Coordination and control of a group of ground robots

Pierre Chassagne

Semester project

Under the supervision of:
Dr. Tony A. Wood and Prof. Maryam Kamgarpour

28.06.2022

sycamore lab
SYSTEMS CONTROL AND MULTIAGENT OPTIMIZATION RESEARCH

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract

The aim of this project was to assemble, interconnect, and program a group of ground robots. The objective of the robot control is to cooperatively complete a set of tasks where each task represents a goal location that needs to be visited. For the hardware implementation so-called JetBots, which are based on the Nvidia Jetson Nano platform, were used in connection with a visual motion tracking system by OptiTrack. In this presentation, the core concepts of the implementation will be presented and the results will be demonstrated with a multi-robot experiment. This report presents the core concepts behind the implementation of such a framework and finally presents how each one of these components had been put together to form the entire system.

Contents

Acknowledgement

I would like to express my deepest appreciation to Dr. Tony A. Wood for its endless support and valuable advice and Prof. Kamgarpour for her kindness and feedback. Finally, I would like to express a very special thanks to Max Polzin who generously provided knowledge and expertise.

Chapter 1

Introduction

Multi-robot systems are becoming increasingly popular and are being used in more and more applications such as warehouses, agriculture, and transportation. Controlling a single ground robot requires planning the desired path and computing actuation inputs to maneuver the robot based on the feedback of its state. Given a set of interchangeable tasks for a multi-agent system, such as a group of robots, the first decision that needs to be made is to determine which agent does what, e.g., which robot goes to which location of interest while avoiding collisions. Numerous algorithms and concepts have been explored in this domain, but research often lacks a system to put theory into practice.

Therefore the aim of this project is to assemble, interconnect, and program a group of JetBots such that every predefined task is completed by one robot. The tasks are given by random goal locations that need to be visited. To achieve this JetBots are been used, they are powerful ground robots provided by Nvidia. However, while the robot and the provided libraries are open-source it appears that their utilisation is mainly recreational or at educative ends, to introduce the basic concepts of Artificial Intelligence (AI) and robotics. Moreover, it appears that no projects have been conducted or made available regarding the coordination and control of a group of Jetbots. Yet, because they embed a Graphic Processing Unit (GPU) and a great computational power thanks to the Jetson Nano, these robots could be used to put in practice some advanced control and coordination algorithms to enable real validation of these concepts. This report presents the core concepts behind the implementation of such a framework and finally presents how each one of these components has been put together to form the entire system. Further documentation and the two packages issued with this work can be found on Github [1, 2].

Chapter 2

Hardware & software presentation

This section will present the hardware and software that were used for the project. Some hardware and software were required, while others have been used for their efficiency and their usefulness. The hardware is composed of five key elements :

- the motion capture system: OptiTrack,
- two computers: the central PC and the motion capture system executor,
- a router : for the wireless communication,
- the robots provided by Waveshare and Sparkfun : the JetBots.

2.1 OptiTrack

OptiTrack is the optical motion capture system provided by the eponymous company, it uses the software Motive [3] in order to compute and make available the positions of the object which were defined by the user. it is used to simulate a reliable tracking system which can stream at 240 Hz precise position and orientation of the JetBots. It is paired with one of the two computers in order to execute the software. This computer is responsible for this only task to reduce at the maximum the latency, it uses a Intel Core i9-10900K CPU @ 3.70 GHz and has 1T of disk capacity as well as 31 GiB of memory, we will call it the OptiTrack PC.

2.2 Central PC

The central PC is running on Ubuntu 20.04.4 LTS. It uses a Intel Core i9-10900K CPU @ 3.70 GHz and has 1T of disk capacity as well as 33,3 GB of memory. This PC is responsible for the centralize data collection, the centralized coordination of the robots while doing the interface between the execution, the robot and the user.

2.3 Jetbots

The Jetbots are open-source ground robots based on NVIDIA Jetson Nano. Jetson Nanos are small powerful computers designed to enable power entry-level edge AI applications and devices. It features a Quad-core ARM Cortex-A57 MPCore @ 1,6GHz processor, a 4GB 64-bit memory and a GPU based on NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores. It has multiple ports for USB communication as well as ethernet. It also supports various communication protocols such as I2C, I2S, SPI, UART and has various GPIO ports. When delivered by Waveshare the JetBots are ready to mount and come with an adafruit PiOLED display, motors, a camera and an intel Dual Band Wireless-AC 8265 for wifi communication.

The Jetson Nano was configured by flashing the SD-card using the image provided by Nvidia [4]. It runs on Linux L4T. NVIDIA Linux4Tegra (L4T) package provides the bootloader, kernel, necessary firmwares, NVIDIA drivers for various accelerators present on Jetson modules, flashing utilities and a sample filesystem to be used on Jetson systems [5].

2.4 Robotic Operating System 2

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications [6]. ROS2 is an upgraded version of ROS, in this project the foxy distro (version in the ROS language) of ROS2 is used, which at the time of the beginning of the project was the most recent distro of ROS2. Compared to ROS1, the design goals of ROS2 are listed below [7]:

- Support multiple robot systems: ROS2 adds support for multi-robot systems and improves the network performance of communication between multi-robots.
- Bridging the gap between prototypes and products: ROS2 is not only aimed at the scientific research field, but also concerned about the transition from research to application of robots, which can allow more robots to directly carry ROS2 systems to the market.
- Support microcontroller: ROS2 can not only run on existing X86 and ARM systems, but also support embedded microcontrollers like MCUs (ARM-M4, M7 cores).
- Support real-time control: ROS2 also adds support for real-time control, which can improve the timeliness of control and the performance of the overall robot.
- Multi-platform support: ROS2 not only runs on Linux systems, but also adds support for Windows, MacOS, RTOS and other systems, giving developers more choices.

