University of South-Eastern Norway
Faculty of Technology, Natural Sciences and Maritime Sciences
–

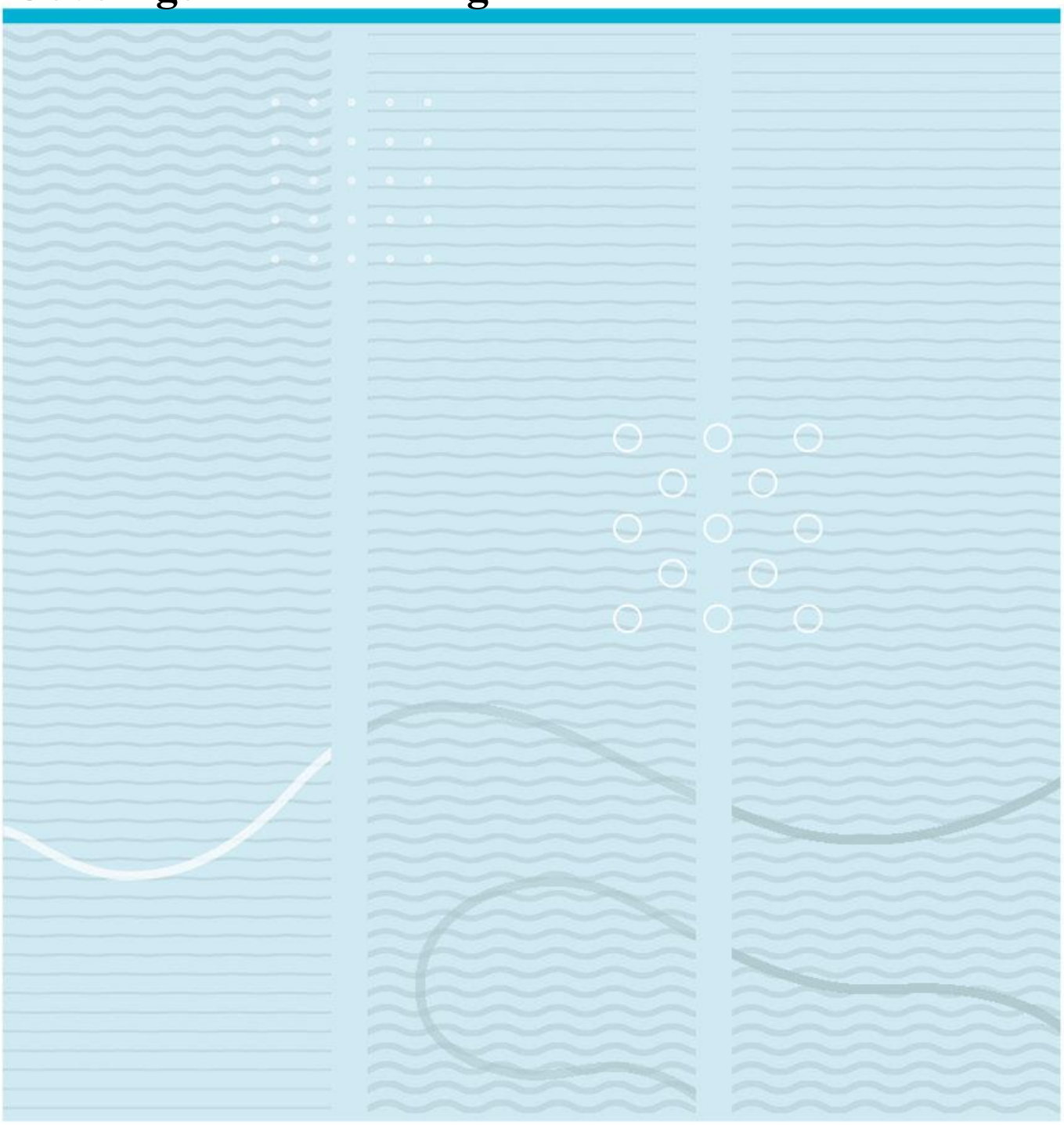
Master's Thesis

Study programme: Master of Science in Micro and Nano Systems Technology

Spring/Autumn 2024

Lluis Prior Sancho

Cooperative Multi-Modal Robots for Autonomous Coverage Path Planning



University of South-Eastern Norway
Faculty of Technology, Natural Sciences and Maritime Sciences
Department of Microsystems.
Raveien 215
NO-3184 Borre, Norway

http://www.usn.no

© 2024 Lluis Prior Sancho

Summary

This thesis is inspired from other works which investigates the collaboration between heterogeneous robots for coverage path planning in dynamic environments. This thesis aims to expand into real-life one of those strategies by using a couple of autonomous vehicles for exploration using coverage path planning.

The method proposed in this project considers a system with one Unmanned Ground Vehicle (UGV) and one Unmanned Aerial Vehicle (UAV), whose cooperation is controlled using a set of Robot Operating System nodes.

The system creates an exploration plan using the wavefront coverage path planning algorithm for each one of the robots. The system must consider the limitations of the robots: an inaccessible area for the Leo rover and the battery scarcity of the DJI Tello drone.

The system integrates a hot swapping subroutine. Once the battery of the drone goes below a threshold, the drone will fly on top of the rover and will land. After landing and outside the scope of this project, the battery can be replaced. With a fully recharged battery, the exploration continuous autonomously.

The location of the robots is obtained using an indoor localization system. This system consists in some antennas installed in the corners of the room that estimates the position of other antennas installed on top of the robots. To increase the accuracy of the estimation for the UGV, a Kalman filter is used. The location of drone is not filtered thus it produces a noisier exploration.

Even though the current studies are focusing in dynamic environments, due to the limitations of the hardware, this project will consider a completely known environment. The incapacity of the robots of obtaining new information regarding present obstacles is outside the scope of this work.

Simulation and real-life testing were conducted in order to validate the behaviour of the system. The resulting exploration in a 6.3x6.3 meters room took approximately 8.30 minutes, including one manual battery change using the hot swap subroutine. The movements done by the UGV and the use of a Kalman filter produced a clear coverage while the disturbances suffered on the drone plus the non-usage of the Kalman filter produced a noisy exploration result.

Preface

This master thesis explores the cooperative capabilities of multi-modal robot strategies inspired by the paper "Heterogeneous Multi-Robot Collaboration for Coverage Path Planning in Partially Known Dynamic Environments" by Gabriel G. R. de Castro et al.

My motivation for this project arises from the potential of robots to transcend human capabilities. Providing mobility to the newly proposed robotic solutions allows a fast, complex and effective solution to everyday life problems. This research presents an opportunity to contribute to the development of innovative solutions that can be applied to real-world scenarios.

The reader is assumed to have a working knowledge of PID controllers, which are essential for maintaining stability and precision in robotic movements, as well as velocity controllers, which manage the speed of the robots. This background knowledge will not be explained in detail during this report.

The computer, UAV, UGV and the indoor localization system have been assembled and configured, ensuring immediate use. This eliminates the need for any hardware assembly or initial setup.

The experimental tests were conducted in the laboratory C-31 at the University of South-Eastern Norway, Campus Vestfold.

Contents

1 Introduction

In recent years, the usage of robots has become a standard practice in our daily life tasks. From medical procedures to autonomous systems, the robotic solutions are being implemented in a multitude of scenarios. Some of the most significant and difficult challenges imply a multi-robot cooperation strategy. The integration of heterogeneous robotic solutions offers significant advantages in several applications such as rescue operations [1] or agricultural monitoring [2].

The use of multi-modal robotic cooperation solutions has been increasingly studied due to their potential efficacy. De Castro et al. [3] present a cooperation strategy for heterogeneous multi-robot collaboration. Extensive simulations using Gazebo sim demonstrate the benefits of combining multi-modal robots into the exploration framework. The hot swapping strategy presented can allow long and continuous exploration in addition to the challenge of the disconnection between batteries. The current research not only includes simulations but also evaluates the hot swapping strategy presented by de Castro et al. in real-life scenarios.

Yan, Jouandeau and Cherif [4] provide a survey and analysis of multi-robot coordination strategies, highlighting the potential of heterogeneous systems and emphasizing the possibility of outperforming homogeneous groups and improving the adaptability of them. The current work also uses heterogeneous robots and therefore its capabilities are enhanced recovering the flaws of the other type of robots, for example the battery scarcity of aerial robots.

Choset [5] provides a detailed survey of various coverage path planning techniques, including grid-based methods, which ensure thorough and systematic area coverage. Wavefront algorithms such as discussed by Zelinsky [6] are particularly effective due to their ability to propagate waves from a starting point to efficiently guide robots to a final destination. In this work, wavefront path-planning algorithm is used for generating the path of the robots. The usage of robust path planning techniques is crucial to develop a fast and efficient coverage, especially in urgent scenarios as the rescue missions.

The integration of multi-modal robots mixed with the use of advanced, robust coverage path planning techniques presents enhanced efficiency and effectiveness for autonomous exploration and coverage tasks.

2 Objectives

The aim of this project is to study, prepare and implement the necessary elements to create a heterogenous multi-robot system for coverage path planning. The system will consist in an Unmanned Ground Vehicle (UGV) and an Unmanned Aerial Vehicle (UAV) that autonomously explore a designated area. The robots will have an offline-computed path that will follow in order to complete the tasks while travelling from the origin to the goal position in a known environment. Each one of the paths will be adapted to the characteristics of the robot such as the accessibility to some areas or the size of the vehicle. It will also implement a battery hot swap mechanism. When the UAV's battery goes below a threshold, the hot swapping routine makes the UAV go to the UGV's position and land. At that moment, the battery of the UAV should be replaced and the system will continue with the exploration. The automatic battery exchange is not considered in the scope of this project thus the battery change will be done manually.

The system will rely upon the Robot Operating System (ROS). ROS is an open-source set of libraries and tools that help build robot applications [7]. Using ROS, a communication network will be established to connect the rover and the drone to the main processing units. Figure 2-1 shows a diagram of the connections. The Master PC will communicate with the UGV via WI-FI, providing the basic information to interpret the goals of the coverage path and computing the necessary commands to move the robot accordingly. The ROSbridge will be used for 2 purposes: enable a WI-FI connection and to work as converter between the different versions of ROS used in the project. On the other hand, the drone is connected to a Nvidia Jetson, doing the same job as the Master PC but for the drone. Further details will be provided in future sections.

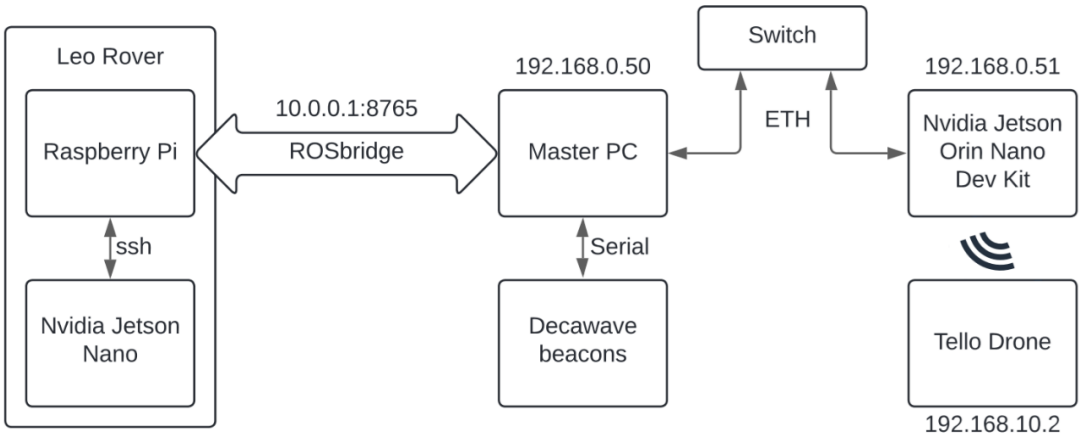


Figure 2-1 Communication diagram.

3 Materials and Methods

3.1 Coverage Path Planning

One practical use of autonomous robotic systems is the coverage path planning (CPP). Coverage path planning is the determination of an optimal path that a robot must follow in order to reach each point in an environment. There exist many planning algorithms [8] whose utility may vary depending on the limitations and characteristics of the environment.

The solution proposed in this thesis uses the wavefront algorithm. The wavefront algorithm consists in a grid-based propagated cost function, for instance the distance to the goal, from the target throughout the map. Once the cost map is computed, the robot should follow the cells with higher cost until the goal is reached. This algorithm presents an inconvenience as it produces large paths with many curves limiting the exactitude of the pose estimation and the odometry. As proposed in [6], the behaviour of the algorithm can be tuned if a discomfort factor is added to the cost function. The discomfort factor should reflect the degree of inconvenience of moving close to obstacles thus creating a more adventurous or conservative exploration nature.

$$J(x,y,\gamma) = D(x,y) + \sum_{n=0}^{n=N} \gamma \cdot O(x,y) \qquad [1]$$

The final cost function is described in the equation [1] where $J(x,y,\gamma)$ is the cost function, $D(x,y)$ is the Manhattan distance from the cell to the goal, N is the number of objects, γ is the discount factor and $O(x,y)$ is the obstacle distance.

The room where the test takes place has a dimension of 6.3x6.3 meters with a set of tables located in the middle of the room employed as obstacles. Due to the inaccuracy of the measurements, the placement of the tables and the grid-based style of the method, the actual position of the corners of the tables have been approximated. Figure 3-1 presents a scheme of the room where the red shape is the real position of the tables and the green one is the approximation.

Since the UGV and UAV have different capabilities, the UAV will be able to travel over the tables while the UGV should avoid them during the exploration. Other important characteristics for the determination of the map, and therefore the path, are the size of the vehicles and the size cells for the map. Additionally, but outside the scope of this

project, an exploration using aerial robots may also include the exploration on a third dimension, including objectives at different heights. The exploration using the UAV will consider constant altitude.

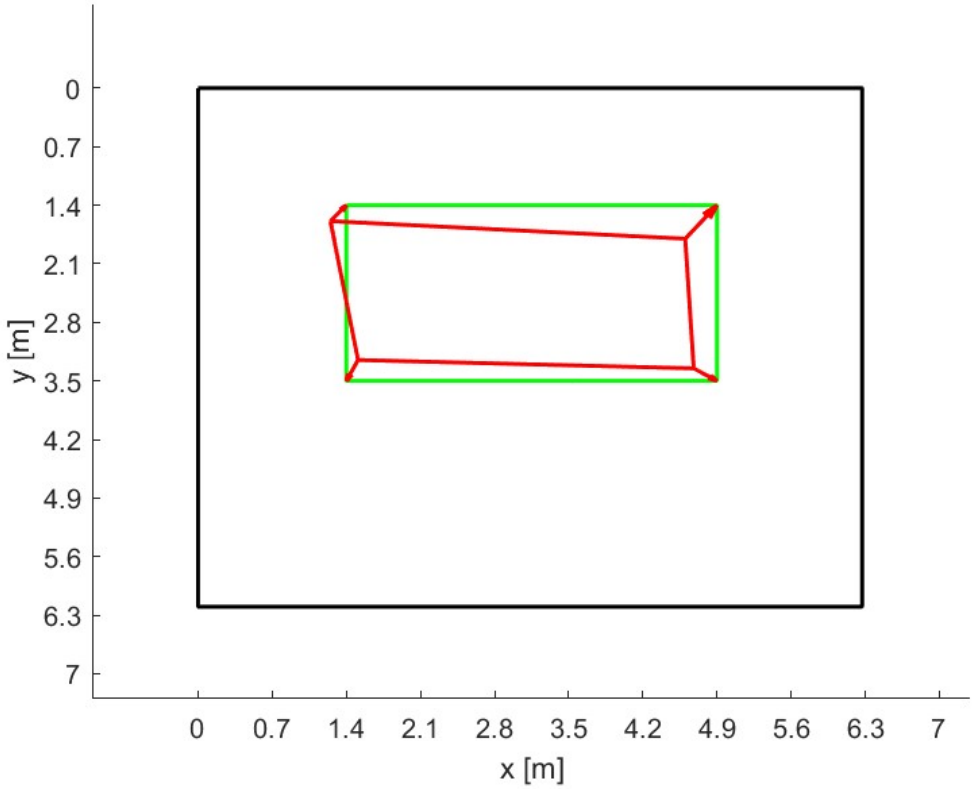


Figure 3-1 Room scheme.

The approximations on the location of the table have been strategically selected to fit a resolution of 0.7 meters for the grid. This grid should allow the rover to move twice in the top and side edges of the table and 4 times at the bottom part of the room. The drone can easily explore all the environment since the tables do not limit its exploration.

The wavefront algorithm requires to select an initial and final cell. The starting cell will be placed at the top left side of the room while the final destination is placed at the opposite corner of the room. The figure 3-2 shows the scheme with the initial and final positions, a green “S” for the initial cell and a red “G” for the goal position.

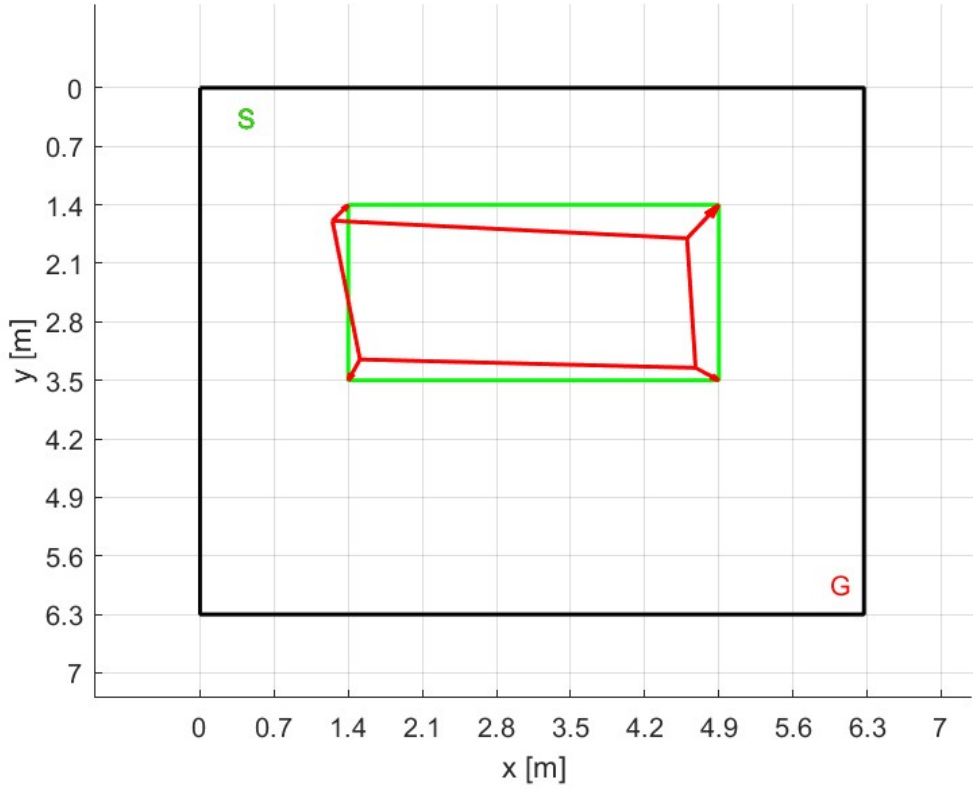


Figure 3-2 Room resolution and initial and final cells.

