École Polytechnique Fédérale de Lausanne

Semester Project, Sycamore Lab, Spring-2022:

Coordination and control of a group of ground robots

Pierre CHASSAGNE
289475

June 21, 2022

Contents

Introduction

Multi-robot systems are becoming increasingly popular and are being used in more and more applications such as warehouses, agriculture, and transportation. The objective of this project is to control and coordinate a group of ground robots. 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.

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 goal locations that need to be visited. The robots can obtain information about their surroundings with an onboard camera and detect objects by processing the images with a neural network. The initial objectives of the project was to set up and calibrate hardware as well as implement machine learning, control, and task assignment methods in software. While also interfacing the robots with a central computer and a visual motion capture system.

1 Hardware & software presentation

This section will present the hardware and software that were used for the project. Some hardware and software were required, while other 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 and finally the robots provided by Waveshare and Sparkfun, the JetBots.

1.1 OptiTrack

OptiTrack is the optical motion capture system provided by the eponymous company, it uses the software Motive in order to compute and make available the positions of the object which were defined by the user. We are using it to simulate a reliable tracking system which can give precise and almost real time 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 its 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.

1.2 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 31 GiB 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.

1.3 Jetbots

The Jetbots are open-source ground robots based on NVIDIA Jetson Nano. Jetson Nanos are small powerful computers designed to power entry-level edge AI applications and devices. It features a 4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s memory, a Quad-core ARM Cortex-A57 MPCore processor 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 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.

We configured the Jetson Nano by flashing the SD-card using the image provided by Nvidia [1]. 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 [2]. Moreover, we are using a docker in order to make ROS2 foxy run on the jetbot through the jetbot_ros container.

1.4 Robotic Operating System 2

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications [3]. ROS2 is an upgraded version of ROS, in this project we are using

the foxy distro (version in the ROS language) of ROS2, 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 [4]:

- 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 [5] 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 [6]:

1.4.1 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.

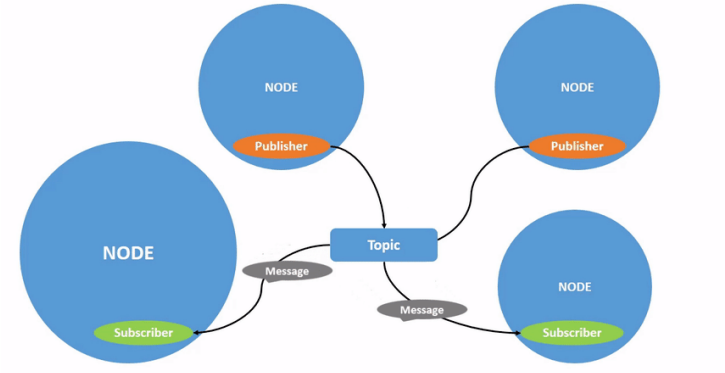

Figure 1: *2 publisher and 2 subscriber nodes communicating through 1 topic.*

1.4.2 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. One generally don't want to use a service for continuous calls; topics or even actions would be better suited.

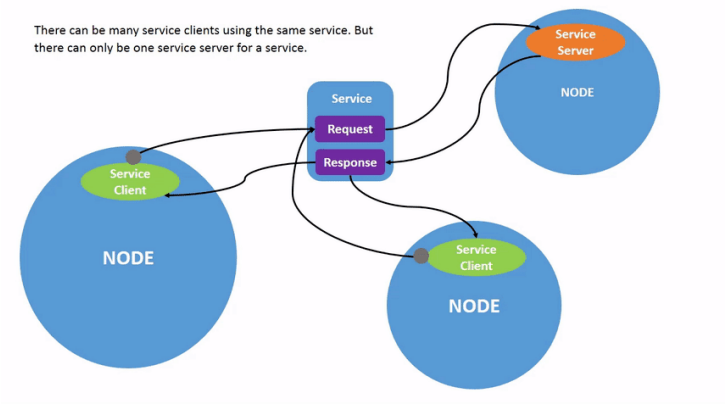

Figure 2: *2 clients and 1 server nodes communicating through 1 service.*

1.4.3 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.

A robot system would likely use actions for navigation. An action goal could tell a robot to travel to a position. While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it's reached its destination.