Presenting ROS2 in detail is out of the scope of this report. However, the article [8] from the European Training Network for Safer Autonomous Systems is a great and concise introduction to ROS2 and its technical particularities, especially regarding its communication protocol the OMG Data Distribution Service (DDS). However, here is a short introduction of the main component of a ROS2 framework, that will help the reader understand the concepts in this report, all of the description and pictures in the following discussion is greatly inspired by the ROS2 documentation [9]:

2.4.1 Topics

Nodes publish information over topics, which allows any number of other nodes to subscribe to and access that information. A Node is a fundamental ROS2 element that serves a single, modular purpose in a robotics system. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Where a Topic subscriber and publisher can both exist in the same node. Figure 2.1, is a schematic of four nodes communicating through a topic.

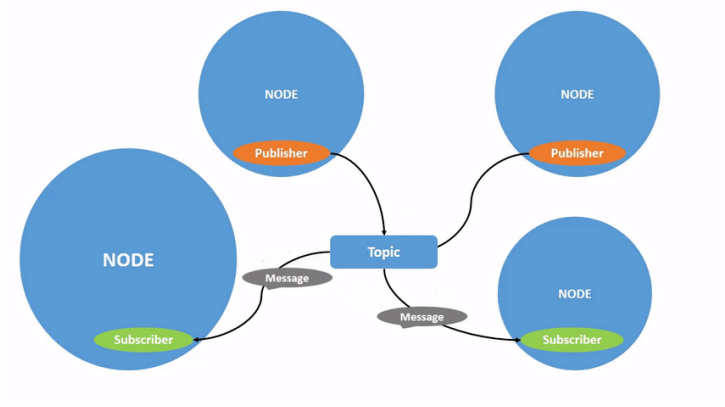

Figure 2.1: *2 publishers and 2 subscribers nodes communicating through 1 topic.*

2.4.2 Services

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model, versus topics' publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client. There can be many service clients using the same service. But there can only be one service server for a service. In general, it is not desired to use a service for continuous

calls; topics or even actions would be better suited. Figure 2.2 shows three nodes using a service to communicate.

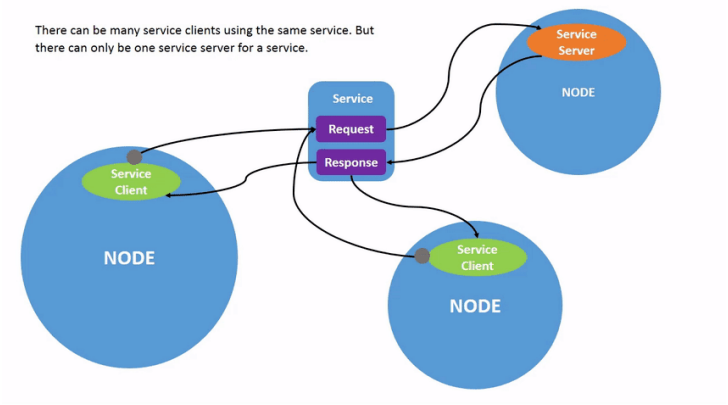

Figure 2.2: *2 clients and 1 server nodes communicating through 1 service.*

2.4.3 Actions

Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result. Actions are built on topics and services. Their functionality is similar to services, except actions are preemptable (you can cancel them while executing). They also provide steady feedback, as opposed to services which return a single response. Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result.

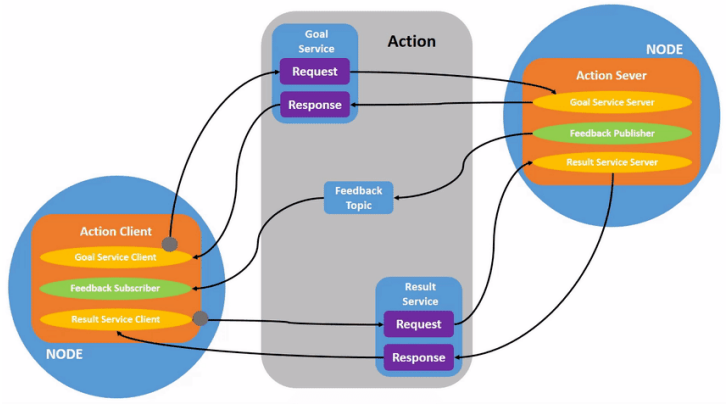

Figure 2.3: *1 action client and 1 action server nodes communicating through 1 action.*

Actions summarize all the basics notions of ROS2 seen previously and a robot system would likely use actions for navigation. For example, using the goal service, a robot could request a goal from the central PC by telling it that it is ready, the PC could then tell the robot to travel to a position in its response. While the robot navigates to the position, it can send updates along the way (i.e. feedback) on its position or state using the feedback topic, and then a final result message once it has reached its destination using the result service.

2.4.4 Parameters, .srv and .msg files

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists. In ROS 2, each node maintains its own parameters.