The program used to create the plans saves the index of the cells in a CSV file that later will be used in the ROS nodes.

3.2 Leo Rover

The Leo rover is an indoor/outdoor mobile robot development kit for researchers. It integrates a Raspberry Pi 4B 2GB board with Ubuntu 18.04 and ROS Melodic, making it easily programmable for prototyping and expandable with a different set of tools, sensors or boards. With a weight of 6.5 kg and a dimension of 447x433x249 mm the Leo rover reaches a maximum speed of 0.4 m/s and 60 deg/s. The rover is powered by a 11,1 Vdc 5000 mAh Li-Ion battery.

Figure 3-3 Leo rover. Source: leorover.tech.

Being open-source allows the community to create and share multiple already-made functionalities [9]. From navigation, SLAM (Simultaneous Location and Mapping) to object detection, the platforms allow the integration of extra sensors and boards. The rover used in this project includes an extra Nvidia Jetson Nano board to handle the lidar and the stereo camera, providing it with the necessary tools to execute autonomous tasks.

One of the peculiarities of the Leo rover is that it uses a skid-steering mechanism. This type of mechanisms controls the directions of the robotic platform by changing the speed of the left and right wheels rather than relying in any external mechanism to change the heading of the wheels. This system is used in multiple mobile platforms but it makes it more difficult to establish an accurate motion model due to the inherent sideslip of the wheels. The use of the camera and the lidar will provide us the necessary tools for a correct pose estimation.

3.3 Nvidia Jetson

Nvidia Jetson is the name of a series of computing boards capable of supporting high performance operations. Even with its reduced size, Nvidia Jetson boards are capable of creating powerful AI applications and software development in the field of robotics. Due to its memory, power management and high-speed interfaces, it makes them suitable in a wide range of scenarios. In this project, two different models are used: Jetson Orin Nano development Kit 8GB that will be used to control the UAV and the Jetson Nano, being the latest the one present in the Leo rover presented before.

Figure 3-4 Jetson Orin Nano dev kit.

The table 3-1 compares some of the basic specifications of the boards [10] [11].

Table 3-1 Technical specifications of the Nvidia Jetsons used.

Technical Specifications	Jetson Orin Nano Dev. Kit	Jetson Nano
GPU	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores (Max. Freq. 625 MHz)	128-core NVIDIA Maxwell™ architecture GPU
CPU	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores (Max. Freq. 625 MHz)	Quad-core ARM® Cortex®-A57 MPCore processor
Memory	8GB 128-bit LPDDR5 68GB/s	4GB 64-bit LPDDR4 25.6GB/s
Networking	1xGbE Connector	1x GbE
Power	7W - 15W	5W – 10 W
Other I/O	40-Pin Expansion Header(UART, SPI, I2S, I2C, GPIO) 12-pin button header	3x UART, 2x SPI, 2x I2S, 4x I2C, GPIOs

	4-pin fan header microSD Slot DC power jack	

Each board needs to be flashed with a NVIDIA JetPack SDK. It includes a board support package with the bootloader and Linux Kernel and Ubuntu as the basic operating system. Depending on the version of JetPack, the version of the application and features of the board may vary. The Jetson Orin Nano dev kit runs the Jetpack 6.0 [12] that includes the Linux Kernel 5.15, Ubuntu 22.04 and ROS2 – Humble installed on top. On the other hand, the Jetson Orin running in the Leo rover uses the Jetpack 4.6.1 [13] that includes the Linux Kernel 4.9, Ubuntu 18.04 and ROS1.

3.4 DJI Tello Drone

DJI [14] is one of the most known drone producers worldwide. Out of the variety of models, one of the drones they produce is the DJI Tello, a low cost, low flight time mini drone equipped with a couple of cameras (frontal and bottom) that makes it suitable for indoor or outdoor usage. The Tello drone offers 13 minutes of continuous flight time with a range of 100 meters and 720p video transmission.

Figure 3-5 DJI Tello drone. Source: djioslo.no.

One of the main characteristics of the Tello drone is the possibility of programming it using Python and UDP (user datagram protocol) communication while connecting it via WI-FI to a computer, making it a good candidate for the development of autonomous systems.

Regarding the physical constraints of the drone, one of the main advantages of using a quadrotor as UAV is the possibility of moving the whole mobile robot in any direction and in any orientation. This opens a wide range of control opportunities and techniques in contrary to the limited range of the UGV.

3.5 Decawave

One of the main drawbacks in the autonomous exploration is the location of the autonomous vehicles. If the system is not capable of knowing where the actual robots are, it is impossible to make them move to their respective goals. There exist multiple approaches to solve this problem but in this project two alternatives have been taken into account: the use of an indoor localization system or the use of local odometry to estimate the change in position of the robots.

3.5.1 Indoor Localization System

The first alternative considered is the indoor localization system. Similarly to the Global Navigation Satellite System (GNSS) but in a lower scale, an indoor localization system relies on several antennas working as beacons to send signals to a receiver that is installed on top of the robots. Knowing the position of the transmitters and the time the signals use to communicate, the location of the receiver can be estimated.

This type of indoor location systems can be purchased for plug-and-play. Specifically, the model used is the Qorvo DWM1001. It provides a 3-axis accelerometer motion with an accuracy within 20 cm while keeping the minimum consumption. A set of four transmitters have been installed in the corners of the room where this project has been developed.

Figure 3-6 Decawave antenna.

Additionally, two more antennas are used as receivers, one in the Leo rover and another one on top of the Tello drone. The one for the Tello drone has been modified to make it smaller, add an attachable support of the drone, and to include a small battery.

Figure 3-7 Decawave antenna for the drone.

This type of systems can be useful to get the approximate location of the robot inside the room but it does not provide enough accuracy for the Leo rover. Furthermore, the accuracy of the system is altitude dependant with the receiver and of the objects between the transmitters and the receivers. For the Tello drone, this technique is good enough due to the nature of the PID controller implemented and because of the lack of obstacles at high altitudes.

3.5.2 Odometry

Odometry is a useful method used in robotics to estimate the position over time of a robot taking into account the data motion sensors. There exist a multitude of sensors that can provide relative motion data such as wheel encoders, IMUs or cameras and some are present in the UGV. The Leo rover includes the RealSense T265 tracking camera [15] that uses VIO. VIO stands for Visual-Inertial-Odometry that is the fusion of visual features obtained from images with inertial measurements from the built-in IMU.

On one hand, the camera uses the images to extract features, small details in images of important relevance of the captured scene. Features should be, preferable, easy to detect, easy to differentiate from other features and as present as possible. The use of features that are unique and distinguishable along images helps obtaining information of the position of the camera between images. Unfortunately, the detection of features is

not an easy task, for example due to the scalability. It is difficult to relate the same feature between similar images that are taking some distance apart. The use of calibrated stereo cameras and artificial intelligence may solve some of these problems but the research is still ongoing [16] [17].

In addition, the camera includes an inertial measurement unit or IMU. These devices are used to report the forces and angular rates of the devices they are attached to. With these measurements the relative angular movement can be estimated and a measurement of the orientation can be computed. The RealSense T265 camera carries a high-speed IMU but the drawback of high-speed IMUs is the tracking loss after long periods of time. Fusing the stereo cameras with the IMU may provide a good tracking odometry measurement to locate the robot but the slippery nature of a skid-steering robot as the Leo rover may shift the odometry measurements overtime.

3.5.3 Kalman Filter

Considering the low accuracy of the proposed methods, the next steps would be to estimate and improve the position of the Leo rover using state estimators. One of the most known and studied technique is the Kalman filter and the non-linear version the Extended Kalman filter. A state estimator is a method based on dynamical modelling that provides an estimate of the internal state of a given system. These types of techniques are often used to approximate some states that are unmeasurable. In our case the state vector related with the localization of the robot can be tracked up to 15 dimensions.

$$x_k = \left[x, y, z, roll, pitch, yaw,\ \dot{x}, \dot{y}, \dot{z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{x}, \ddot{y}, \ddot{z}\right]^T \quad [2]$$

The Kalman filter consists in two basic steps: the prediction and the correction. It consists in a recursive algorithm were given an initial state x_k with covariance P_{k-1} and an observation z_k at a time k, the Kalman filter allows to estimate the real non-measurable state even when the input data is under white noise. The algorithm works as follows:

0. For an initial estimation of $\hat{x}_{k-1}$ and P_{k-1}
1. Project the state ahead.

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_k, 0) \quad [3]$$

2. Project the error covariance ahead.

$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \quad [4]$$

3. Compute the Kalman gain.

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \quad [5]$$

4. Update the estimate with the measurement z_k.

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0)) \quad [6]$$

5. Update the error covariance.

$$P_k = (I - K_k H_k) P_k^- \quad [7]$$

Where:

- x_k and z_k are the actual state and measurement vectors.
- $\hat{x}_k$ and $\hat{z}_k$ are the approximate state and measurement vectors.
- w_k and v_k are the process and measurement noise.
- f is the non-linear function that relates the state at the previous time step $k-1$ to the state at the current time step k: $f_{k-1} = \frac{\partial f}{\partial x}\Big| \hat{x}_{k-1|k-1}, u_{k-1}$
- h is the non-linear function in the measurement equation that relates the state x_k to the state z_k: $h_{k-1} = \frac{\partial h}{\partial x}\Big| \hat{x}_{k|k-1}$
- A is the Jacobian matrix of partial derivatives of f with respect to x.
- W is the Jacobian matrix of partial derivatives of f with respect to w.
- H is the Jacobian matrix of partial derivatives of h with respect to x.
- V is the Jacobian matrix of partial derivatives of h with respect to v.

The behaviour of the Kalman filter ends up consisting in a weighted sum between the prediction step and the measurement done by the sensors. When the sensors produce high noise measurements, the Kalman gain will reduce the weight and will increase the importance of the prediction.

This state estimator can be used to estimate the position of the Leo rover given measurement data such as the odometry obtained with the camera and the location obtained with the decawave beacons. The figure 3-8 represents the difference between the raw data obtained with the indoor localization system and the corrected odometry, fusing the raw data of the indoor localization system with the camera visual using the Extended Kalman filter.

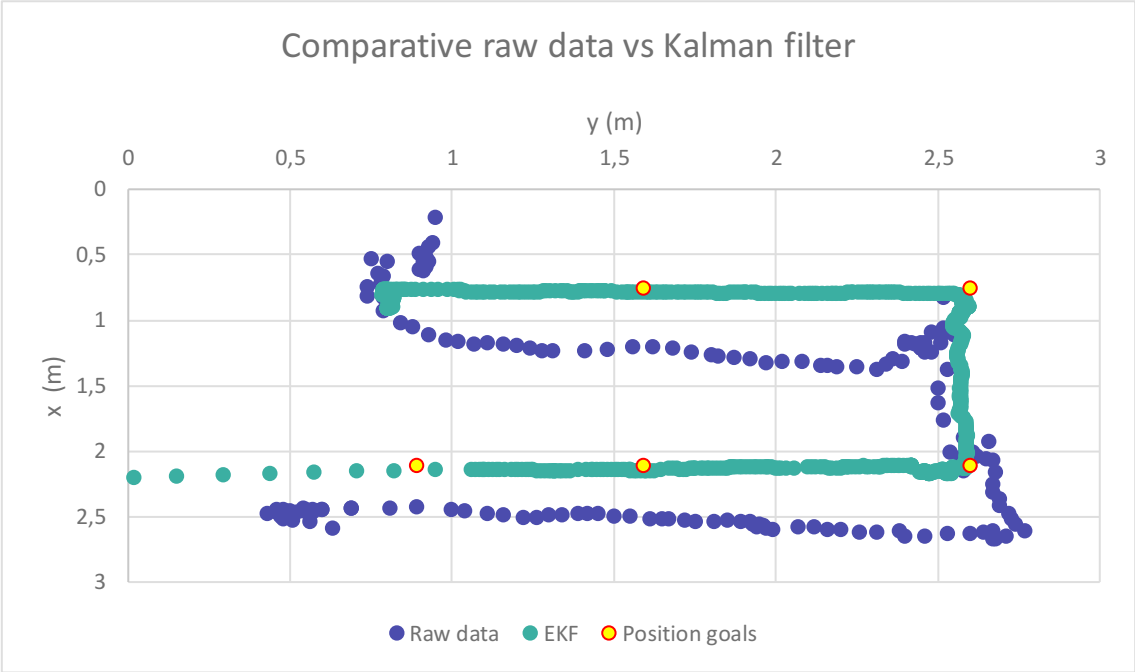


Figure 3-8 Kalman filter comparative.

4 Robot Operating System

ROS or Robotic Operating System is a middleware layer that helps to interconnect different interfaces with the objective of developing robotic solutions. Currently there are two versions of ROS: ROS1 that will be supported until 2025, and the next generation, ROS2. Both versions present the same main ideas but using different communication protocols, focusing more on the security or the environment building such as the building tool, the command lines, the package structure and the overlays.

4.1 Nodes

ROS presents the concept of *nodes*. The basic idea is to separate large coding projects into small pieces that can be handled in each one of the nodes. Each ROS node should be responsible of simple, single-handed tasks. The segmentation allows a more general method of programming, debugging and problem solving. Each node is connected to the others by different methods such as topics, services, actions or parameters. A final robotic system will consist in multiple nodes running in parallel communicating using the tools that ROS provide. It is important to mention that ROS is a middleware and provides a communication protocol and programming structure but it does not stop nor eliminates the native functionalities and capabilities of the programming languages as it can be C++ or Python.

Having a large number of nodes can be beneficious for multiple reasons but managing them can be tedious. ROS also includes the launch files, allowing to run multiple nodes at the same time using only one command. In ROS2 those launch files have, the biggest part of it, migrated to a Python file that allows multiple configurations, from adding parameters, to timings or physical conditions.

4.2 Topics

The most basic form of communication in ROS is using *topics*. When new information is generated by a node, the information can be *published* to the rest of the nodes under a topic. This topic is visible for all the rest of the nodes. The rest of the nodes that need this information can *subscribe* to the topic and receive continuous updates. The topics are a

unidirectional communication channel which has no feedback involved. Each node can subscribe to as many topics as necessary.

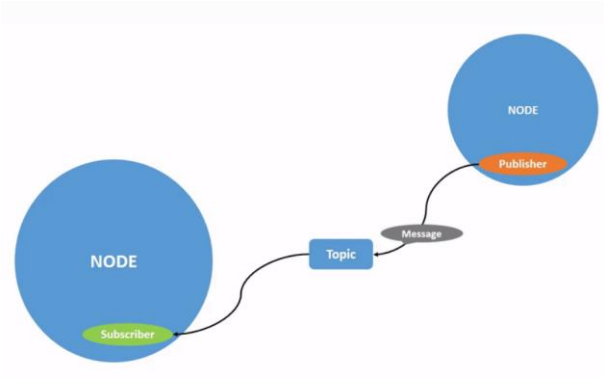


Figure 4-1 ROS topic. Source: docs.ros.org

When a node subscribes to a topic and a new message is obtained a subroutine or callback function is activated automatically. Those functions may be useful to activate other parts of the node, to save the data or to execute some process. Often the number of incoming messages does not suppose a problem but high-speed publications may be origin of problems in some situations.

There exist an infinite number of message types or custom packages that use topics in them but it is important to remark *tf* package. It provides a popular manner of managing a set of coordinate frames in the space and over time. It is useful to represent positions and rotations in space between two frames, allowing the user to build an acyclic graph to obtain in an efficient manner the transformation between frames that are connected to the same tree.

4.3 Services

The *services* are a different method of communication in ROS. To create a service two nodes are necessary: one that work as a *server* and another that work as a *client*. The server node offers the service to the rest of the nodes allowing them to call it. The clients will call the service and, at the same time, they will provide some information. The server will receive the information, and will send back a response. Each communication may be different between clients and, contrary to the topics, it allows a response from the server to the client. Only one node can advertise each type of service.

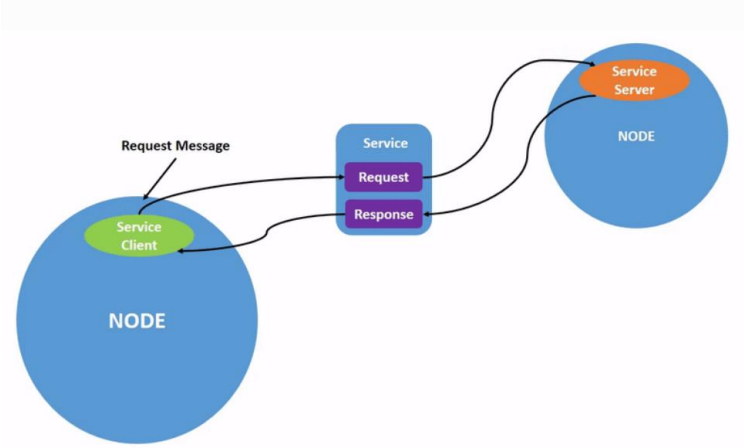


Figure 4-2 ROS service. Source: docs.ros.org

4.4 ROS1 – ROS2

ROS1 was presented in 2007 with the intent of providing a set of tools and libraries for the development of robotic solutions. Over the years a large set of modules have been built on top of ROS1 but it did not prioritize features like security, real-time applications or network topology. Those aspects made the founders of ROS, Open Robotics, to release a new version in 2015. Both versions present different qualities but there are some relevant changes:

- Architecture: ROS1 presented a master – slave architecture while ROS2 uses a data distribution service (DDS) which provides configurable settings, higher efficiency and reliability.
- ROS client library: While ROS1 presented two different libraries for C++ (rclcpp) and Python (rclpy), the next version of ROS implements a single library written in C with libraries build on top that adapts to each programming language, allowing a faster adaptation to new programming languages as java or C#.
- Ecosystem: The new version of ROS provides a convention on how to write nodes and executables. The new style allows the system to create multiple nodes on the same executable, admitting to run several nodes in parallel. Additionally, the nodes are created from an inherit node class, that implements all the basic functionalities.
- Communication: With the use of DDS as communication protocol in ROS2, systems sharing the same physical network can discover and share information. This mechanism is called ROS_DOMAIN_ID and allows a fast interconnection between groups of computers. The use of DDS also allows the customization of the message protocol like priority, reliability, deadline and more.

With the end of the lifetime present for ROS1 prepared for 2025, the community has started to port common public packages from ROS1 to ROS2. The migration from one version to the following one is slow, even more when such a large ecosystem has been created over the years. Due to the slow migration of some popular packages, the community created the ROSbridge [18] server, that creates a webserver that allows the communication between ROS1 and ROS2 messages.

4.5 Messages

The whole ROS system relies in sharing information between nodes, using topics or other types of communication. To share information a set of *messages* or type of variables are used. Usually, the messages are already standardized but it is possible to create custom messages to use in our personal solution. Here are some of the messages used in this project.

1. Empty()
   As part of the std_msgs package [19], it represents an empty message, without context nor data. Sometimes it is used to indicate the start of something, where the information is the message, and not any data attached to it.

2. Bool()
   As part of the std_msgs package, the data attached to this type of message is of type boolean, true or false.

