University of Liege
Academic year 2022-2023

INFO0948-2 : Introduction to intelligent robotics

Pizza delivery

Authors
Baguette Brice S181482
Kamps Nathan S183601
Destexhe Clara S191103

Instructors
Sacré Pierre
Goffin Sven
Marlier Norman

1 Introduction

Nowadays, robots are increasingly surrounding us. Everything tending towards automation and soon, science fiction films will become reality. Moreover, a lot of products can be bought on internet and-delivered directly to our homes in just a matter of days. Since the onset of Covid, this online market has experienced significant growth. Therefore, one big market that will be invaded by robots in the future is the delivery given the rising demand from customers.

Consequently, the project is focusing on this delivery demand and more precisely on pizza delivery. This year, the university bought several TurtleBot3 Burger. These robots are able to move around in a closed environment thanks to a front camera and a LiDAR sensor. We will be given a list of customers that ordered pizzas and their "adress". The goal will be to visit each of these customers in the order of our choice. Obviously, the time of delivery needs to be optimal. Each client is assigned to an aruco marker. The task is accomplished when the robot has stopped to each marker.

2 Software

2.1 ROS2

The robots that are given to us use the Robot Operating System(ROS2). ROS2 provides a set of software libraries and tools for developing robust and scalable robotic systems. It allows the communication between different components of a robotic system. This is working thanks to a publish-subscribe messaging model. Nodes can publish messages to specific topics and others nodes can subscribe to those topics to receive the messages.

A few words need to be add on the node concept present in ROS2. A node is a fundamental building block of a robotic system. It is an entity performing communication, computation and control in a distributed ROS2 system. Each node is an independent module communicating with other nodes through a publish-subscribe messaging mechanism. It means some nodes are publishing information (publisher nodes). The information published is often data structures useful to specific topics. Other nodes that find these information relevant can subscribe to these (subscriber nodes).

A topic is the communication channel between the different nodes in ROS2. In other words, it is the place where nodes can publish messages and the place where other nodes can receive these messages. The messages are information contained in data structures.

Finally, let's explain what is an action in ROS2. Actions are a way to define and execute long-running tasks. These tasks need some kind of feedback and goal progress tracking. It is composed of three different components: The action server, the action client and the action message. The action server is a node exposing an action and handle the corresponding goal. The action client is a node sending requests to the action server and receiving results updates. The action message fixes the structure of the goal that must be reached by the action server.

2.2 Gazebo

In order to work from home, a system has to be used to simulate the robots and its behavior. Gazebo is an open-source environment that allows modelisation and simulation of robots and their environments. Gazebo can simulate complex systems and accurately model the laws of physics behind them. It simulates the behavior of objects, their dynamics, collisions, and interactions within a simulated environment. This program is essential to observe and analyze the behavior of robots and their interactions with the surroundings.

2.3 Aruco

As said before, the customers are assigned to Aruco markers. These markers are stuck to the walls of the closed environments and the robots needs to move around in order to scan them. Aruco is an application that enables the detection and tracking of square markers. ArUco markers are black-and-white square patterns that contain an unique identifier. These markers are designed to be easily detectable and robust to different lighting conditions and viewpoints.

3 Code

We can observe in the following figures 1 and 2 the topics list linked to the nodes "path_planner_node" and "pizza_delivery_node".