.srv and .msg files are files which describe different interfaces (i.e. type and format of messages) for services, actions and topics. Most of the time it is good practice to use the predefined ones. However,

there are cases where it is useful to write our own file. Figure 2.4 shows a custom .srv file which defines the type of message passed through a service. The information given above the — line defines the format of the request, whereas the one under this line defines the format that the response should take. A .msg is define the same way except that it does not allow for a — line, indeed messages are send through topics that are unidirectional, however a message could take as many arguments as needed.

```
# Custom .srv file

float x
float y
float z
---
bool start
```

Figure 2.4: *Custom .srv file*

Finally, we can refer to the connected set of element that constitute a ROS2 system as the "ROS graph". It is a network of ROS2 elements processing data together at one time. It encompasses all executables and the connections between them if you were to map them all out and visualize them. Later in this report we will see how these different elements can interact to form the entire system.

Chapter 3

Architecture

In this section we will present the general architecture of the system by looking at diagrams which represent the ROS graph as well as some implementation examples. Figure 3.1 shows the legend for the diagrams in this section, please refer to it.

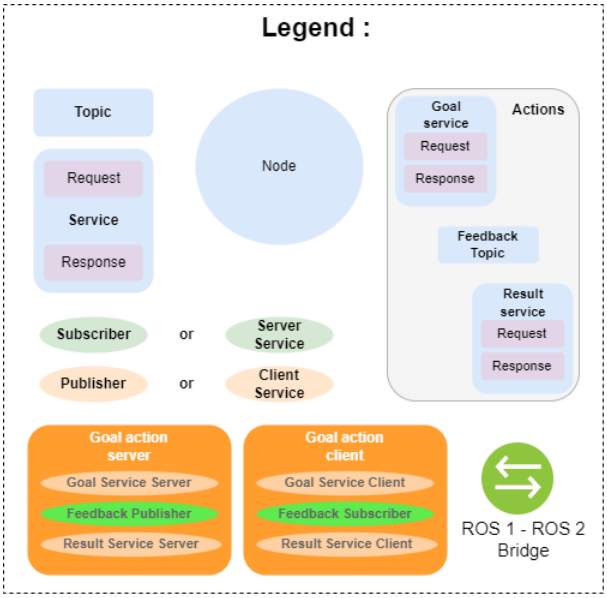

Figure 3.1: *Legend for the diagrams.*

3.1 General Overview

Figure 3.2 shows the ROS graph for the system with the central PC communicating with one JetBot called SycaBot_Wx where $x = 1, ..., N$ corresponds to the identifier of the JetBot. All the topics published by the JetBot and the Central PC are available through the network and can be discovered by every node through the DDS protocol that ROS2 uses. On the links between the different topics and services are the message names and their types, displayed in brackets. The system was designed to be scalable to any number of robots by asking for an user input at startup of the Central PC nodes. Since each robot is responsible for its own control the usage of an action is not necessary in this configuration. However, for more complex applications, actions could become essentials. While the Central PC has information on every Jetbos, the Jetbots do not communicate with each other directly in this configuration.

Nevertheless, and this might be on of the main advantage of ROS, if one would want to implement distributed control this could easily be achieved by simply creating a new node in each Jetbot which subscribe to the pose topics of the other Jetbots and create a new service for the other Jetbots to exchange information.