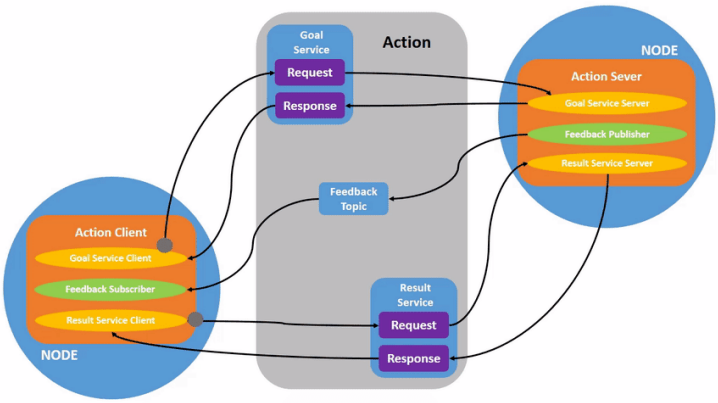

Figure 3: *1 action client and 1 action server nodes communicating through 1 action.*

1.4.4 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. Fig. 4 shows a custom .srv file which define the type of message passed through the service that would use it. The information given above the — line tells the format of the request, whereas the one under this line tells the format that the response should take. A .msg is define the same way except that it doesn't 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 4: *Custom .srv file*

Finally, we can refer to the 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.

1.5 Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. [7]

Why using Docker with ROS ? Sebastian Castro gives a simple answer in his blog post, while Docker is meant to solve the "it works on my machine" headache. It has also many advantages, especially when it comes to multi agent coordination and control : [8]

- **Reproducibility**: If you want others to reproduce your work as painlessly as possible, you could provide a detailed README with all the necessary dependencies, installation steps, troubleshooting tips, etc. Alternatively, you could handle all those potentially tricky dependencies inside a Docker container that you can test yourself before sharing that with a (hopefully) much easier set of instructions to get things running.
- **Switching Between Projects**: Even if you're a pro ROS system integrator or you have developed a watertight installation guide, chances are multiple projects will have conflicting dependencies. If you're working on projects that use different versions of ROS (or different versions of software in general), then tinkering with your host machine's environment for context switching may be painful, if not impossible, to get right.

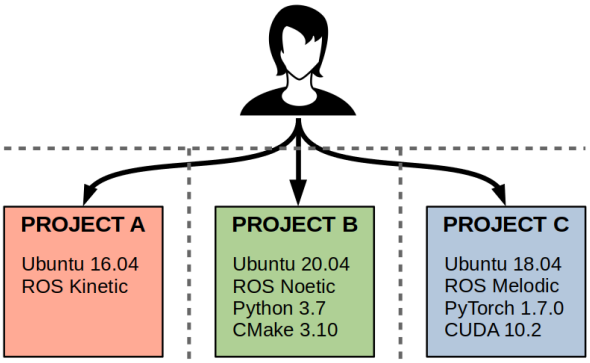

Figure 5: *A typical, exaggerated, project architecture without Docker. [8]*

Docker allows to go from the project architecture displayed in Fig.5 to the one displayed on Fig.6. Docker provides flexibility as well as robustness to any project, while saving a lot of time, especially when it comes to pass the project over. Therefore, in the project we are using Docker and especially a base image provided by Nvidia : **dustynv/jetbot_ros:foxy-r32.5.0**. This image provides example to run the JetBot using ROS and is based on another image that install ROS2 foxy for the JetBot as well as the necessary python and system packages, it corresponds to the Linux version we are using on the JetBot.

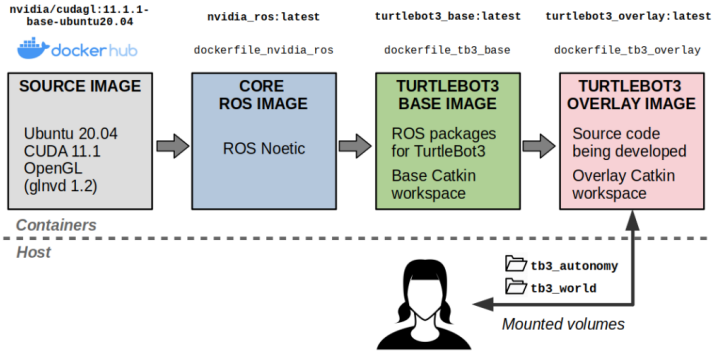

Figure 6: *A project hierarchy using ROS, Docker and TurtleBot3. [8]*

2 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 example. Fig.7 shows the legend for the diagrams in this section, please refer to it.