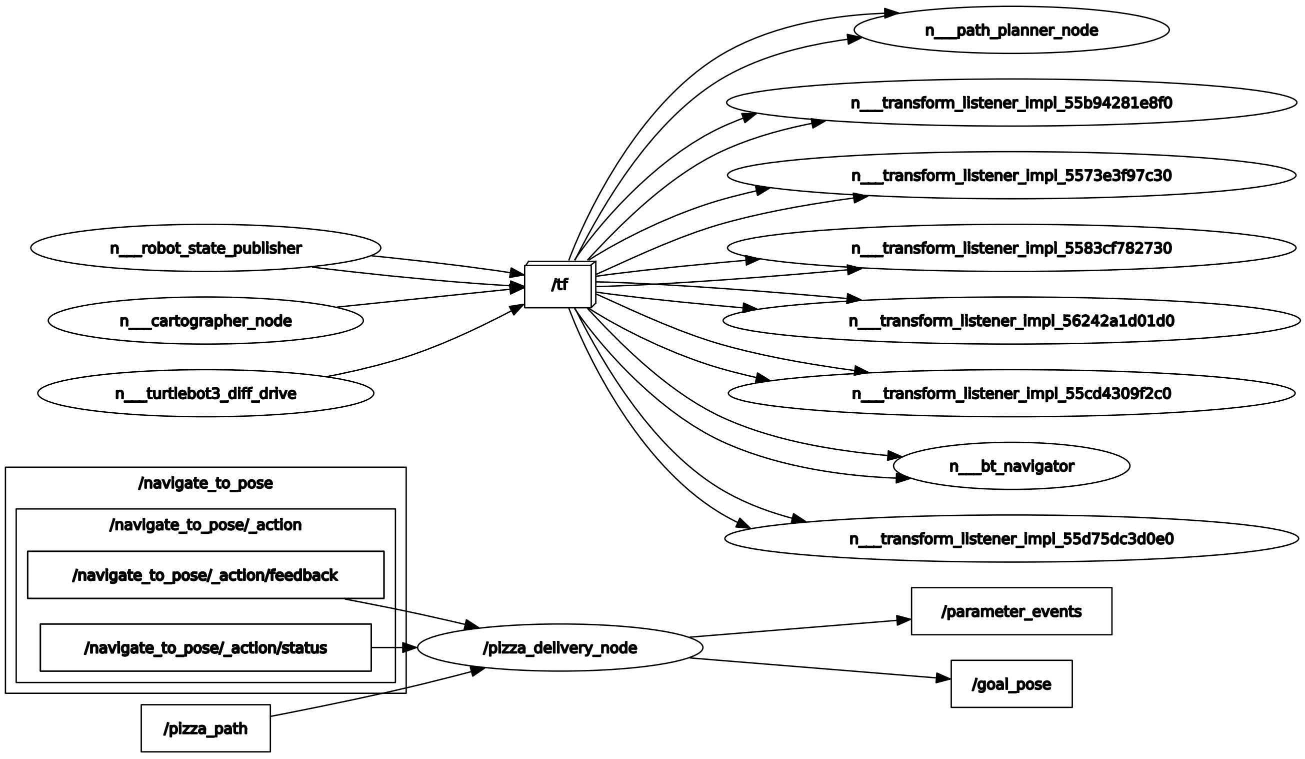
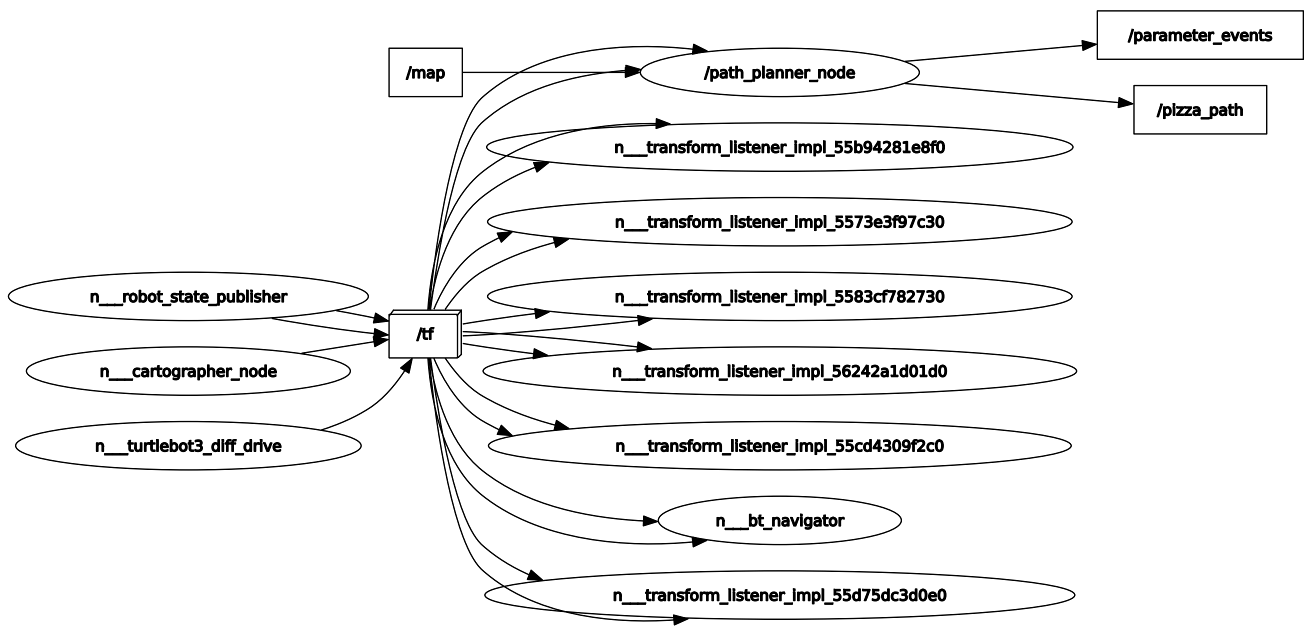