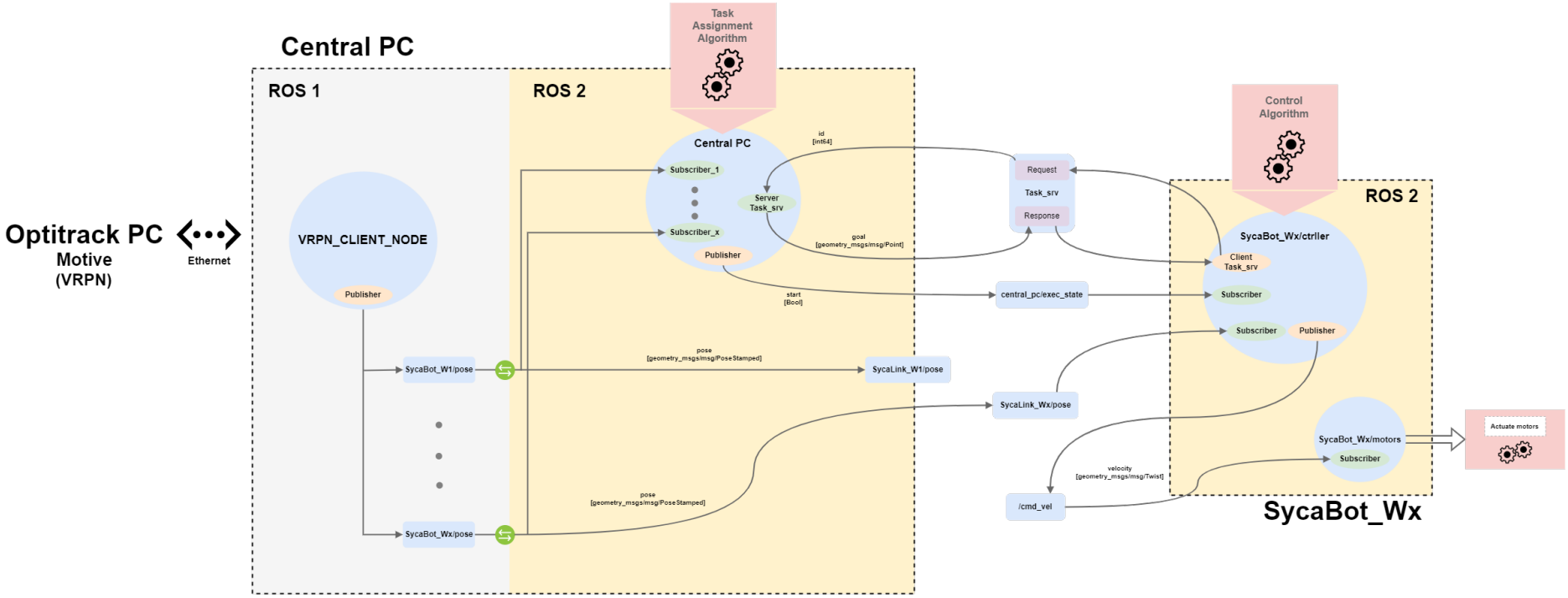

Figure 3.2: *ROS Graph for the the system with one JetBot.*

3.2 Central PC

Figure 3.3 shows the central PC's ROS graph. The Optitrack PC is responsible for gathering the poses of the robots at anytime using the motive software [3] which sends to the Central PC them through Virtual-Reality Peripheral Network (VRPN). The central PC gets the poses using the ROS1 **vrpn_client_ros** package [10]. The built-in ROS1->ROS2 bridge is then used to make these poses available on ROS2 topics which are accessible by any other nodes running on ROS2.

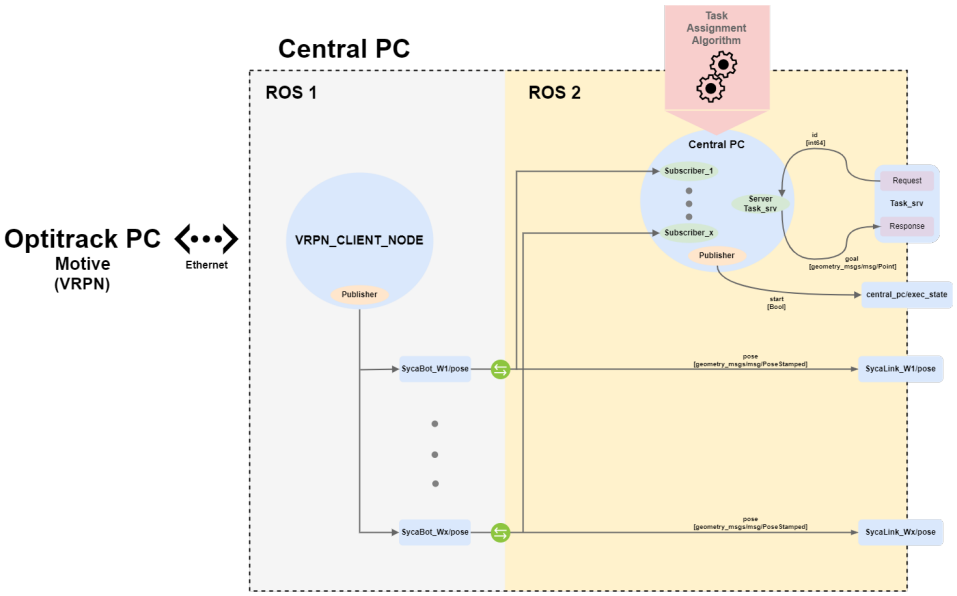

Figure 3.3: *Close-up view for the central PC's ROS graph.*

Furthermore, the central PC node is responsible for generating random tasks and assigning them to a Jetbots. The task assignment algorithm will be discussed in the next section. In order for the Jetbot to get its task it needs to request it through the **task_srv** service by sending its id, then the central PC sends back its task and updates the list of ID to know which Jetbot already knows its task. Once all the Jetbots have requested their task, the central PC puts the start boolean to True and publishes it on the **exec_state** topic so that all the Jetbots can start executing the control approximately at the same time.

3.3 Jetbot

Figure 3.4 shows the jetbots' ROS graph. Obviously, the robots are responsible for the actuation of the motors, however, in a more decentralized fashion they are also responsible for their own control algorithm. Therefore, the controller node communicates velocity commands to the motors node which actuate the motors of the robot by generating a PWM to the Adafruit servo driver which operate has describe in the SunFounder wiki [11]. The motors node responsible for this task is the only node of the system that has not been fully implemented and which is directly taken from the GitHub project **jetbot_ros** [12] and adapted by modifying the topic name to which it subscribes to suit the one of our system. Finally, the Jetbots wait for the start signal - published on the **/exec_state** topic by the central PC - to initialise the control loop.