3. Point()
   As part of the geometry_msgs package [20], this variable indicates a point in a 3-dimensional space.

4. Quaternion()
   Similarly to the point and also from the geometry_msgs package, the quaternion indicates a rotation in the space in quaternion form.

5. Vector3()
   From the geometry_msgs package, this message represents a vector in the space. The use of this type of variable is meant to represent a direction so it does not

make sense to apply a translation to it. This can be affected by the behaviours of other packages.

6. Twist()
   From the geometry_msgs package, the twist message is formed by two Vector3() variables, one indicating a linear velocity while the other representing the angular velocity.

7. Header()
   As a part of the std_msgs package, this type of variables is used in messages to represent the timestamp of the data associated to a frame. It consists in three fields: the sequence ID, the stamp (seconds and nanoseconds) and the frame id the data is associated to.

8. Pose()
   This geometry_msgs class is the arrange of a Point() and a Quaternion() variable to represent a pose in space.

9. Transform()
   This geometry_msgs class is the arrange of a Vector3() and a Quaternion() variable to represent a translation in space. The vector3() represents the translation while the Quaternion() represents the rotation of the frame.

10. TransformedStamped()
    A TransformedStamped() is used to represent a transform() at a specific timestamp. It is created by arranging a Header(), a transform() and the name of the frame the transform leads to.

11. Odometry()
    As part of the nav_msgs package [21], this variable represents an estimate of the position and velocity in free space. The message is composed by a header, by a frame id, by a PoseWithCovariance() and by a TwistWithCovariance(). The PoseWithCovariance() and the TwistWithCovariance() are similar to the Pose()

and Twist() but it also adds a matrix that represents the covariance of the estimate.

12. DecawavePosition()
    This custom message is used to represent a position in the space of one of the receivers of the decawave location system. It includes the position, the stamp and the name of the tag used as a receiver.

13. DecawaveList()
    Similarly to the DecawavePosition(), this message creates a list of DecawavePosition() messages for each one of the tags the decawave system estimates. It also includes the number of tags and the time that the list was created.

14. Marker()
    There exist multiple visualization programs that allow the representation of some type of messages. One of the most important is rviz2, connecting directly to the ROS2 environment and displaying the messages as some type of visual figure. Some messages that are not standard may need to be adapted to a Marker() type of message. As part of the visualization_msgs package [22], this message provides a set of configurations such as colour, size, type of marker, position, etc, in order to make it visible in rviz2.

There exist a lot of packages that offer standardized messages with a lot of variations, for example adding a Header() or a covariance matrix to the already known messages, not to mention the infinity of possibilities of custom messages, making ROS one powerful system for data transmission.

5 Implementation

The solution presented in this project includes all the necessary files to perform an autonomous navigation inside a room using coverage path planning algorithms. The solution has been tested in simulation using Gazebo and in real-life using the hardware presented before.

The system can be easily divided in four aspects: the localization, the path controller, the velocity controller and the behaviour manager. The first one is related to the position of the robots and the filtering using the Kalman filter. The second one is related to the management of goals and how to send the correct goals to the robots and when. These ones are highly related with the velocity controller, that is in charge of getting the goals and the position of the robot and to compute the correct velocities so that the robots move to their respective destinations. Finally, the behaviour manager is in charge of arranging the behaviour of the system, sending the velocities or stopping the robots, or taking off or landing the UAV if necessary for the hot swapping routine.

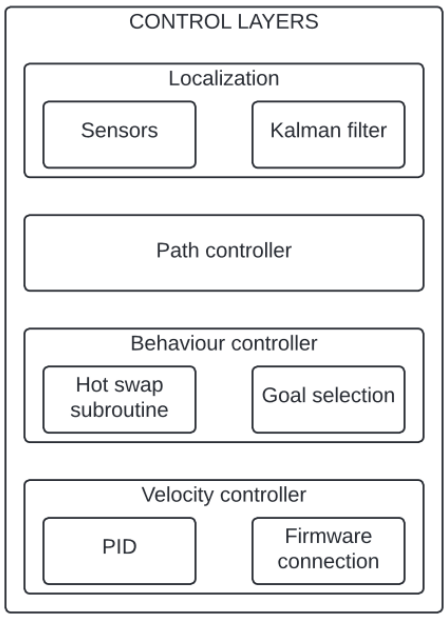


Figure 5-1 Control layer scheme.

5.1 Simulation

The first approach is done using simulation, in particular using Gazebo sim. Gazebo is a powerful toolbox for simulating robotic environments. It is highly used to accurately simulate real-life conditions offering high fidelity in the response of sensors, physical

contacts or interactions while providing an easy interaction with other software interfaces.

The first step for the simulation is to generate the elements the scene must have. In our case the room will be done using four walls while the tables are replaced by a red square of the same size. Additionally, the robots should be included together with a set of controllers that simulate the drivers in Gazebo. The controllers used are a set of files inside the ROS2 control framework that interpret the velocities and make the simulated robots move accordingly.

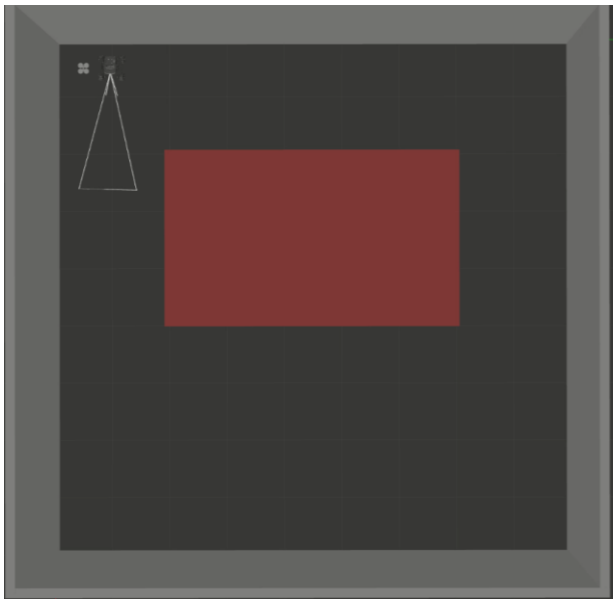


Figure 5-2 Gazebo simulation scenario.

On one hand, for the UGV, the developers of the Leo rover provide an accurate description of the rover ready to use in Gazebo. As in real-life, the control of the UGV in done using a Twist(), where the linear x component represents the speed moving forward and backward while the angular z component will spin the robot to the left or to the right. The velocities will be interpreted by the Raspberry Pi, rejecting any other velocity not compatible, for example movement in the z axis or using outbound velocities, and computing the precise motor speed for each wheel.

On the other hand, the UAV description is made by the community using some functionalities from the official API (Application Programming Interface). Even though not being supported by DJI, the behaviour of the simulated UAV is accurate to the real one. In the case of the UAV and contrary to the limitations of the UGV, the UAV accepts velocities, linear and angular ones, in each of their axis and allowing it to move freely in

space. Because it is easier to control the linear velocities than the angle of rotation, the drone will move in the x and y axis, without rotating around any of its axis.

Thanks to Gazebo, we can solve the problem of the localization of the robots since it offers a service to get the exact positions in the simulated environment. This position is published as a Point() message under the topics *"/tello1_position"* and *"/leo1_position"*.

We can divide the simulation in two big parts: the goal manager and the velocity controller for each one of the robots. The aim of the goal manager is to publish the correct goals at the correct moment taking into account the position of the robots and the hot swap subroutine. On the other hand, the velocity controllers take the goals provided by the goal manager and compute the correct Twist() message for each one of the vehicles. Remark that since it is a simulation, all information can travel in form of topics and no external API, driver or hardware is required.

To correctly simulate the battery state of the UAV is necessary to take into account a lot of factors: battery health, temperature, consumption of the motors, etc. To avoid complex simulations at this stage of the project, the hot swap subroutine will be triggered manually by an Empty() message published in the topic *"/hotswap"*. When using real hardware, this triggering will be done using the real capacity of the battery from the drone provided by the official driver.

5.2 Goal Manager

The goal manager is the process in charge of sending the goals to the velocity controller at the current pace and to supervise the hot swap subroutine. The basic work of the node is to consider the position of the vehicles and check its distance to the current goal. When the UAV or the UGV is close enough to its current goal, the node gets the next one from a CSV file. This CSV file contains the plan that each one of the robots has to follow. It has been generated at the time of computing the wavefront algorithm and, because of the capabilities of the robots, each one needs its own set of targets. Because the wavefront heuristic is a grid-based algorithm, the goals in the CSV file must be converted to Point() positions in the simulated environment so they can be compared to the actual position of the robots inside Gazebo.

The hot swap subroutine starts when the battery of the UAV is below 20% of its full capacity. During that time, the UAV should approach the position of the UGV while it is

stationary, land, wait until the battery is charged or changed with a full one, and continue with the exploration. This behaviour is controlled by the goal manager but the simulation of the battery is triggered manually. While this message is not received, the goals that sends the manager are the ones obtained by the planner. When the subroutine is triggered, the goals are replaced. The new goal for the UAV will be the current position of the UGV while the new goal of the UGV is its own position. When sending a goal that is really close to the position the robot is in, the velocity controllers will just stop the movement of the robot, allowing it to wait until the full subroutine ends.

The figure 5-3 presents a scheme of the behaviour of the goal manager, taking as inputs the positions of the robots, the hot swapping flag and the different plans and having as output the new goals for each one of the robots.

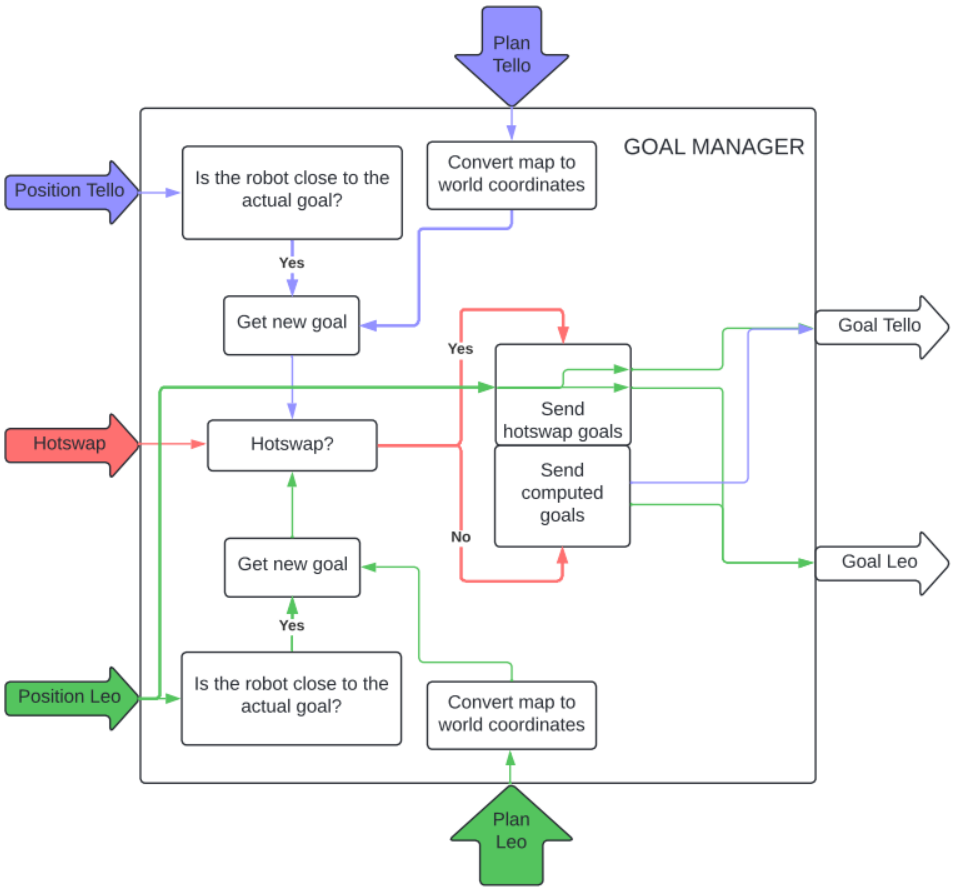


Figure 5-3 Goal manager diagram.

5.3 Velocity Controllers

The velocity controllers are the responsible for computing the adequate velocities for each one of the robots. They are similar but there is a remarkable difference between

the type of control. The possibility of the UAV to move freely in all directions allows a more direct and simple control strategy. On the opposite part, the UGV can perform movements apart from translation along the x direction and rotation in the z direction.

To compute the velocities, we must have the position of the robots. To obtain the position of the robots we can make use of the tf library that allows us to get the transforms between different frames. If the transform is between a fixed frame attached to the robot (usually called *base_link* or *base_footprint*) and the origin of coordinates (usually called *map* or *world*), the location of the robot is given directly by the transform. Similarly happens with the orientation of the robots. Once the pose is obtained, the direction and distance to the goals can be obtained. Depending on the vehicle, the control changes. For the UAV, the velocities are proportional to the distance to the goal. For the UGV, the linear movement is also proportional to the distance to the goal. On the other hand, for the angular speed, first it is necessary to compute the rotation. It can be obtained as the difference between the heading of the robot and the imaginary line formed between the current position of the robot and the destination. The figure 5-4 shows a scheme of the situation for the UGV.

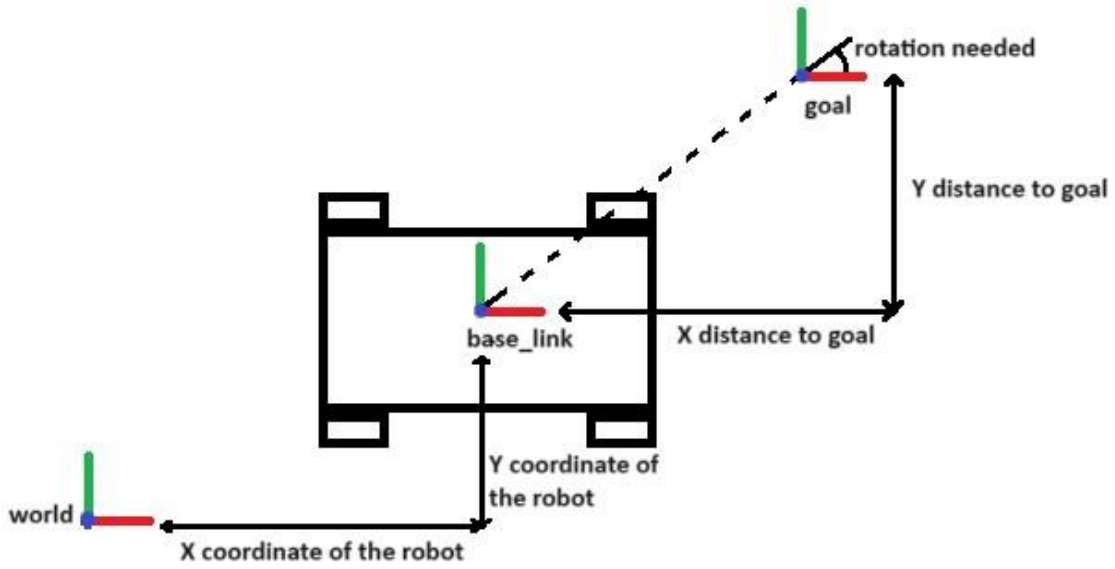


Figure 5-4 Scheme of distances.

Once the necessary rotation is obtained, a proportional strategy is used to control the angular velocity. Because we are working on a simulation that does not include any modelled noise or uncertainty, the poses, velocities and control strategies can be kept

simple and precise. When using the real hardware those strategies will change and there will be multiple sources of disturbances.

The figure 4 shows a scheme of the velocity controllers used for the UGV and the UAV.

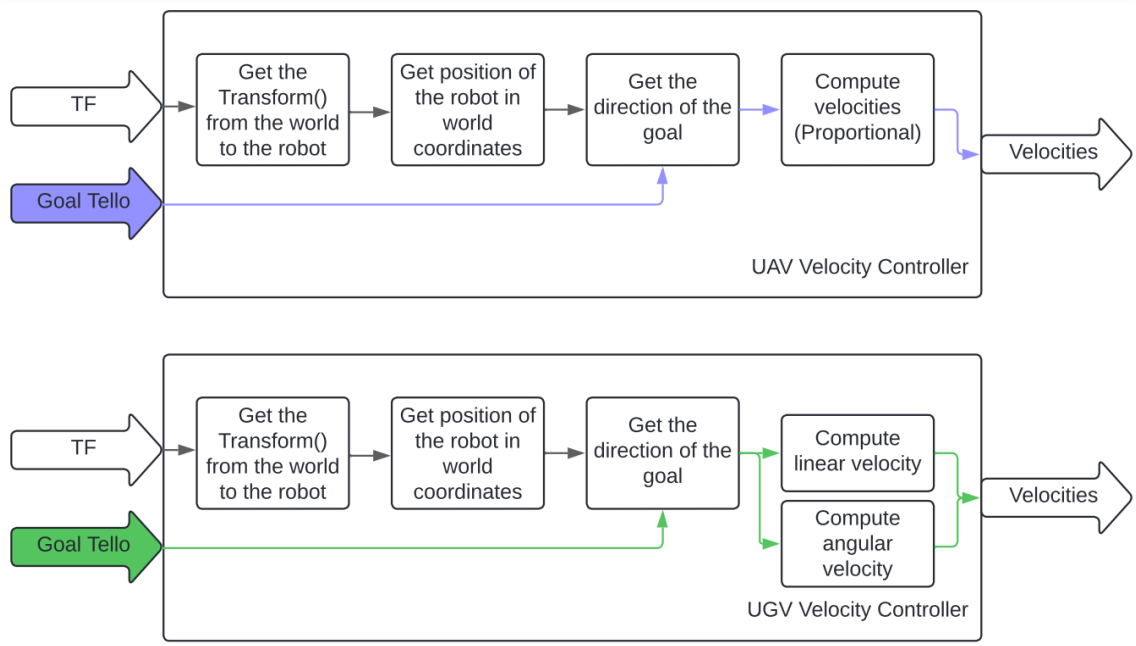


Figure 5-5 Velocity controllers' diagram.

6 Real System Integration

Thanks to the computer simulations, we could study and experiment some of the difficulties of working with multi-model robots. This time has allowed us to create a basic architecture of how this system can be implemented in real-life. In this part of the project, we are going to present the challenges endured when trying to apply the previous solution in real-life and the solutions carried out.

The system used have different hardware stages: the main computer (Master PC) together with the Nvidia Jetson Orin Nano dev kit will work together to execute as many nodes as possible in ROS2 and to control the Tello drone. On the other hand, the Leo Rover (using the Raspberry Pi) and the Nvidia Jetson Nano running ROS1 will execute just the minimum number of nodes.

The system it is divided in different files that run different nodes that must be run in the different systems. Two of the files are already made files that sets some configurations in the Leo rover in ROS1. The other files are inside the ROS2 framework and runs the main package of the system.

6.1 ROS1

As mentioned before, the packages in the UGV are community made packages inside the ROS1 framework. We are not going to enter into detail but just comment some important details that may be relevant in the main system.

6.1.1 Nvidia Jetson

As part of the Leo rover, this small computer is in charge of controlling, connecting and managing some extra sensors installed on top of the UGV. In specific there are two main sensors: the RealSense T265 tracking camera and the Rplidar A1 lidar. These two sensors are highly used in multitude of real-life applications so it is easy to find several ready-to-use packages.