n___path_planner_node
n___transform_listener_impl_55b94281e8f0
n___transform_listener_impl_5573e3f97c30
n___robot_state_publisher
n___transform_listener_impl_5583cf782730
/tf
n___cartographer_node
n___transform_listener_impl_56242a1d01d0
n___turtlebot3_diff_drive
n___transform_listener_impl_55cd4309f2c0
n___bt_navigator
/navigate_to_pose
/navigate_to_pose/_action
n___transform_listener_impl_55d75dc3d0e0
/navigate_to_pose/_action/feedback
/parameter_events
/navigate_to_pose/_action/status
/pizza_delivery_node
/goal_pose
/pizza_path

Figure 1: Liste des topics utilisé par le code Navigation



/map
/path_planner_node
/parameter_events
/pizza_path
n__transform_listener_impl_55b94281e8f0
n__transform_listener_impl_5573e3f97c30
n__robot_state_publisher
/tf
n__transform_listener_impl_5583cf782730
n__cartographer_node
n__transform_listener_impl_56242a1d01d0
n__turtlebot3_diff_drive
n__transform_listener_impl_55cd4309f2c0
n__bt_navigator
n__transform_listener_impl_55d75dc3d0e0

Figure 2: Liste des topics utilisé par le code Path Planner

The "pizza_delivery_node" has 3 subscriptions and 2 publishers. The "path_planner_node" has 3 subscriptions and 2 publishers.

3.1 Path planner

Ignoring the code to import the different packages used, we start with :

```
        self.occupancy_grid = OccupancyGrid
        self.matrix_map = []
        self.waypoints = []
        self.subscription = self.create_subscription(
            OccupancyGrid,
            'map',
            self.load_map,
            10
        )
```

This is used to initialize the 'occupancy_grid' which is used to store the map. The variable 'matrix_map' is used to convert the map into a matrix and therefore be used more effectively by the search algorithm. The 'waypoints' is utilized to the Pose configuration of the different markers. We also create a subscription to the topic 'map' and starts a callback to the function 'load_map' when the topic receives a message.

```
        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        self.publisher = self.create_publisher(
            Path,
            'pizza_path',
            10
        )
```

Furthermore, we add a listener to know the position of our robot and we create a publisher to the 'pizza_path'. Thanks to that, we can transfer our optimal path to the 'pizza_delivery_node'.

```python
def load_map(self, msg):
        self.occupancy_grid = msg
        map_data = msg.data
        map_width = msg.info.width
        map_height = msg.info.height

        matrix = [[0 for _ in range(map_width)] for _ in range(map_height)]

        for y in range(map_height):
            for x in range(map_width):
                index = y * map_width + x
                map_value = map_data[index]
                if map_value == 100:
                    matrix[y][x] = 1  # Represents a wall or obstacle
        self.matrix_map = matrix
        if len(self.waypoints)==0:
            self.pose_callback()
        self.get_path()
```

