[OK] File '{OUTPUT_FILE}' creato/sovrascritto con successo.")

except requests.exceptions.RequestException as e:
```

```
    print(f"[ERRORE] Richiesta HTTP fallita: {e}")
except Exception as e:
    print(f"[ERRORE] Generico: {e}")
```

Listing 3.3: API client

To execute the script, having already been positioned on the folder where it is present, just type the following command:

```
python3 client.py
```

If everything goes well, the output should be: "[OK] waypoints_file.json file successfully created/overwritten."

3.4 Structure of the generated JSON file

The structure of the json that will be created is as follows:

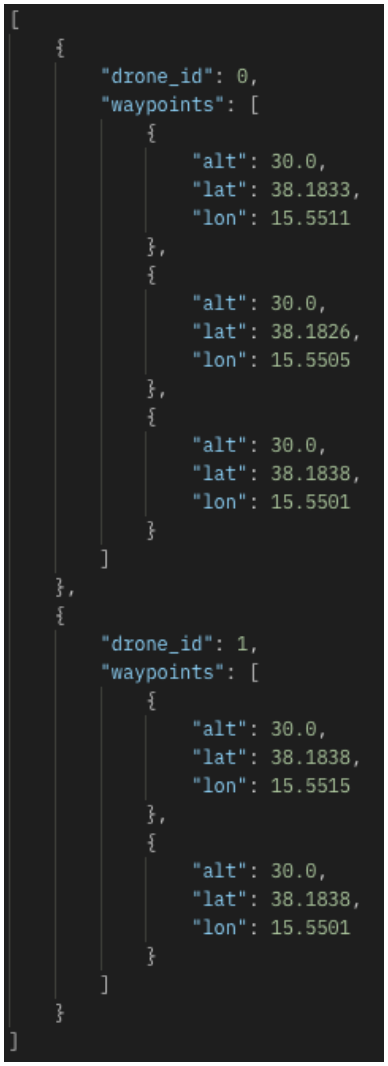

Figure 3.3: Server output

Chapter 4

Conclusions

In this project, a complete system for the autonomous flight management of one or more simulated drones was created, taking advantage of the integration between PX4, ROS 2, MAVROS and QGroundControl. The system has been designed to receive input missions consisting of a sequence of waypoints, processed through a greedy algorithm implemented on the server side, with the aim of dynamically optimizing the route based on the position of the drone and the points to be reached.

The logic of the system is divided between a ROS 2 node developed in Python, responsible for communication with MAVROS and direct control of the drone, and a Flask server that manages the generation and sending of missions in a compatible format. The ROS 2 node manages the main flight features: position reception, drone status monitoring (connection, flight mode, weapon status), mission loading and automatic mission mode activation. This modular architecture allows flexible and responsive management, also suitable for multi-drone contexts or adaptive missions.

The use of the PX4-Gazebo simulation environment made it possible to validate the system under controlled conditions, with the support of the QGroundControl interface for real-time monitoring of the drone's mission and behavior.

In conclusion, the work carried out has led to the creation of a reliable and reusable software infrastructure, capable of managing complex missions in an autonomous and modular way.

4.1 Future works

Currently, the proposed multi-drone configuration, although correctly implemented at the drone_launch.py level, presents operational difficulties. In particular, the control node is able to start correctly for each drone, but the connection with MAVROS does not take place as planned. This problem prevents the simultaneous control of multiple drones, limiting the interaction to only a single instance.

As a future development, it will be necessary to further analyze the management of UDP ports and MAVROS interfaces to ensure that each instance correctly uses distinct communication channels towards PX4. It will also be useful to implement automatic connection verification mechanisms for each drone, so as to quickly identify any malfunctions in the communication chain.