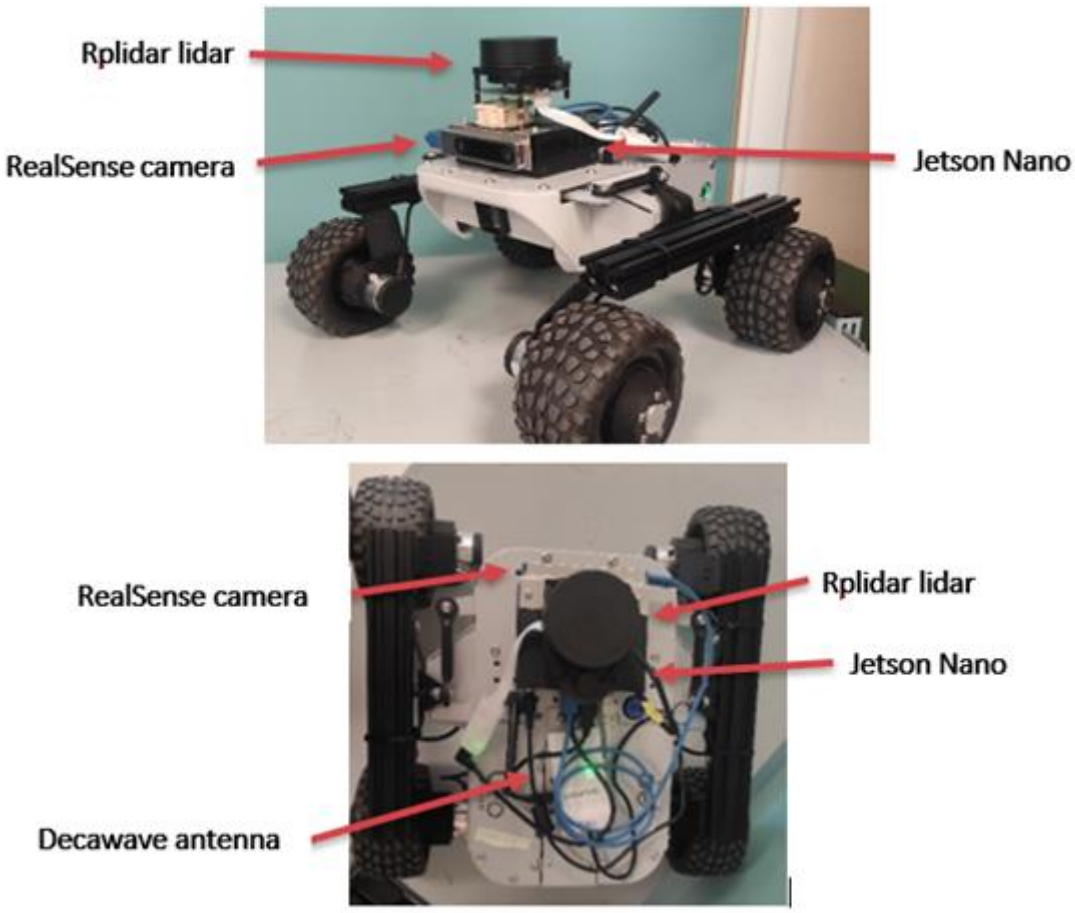


Figure 6-1 Sensor set up on the Leo rover.

1. RealSense camera

   RealSense offers a set of ROS1 packages prepared for some models of their cameras. When running the package for the T265 camera installed of the Leo rover, a set of TransformedStamped() messages are created. Those messages are of particular interest because they offer an estimation of the position of the camera with respect to the starting point of the nodes. That means that when the nodes are running, it creates a frame at the starting position and, using VIO, it estimates the transformation between the camera and the initial point over time. These transformations are the ones used as the odometry measure for the robot.

2. Rplidar lidar

   The community has created a lot of packages ready-to-use on the Leo rover, including some for autonomous navigation or SLAM. This type of packages usually

needs a fix reference in the space. The references REP 103[1] "Standard Units of Measure and Coordinate Conventions" and REP 105[2] "Coordinate Frames for Mobile Platforms" describe the system conventions used in general in ROS and for mobile robots. It defines the graph of coordinates as *world -> map -> odom -> base_link.* The world frame is the highest level in the graph and it is often use for the interaction between multiple robots that have different *map* frames. The *map* frame is a space fixed frame used for long term global reference. Finally, the *odom* frame is a more or less fixed frame that may drift over time as it describes the best continuous estimation of the robot to its attached frame *base_link*.

Using the lidar and the gmapping package, we can create a set of frames to use the graph of coordinates described before. The sensor data will provide a basic foundation for a good estimation of the *map* frame and, indirectly, for the *world* frame that will be used in later sections of the project.

6.1.2 Raspberry Pi

The Leo rover integrates a Raspberry Pi 4B 2GB board running Ubuntu 18.04 and ROS Melodic. This board is in charge of starting a set of basic nodes that allows the interactions with the motors, the camera and the connection with the external local webserver. These set of nodes also advertise the topic *"/cmd_vel"* to capture velocities and to convert them to the corresponding motor commands.

To send the velocities computed by the Master PC, we need to start a new communication channel using the ROSbridge package. This package will allow us to close the channel in the Master PC using ROS2 and to send the necessary messages. To keep it simple, we will be sending and receiving some string messages from the ROS2 nodes and so we need a set of nodes in the Leo rover that get those strings with data and process them to the rest of the ROS1 environment.

6.1.2.1 Traductor.cpp

To send messages from the Master PC to the Leo rover we need to create a string of numbers separated by commas containing the desired speed for the robot. The

[1] https://www.ros.org/reps/rep-0103.html
[2] https://www.ros.org/reps/rep-0105.html

traductor.cpp file runs a node that take those strings, rebuilt the Twist() message in ROS1 and publish them in the topic *"/cmd_vel".* The rover does not accept movement in other axis that are not the linear x and angular z so the string can be reduced to just two numbers separated by a comma.

6.1.2.2 Tf_listener.cpp

In a similar but opposite way as the *traductor.cpp* file, the *tf_listener.cpp* file runs a node that instead of receiving string messages and converting them to the corresponding message type, it creates string message from a topic. In specific, the node uses the tf library to keep track of the transform between the frames *odom* and *base_link* and publishes a string with the data separated by commas. The string will be sent to the ROS2 environment using the ROSbridge webserver.

6.2 ROS2

The introduction of new communication protocols and features together with the simplicity on the integration of multiples machines connected to the same physical network makes ROS2 a good candidate for real hardware applications. The combination of multiple nodes running in parallel in the same executable simplifies the shareability of data, including the connection to the hardware in use.

The major part of this project has been developed using ROS2 because of their features and novelties compared to its predecessor, ROS1. For this project, there will be two main processing units: the Master PC that will be running all the common nodes and that will work as a bridge between the different elements in the set up and the Nvidia Jetson Orin Nano dev kit also running ROS2 that will be executing the drone-related nodes.

One of the main advantages of ROS2 is the shareability of data between nodes in the same physical network. The Master PC connects to the Leo rover Raspberry Pi using a webserver while the connection with the Nvidia Jetson Orin Nano dev kit that controls the drone is done using an ethernet connection throughout a switch. The usage of a switch enables the possibility of including more UAVs to the system. With the use of the switch and with some namespaces (small tags that allow to differentiate processes with the same name and functionality), the system proposed here could be easily adapted to include extra Jetson boards and Tello drones. Unfortunately, ROS2 has not implemented

yet the possibility of launching nodes in external machines from a master host. This functionality was present in ROS1 with the *<machine>* tag but in the new version we are required (at the moment) to launch the nodes manually in each machine.

As explained before, ROS2 uses DDS as communication protocol. To enable the communication of messages and nodes between machines inside the same physical network, we need to configure the Domain ID. The Domain ID (or ROS_DOMAIN_ID in the ROS jargon) is the primary mechanism used in DDS to enable different logical networks. The nodes launched in different Domain IDs will not be able to communicate between them nor to receive topics, advertise services or use other ROS features.

To configure the ROS_DOMAIN_ID it can be done using the command:

```
$ export ROS_DOMAIN_ID=<your_domain_id>
```

Now the nodes or launch files executed in the shell are run directly to that domain. To use ROS2 commands from a terminal in a different ID than the configured one use:

```
$ ROS_DOMAIN_ID=<other_id> ros2 <your_command>
```

To maintain a ROS_DOMAIN_ID setting between shell sessions:

```
$ echo "export ROS_DOMAIN_ID=<your_domain_id>" >> ~/.bashrc
```

If those commands are run in all the machines with the same ROS_DOMAIN_ID, they will be able to interact. Different ROS_DOMAIN_IDs will not be able to communicate unless special commands are used.

6.2.1 Master PC

The Master PC is the main computer of the project. Even though not existing ROS Master in ROS2, the name designated to this computer is because of the given naturality of implementing a base for the interaction with the rest of the hardware. On one hand, to interact with the Nvidia Jetson Orin Nano dev kit, the computers must have the same ROS_DOMAIN_ID. On the other hand, the communication with the Leo rover is done using a webserver provided by the ROSbridge package, that at the same time enables the conversion between ROS1 and ROS2.

This computer will also run some complementary nodes that may be used for tasks like visualization or results recovery. Those extra nodes are made by the community or are features implemented by the native software of ROS.

6.2.1.1 Decawave.py

The *decawave.py* file is one of the most important files of the system because it provides the positions of the robot thanks to the indoor localization system. The localization system uses some antennas in known places and some receivers (or tags) installed on top of the robots. Measuring the propagation delay of the signals, the position of the receivers can be estimated. This system uses 2 types of receivers, the ones used for the position estimation and one that is used as an interface. This interface can be connected to a serial port and will provide the measurements without the need of WI-FI connections or external APIs.

What this file does is to run a node that opens the serial communication between the antenna of the decawave system and the computer it is connected to. The antenna should be connected and get prepared to receive estimation messages from the rest of the tags. To open the antenna the signal *"\r\r"* should be sent via serial communication. Additionally, depending on the working mode, the signal *"lep"* should also be sent through the serial. Those signals are only needed to be sent once, just when the interface is connected. The configuration will remain set as long as the interface has electrical connection, even when the ROS node finishes. If the interface is disconnected, runs out of battery or the commands are sent twice, it will stop receiving estimations thus the configuration must be sent. To control the signals, the argument *"first_start"* can be used in any of the launch files. Once it receives a new lecture (or estimation from one of the tags) it parses the position, the ID of the tag and the time and adds it to the DecawaveList() with the rest of tags that are in the system. If the tag already exists, it updates the position on the list.

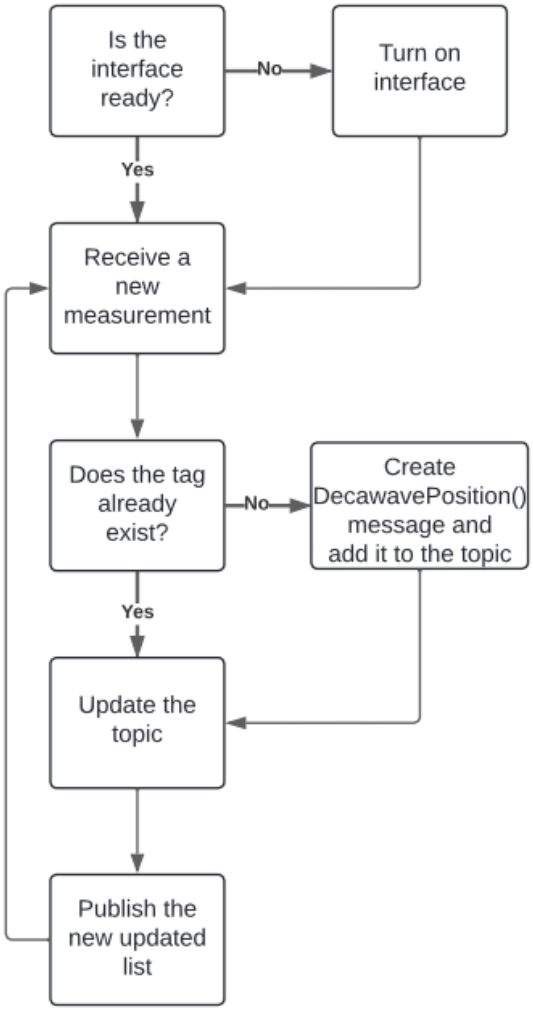


Figure 6-2 Decawave.py diagram.

At the end, this node publishes a DecawaveList() under the topic *"/decawave_data"* with the positions of the tags.

6.2.1.2 Efk_node

The *ekf_node()* is a node from the robot_localization package [23] that implements the extended Kalman filter; using omnidirectional model motion to predict the state and corrects it using sensor data. Because of the bad accuracy of the beacon installed on the UGV and the possibility of using more sensors without weight and consumption limitation, the localization of the rover is improved using the Extended Kalman filter for a planar robotic platform. The node must be configured by a set of parameters that ensures the correct integration of the sensors with the model. We can differentiate 2 types of parameters, the ones related with the configuration of the node and prediction step and the ones related with the sensors data. The robot_localization package offers a

wide variety of state estimators and configurable parameters but here are the ones used in our use case:

- Frequency: Speed at which the node produces an estimation. It is set at 30 Hz.
- Sensor_timeout: Real time value expressed in seconds when it is considered that the sensor will not provide new data thus the correction step cannot take place. It is set at 0.3 seconds.
- Two_d_mode: If the robot is moving in a planar space, this parameter allows to avoid estimation in the Z axis, ensuring an estimation inside the X-Y plane. It is set to true.
- Map_frame, odom_frame, base_link_frame, world_frame: Names of the frames of the system. All those frames are created in the Nvidia Jeton Orin thanks to the community made packages. The names are set to *map*, *odom*, *base_link* and *room_map* respectively. Remember that the *map* frame is created by the gmapping package, the *odom* frame is created by the camera using VIO, the *base_link* is the frame attached to the robot base and that the *room_map* frame is a fixed frame at (-0.9, -0.8, 0) meters from the *map* frame that is used to have a common reference between the rover position and the drone position using the decawave beacons.
- Input topic name: For each sensor data that can be fused, it is necessary to give a name and a topic where the *efk_node()* can subscribe to. In our case there are 2 sensors that produce data, the odometry provided by the camera and the localization beacons using the decawave system. The topic names are *"/odom/robot"* for the odometry from the camera and *"/odom/gps"* for the data from the beacons. Because the node only accepts Odometry(), PoseWithCovarianceStamped(), TwistWithCovarianceStamped() and sensor_msgs/Imu() messages, the topics are prepared by another node that will be explained in future sections of this report.
- Input topic configuration: The different sensors provide data regarding the state of the mobile platform but not all data provides information about the different changes in the state of the robot. For instance, the position *(x, y, z)* given from the decawave should not change the estimation in the orientation *(qx, qy, qz, qw).* To configure these aspects, each sensor should include a boolean matrix-like

configuration stating which variables of those messages should be fused in the state estimation. The state of our mobile robot is described in [2].

The natural drifting of the odometry estimation makes the odometry of the camera a bad estimation for the position but the presence of the IMU makes it a good estimation for the rest of the state. Because of that, the configuration of the odometry of the camera is:

$$x_k = \begin{bmatrix} x, y, z, \\ roll, pitch, yaw, \\ \dot{x}, \dot{y}, \dot{z}, \\ \dot{roll}, \dot{pitch}, \dot{yaw}, \\ \ddot{x}, \ddot{y}, \ddot{z} \end{bmatrix} = \begin{bmatrix} false, false, false, \\ true, true, true, \\ true, true, true, \\ true, true, true, \\ true, true, true \end{bmatrix}$$

On the other hand, the position obtained with the beacons cannot provide information other than the position, leaving the configuration matrix as:

$$x_k = \begin{bmatrix} x, y, z, \\ roll, pitch, yaw, \\ \dot{x}, \dot{y}, \dot{z}, \\ \dot{roll}, \dot{pitch}, \dot{yaw}, \\ \ddot{x}, \ddot{y}, \ddot{z} \end{bmatrix} = \begin{bmatrix} true, true, true, \\ false, false, false, \\ false, false, false, \\ false, false, false, \\ false, false, false \end{bmatrix}$$

After running this node, the topic *"/odometry/filtered"* will be published giving an Odometry() message of the estimated position of the robot. This estimation will be renamed as *"/odom/filtered"* and will be the one provided to the rest of the nodes as the real position of the Leo rover. Due to the configuration parameters and the frames' structure, the position of the Leo rover is already given with respect to the *room_map* frame, making it ready to use for the rest of the nodes and avoiding to produce a transform to a different frame before using it.

6.2.1.3 System_control_leo.py

The *system_control_leo.py* file is the responsible of getting the position of the Leo rover and send the correct goals through the corresponding topic. It is also the responsible of computing the path in world coordinates and stopping the robot when the hot swap subroutine starts.

Thanks to the new standardized way of programming a node in ROS2, it allows, in a very simple manner, the execution of multiple nodes in parallel that are inside the same file.

We can divide this file in the 2 nodes that forms it: the *goal_sender_leo()* and the *position_supervisor_leo().*

6.2.1.3.1 Goal_sender_leo()

This node implements a simple routine to publish under the topic *"/new_goal/leo"* the correct goal. Every second and with the activation of a timer, the node checks if the hot swap subroutine has started and, if not, it will check the distance from the robot to the current goal. If the robot is close enough to the current goal, it will move to the next cell of the path created using the coverage path planning algorithm. After obtaining the cell, it will compute the actual position of the goal in world coordinates and the new goal will be published. If the distance to the current goal is significant, the cell will remain the same and the published goal will be the same as in the previous activation of the timer. Additionally, if the hot swap subroutine has started, the published goals correspond to the exact current location of the robot. Sending the same location as the current one makes the robot stop thanks to a condition implemented in the velocity controller of the rover.

6.2.1.3.2 Position_supervisor_leo()

As its name indicates, this node has the purpose of getting the position of the Leo rover. Due to the fast and constant publication of the estimation of the Leo by the *ekf_node(),* the *goal_sender_node()* cannot handle the callbacks with enough speed and simultaneously be publishing at a constant pace the goals for the mobile platform.
Thanks to the features of ROS2, this problem can be solved by creating a new node that focus on receiving the messages at high speed and sharing it to the *goal_sender_leo()* using a global variable. Also, this node subscribes to the *"/hotswapping"* topic to let the *goal_sender_leo()* node know when the hot swap subroutine is taking place. The figure 6-3 illustrates the interaction between the 2 nodes and the global variables.