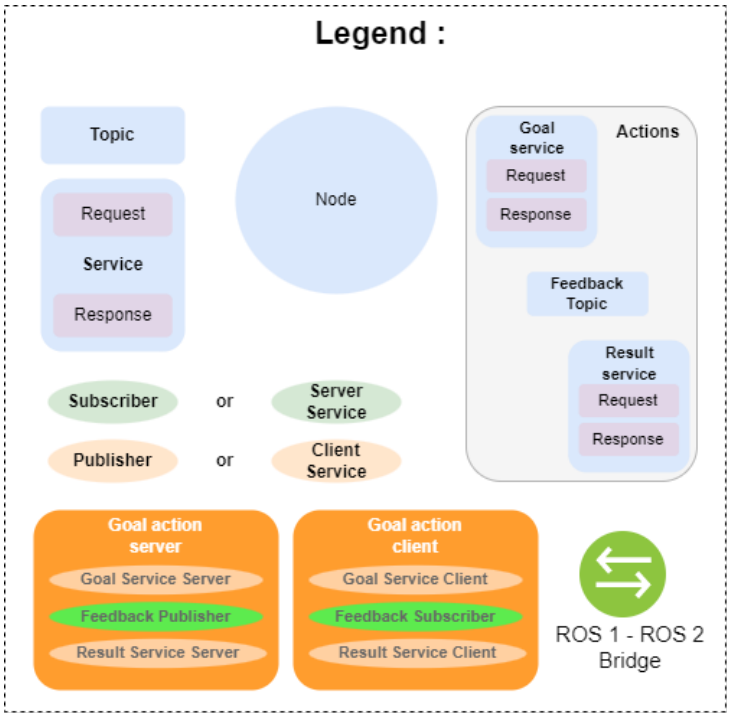

Figure 7: *Legend for the diagrams.*

2.1 General Overview

Fig.8 shows the ROS graph for the system with the central PC communicating with one JetBot called SycaBot_Wx where $x \in 1, ..., N$ and corresponds to the identifier of the JetBot. All the topic 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 to the different Topics and Services are the messages name and their type is displayed into bracket. The system was designed to be flexible to any number of robot by asking 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 at the moment. However, for more complex applications, actions could become essentials. While the Central PC knows every Jetbots for now the Jetbots are not communicating between each other.

Nevertheless, and this might be on of the main advantage of ROS, if one would want to achieve decentralize CAPT 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 Jetbot to communicate information with it.