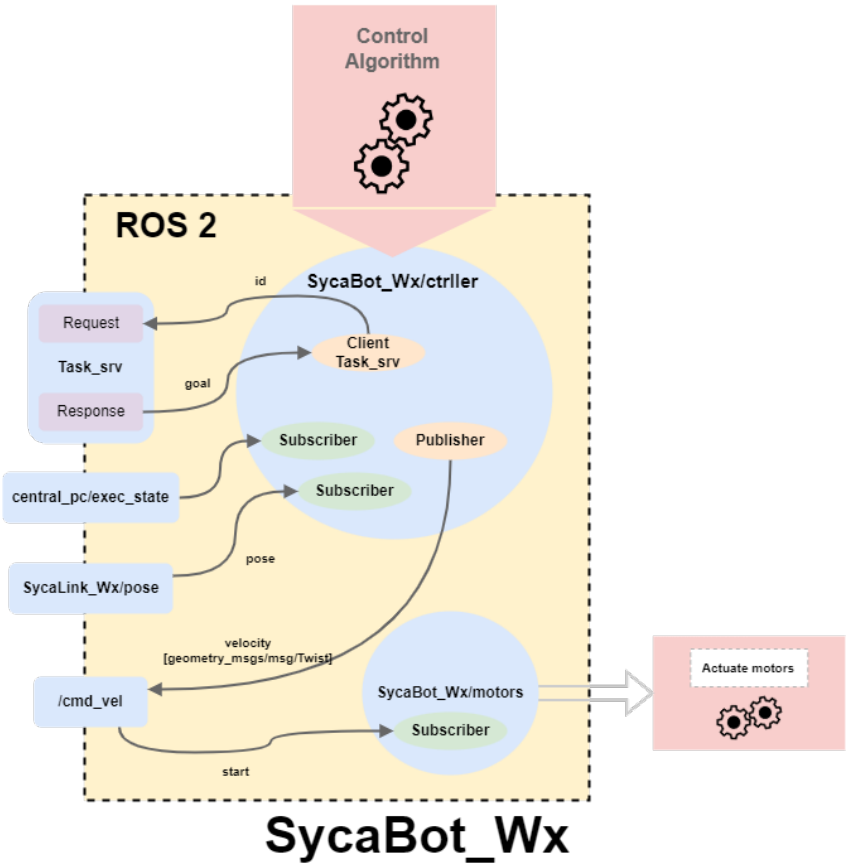

Figure 3.4: *Close-up view of the Jetbot's ROS graph.*

Finally, it is also worth mentioning that some communication issues arose at the very end of the project. It seemed that these issues were due to the communication between the Optitrack PC and the Central PC and especially because the poses are sent using the UDP protocol which is known to cause a lot of network trafic. It has been solved by plugging both PC to the router using a ethernet cables to ensure that the Wifi is not overloaded.

Chapter 4

Task Assignment Algorithm

4.1 Implementation

The task assignment [TA] algorithm used by the central PC is based on the paper [13] for the case where the number of goals M is equal the number of robots N, and the goals are generated at random. From this paper, we know that straight-line trajectories do not intersect and that if the goals and initial positions are spaced by $\Delta = 2\sqrt{2}R$, where R is the radius of the robot collision box, the individual robot velocities can be set such all robots reach their assigned goals at the same time and no collisions occur along the way. The following is the pseudo-code of the algorithm, where $\mathcal{I}_N$ and $\mathcal{I}_M$ are the set of robots and goals indices respectively, ϕ is a matrix of binary decision variables and ϕ^* is the optimal distance squared assignment matrix :

Do : generate M>N goals spaced by Δ

While M>N : Remove one goal randomly

Compute : $D_{i,j} = \|x_i(t_0) - g_j\|^2 \qquad \forall i \in \mathcal{I}_N, \forall j \in \mathcal{I}_M$

Solve : $\phi^* = \underset{\phi}{argmin} \sum_{i=1}^{N} \sum_{j=1}^{M} \phi_{i,j} D_{i,j}$

Therefore, once ϕ^* is obtained, if robot i requests its task, it is given $j-th$ task where j is the index of the 1 at the $i-th$ row of the matrix. The goals are generated using an adapted code found in this note [14]. The implementation in Python is shown on the next page.

4.2 Results

Fig.4.1 to Fig.4.3 show the goals, the robots initial positions and the trajectories generated for $N = 10, 20, 30$, the point are generated in cm in a box of 1000x400 cm which approximately corresponds to the dimension of the lab in which the experiment are conducted and the line in the trajectories have a width of 71 cm which corresponds to the space Δ required for the collision free TA algorithm. As expected, and as explained in Turpin's paper [13], all the trajectories are straight and robots are not colliding if their velocities are correctly set.

```
def set_task_cb(self, request, response):
        '''
        Compute tasks if it has never been init and give it to the asking jetbot.
        arguments :
            request (interfaces.srv/Start.Response) =
                id (int64) = identifier of the jetbot [1,2,3,...]
        ------------------------------------------------
        return :
            response (interfaces.srv/Task.Response) =
                task (geometry_msgs.msg/Pose) = pose of the assigned task
        '''
        # Step 1 : if central has never initialised goals locations, do it
        if not self.init :
            self.init = True
            k = 20 # Sample before rejection
            n_goals = 0
            while n_goals < self.n_sycabots : # We always want more goals than jetbot fo
                self.goals = generate_points(x_bound=400,y_bound=1000,r=DELTA, k=k)
                n_goals = self.goals.shape[0]
                k+=10

            while self.n_sycabots < self.goals.shape[0]: # Since we want N=M we suppress
                idx = np.random.randint(0,self.goals.shape[0])
                self.goals = np.delete(self.goals, idx,0)
            self.goals = self.goals/100
            self.cCAPT(vmax = MAX_LIN_VEL, t0=0.)

        # Step 2 : Compute and send response
        task = Point()
        task.x = self.tasks[request.id-1][0]
        task.y = self.tasks[request.id-1][1]
        task.z = 0.
        response.task = task
```