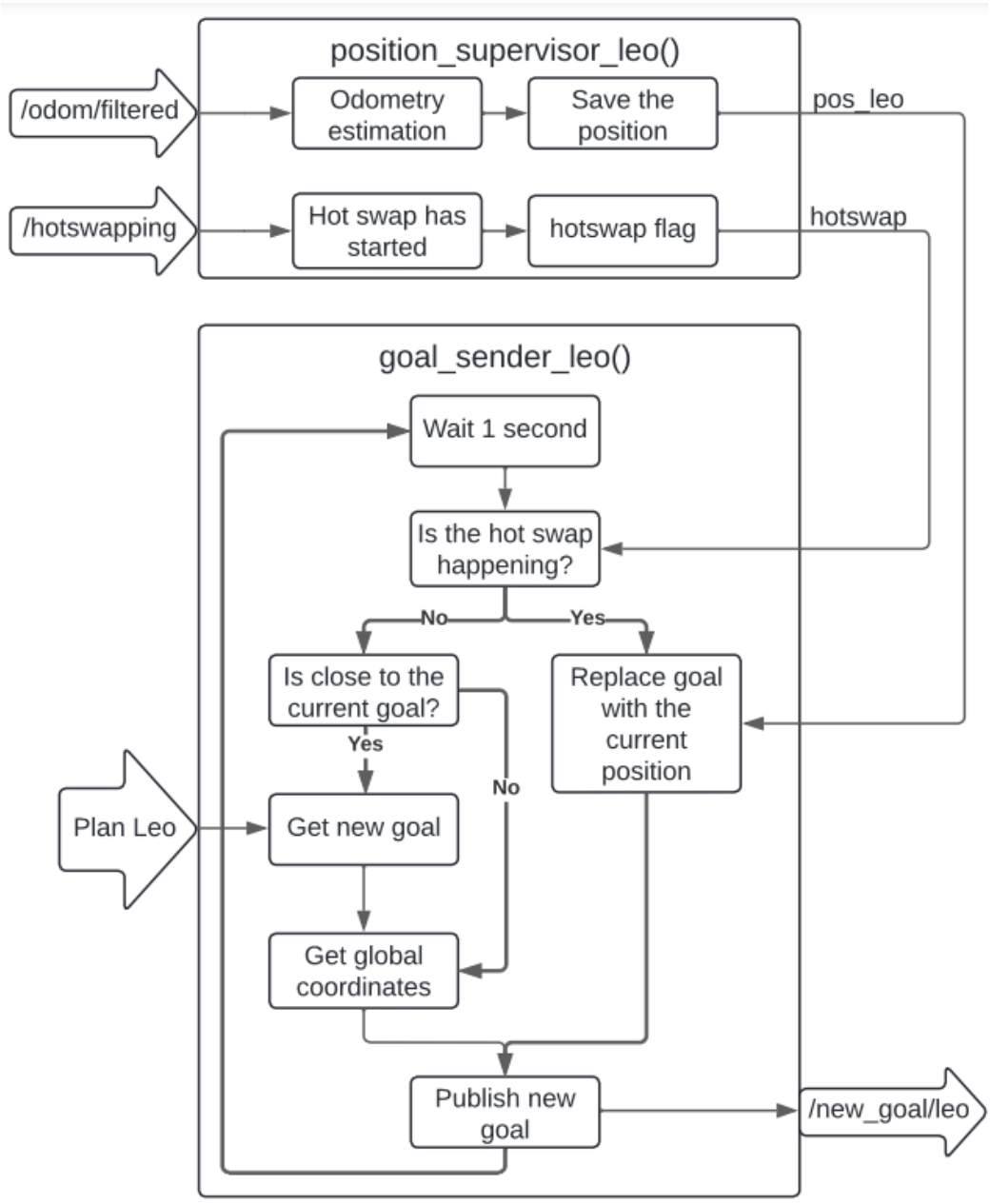


Figure 6-3 System_control_leo.py diagram.

At the end, the *system_control_leo.py* file publishes a Point() message under the topic *"/new_goal/leo"* with the position of the goal.

6.2.1.4 Leo_control.py

The *leo_control.py* file is the responsible of setting the communication between the Leo rover and the Master PC using the webserver generated by the ROSbridge package. There are multiple libraries that connect to the webserver but the used one in this system is the *roslibpy* library. The *roslibpy* library enables the connection to the webserver using Python, allowing the subscriptions and publications of topics.

The *leo_control.py* file implement a pair of nodes that interconnects multiple information between the different ROS environments. The first one is the *odom_builder(),* a node that is in charge of sharing and processing the topics needed to create a good estimation of the pose with the *efk_node().* That means that the *odom_builder()* will prepare all the necessary topics for the *ekf_node()* to work as intended. The second node inside this file is the *leo_controller(),* that implements a velocity controller for the rover. Once a new goal arrives, it will compute the correct velocities to reach the goal and those will be sent to the Leo rover. Even though these nodes do not share many common variables or have similar purposes, the connection with the Leo rover using the webserver is much easier and without the need of extra configuration if they are kept in the same file. The figure 6-4 represents a diagram of the interconnection of the different nodes.

6.2.1.4.1 Odom_builder()

As explained before, the *ekf_node()* will use two sensors' information, the odometry from the camera and the position provided by the indoor localization system. The camera odometry does not provide the information in a manner that ROS2 understands it and the position from the beacons are given in form of a custom message. The *odom_builder_node()* will subscribe to those topics and will reformat the information to one of the accepted formats by the *efk_node()*.

Because the *roslibpy* library does not require a ROS environment, the messages received are in form of a Python dictionary. The dictionary follows the ROS1 message structure so all the incoming messages need to be translated from a dictionary to an acceptable message structure but this time in ROS2. Luckily, the odometry measurements from the camera are available in both versions of ROS so the translation is direct. Thanks to this set of packages, an Odometry() message enters throughout the topic *"/camera/odom/sample"* in ROS1 at the Leo rover and leaves as an Odometry() message under the topic *"/odom/robot"* in ROS2 at the Master PC. The comparative can be seen in the table 6-1.

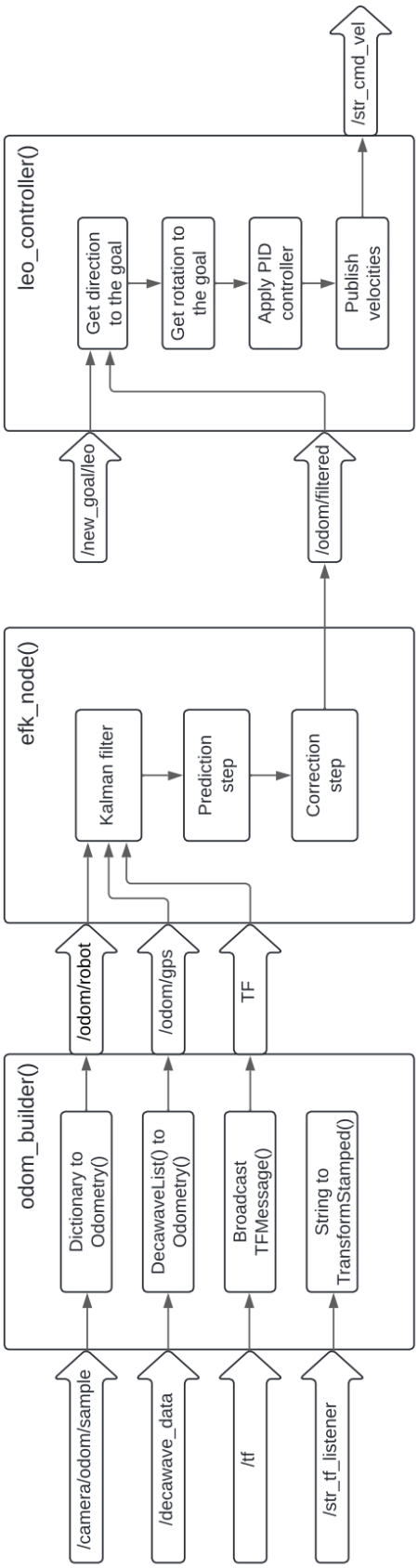


Figure 6-4 Leo_control.py and ekf_node() interconnection scheme.

Table 6-1 Comparative between incoming and outgoing messages types.

Incoming message (python dictionary)	Outgoing message (Odometry() message)

`msg = {'header': {'seq': 0, 'stamp': {'secs': 0, 'nsecs': 0}, 'frame_id': 'room_map'}, 'child_frame_id': 'base_footprint', 'pose': {'pose': {'position': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'orientation': {'x': 0.0, 'y': 0.0, 'z': 0.0, 'w': 0.0}}, 'covariance': [...]}, 'twist': {'twist': {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x': 0.0, 'y': 0.0, 'z': 0.0}}, 'covariance': [...]}}`	`odom_robot = Odometry()`
	`odom_robot.header.stamp`
	`odom_robot.header.frame_id`
	`odom_robot.child_frame_id`
	`odom_robot.pose.pose.position.x`
	`odom_robot.pose.pose.position.y`
	`odom_robot.pose.pose.position.z`
	`odom_robot.pose.pose.orientation.x`
	`odom_robot.pose.pose.orientation.y`
	`odom_robot.pose.pose.orientation.z`
	`odom_robot.pose.pose.orientation.w`
	`odom_robot.pose.covariance`
	`odom_robot.twist.twist.linear.x`
	`odom_robot.twist.twist.linear.y`
	`odom_robot.twist.twist.linear.z`
	`odom_robot.twist.twist.angular.x`
	`odom_robot.twist.twist.angular.y`
	`odom_robot.twist.twist.angular.z`
	`odom_robot.twist.covariance`

For the position of the robot given by the beacons, the message is already in the ROS2 environment so there is no need of using the *roslibpy* library. The main difficulty is the type of message, DecawaveList()/DecawavePosition(), that is not accepted by the *efk_node()*. Every time the node receives an update of the position for the tag attached to the UGV, it will create an Odometry() message with the updated position, publishing it. This type of messages will be accepted by the *ekf_node()* to produce its estimation.

The *efk_node()* also makes use of the ROS tf graph to understand the names of the frames and the transformations between them. The node should also bring to the Master PC the full tf tree generated in the rover. Furthermore, this node receives the information that

has been sent by the *tf_listener.cpp* node in the Raspberry Pi as it is the transform between the frames *odom* and *base_link*. With the tf package for ROS2, the full tf tree can be broadcasted easily.

The only variable shared with the *leo*_controller() node is the transformation between the *odom* frame and the *base_link* since it produces an accurate estimation of the orientation of the robot thanks to the IMU incorporated at the RealSense camera.

6.2.1.4.2 Leo_controller()

The *leo_controller()* node implements a velocity controller to move the UGV to the desired position. To do that job, the node subscribes to the *"/odom/filtered"* topic provided by the *efk_node()* and to the *"/new_goal/leo"* topic published by the *goal_sender_leo()* node.

The process to compute the velocities is similar to the one used in the simulation. When a new goal is received, the direction of the imaginary line between the position of the robot and the goal is computed. Comparing the current orientation of the rover using the transformation given by the *odom_builder()* node and the desired orientation, we can establish the necessary angular velocity. The similar case happens with the distance and the linear velocity but in this case, if the distance is too close to the objective, the system will stop the robot. This procedure ensures that the UGV is steady when the hot swap subroutine begins.

It is important to notice that the controller implement a PID control loop. For the linear speed the behaviour keeps proportional as in the simulation. Due to the physical limitations of the hardware, the effects that the integral and the derivative parameters are almost no noticeable. The Leo rover will saturate the speed so the changes that those parameters offer are almost not perceptible until the goal is almost reached. For the rotational control, the parameters used are kp = 1.0, ki = 0.1 and kd = 0.5.

Because of the physical limitations of the hardware, large values of linear velocities can shadow the effect of the angular speed so if the error on the rotation is larger than 0.15 radians, the rover will send a null linear velocity.

Once the velocity is computed, it is sent to the topic *"/str_cmd_vel"* to the Raspberry Pi using the *roslibpy* library so the *traductor.cpp* node can transform it to the correct format.

It is important to mention that both robots have implemented a PID controller in both their axis. Multiple alternatives such as the move_base package [24] or using fuzzy controllers has been tested but the simplicity and direct way of tunning the PID made it the best option.

6.2.1.5 Other launched nodes

Part of the responsibilities of the Master PC is to run all the extra nodes that are not essential for the system, including visualization tools or results recovery programs. In this project the program rviz2 has been used for visualization. Rviz2 is a visualization tool that allows direct connection with the ROS environment, integrating numerous types of visualization techniques depending on the type of message. The easiest way of using rviz2 is to connect it to the topic we want to visualize and, if the type of message is standardized, rviz2 will recognize it and will plot it. On the other hand, there are some custom messages that may need some transformation before been correctly visualized in rviz2. One example of that is the DecawaveList() custom message, that in order to visualize it we need to adapt it to a Marker() type.

To make this transformation, the file *demo.py* has been created. This file runs two nodes with almost the same purpose, preparing some of the topics data so that can be visualized in rviz2. The outputs of these nodes are a series of Marker()s with the data correctly configured for visualization. Including the appropriate settings in rviz2, the data will display the results of this project.

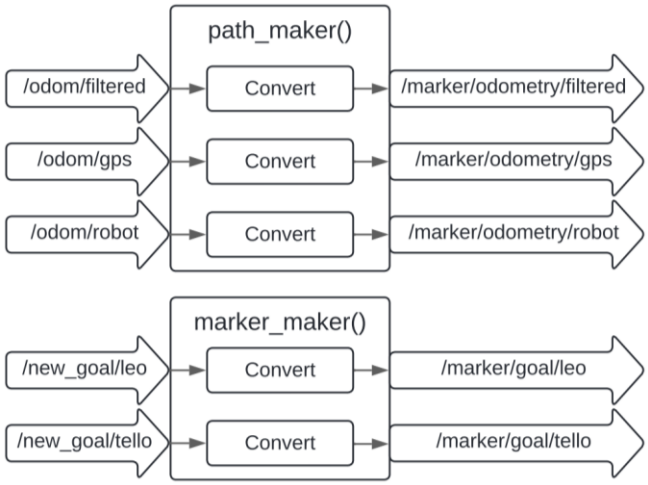


Figure 6-5 Demo.py diagram.

Lastly, ROS implements a functionality to record and save data: rosbag. In our case it may be useful to record the topics used for later purposes, like results recovery. This functionality is controlled by an argument in one of the launch files that will be presented later. This functionality is configured to listen to all the topics and save the messages to a file under a folder named *data*. Is important to observe that if this utility is used, it will overwrite the previous saved recording with a new set of data.

6.2.2 Nvidia Jetson Orin Nano dev kit

The usage of multiple machines is one of the main advantages of ROS2 compared to its predecessor ROS1. Two machines using the same physical network and configuring the same ROS_DOMAIN_ID will be able to communicate immediately. The Nvidia Jetson Orin Nano dev kit is added to this project to communicate and control the Tello drone using WI-FI connection while the Master PC uses the webserver.

As mentioned before, between the connection of the Master PC and the Jetson board a switch is installed. A switch in networking jargon is a device that interconnects multiple devices allowing them to share the same network and enabling multiple devices to talk to each other. Even though the system here presented is limited to just one drone, the use of this switch conceives expandability to the system and makes possible the use of multiple Jetson boards and drones. The use of some namespaces and with little extra configuration, this system can be expanded to include multiple drones for a multi-robot exploration.

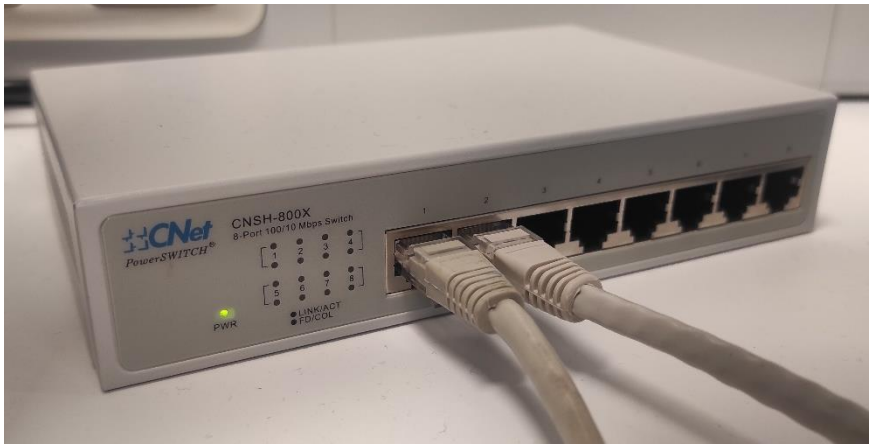


Figure 6-6 Switch in use.

Thanks to the Master PC, the Jetson board only needs to run one file containing 3 nodes. Unfortunately, it is not possible to execute files in other machines even when they are connected to the same network. This procedure was possible in ROS1 and the <*machines*> tag in the launch files but is not yet implemented in ROS2. To run the node, we will have to do it manually using a keyboard connected to the board.

6.2.2.1 System_control_tello.py

The *system_control_tello.py* is the most complex file of the project. It runs 3 different nodes that may not highly interact between one another but the connection and the use of the official API for the drone makes them be in the same file. Firstly, we have the *position_supervisor_tello()*, a similar node to the *position_supervisor_leo()* that helps to control the income flow of messages regarding the position of the robots. Secondly, we have the *tello_velcontrol()* node, in charge of computing the velocities for the drone. Finally, we have the *goal_sender_tello()* node, that manage the flow of goals and the hot swapping subroutine. The figure 6-7 represents some of the main interaction between the three nodes and the main workflow of the nodes. In the image, there are some cases or situations that are not represented but that are taken into account at the time of running the nodes.

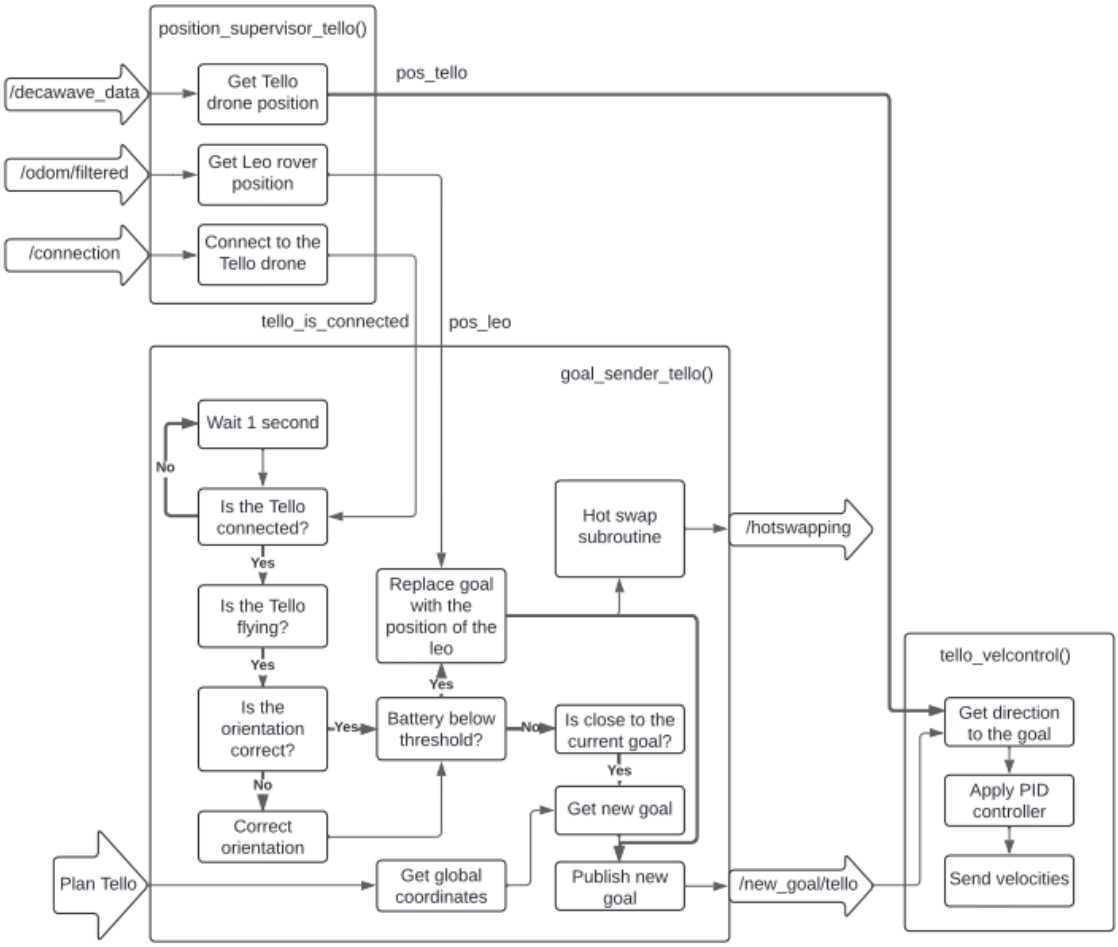


Figure 6-7 System_control_tello.py diagram.