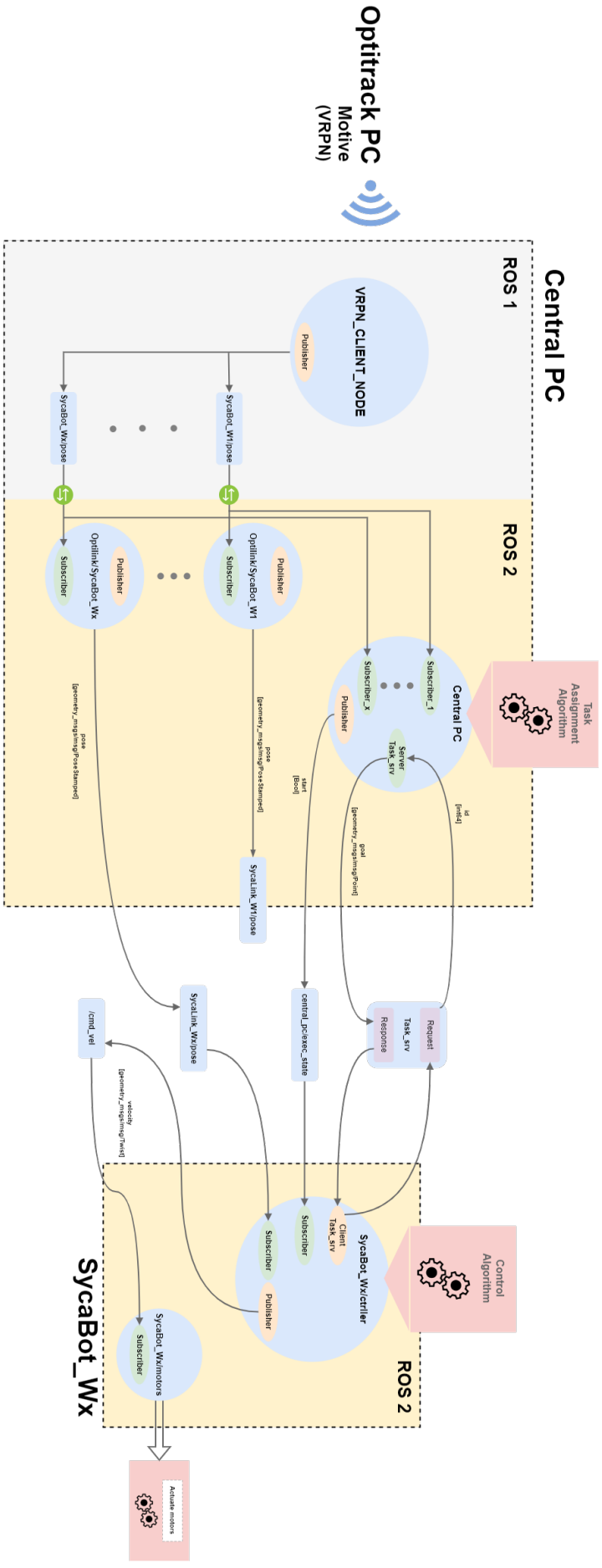

Figure 8: *ROS Graph for the the system with one JetBot.*

2.2 Central PC

Fig.9 shows the central PC's ROS graph. The Optitrack PC is also displayed here since its role is mainly to gather and communicate position points to the central PC using motive and the Virtual-Reality Peripheral Network (VRPN) library. The central PC uses the vrpn_client_ros package [9], a ROS1 package which creates client nodes to transfer the positions transmitted by the Optitrack PC. The positions are then published over ROS1 topics. These topics can be accessed in ROS2 using the ROS1 to ROS2 bridge which is built in ROS2. Then nodes are created to make the position accessible by every devices (here the robots) on the network and through ROS2.

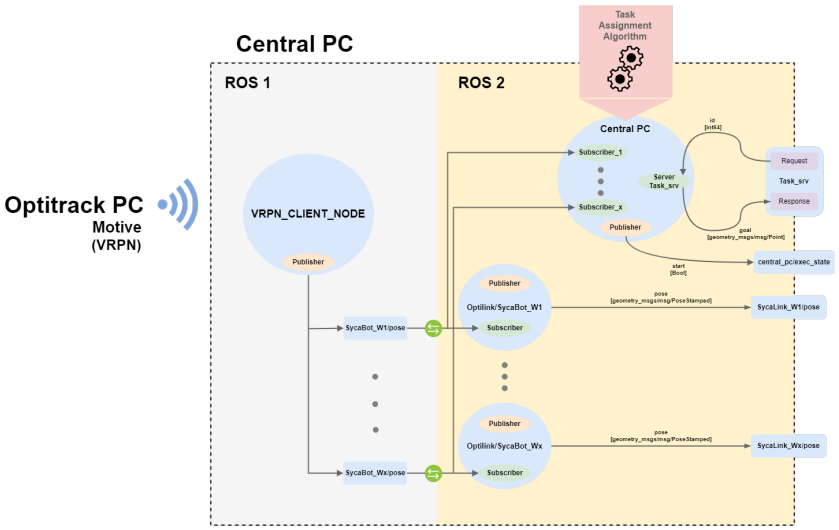

Figure 9: *Zoom on the central PC's ROS graph.*

Furthermore, the central PC node is responsible for generating random tasks and assigning them to the Jetbots. The task assignment algorithm will be discussed in the next section. In order for the Jetbot to get its task it needs to require it through the **task_srv** service by sending its id, then the central PC sends back its task and update 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 almost at the same time.

2.3 Jetbot

Fig.10 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 using an I2C protocol. As said above, after having requested its task, the Jetbots wait for the start signal and then begin the control loop.

