Group 8
Javier Alquézar Alquézar
Alejandro Cano Caldero
Jesús Moncada Ramírez

Assignment 1 Report

Introduction

This report will start by explaining the **action client** node, which is in charge of printing the final results. Subsequently, it will move on to the explanation of both the **action server** node and an **obstacle detection** node we have developed to modularize the code.

Client

The **client node** (*src/client.cpp*) asks the user for a position goal for the robot, uses the action server (*tiago_controller*), and prints the result (the detected obstacles) of the process. The most relevant functions are:

- *printPoses()* → receives a vector of poses (object *Pose*) and prints the position of the axes *(x, y, z)* and the orientation *(x, y, z, w)* of each element.
- *printObstacles()* → receives a vector of obstacles (object *Point*) and prints them. Note that the server will return the middle point of each obstacle, and this is what we print.
- ***doneCb****()* → after the action server has finished its execution, the client obtains the results of the goal, there are 3 possible states: a “Succeeded” goal (everything okay), an “Aborted” goal (some error happened, see server node for more information) and an “Unknown state”. If the goal succeeded the movable obstacles are printed.
- ***feedbackCb****()* → receives a feedback message from the action server. (See section *Server* to discover all possible messages).

In the *main* function, the node asks the goal position to the user (from the command line if not specified in the arguments). After this, sends the goal position to the action server. When the robot reaches the goal, the function *doneCb* will be executed.

Server

The **server node** (*src/server.cpp*) will both interact with the ***move_base*** **action server** so as to get the robot moving and **respond** to the **client**’s request. It will also call the obstacle detection server (*obstacle_detection*) at the end of the robot’s movement. In the ROS environment, it will create an action server called *tiago_controller*.

The method ***executeCB*** will be executed when this action server is called. Not only will it **send** some **feedback** to the client node, but it will also take the **goal’s position** (coming from the client) and **send it to *move_base*** server. Moreover, this class implements two important additional methods:

- ***doneMoveBase()*** → called once the robot has reached the desired position. It is in charge of **sending the results** and the robot’s **final state** to the client.
- ***feedbackMoveBase()*** → this method is called every time the robot is moved. It receives a feedback message with the robot's current pose from *move_base* and sends it as a **feedback message** to the action client.

There is another method implemented called ***sendFeedback()*** used by the former method *feedbackMoveBase()* in order to add the **feedback mode** value to the action class attribute. This method helps us to manage the different feedback messages we have established (as we are asked to do in the statement of the assignment). Each feedback message is identified with a *mode* that specifies its meaning, they are: (0) the goal was received from the action server, (1) the goal position was sent to the navigation stack using the *move_base* action server, (2) the robot reached a new position, and (3) the robot is in the final position and the obstacle detection procedure begins.

Another additional aspect that is worth highlighting is the fact that the client node is nourished with a list of poses, that is to say, the implemented feedback coming from the server node (*server.cpp*) provides, not only the last pose but a list containing all the robot's poses. However, unlike our implementation, the *move_base* action server sends just the last pose.

Obstacle detection

The obstacle detection node (*src/obstacle_detection.cpp*) creates a **ROS server** (called *obstacle_detection*) that is in charge of detecting the mobile obstacles once the robot has successfully moved. The server file (*srv/detection.srv*) has a request with two parameters (*scanMinusMapThreshold, obstacleDetecionThreshold*) and a response with an array of obstacle points (*obsPoint*).

When this server is called it performs the following operations: **(1)** get all the map points by getting a message from the topic */map*; **(2)** get a scan message by getting a message from the topic */scan*; **(3)** transform all the scanned points to the reference frame of the map; **(4)** discard those scanned points close to the map points; **(5)** apply an obstacle detection algorithm.

We get to obtain all the map points **(1)** by iterating over the array *data* of a *nav_msgs::OccupancyGrid* object. These points are translated to the cartesian coordinates we need by using the map's attributes *resolution* and *origin.position*. Transforming all the scanned points **(3)** from their reference frame (*base_laser_link*) to the map reference frame (*map*) is done by using the library *tf2*.

To discard all the scanned points (transformed) close to the map points **(4)** we have come up with an algorithm that iterates over each scan point comparing it to all the map points. If the point in the scan is close (distance *scanMinusMapThreshold*) to any of those in the map, it is eliminated.

The obstacle detection algorithm **(5)** maintains a list for each obstacle, where each element is another list with all the points of the obstacle. This way, we iterate over the scanned points, and for each point:

- If there is no obstacle saved yet, create an obstacle and add the point as the only one.
- If there are obstacles, compare the point to each obstacle.
    - If the point is close (distance *obstacleDetectionThreshold*) to an obstacle's middle point (mean of all the points), it will be added to that obstacle.
    - If the point is not close to any obstacles, a new obstacle is created with that point as the only one.