The main advantage of having these nodes together is the connection with the drone. The Tello drone is controlled using the WI-FI module of the Jetson board. When the drone is connected, it creates a UDP socket to transmit data from, in this case, a node. The communication using the sockets can be exhausting depending in the number of commands and the required speed. To simplify the job, the *djitellopy* library offers a set of Python functions to obtain and control almost every aspect of the drone. Thanks to this library, common commands used by the three nodes can be executed without external and complicated protocols.

6.2.2.1.1 Position_supervisor_tello()

The part of the *system_control_tello.py* that is in charge of obtaining the positions of the robots is the *position_supervisor_tello()*. This node subscribes to the *"/odom/filtered"* and *"/decawave_data"* topics to obtain the positions of the UGV and the UAV respectively.

Comparing it to the *position_supervisor_leo()* node, the position of the rover is also required for these nodes as it will be used as goal when the hot swapping routine starts. The positions are stored in the global variables *pos_tello* and *pos_leo*. Another important global variable created and managed by the node is *tello_is_connected*. This variable indicates if the Tello drone is correctly connected to the Jetson board and ready to receive commands using the API. All the system relies in the status of this variable that is controlled by the external function *connect().* This function is called at the beginning of the program when the nodes are not even running and ensures an automatic start for the exploration.

When the hot swap subroutine ends, the drone has lost electrical power thus the connection to the Jetson board is also lost. To reconnect the drone, *connect()* is called using the topic *"/connection"*. The topic *"/connection"* is an Empty() message whose only purpose is to connect the drone and make sure it is ready to receive commands. Because of the manual battery swap, the topic is also manually triggered.

6.2.2.1.2 Tello_velcontrol()

Tello_velcontrol() is the name of the node that takes the position of the drone and generates the corresponding velocities to move the drone to a specific destination. The advantages of using drones in this type of controllers is the possibility of moving it freely in every direction. Unlike the Leo rover, the Tello drone allows the movement and rotation in every axis, avoiding the problem of been constantly correcting the orientation. Every time a new goal is received from the topic *"/new_goal/Tello"* the callback function will generate the corresponding velocities and will send them using the *djitellpy* library. To generate the velocities, a pair of PID controllers are used, one for each direction of movement. Those PID controllers have the same parameters for each direction being them kp = 20, ki = 0.1 and kd = 0.5. For safety, the velocities inside the room are limited to a ±15% on the x direction (+ forward) and to the range [-15, 20] on the y direction (+ right). This change of range depending on the direction is due to the disturbances generated by the ventilation of the room. More details will be provided in future sections of the document.

Using an aerial robot implies working on the control of at least one extra dimension. The altitude of the drone can be controlled automatically by the drone or manually for us.

The altitude stablished by the automatic controller is not enough to dodge the terrestrial obstacle that the rover already needs to avoid. Using the altitude provided by the drone a basic controller for the heigh is implemented. If the height of the drone is below the 115cm, the drone will increase its propellers a 15%, in the contrary, if the drone is over the 145cm, the drone's propellers will decrease a 15%. Anything between is proportional, aiming for a constant 130 centimetres in altitude.

6.2.2.1.3 Goal_sender_tello()

The *goal_sender_tello()* node has the task of controlling the goals for the Tello drone and to know when to execute the hot swap subroutine. The node consists in a timer that gets triggered every second. When the timer triggers, the first step is to check if the Tello is correctly connected. Not checking the global variable *tello_is_connected* may lead to errors on the code, stopping the node and in consequence, the whole exploration. The second step is to check if the drone is flying or not. This can be easily checked using the API. In the case that the drone is not flying, it may be because one of two situations: because of the hot swap subroutine or because the exploration has not started yet. In both cases, if the drone is correctly connected, it can take off, correct its orientation and mark that the hot swap subroutine is not taking place. The correction step is done to avoid small errors in the control by sending velocities that, in fact, may not be approaching the drone to its destination. In this situation the hot swapping subroutine is not taking place because the drone is correctly connected. Being correctly connected can only occur when the exploration begins or by triggering manually the topic *"/connection"*.
In the case that the drone is flying, we may need to correct the orientation. This procedure is done every time the heading of the drone has drifted more than 5 degrees with respect to the *map* frame. The drift can be produced by external disturbances, by errors on the motors or by small vibrations at the landing and taking off stages. To correct the orientation, the API provides two functions: the first one to get the orientation of the drone using the integrated IMU, and the second one to rotate the drone some degrees in the z axis. Stopping the movement of the robot and using those 2 functions, the drone can be always aligned with our reference frame.
Once the orientation is corrected, we need to check the battery of the drone, again, using the API. When the battery is below the 25%, the hot swap subroutine starts. During this

subroutine the robot will first check if the position of the Leo is available. In case it is not available, the drone is moved back to the home position. In case of the position being available, the drone will move to the estimated position. Additionally, the topic *"/hotswapping"* changes its state from false to true. Remember that the *goal_sender_leo()* node subscribes to the *"/hotswapping"* topic and it will be sending the goals to the Leo. Using the *"/hotswapping"* topic, the goals of the Leo equal its own current position, stopping it wherever it is. When the drone arrives to the location of the rover, it will land and will wait until the battery is changed. Due to the loss of communication, the drone will no longer be connected, making necessary the manual triggering using the *"/connection"* topic.

In case that the battery of the drone is enough to continue the exploration, the node will not start the hot swapping subroutine. Next, the node will get the distance to the goal and, in case it is close to the goal, it will take the next goal in the path, transform it to global coordinates, and publish it so the *tello_velcontrol()* node can move the drone to its destination.

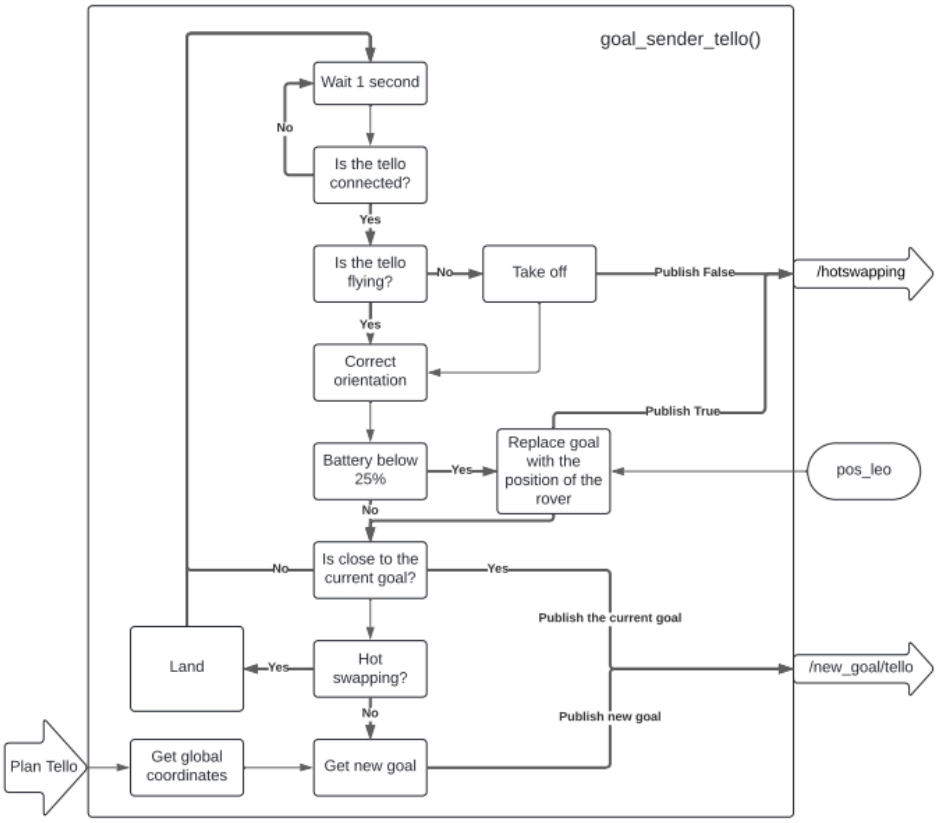


Figure 6-8 Goal_sender_tello() diagram.

7 Usage of the Proposed Solution

The final system consists in multiple computers and boards, of two different robots and a multitude of nodes and packages that has been created specifically for this project, for simulation and to use the real hardware. In order to use the set of packages, we need to execute a number of steps here summarized.

7.1 Jetson Orin Nano dev kit Preparation

The whole program runs in a Linux based operated system. The first element to configure is the Jetson Orin Nano dev kit. To prepare these types of boards is necessary to flash them with a Jetpack, a package providing the operating system and other characteristic functionalities of those boards. The Jetson Orin Nano dev kit implements the Jetpack 6.0 that includes the Linux Kernel 5.15, Ubuntu 22.04 and ROS2 – Humble installed on top. Preparing the board requires some assumptions:

- At least 1 Ubuntu-based computer (Master PC) with internet and ethernet connection is available.
- A screen, keyboard and mouse connected to the board. These elements are needed to complete the final ubuntu configuration.

To flash the system, we must download the files from the Nvidia developer webpage[3] that includes the driver package and the sample root filesystem. Once downloaded, the Jetson board should be connected to the computer using the USB-C port and turned on in force recovery mode. To turn on the board in recovery mode, place a jumper between the pins REC and GND from the 12-pin button header. When connecting the power cable to the board, it will initialize in force recovery mode.

[3] https://developer.nvidia.com/embedded/jetson-linux-r363 (last update May 2024)

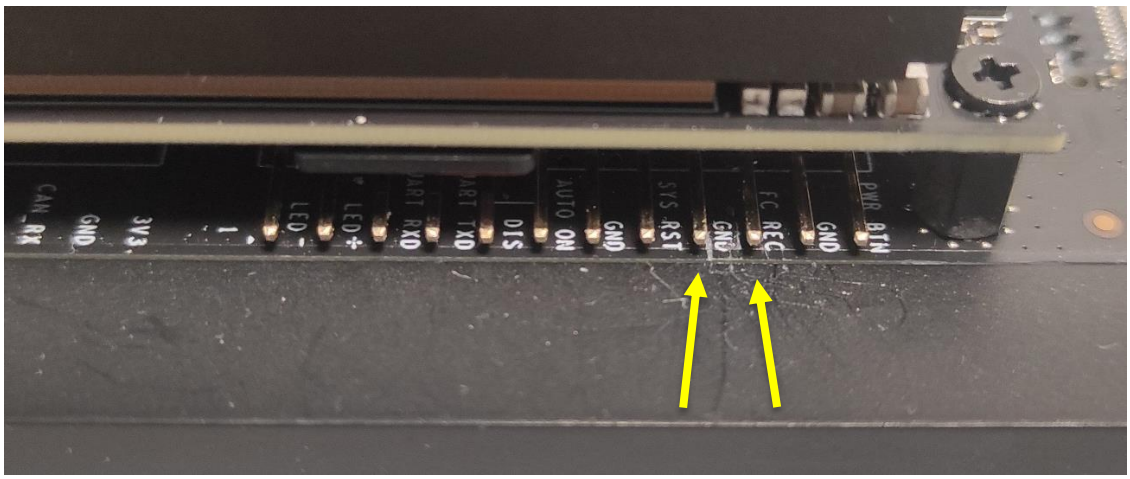


Figure 7-1 Pins to start the Jetson Orin Nano dev kit in force recovery mode.

To check that the board is in recovery mode execute:

```
$ lsusb | grep Nvidia
```

If the system is correctly in force recovery mode it should appear a message. At this point the board can be flashed with the downloaded software. For exact instructions, go to *Appendix A "Flashing the Jetson Orin Nano dev kit".*

Once the board is flashed, the system should reboot and the jetson is operational. Now, the ROS system can be installed. For extensive instructions on how to install ROS2, go to *Appendix B "Installing ROS".*

7.2 Workspace Preparation

The next step is to create the workspaces for the packages and to build them but first we need to connect to the Raspberry Pi and the Jetson Nano using ssh connection. To connect to the Raspberry Pi, the computer must be connected using WI-FI to the Raspberry Pi. Using the program MobaXterm a connection to the IP address 10.0.0.1 can be created.

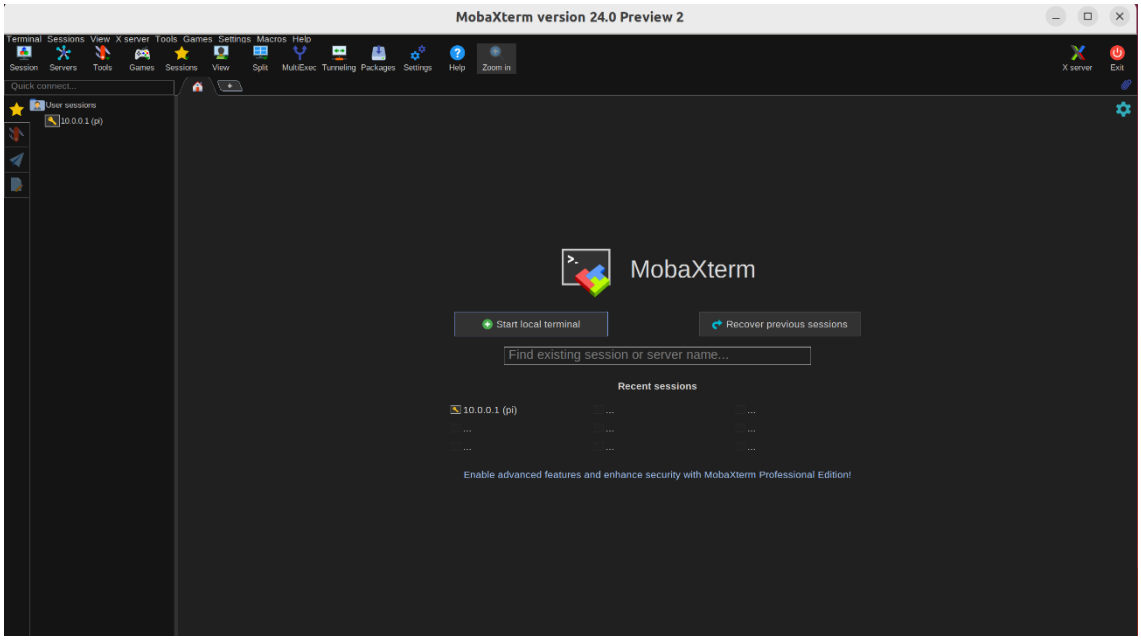


Figure 7-2 MobaXTerm.

Once connected to the Raspberry Pi, we can enter to the Jetson Nano using the command:

```
$ ssh ubuntu@leorover1
```

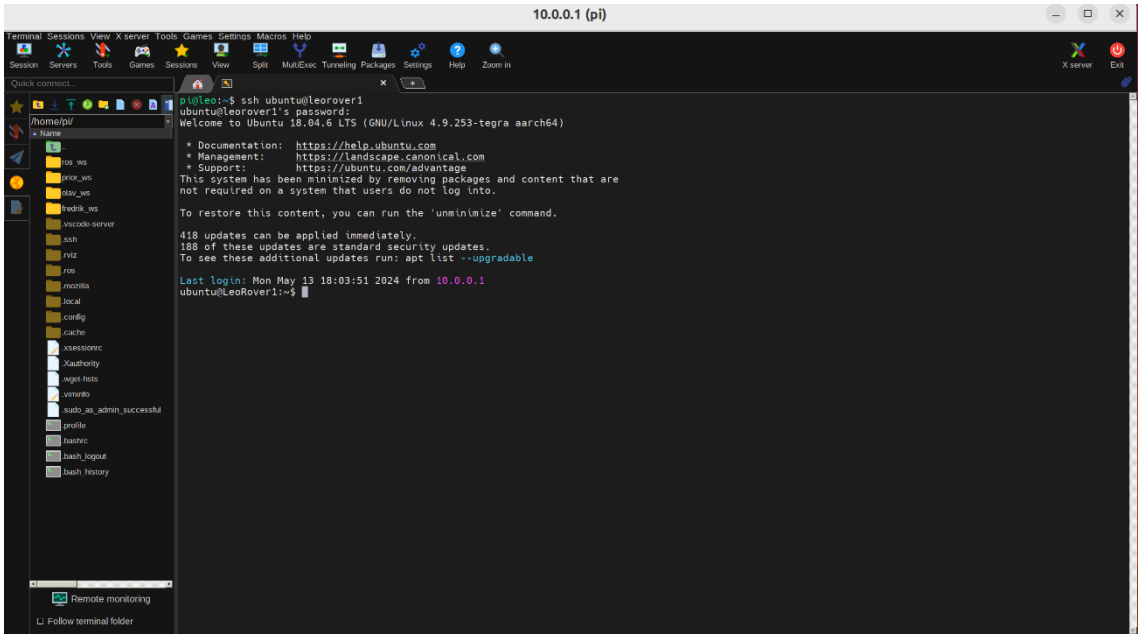


Figure 7-3 Ssh connection to the Jetson board using MobaXTerm.

At this time, the window we created is connected to the Jetson Nano. MobaXTerm allows multiple connections so in order to return to the Raspberry Pi, we can create a new connection in a different tab of the screen.