Just above is the callback function that we start when our subscription receives a message. We begin to save the OccupancyGrid variable into our variable. Then, we create a matrix with the dimensions of the map filled with zeros. We check every position to see if the value stored at a position is equal to 100, in order to input a 1 in our matrix for the positions of a wall or obstacle. At our first call, we call the function 'pose_callback' to get the markers positions. This function is only called once, i.e. when we run the node. Finally, we launch the 'get_path' function.

```python
 1     def pose_callback(self):
 2         success = False
 3         while not success:
 4             try:
 5                 transform = self.tf_buffer.lookup_transform(
 6                     'map',  # Target frame (map frame in this example)
 7                     'base_footprint',  # Source frame (base_link frame in this example)
 8                     rclpy.time.Time().to_msg(),  # Use latest available transform
 9                     timeout=rclpy.duration.Duration(seconds=1)
10                 )
11                 # Access the pose information
12                 pose = transform.transform
13                 self.waypoints = [Pose(position=Point(x = pose.translation.x, y= pose.translation.y,
14 
15                 quaternion1 = self.euler_to_quaternion(0.000400, 1.570000, 1.570396)
16                 quaternion2 = self.euler_to_quaternion(0.005486,1.570000,2.618564)
17                 self.waypoints.extend([Pose(
18                     position=Point(x=0.676417, y=0.026173, z=0.149815),
19                     orientation=quaternion1
20                 ),
21                 Pose(
22                    position=Point(x=0.620023, y=1.416080, z=0.149815),
23                    orientation=quaternion2)
24                 ])
25                 success = True
26             except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros.ExtrapolationExce
27                 # Handle exceptions
28                 self.get_logger().warning('Failed to retrieve TurtleBot3 pose')
```

The lines 5 – 12 allow us to look up the transform of the robot at its initial position and add it to the 'waypoints' matrix. The following lines then store the Pose configurations of the matrix into the 'waypoints' one, while changing the Euler angles into Quaternion vector. These positions are hard coded. The last lines are just a warning message.

```python
 def get_path(self):
            matrix_representation = [self.pose_to_occupancy_grid(pose) for pose in self.waypoints]
            path = self.find_best_order(matrix_representation)
            converted_path = [self.occupancy_grid_to_pose(pose[0], pose[1], pose[2], pose[3]) for pos
            self.publish_path(converted_path)
```

The 2nd line calls a function 'pose_to_occupancy_grid' to translate the Pose configurations into indexes of 'matrix_map'. It represents the 'waypoints' matrix as the 'matrix_representation'. The 3rd line calls the function 'find_best_order' explained in this section 3.1.1. The 4th line is to convert back the indexes into Pose configurations, so that our Navigation node can understand the path to chose. Finally, we call the function 'publish_path' to publish our path to the 'pizza_path' topic.

3.1.1 Find Best Order function

```python
def find_best_order(self, waypoints):
        """Find the best order to visit the list of waypoints."""
        num_waypoints = len(waypoints)
        best_order = []

        # Start with the first waypoint as the initial current position
        current_position = waypoints[0]
        best_order.append(current_position)

        # Continue until all waypoints have been visited
        while len(best_order) < num_waypoints:
            closest_distance = float('inf')
            closest_point = None

            # Find the closest unvisited waypoint
            for waypoint in waypoints:
                if waypoint not in best_order:
                    distance = self.calculate_distance(current_position, waypoint)
                    if distance < closest_distance:
                        closest_distance = distance
                        closest_point = waypoint

            # Update the current position and add the closest waypoint to the best order
            current_position = closest_point
            best_order.append(current_position)

        return best_order
```

We start from the initial position of the robot then compare its distance with all markers. Afterwards, the closest marker is added to the 'best_order' vector. The loop is repeated, but this time the distances are calculated from the previous closes waypoints to our initial position with all markers left. This loop is repeated until all markers are found in the 'best_order' vector.