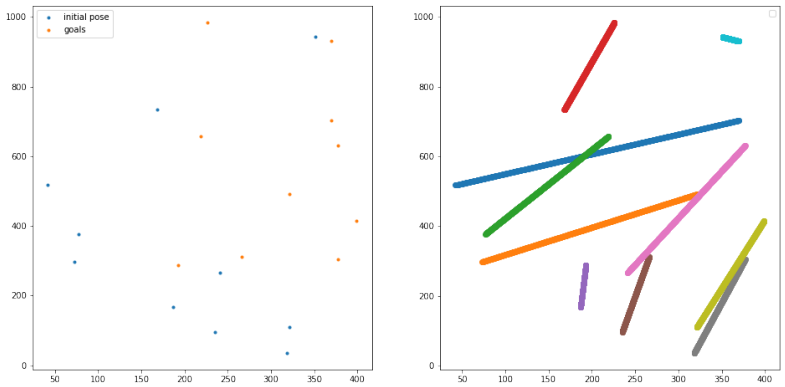

Figure 4.1: *Trajectories for N=M=10.*

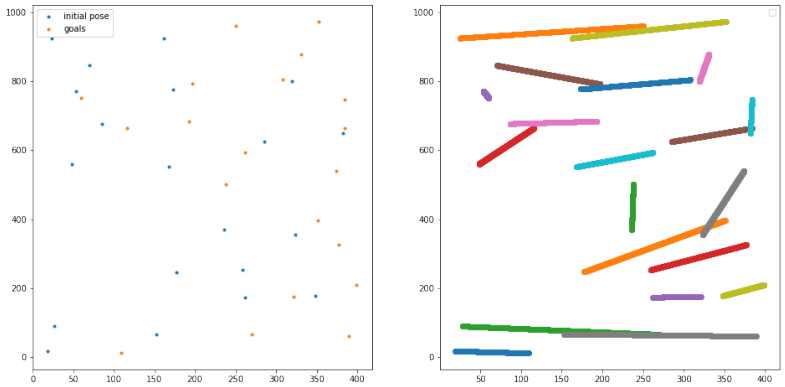

Figure 4.2: *Trajectories for N=M=20.*

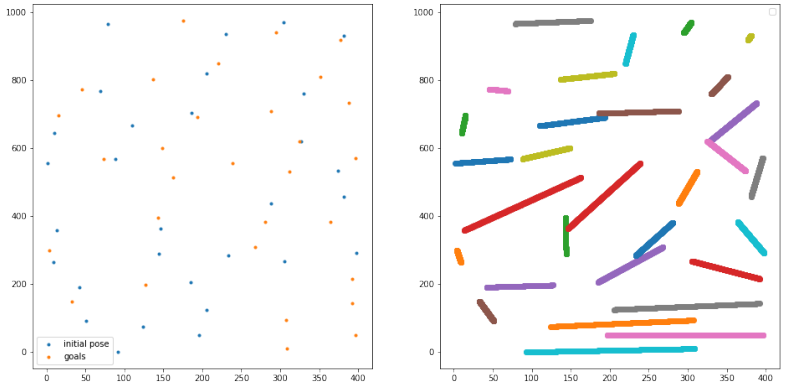

Figure 4.3: *Trajectories for N=M=30.*

Chapter 5

Control of the Jetbot

5.1 State space representation

The states of the Jetbot is given by $\mathbf{x} = [x, y, \theta]$, which respectively corresponds to the 2D coordinates of the robot and its angle with the x-axis. Fig.5.1 shows a schematic of the robot and the correspondence of each variable.

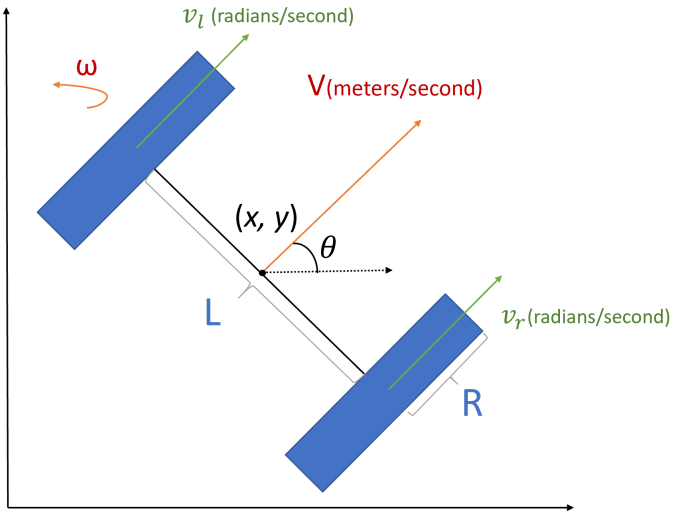

Figure 5.1: *Schematic of a differential drive robot [15]*

The dynamic of the robot is given by

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v * cos(\theta) & 0 & 0 \\ 0 & v * sin(\theta) & 0 \\ 0 & 0 & w \end{bmatrix},$$

where v is the linear velocity and w is the angular velocity, given by

$$v = \frac{R}{2}(V_r + V_l),$$
$$w = \frac{R}{L}(V_r - V_l).$$

It is therefore possible to build a controller that generates these two inputs, and send them through the **/cmd_vel** topics to the motor node which will take care of generating the PWM required to make the motors rotate.