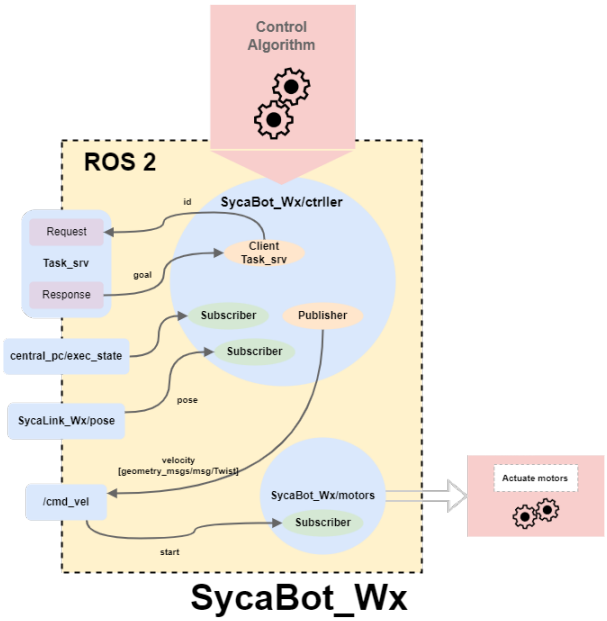

Figure 10: *Zoom on the Jetbot's ROS graph.*

3 Task Assignment Algorithm

3.1 Implementation

The task assignment algorithm used by the central PC is taken from the paper [10] in the case where the number of goal M is equal the number of robots N, which is easily implemented since the algorithm randomly generates goals. From this paper, we know 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, then the task assignment algorithm guarantee collision free trajectories. Hereafter 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 binary decision variable 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 : $D_{i,j} = \|x_i(t_0) - g_j\|^2 \quad \forall i \in \mathcal{I}_N, \forall j \in \mathcal{I}_M$

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 $i-th$ position of the matrix. The points generated for the goals are generated using an adapted code of this thread [11] which is taken from this note [12]. The implementation in Python is shown on the next page.

3.2 Results

Fig.11 to Fig.13 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 approcimately corresponds to the dimension of the lab in which we work and the line in the trajectories have a width of 71 cm which corresponds to the Δ of the Waveshare Jetbots. As expected, and as explained in Turpin's paper, all the trajectories are straight and not colliding.

```
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 for now
                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 the number of remaining
               goals
                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
```