3.2 Navigation

```python
        self.waypoints = []
        self.callback_processed = False

        self.publisher = self.create_publisher(
            PoseStamped,
            'goal_pose',
            10
        )

        self.client = ActionClient(
            self,
            NavigateToPose,
            'navigate_to_pose'
        )
```

We initialize the variable 'waypoints' to store the informations sent from 'pizza_path' topic. We create a publisher to the topic 'goal_pose'. Finally, we create an action client which is based on the client-server model.

```
        while not self.client.wait_for_server(timeout_sec=1.0):
            self.get_logger().info('Action server not available, waiting...')

        self.subscriber = self.create_subscription(
            Path,
            'pizza_path',
            self.start_navigate_callback,
            10
        )
```

We display a message while the action server doesn't have anything. A subscription is then created to the topic 'pizza_path', which will call the function 'start_navigate_callback' when it receives a message from the topic.

```
    def start_navigate_callback(self, msg):
        if self.callback_processed:
            return
        self.callback_processed = True
        for pose in msg.poses:
            self.waypoints.append(pose.pose)
        self.current_waypoint_index = 0
        self.navigate_to_next_waypoint()
```

This function starts the 'navigate_to_next_waypoint' function with the initial robot position.

```
    def navigate_to_next_waypoint(self):
        if self.current_waypoint_index < len(self.waypoints):
            waypoint = self.waypoints[self.current_waypoint_index]
            pose_msg = PoseStamped()
            pose_msg.header.frame_id = 'map'
            pose_msg.pose = waypoint

            self.publisher.publish(pose_msg)

            goal_msg = NavigateToPose.Goal()
            goal_msg.pose = pose_msg
            print(goal_msg.pose)
            goal_handle_future = self.client.send_goal_async(goal_msg)
            goal_handle_future.add_done_callback(self.navigation_goal_sent_callback)
        else:
            self.get_logger().info('All waypoints reached.')
```

Until we arrive at the last marker, we publish the Pose configuration of the next marker in the waypoint list to the 'pose_msg' topic, which can then be used in the Action client. This sends the next marker's position to the server. We then registers a callback for the goal response to the function 'navigation_goal_sent_callback'. Once it is called enough times, the 'navigation_result_callback' will call this function one last time, but the waypoints have been all reached.

```
    def navigation_goal_sent_callback(self, future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().info('Navigation goal rejected.')
            return

        self.get_logger().info('Navigation goal accepted.')
        goal_handle.get_result_async().add_done_callback(self.navigation_result_callback)
```

This function basically checks if the navigation goal is accepted and callbacks the function 'navigation_result_callback'.

```
def navigation_result_callback(self, future):
        result = future.result()
        print(result)
        if result.status == 4:
            self.get_logger().info('Waypoint reached.')
            self.current_waypoint_index += 1
            time.sleep(1)
            self.navigate_to_next_waypoint()
        else:
            self.get_logger().info('Navigation failed.')
```

This function checks if the waypoint is reached, increases the index if so to move to the next waypoint and wait 1 second to stay in front of the marker. Then calls the function 'navigate_to_next_waypoint' with the next waypoint.

3.3 Difficulties

Some difficulties have been encountered during the realisation of the project. A little list of these problems is exhibited just below :

- **Installation**
  Indeed, the installation of every software environment needed was a little bit tricky. The linux environment was the hardest part. Finally, we managed to install ubuntu thanks to a bootable usb key. One member of the group has a MACOS environment. The installation of ubuntu on this type of environment is almost impossible. She had to take another laptop with a Windows environment on it.

- **Quaternion angles**
  The Aruco markers in Gazebo are assigned to a quaternion in order to characterized their orientation. A quaternion is a mathematical concept that extends the idea of complex numbers into a four-dimensional space. While complex numbers have a real part and an imaginary part, quaternions have four components: a scalar (real) part and a vector (imaginary) part consisting of three elements. Its main goal is to represent 3D rotation. We needed to convert these quaternions into Euler angles in order to position the robot in the good orientation. Fortunately, some functions have been already implemented into Python to do the conversion in the two directions.

- **Optimal path**
  One other tricky part was the implementation of an algorithm able to compute the ideal path which would take less time possible to deliver all the pizzas. The algorithm used to complete this task was the TSP algorithm. A TSP algorithm use iterative processes to search for good solutions by iteratively improving the route.

- **Interaction with nav2**
  Also, it tooks some time to understand how to communicate with nav2. As a reminder, nav2 is a set of nodes and libraries that enable robots to plan path and navigate their environment autonomously.