5.2 Control Algorithm

The control of the robot is performed by one LQR controller which tracks a target angle. Even though the system is non linear it appears that the LQR controller is achieving sufficient performances in practice. Figure 5.2 shows a schematic of the control feedback loop for the angle of the robot. For simplicity the linear velocity v of the robot is not controlled and a constant input is given. The input depends on the robot and its distance to its assigned task, so that all the robots reach their assigned goals at the same time and no collisions occur along the way. Hence, the linear velocity of the robot n is scaled by

$$v_n = \frac{d_n}{d_{max}} v_m ax$$

Where d_n, d_{max}, v_{max} are the distance from robot n to its goal, the maximum assignment distance and the maximum velocity. As explained above the controller generates an angular velocity input w that is transmitted to the motors node through the **/cmd_vel** topic. The LQR controller is designed to minimize the proportional error and the integral error between the actual angle θ_k where $k = 1, \ldots, T$ and the target angle θ^*. The proportional error and the integral error are given by $e_k^\theta = \theta^* - \theta_k$ and $e_k^{\theta_i} = \sum_{i=1}^{k} e_i^\theta$ respectively. Therefore the equations governing the LQR controller are given by

$$\min \sum_{k=0}^{\infty} J_k = \mathbf{x_k^T} Q \mathbf{x_k} + u_k^2 R$$

$$u_{k+1} = c_k^\theta e_k^\theta + c_k^{\theta_i} e_k^{\theta_i}$$

$$\begin{bmatrix} e_{k+1}^\theta \\ e_{k+1}^{\theta_i} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ T_s & 1 \end{bmatrix} \begin{bmatrix} e_k^\theta \\ e_k^{\theta_i} \end{bmatrix} + \begin{bmatrix} T_s \\ T_s^2 \end{bmatrix} u_k$$

Finally, the Listing 5.1 shows the implementation of the LQR controller in Python and Figure 5.2

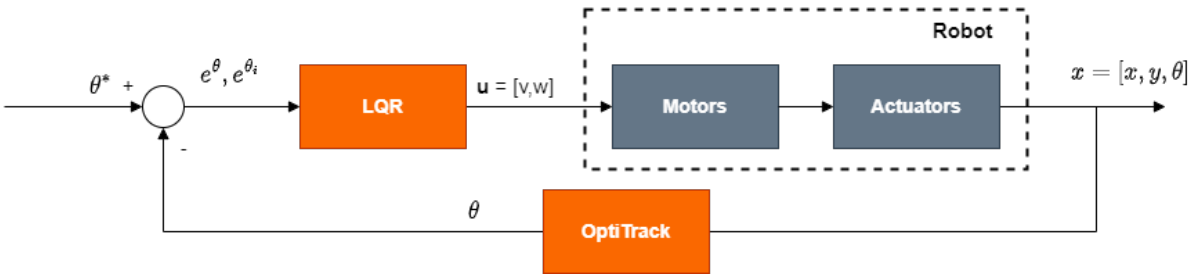

Figure 5.2: *Control feedback loop for the angle of the robot*

Listing 5.1: *Implementation of the LQR controller*

```
def __init__(self, R=None, Q=None, Ts=0.01):
        self.A = np.array([[1,0],[Ts,1]])
        self.B = np.array([[Ts],[Ts*Ts]])
        self.R = R
        self.Q = Q
        self.Ts = Ts
        self.error = [0.]
        self.i_error = [0.]
        self.previous_value = 0
        self.set_point = 0
        self.transition = False

    def update(self, current_value):
        '''
        1-step update of the controller.
        arguments :
            current_value = current value of the controlled state(s).
        ------------------------------------------------
        return :
            Velocity command
        '''

        K,_,_ = control.dlqr(self.A,self.B,self.Q,self.R)
        inp = m.atan2(m.sin(self.set_point-current_value),
              m.cos(self.set_point-current_value))
        self.error.append(inp)
        self.i_error.append(self.Ts*self.error[-1] + self.error[-2])
        return K[0,0]*self.error[-1] + K[0,1]*self.i_error[-1]

    def setLQR(self, Q, R, A=None, B=None):
        if A != None : self.A=A
        if B != None : self.B=B
        self.R=R
        self.Q=Q

    def setPoint(self, set_point, reset=True):
        '''
        Set the target value and eventually reset the errors variables.
        arguments :
            set_point = Setpoint to follow.
            reset = if you want to reset controller params
        ------------------------------------------------
        return :
        '''
        self.set_point = set_point
        if reset :
            self.error = [0.]
            self.i_error = [0.]
```

Tuning is quite intuitive and consist of finding a trade off between how much the input and each error should be penalized. This is achieved by tuning the coefficients of the Q and R matrices, which was done using a try and error approach. Finally we have

$$Q = \begin{bmatrix} 100 & 0 \\ 0 & 1 \end{bmatrix} \text{ and } R = 60.$$

Here is the feedback control algorithm :

- Suppose the robot's current orientation is θ, the desired orientation is θ^*, the current position on X-Y plane is (x, y), and the desired position on X-Y plane is (x_g, y_g).
- Calculate the angle α between the line joining (x, y) and (x_g, y_g) and the x-axis; set it as the desired orientation θ^*.
- Initialize the LQR controller with the setpoint θ^* and the Q and R matrices. Adjust the angle according to the angular velocity computed by the controller.
- Once $\theta \to \theta^*$, start moving forward using a constant linear velocity, and keep adjusting the angle and checking the remaining distance toward desired position (x_g, y_g).
- Once $(x, y) \to (x_g, y_g)$, stop and repeat the process for the next waypoint.

Fig.5.3 shows a control sequence for one robot. Note that the **motors** node clips the angular velocity to [-1,1]*100% of the max speed. Therefore, even if in the Figure the angular velocity is less than -100% of the max speed, the command sent to the motors will always be -100%.