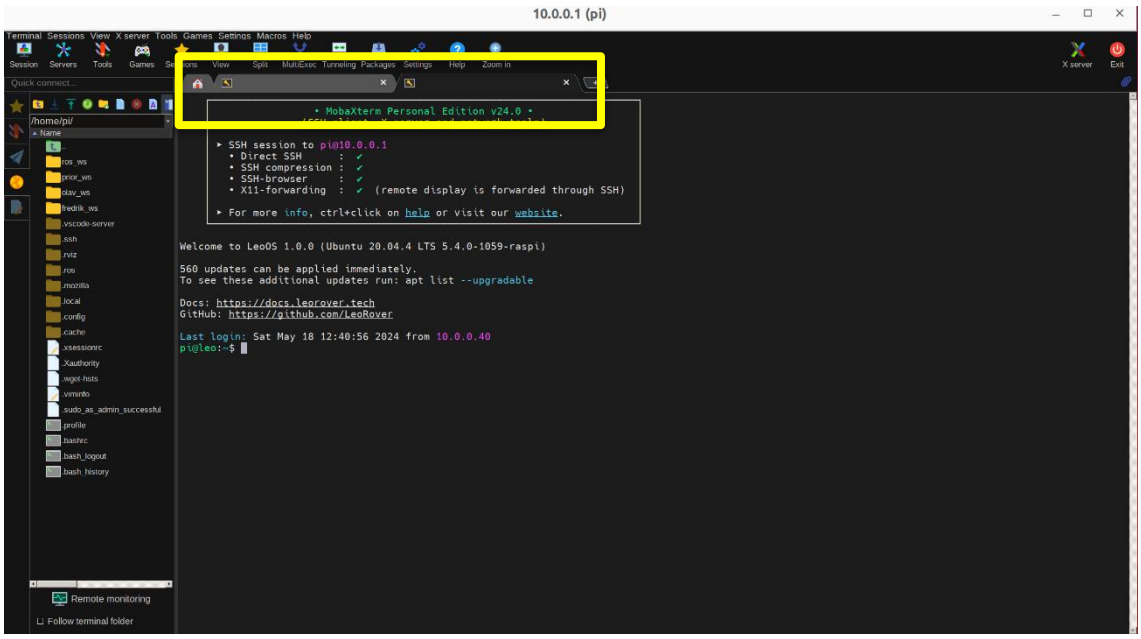


Figure 7-4 Multiple ssh connections using MobaXTerm.

Now that all that the computers and boards are accessible, we can create the workspaces in each one of the systems.

1. Jetson Nano: In this system the workspace is already prepared, we just need to access it and source it:

   ```
   $ cd ros_ws/
   $ source devel/setup.bash
   ```

2. Raspberry Pi: In this board the workspace needs to be created from scratch. To do so execute these commands:

   ```
   $ mkdir -p ~/prior_ws/src
   $ cd prior_ws/src
   $ git clone git@github.com:LluisPriorSancho/CMMR_for_ACPP.git
   $ git checkout ROS1
   $ cd ..
   $ catkin build   #or catkin_make, depending on the catkin tool
   $ source devel/setup.bash
   ```

   With these commands a workspace is created, the packages are cloned and build and the environment is sourced. Now the Raspberry Pi is ready to launch some nodes.

3. Master PC: Similarly to the Raspberry Pi, the workspace needs to be created. The difference is that now, the tools used are in the ROS2 environment:

```
$ mkdir -p ~/ros2_ws/src
$ cd ~/ros2_ws/src
$ git clone git@github.com:LluisPriorSancho/CMMR_for_ACPP.git
$ git checkout ROS2
$ cd ..
$ colcon build –symlink-install
$ source install/setup.bash
```

   Now the Master PC is ready to execute some nodes.

4. Jetson Orin Nano dev kit: The process here is identical to the one in the Master PC.

7.3 The Launch System

Thanks to the launch files created in the different packages, the system can run several nodes using single line commands. We can use these files in a multitude of ways to execute the system and to adapt it to our situations. The launch files include a series of arguments that allow the customization or selection of the nodes that will run.

- *rover_control.launch.py*: Runs the files related to the Leo rover; *decawave.py*, *leo_control.py*, *system_control_leo.py* and *ekf_node()*
- *tello_control.launch.py*: Runs the files related to the Tello drone; *decawave.py* and *system_control_tello.py*
- *complementary.launch.py*: Runs the non-essential files; *demo.py*, the program rviz2 and the rosbag functionality.
- *sim_control.launch.py*: Runs all related node, file or program related to the simulation, including *simulation.py*.

There are several important files to consider when running the system and each of them presents different arguments:

Table 7-1 Descriptions of the available arguments present in the system.

Name of the argument	Description	Launch file available

first_start	This argument controls if the commands *"\r\r"* and *"lep"* are sent to the decawave interface when executing the *decawave.py* file. (**Default:** false)	*total_control.launch.py*
decawave_leo	This argument controls if the decawave node should be executed with the nodes related to the Leo rover. If using the total_control.launch.py, this node should be set to false. (**Default:** false)	*rover_control.launch.py* (can be inherited from *total_control.launch.py*)
decawave_tello	Similar to the decawave_leo argument but with the nodes related to the Tello drone. (**Default:** false)	*tello_control.launch.py* (can be inherited from *total_control.launch.py*)
complementary	This argument allows to use the set of non-essential nodes: rviz, demo.py and rosbag. (**Default:** true)	*total_control.launch.py*
rviz	This argument decides if rviz2 is executed. The complementary argument must be set to true. (**Default:** true)	*complementary.launch.py* (can be inherited from *total_control.launch.py*)
path_visual	This argument decides if demo.py is executed. The complementary argument must be set to true. (**Default:** true)	*complementary.launch.py* (can be inherited from *total_control.launch.py*)
rosbag	This argument decides if the rosbag functionality is executed. The complementary argument must be set to true. (**Default:** false)	*complementary.launch.py* (can be inherited from *total_control.launch.py*)
leo	This argument enables the execution of the nodes related to the Leo rover. Launches *rover_control.launch.py* (**Default:** true)	*total_control.launch.py*

tello	This argument enables the execution of the nodes related to the Tello drone. Launches *tello_control.launch.py* (**Default:** false)	*total_control.launch.py*

7.4 Running the System

The system has been tested using simulation and real hardware. Even though the simulation is a much simpler scenario, has been a good first approach to familiarize with some of the problems encountered during the process. The simulation can be executed using:

```
$ ros2 launch real_hardware sim.launch.py
```

To take off and land the drone:

```
$ ros2 service call /drone1/tello_action
tello_msgs/srv/TelloAction "{cmd: 'takeoff'}" # or cmd: 'land'
```

And to simulate the hot swap subroutine:

```
$ ros2 topic pub -1 /hotswap std_msgs/msg/Empty "{}"
```

On the other hand, before launching any file in the real hardware, we need to make sure that our system is ready to execute it properly:

a. The Leo rover is turned on and placed at the start point.
b. Using MobaXTerm, we have ssh connection to the Jetson Nano.
c. In a second tab of the MobaXTerm, we have connection to the Raspberry Pi equipped in the Leo rover.
d. The Master PC and the Jetson Orin Nano dev kit are connected using ethernet connection, with the same ROS_DOMAIN_ID (by default falls into the 0).
e. The decawave interface is connected using USB to the Master PC.
f. The drone is turned on, placed at the starting position and connected using WI-FI to the Jetson Orin Nano dev kit.
g. The drone carries one decawave antenna (modified) that is turned on.
h. The rover carries one decawave antenna that is turned on.

i. Every board is in the workspace created and with the packages correctly sourced.

Now that the boards are set up, ROS is prepared in the different machines and the workspaces have the packages ready, we can execute the correct launch files with the correct arguments.

- On the Jetson Nano:

```
$ roslaunch leorover frames_launch.launch
```

- On the Raspberry Pi:

```
$ roslaunch conversion_pkg connection.launch
```

- On the Master PC:

```
$ ros2 launch real_hardware total_control.launch.py
```

- On the Jetson Orin Nano dev kit:

```
$ ros2 launch real_hardware tello_control.launch.py decawave_tello:=false
```

The previous commands should start the frames of the rover, create a ROSbridge connection, execute all the nodes for the Leo and execute the nodes for the Tello. The usage of the *total_control.launch.py* file makes the system run also the complementary files as *demo.py*, rviz2 and to record the topics into the *data* folder. Thanks to the shareability of ROS2, in the Jeton Orin Nano dev kit, we can just run the nodes for the Tello drone. For the same reason, the *decawave.py* file is running in the Master PC thus is not necessary to run it again in the Jetson board. The figure 7-5 present the full system and architecture of the nodes and topics. The squares represent the topics, the balls represent the nodes, in green the nodes running from the file *system_control_tello.py*, in blue the nodes running from the file *system_control_leo.py*, in orange the nodes running from the file *leo_control.py* and in purple the nodes running from the file *demo.py*. Additionally, from the *leo_controller()* node, there is the connection to the Leo rover in ROS1.

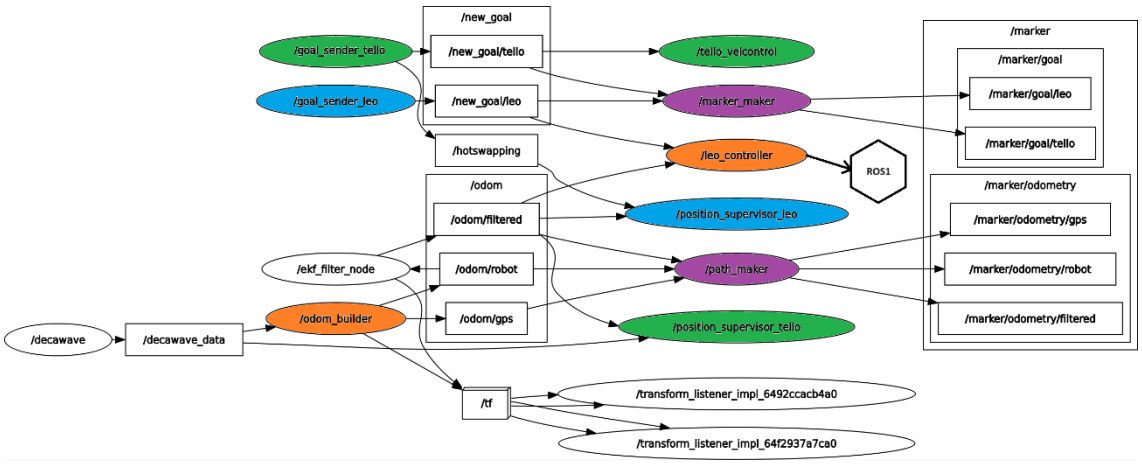


Figure 7-5 Final graph of nodes.

Once the hot swap subroutine starts, the drone will approach the rover and land. At that moment the drone should be manually turned off, the battery must be replaced by a full one and the drone should be turned on again. At that moment, an Empty message must be sent to the topic *"/connection"*. The rest is done automatically by the system. To send the topic:

```
$ ros2 topic pub -1 /connection std_msgs/msg/Empty "{}"
```

It is possible that the topic fails in the connection. Make sure that the drone has completely started the WI-FI before sending the connection topic (this can be done checking if the WI-FI of the jetson board is connected to the drone).

8 Results

After executing the commands, the robots have performed an autonomous exploration with a hot swap subroutine. This behaviour has been tested in simulation and in real-life. The video-results can be found at:

Simulation

Cooperative Multi-Modal Robots for Autonomous Coverage Path Planning

The first step is to compute the plans for the robots using the wavefront algorithm. The presence of multiple robots requires to compute multiple paths for each one of the robots. The figure 8-1 represents the plans of the robots, at the left the one for the UGV and at the right the one for the UAV. It is interesting to notice how the UGV reaches the goal position without covering the whole area so the exploration should continue until the whole region is covered. After reaching the final cell, the robot must go back to the final destination. This behaviour can also be seen in the real exploration. Contrary to the rover plan, the drone plan is simple, direct and straightforward due to the lack of obstacles.

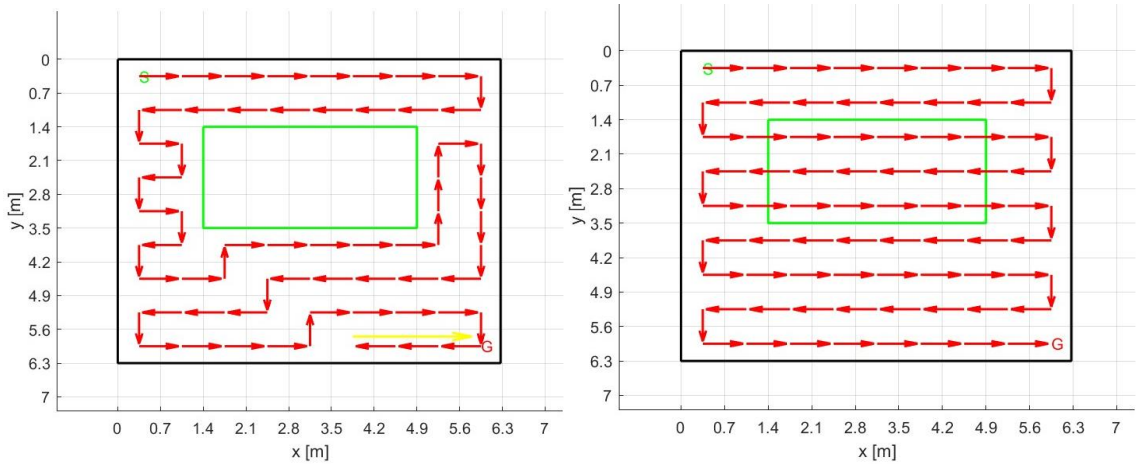


Figure 8-1 Final path plans for the robots.

Now we will focus on the results when using the real hardware. Using the data obtained using the rosbag functionality, we can plot the trajectory of the robots and the goals while performing the exploration.

On one hand, in the figure 8-2 we can see how the Leo rover performs a nice exploration, being the green line the location of the robot using the Kalman filter while the dots

represent the goals the rover had to reach. The starting point of the exploration is the top left corner while the bottom right goal is the final destination. The exploration has sent the goals in the correct order and within a range of 30 cm from the previous one, reducing the number of turns thanks to the wavefront algorithm implemented. It is notable how at the end of the exploration, the rover, even being at the goal position, continues three more goals to cover the whole area to finally go back to the goal destination.

One the other hand, the pose of the drone is obtained using the decawave beacons, providing a more inaccurate and noisy measurement. The figure 8-3 represents the journey of the drone with a green line and the goals with purple dots. The positions of the drone during the hot swap have been removed.

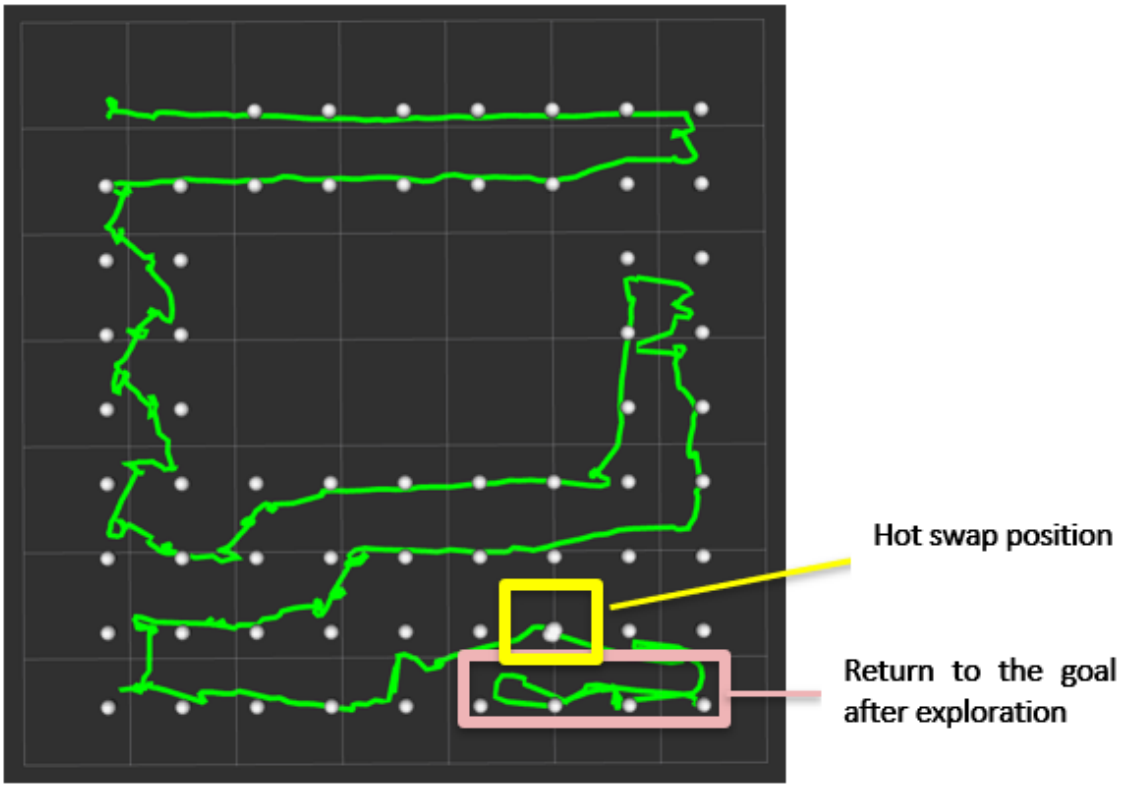


Figure 8-2 Path of the Leo rover.

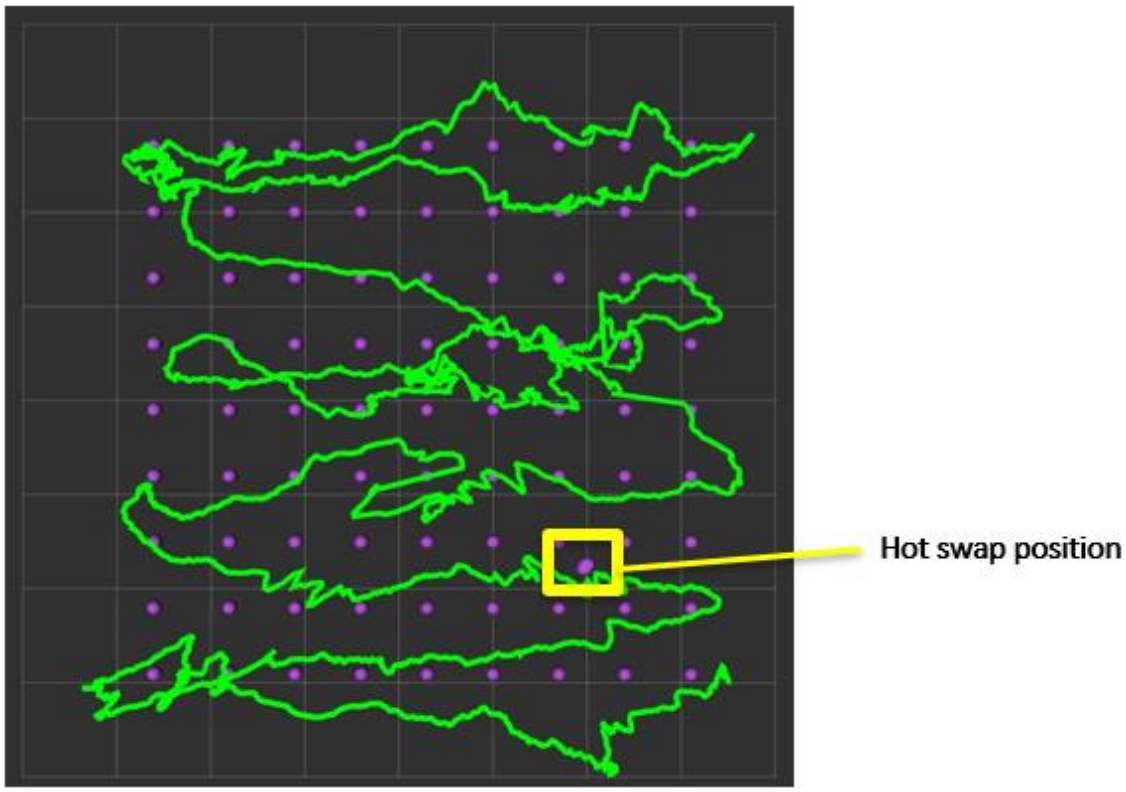


Figure 8-3 Path of the Tello drone.