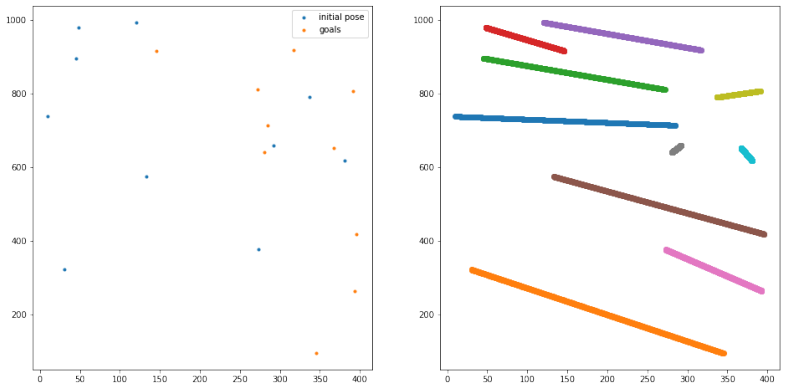

Figure 11: *Trajectories for N=M=10.*

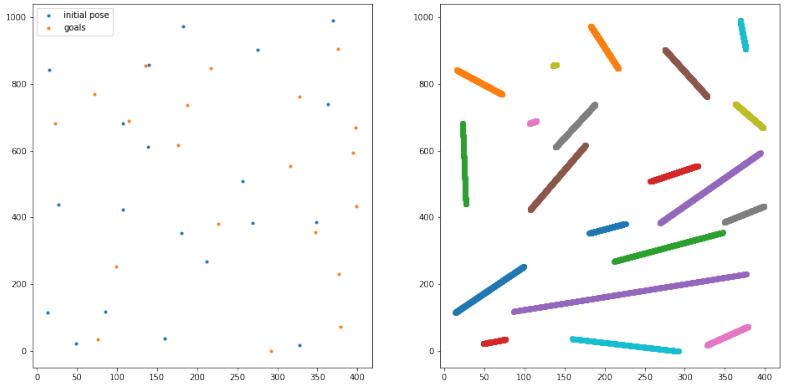

Figure 12: *Trajectories for N=M=20.*

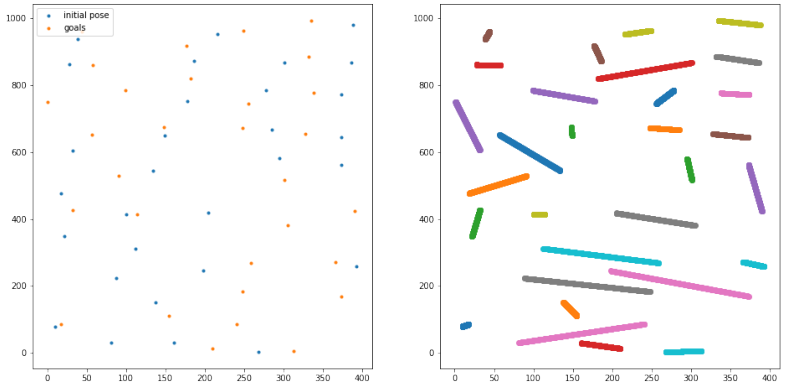

Figure 13: *Trajectories for N=M=30.*

4 Control of the Jetbot

4.1 State space representation

The states of the Jetbot is given by $\mathbf{x} = [x, y, \theta]$, which respectively corresponds to the coordinates of the robot and its angle with the x-axis. Fig.14 shows a schematic of the robot and the correspondence of each variable.

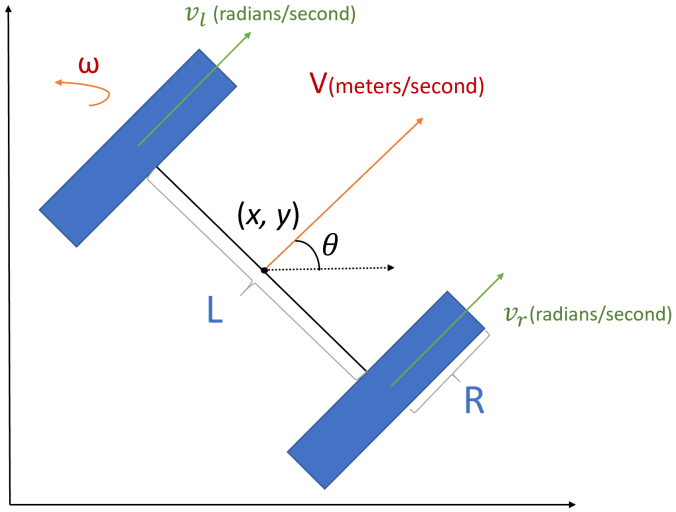

Figure 14: *Schematic of a differential drive robot [13]*

The state transition matrix 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 the angular velocity, given by :

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

We can therefore build a controller that generates these two inputs, we can then transform them two give the right commands to the motors, this is handled by the motors node in each Jetbot.

4.2 Control Algorithm

The control of the robot is performed by one simple PID controller which tracks a target angle. Even though the system is non linear it appears that the PID controller is performing well, even in presence of noise. Fig.15 shows a schematic of the control feedback loop for the angle of the robot. The choice of not having a controller which control the linear velocity of the robot is done purposely, indeed, since there is no odometry sensor on the Jetbot and the Optitrack system is extremly reliable, directly controlling

the linear velocity of the robot is not relevant. However, one could use the camera of the robot and optical flow to compute the speed of the robot and therefore use another PID controller to control its linear velocity.

The controller generates angular velocities that are transmitted to the motors node through the **cmd_vel** topic. Here is the PID class written in python :

```
class PIDController:
    def __init__(self, P=0.0, D=0.0, I=0.0, set_point=0):
        self.Kp = P
        self.Kd = D
        self.Ki = I
        self.set_point = set_point
        self.previous_error = 0
        self.sum_error = 0

    def update(self, current_value):
        # calculate P_term and D_term
        error = self.set_point - current_value
        P_term = self.Kp*error
        D_term = self.Kd*(error - self.previous_error)
        I_term = self.Ki*self.sum_error
        self.previous_error = error
        self.sum_error += error
        return P_term + D_term + I_term

    def setPoint(self, set_point):
        self.set_point = set_point
        self.previous_error = 0

    def setPID(self, P=0.0, D=0.0,I=0.0):
        self.Kp = P
        self.Kd = D
        self.Ki = I
```