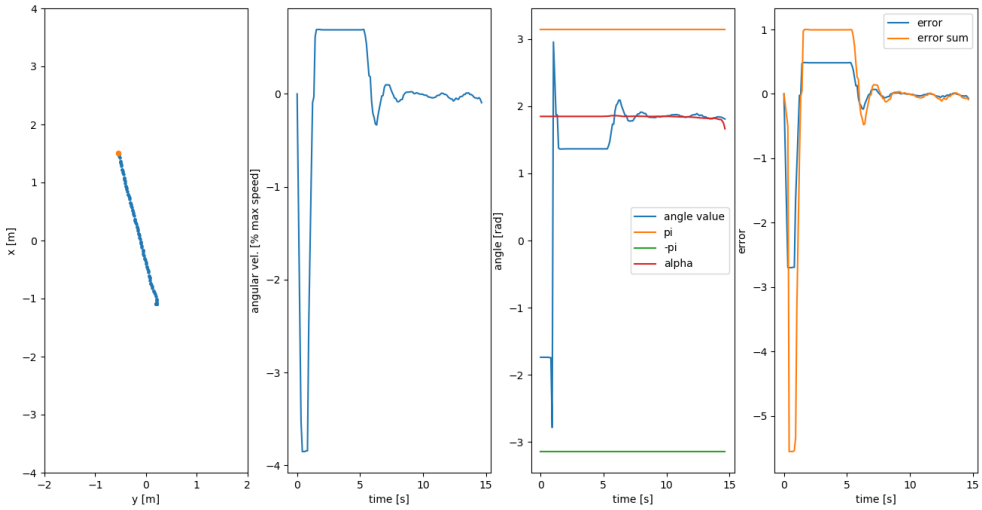

Figure 5.3: *control sequence for one robot where the orange point on the left figure represents the goal position*

Chapter 6

Conclusion

We have seen how we build the entire system in order to assemble, interconnect, and control a group of JetBots such that every predefined task is completed by one robot. Even though the full computing capabilities of the Jetson Nano are not fully used and the control algorithm is still quite simple, this project serves as a proof of concept and has vocation to help future researcher or engineers to achieve the same objectives for more complexe approaches. Moreover, the benefit of the built system is that it is built with ROS2 which provides modularity thanks to its architecture which allows to easily add nodes that communicate through topics, services and actions. Indeed, it is really straightforward to add components and nodes because of the encapsulation of each part of the system. All of the system and the setup was coded with emphasis on flexibility so that anyone with some ROS2 knowledge can update it. Nvidia offers Docker containers [16] to enhance and facilitate the deployment, the use and the efficiency of their hardware with neural networks, it would be interesting to investigate in details its compatibility with the current architecture to pursue with future works such as the use of neural networks for obstacle recognition and avoidance. Another future work will be to investigate and implement more complex concepts such as MPC for the control of multi-agent systems.

Bibliography

[1] *GitHub lalaorome/SycaBot$_r$os*. https://github.com/lalaorome/SycaBot_ros. Accessed: 2022-06-15.

[2] *GitHub lalaorome/Central$_P$C*. https://github.com/lalaorome/Central_PC. Accessed: 2022-06-15.

[3] *OptiTrack Motive Documentation*. https://v22.wiki.optitrack.com/index.php?title=Motive_Documentation. Accessed: 2022-07-08.

[4] *NVIDIA Jetbot Software Setup (SD Card Image)*. https://jetbot.org/master/software_setup/sd_card.html. Accessed: 2022-06-11.

[5] *NVIDIA L4T Base What is L4T ?* https://catalog.ngc.nvidia.com/orgs/nvidia/containers/l4t-base. Accessed: 2022-06-11.

[6] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[7] *Why ROS 2?* https://design.ros2.org/articles/why_ros2.html. Accessed: 2022-06-11.

[8] Yuan Liao. *Introduction of Robot Operating Systems 2: ROS2*. https://etn-sas.eu/2020/03/23/introduction-of-robot-operating-systems-2-ros2. Accessed: 2022-06-11.

[9] *ROS 2 Documentation*. https://docs.ros.org/en/foxy/index.html. Accessed: 2022-06-11.

[10] Paul Bovbel. *ROS wiki vrpn_client_ros*. http://wiki.ros.org/vrpn_client_ros. Accessed: 2022-06-15.

[11] *PCA968516 Channel 12 Bit PWM Servo Driver*. http://wiki.sunfounder.cc/index.php?title=PCA9685_16_Channel_12_Bit_PWM_Servo_Driver. Accessed: 2022-07-08.

[12] *dusty-nv/jetbot$_r$os*. https://github.com/dusty-nv/jetbot_roshttps://catalog.ngc.nvidia.com/orgs/nvidia/containers/l4t-base. Accessed: 2022-06-11.

[13] Matthew Turpin, Nathan Michael, and Vijay Kumar. "Capt: Concurrent assignment and planning of trajectories for multiple robots". In: *The International Journal of Robotics Research* 33.1 (2014), pp. 98–112. DOI: 10.1177/0278364913515307. eprint: https://doi.org/10.1177/0278364913515307. URL: https://doi.org/10.1177/0278364913515307.

[14] Robert Bridson. "Fast Poisson Disk Sampling in Arbitrary Dimensions". In: *ACM SIGGRAPH 2007 Sketches*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, 22–es. ISBN: 9781450347266. DOI: 10.1145/1278780.1278807. URL: https://doi.org/10.1145/1278780.1278807.

[15] *UCR Robotics Kinematics and Control for Wheeled Robots*. https://ucr-robotics.readthedocs.io/en/latest/tbot/moRbt.html. Accessed: 2022-06-15.

[16] *Docker Docs Docker overview*. https://docs.docker.com/get-started/overview/. Accessed: 2022-06-12.