The drone has completed the full coverage using 2 batteries, that is 1 hot swap as it can be seen in the video. During this exploration the drone has correctly reach each goal within a range of 40 cm but there are some interesting remarks. First of all, the drone has been able to fly over the table, reaching those goals that the Leo could not. Secondly, we can see a high perturbance area at the mid-right part of the room. This behaviour is due to the ventilation system inside the room creating air currents opposite to the movement of the drone. The safety limits imposed in the drone had to be changed so that the drone could manage the ventilation zone more comfortably. We can notice one more time the lack of accuracy of the decawave beacons, specially the figure 8-4 represents in red an error in the measurement of the position. That measurement is clearly wrong since it would imply a teleportation of the drone and it would also suggest the drone hitting the window of the room. The behaviour of the drone during that fail can also be seen in the videos presented before (minute 3.50). Even with the wrongful actions, the drone can manage to return to its path.

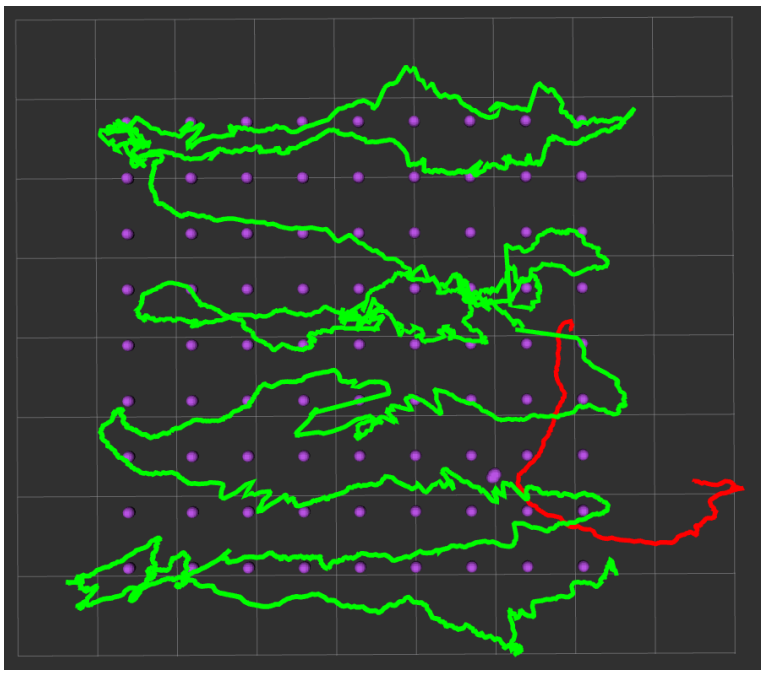


Figure 8-4 Path of the Tello drone including error in the measurement in red.

The figure 8-5 show the behaviour of the drone during the hot swap subroutine. Once the hot swap subroutine starts, the drone moves to the position of the rover and lands. Once the hot swap finishes, the drone takes off and continues the exploration. We can see how the drone does not go directly to the position on top of the robot due to the controller implemented. The controller creates an overshoot, making the robot to move more to the right area of the room instead of arriving to the rover directly. The comparative between the expected behaviour and the real trajectory can be seen in the figure 8-5.

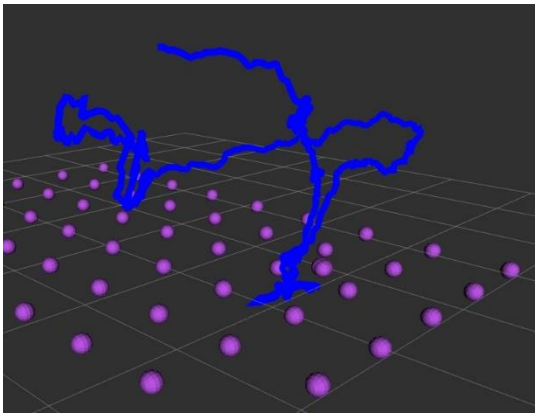


Figure 8-5 Positions of the Tello drone while hot swapping.

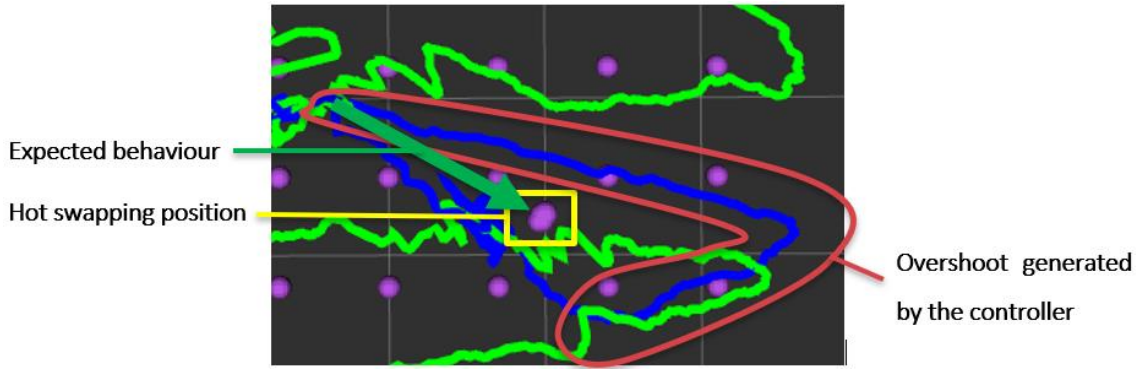


Figure 8-6 Expected behaviour and overshoot of the drone.

The complete track of the drone, including the error and the hot swap can be seen in the figure 8-7.

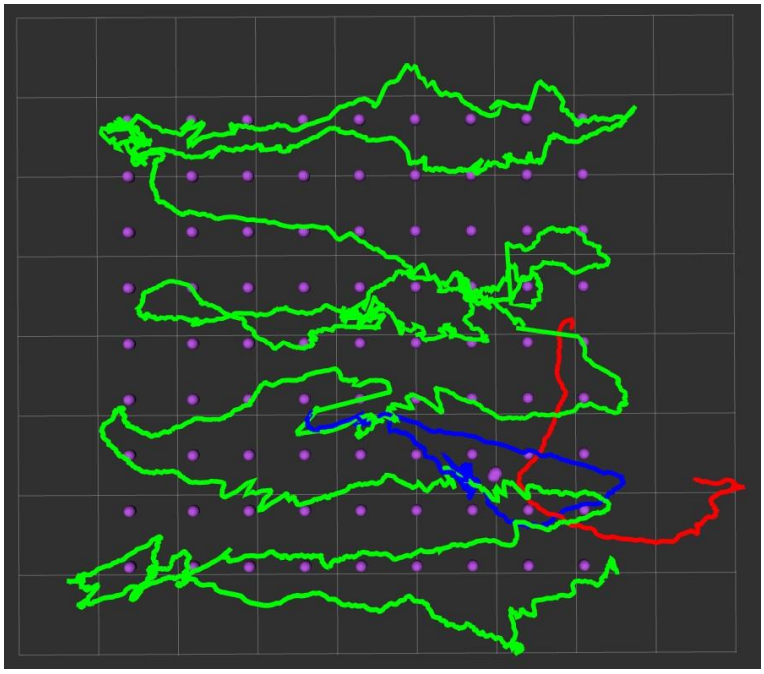


Figure 8-7 Path of the Tello drone including error in the measurement in red and hot swap in blue.

9 Discussion

The results show that is possible to implement a heterogeneous multi-robot system for exploration using coverage path planning in real-life scenarios. The use of the wavefront algorithm has generated a couple of effective plans for the UAV and the UGV, taking into account their limitations and capabilities. Through the collaboration of the robots, the limitation of the UAV's battery can be controlled, allowing for larger and continuous explorations.

The use of ROS and its modular capabilities have demonstrated, in a straightforward manner, that the communication between multiple robots is possible, even when they use different software versions. The switch, the easy shareability of ROS2 and the use of namespaces, have demonstrated that the system can be easily expanded and adapted to include multiple UAVs and UGVs. To include extra UAVs the only necessary changes in the code would be to replace the topics *"/connection"* and *"/new_goal/tello"* to *"/UAV_x/connection"* and *"/UAV_x/new_goal/tello"* where x represents an ID for each one of the drones. This change can be done using the tag *"ns"* in the launch files or using the remap option of ROS.

The system implements a series of PID controllers to compute the velocities. The results show that the use of these PID controllers produce satisfactory outcomes as the robots reach all their goals. However, despite the goals being reached, the actual position of the robots presents some overshoot and room for improvement in the control strategy. The inherit simplicity of the PID controller and its tunability allow for enhancements or the application of more advance control techniques such as model predictive control (MPC), fuzzy logic or neural networks. It is important to highlight that both robots had safety measures implemented, as limiting the velocities and a return-to-home feature.

Even though the system implements a lot of functionalities in ROS2 thanks to the Master PC, there still exist the possibility of separating the computations between robots. The robot_localization package can be implemented in the Jetson Nano in ROS1. Once the localization is ready, the same velocity control nodes used in this project could be implemented in the Jetson board, making the rover to drive to the destination. The same logic applies to the drone.

On the other hand, the decawave system allows to create multiple interfaces so multiple computers can receive the position of the tags. Using one interface in each one of the

robots reduce the computational effort of the network. It is important to mention the lack of accuracy in the localization system and the errors in the position it generates. Precise localization is required no matter the approach selected thus it is imperative to consider those aspects in future works.

Once those changes are applied, the only communication required between the robots would be the *"/hotswapping"* topic, to indicate the rover that it needs to stop. The new structure would set the bases for a decentralized optimization problem in coverage path planning.

The novel functionalities, features and characteristics of ROS2 and the upcoming discontinue of ROS1 in 2025 made us approach the system using this type of structure.

10 Conclusions

This project has shown how to implement a cooperative multi-modal robotic system for coverage path planning that involves a UGV and a UAV. The system can be executed in simulation using Gazebo sim or running it in real hardware, using the Leo rover and the Tello drone to perform the real-life exploration.

The robots are capable of cooperating in order to produce an effective exploration in a known environment. The system incorporates a set of plans developed using a robust coverage path planning technique allowing successful exploration of the environment.

The physical limitations of the UAV have limited the capabilities of the exploration but the implementation of a hot swap subroutine has increased the adaptability of the UAV to larger environments.

The usage of ROS as middleware has been crucial for the development of this project. The large community behind the open-sourced software has been essential to implement some of the functionalities used during the development of the system, guaranteeing a successful execution. Due to the transition from ROS1 to ROS2, some of the packages used had to be adapted, modified or adjusted to obtained expected behaviour. Those changes had met the requirements in our case but it cannot guarantee successful performance in other situations.

11 Future Work

The system had correctly implemented a multi-robot exploration. During the development of the project there have been some challenges that had left some room for improvement.

On one hand, the exploration of the environment is limited to the prior knowledge of the scenario. Including extra sensors into the robots may allow a dynamic exploration of a partially unknown environment. Including extra sensors as well has a map analysis, a dynamic coverage path planning could be implemented. This dynamic coverage path planning may be able to recalculate the path of a certain robot after, for example, an obstacle is present, when a hot swap routine takes place or when new information is gathered from other robots inside the exploration. Additionally, the presence of the sensor may include some collision avoidance mechanism, reducing the danger and increasing the safety of the exploration.

Other area of enhancement is the controller applied. The PID implemented leave space for improvement as using other type of controllers may produce a more robust and direct approach to the goals and to the UGV during the hot swap subroutine. Those robust controllers may enable the possibility of dynamic hot swapping, where the drone lands on the rover while moving. This approach can include the development of robust control techniques or visual servoing and setting the bases for future research projects in the cooperative navigation field.

During the report has been mentioned multiple times that the system is practically prepared to include more drones but it is almost not mentioned the inclusion of extra UGVs. The addition of extra UGVs would require extra attention on how to manage multiple hot swap points and the path planning: is each drone assigned to one rover? What happens when a drone has low battery and all charging points are taken? It is efficient to continue the exploration where it was left before charging or it is necessary to recompute the path considering the other drones? These questions and others could imply a reconstruction of the system here presented.

It is important to highlight the challenges presented by the localization and heading of the robots. Including special heading sensors for the robots as well as improving the accuracy of the location system will create a more exact and precise control system. Another important improvement could imply a Kalman filter for the localization for the

drone. This step would require the fusion of IMU sensor data obtained from the drone as well as the localization estimation of the beacons.

References

[1] Pillai, B. M., Suthakorn, J., Sivaraman, D., Nakdhamabhorn, S., Nillahoot, N., Ongwattanakul, S., … Magid, E., “A heterogeneous robots collaboration for safety, security, and rescue robotics: e-ASIA joint research program for disaster risk and reduction management.,” *Advanced Robotics,* pp. 129-151, 2024.

[2] Berger, Guido S., Marco Teixeira, Alvaro Cantieri, José Lima, Ana I. Pereira, António Valente, Gabriel G. R. de Castro, and Milena F. Pinto., “Cooperative Heterogeneous Robots for Autonomous Insects Trap Monitoring System in a Precision Agriculture Scenario,” *Agriculture,* vol. 2, p. 239, 2023.

[3] G. de Castro, T. Santos, F. Andrade, J. Lima, D. Haddad, L. Honório and M. Pinto, “Heterogeneous Multi-Robot Collaboration for Coverage Path Planning in Partially Known Dynamic Environments,” *Machines,* vol. 200, p. 12, 2024.

[4] Z. J. N. &. C. A. A. Yan, “A Survey and Analysis of Multi-Robot Coordination,” *International Journal of Advanced Robotic Systems,* 2012.

[5] H. Choset, “Coverage for robotics - A survey of recent results,” *Annals of Mathematics and Artificial Intelligence,* 2001.

[6] Alexander Zelinsky and R.A. Jarvis and J.C. Byrne and Y. Shin'ichi, “Planning Paths of Complete Coverage of an Unstructured Environment by a Mobile Robot,” 2007.

[7] Open Robotics, “Ros - Robot Operating System,” 19 05 2024. [Online]. Available: https://www.ros.org/. [Accessed 19 05 2024].

[8] C. S. Tan, R. Mohd-Mokhtar and M. R. Arshad, “A Comprehensive Review of Coverage Path Planning in Robotics Using Classical and Heuristic Algorithms,” *IEEE Access,* pp. vol. 9, pp. 119310-119342, 2021.

[9] Leo Rover, “Leo Rover - Github,” 2024. [Online]. Available: https://github.com/LeoRover. [Accessed 19 05 2024].

[10] Nvidia, “NVIDIA Jetson Nano,” 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/. [Accessed 22 05 2024].

[11] Nvidia, “Robotics and Edge Computing,” 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. [Accessed 23 05 2024].

[12] NVIDIA Corporation , “Nvidia Developer,” [Online]. Available: https://developer.nvidia.com/embedded/jetpack. [Accessed 19 05 2024].

[13] Nvidia Corporation, 2024. [Online]. Available: https://developer.nvidia.com/embedded/jetpack-sdk-461. [Accessed 22 05 2024].

[14] DJI, 2024. [Online]. Available: https://www.dji.com/no. [Accessed 22 05 2024].

[15] IntelRealSense, 2024. [Online]. Available: https://www.intelrealsense.com/visual-inertial-tracking-case-study/. [Accessed 22 05 2024].

[16] Y. Mohd Mustafah, A. Azman and M. H. Ani, “Object Distance and Size Measurement Using Stereo Vision System,” *Advanced Materials Research,* pp. 622-623, December 2012.

[17] A. Saxena, S. H. Chung and A. Y. Ng, “Learning Depth from Single Monocular Images,” *Computer Science Department.*

[18] Open Robotics, “Wiki ROS - rosbridge_suite,” 2024. [Online]. Available: https://wiki.ros.org/rosbridge_suite. [Accessed 19 05 2024].

[19] Open Robotics, “Wiki ROS - std_msgs,” 2024. [Online]. Available: http://wiki.ros.org/std_msgs. [Accessed 19 05 2024].

[20] Open Robotics, “Wiki ROS - geometry_msgs,” 2024. [Online]. Available: http://wiki.ros.org/geometry_msgs. [Accessed 19 05 2024].

[21] Open Robotics, “Wiki ROS - nav_msgs,” 2024. [Online]. Available: http://wiki.ros.org/nav_msgs. [Accessed 19 05 2024].

[22] Open Robotics, “Wiki ROS - visualization_msgs,” 2024. [Online]. Available: http://wiki.ros.org/visualization_msgs. [Accessed 19 05 2024].

[23] Open Robotics, “Wiki ROS - robot_localization,” 2024. [Online]. Available: https://wiki.ros.org/robot_localization. [Accessed 19 05 2024].

[24] O. Robotics, “Wiki ROS - move_base,” 2024. [Online]. Available: https://wiki.ros.org/move_base. [Accessed 19 05 2024].

List of tables and charts

Appendix

Appendix A: Flashing the Jetson Orin Nano dev kit

Once the system is connected and in recovery mode, inside the folder containing the downloaded file execute the following commands to prepare the environmental variables:

```
$ export L4T_RELEASE_PACKAGE = Jetson_Linux_r36.3.0 _aarch64.tbz2
$ export SAMPLE_FS_PACKAGE = Tegra_Linux_Sample-Root-
Filesystem_r36.3.0_aarch64.tbz2
$ export BOARD = jetson-orin-nano-devkit
```

To untar the files:

```
$ tar xf ${L4T_RELEASE_PACKAGE}
$ sudo tar xpf ${SAMPLE_FS_PACKAGE} -C Linux_for_Tegra/rootfs/
$ cd Linux_for_Tegra/
$ sudo ./tools7l4t_flash_prerequisites.sh
$ sudo ./apply_binaries.sh
```

The flashing phase can be done to a SD card, a USB or a NVMe. In this tutorial we consider the SD card. To flash the system to the SD card, execute:

```
$ sudo ./tools/kernel_flash/l4t_initrd_flash.sh --external-
device mmcblk0p1 \ -c
tools/kernel_flash/flash_l4t_t234_nvme.xml -p "-c
```

```
bootloader/generic/cfg/flash_t234_qspi.xml" \ --showlogs --
network usb0 jetson-orin-nano-devkit internal
```

Appendix B: Installing ROS2

To install the middleware, execute the next commands in the terminal:

```
$ sudo apt update && sudo apt install locales
$ sudo locale-gen en_US en_US.UTF-8
$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
$ export LANG=en_US.UTF-8
$ sudo apt install software-properties-common
$ sudo add-apt-repository universe
$ sudo apt update && sudo apt install curl -y
$ sudo curl -sSL
https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
-o /usr/share/keyrings/ros-archive-keyring.gpg
$ echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/ros-archive-keyring.gpg]
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release &&
echo $UBUNTU_CODENAME) main" | sudo tee
/etc/apt/sources.list.d/ros2.list > /dev/null
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install ros-humble-desktop
$ sudo apt install ros-dev-tools
$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```