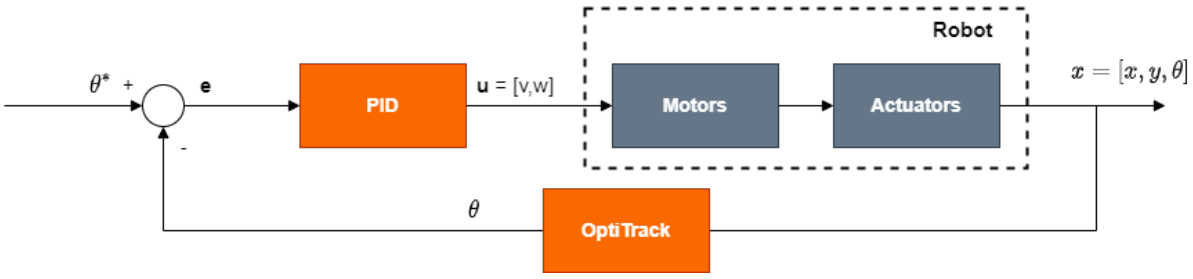

Figure 15: *Control feedback loop for the angle of the robot*

In order to tune the PID controller, it has been implemented and tested with a simulated noisy system. The noise is constituted by a Gaussian white noise on the measurement of the position and a random drift which range from 0 to 0,2 on one of the wheel's velocity. 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 a PID controller with the setpoint θ^* and a set of parameters (Kp, Kd). Adjust the angle according to the angular velocity computed by the PID controller.
- Once $\theta \rightarrow \theta^*$, 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 $(x, y) \rightarrow (x_g, y_g)$, stop and repeat the process for the next waypoint.

Fig.16 shows a control sequence for the robot. The robot has to go from its initial state $[x, y, \theta] = [0, 0, 0]$ to its goal in $[x_g, y_g] = [-8, -4]$, the target velocity is given by 0.3 which represents a third of the maximum velocity of the robot.

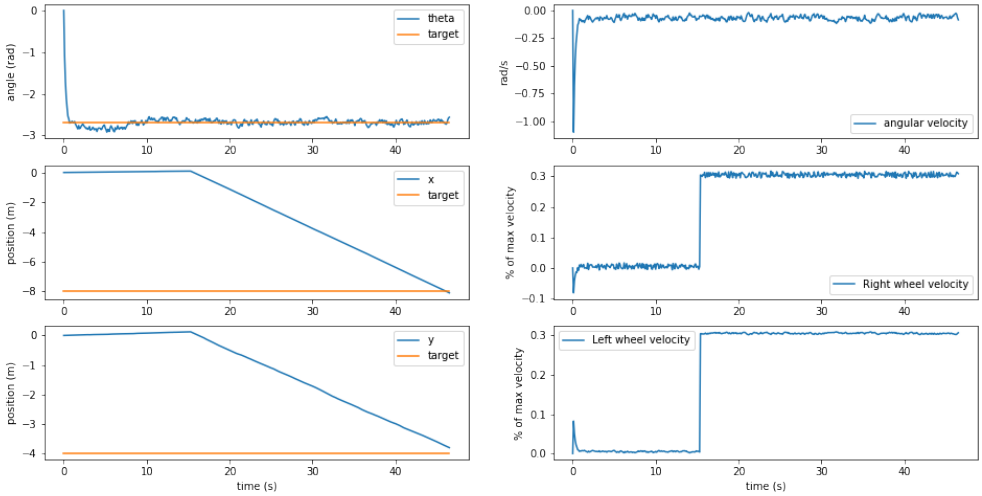

Figure 16: *Result of the simulation with a noisy system*

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 we are not using the full computing capabilities of the Jetson Nano and the control algorithm is still quite simple, the strength of ROS2 resides in its modularity. It is really straightforward to add components and nodes because of the encapsulation of each part of the system. Moreover, all of the system and the setup was coded with emphasis on flexibility and its ease to be taken over so that anyone with some ROS2 knowledge can update it. The next steps for this project will be to try and implement MPC for the control of the robot as well as using the Jetson Nano to perform some image recognition and deep learning.

References

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

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

[3] 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.

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

[5] 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.

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

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

[8] Sebastian Castro. *Robotic Sea Bass A Guide to Docker and ROS*. https://roboticseabass.com/2021/04/21/docker-and-ros/. Accessed: 2022-06-12.

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

[10] 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.

[11] Paul Bovbel. *Stack Overflow Generating multiple random (x, y) coordinates, excluding duplicates?* https://stackoverflow.com/questions/19668463/generating-multiple-random-x-y-coordinates-excluding-duplicates. Accessed: 2022-06-15.

[12] 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.

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