Intelligent Robotics course

Assignment 1 Report - Tiago navigation and obstacles detection

Prof. Emanuele Menegatti

Group 23: Simone Bastasin, Andrew Zamai & Alessandro Manfè

23rd December 2022

1 How we designed the ROS nodes architecture

As the object-oriented programming style teaches, having a structured, modular and easily extendable code is the key-point of any well-coded program. In this regard, ROS makes it possible to implement different programs in different nodes and to make them communicate through messages, services and client/server actions. How does this translate into assignment 1? Looking at the assignment specifications we see that the action client has to perform as little computations as possible, receiving only the input Pose_B from the user through terminal and printing back robot's current status periodically and, only lastly, the positions of the detected movable obstacles. Robot's navigation and obstacles detection must be implemented in a separated action server. We can at this point observe that these two tasks are independent from each other and thus their code implementations should be placed in two different nodes. Due to the frequency by which the tasks have to be implemented we see that the node responsible for the navigation must implement an action server (to frequently inform the client about it's current status), while the obstacles detection task, which has to be performed only once when the robot reaches pose_B, could be implemented through a service of type request/response. We have up to this point already defined 3 nodes. Before starting to code we looked also at the extra points request. We immediately figured out that the motion control law that we were asked to implement had to be defined in another node acting as action server. The client thus would have been required to first send the goal to our motion control law server (which would have stopped at the end of the corridor) and only then send the goal again to move_base server to continue navigation. Finally call the node for the detection. We can see how all this overheads what the action client is required to do. We therefore decided to implement an intermediate node acting as "proxy" that would get the goal from the client and dispatch it firstly towards the motion control law server, then to the move base server and lastly sending the request for the detection. The client was then only required to send the goal and print the current status and final positions, while the proxy acting as dispatcher to the other nodes. Despite allowing this solution to have one only proxy we decided to separate the mandatory part of the assignment from the extra points defining then 2 proxy nodes named respectively "proxy" and "proxyMCL". The latter one is identical to the first one but interfaces to motion control law server before interfacing to move base and detection service.

In Fig. 1 the ROS nodes architecture we designed.

2 Work and data flow

We have above presented the ROS nodes architecture we came up with and the motivations that led us to it. We now briefly describe how the work and data flow from the client to the proxy, to all the other nodes and back to the client:

1. the client node is run requiring in input the pose_B goal towards to Tiago is required to be sent. Since Tiago is a differential wheeled ground-based robot the z position will always be 0, as well also the roll and pitch rotation parameters cannot change. We therefore decided to require in input for the pose_B goal only the x, y Cartesian coordinates (w.r.t. to the robot reference frame) and the yaw angle in degrees.

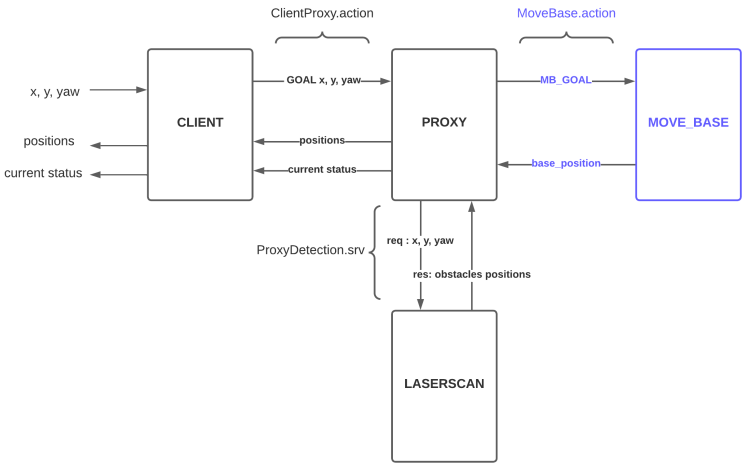

Figure 1: ROS nodes architecture.

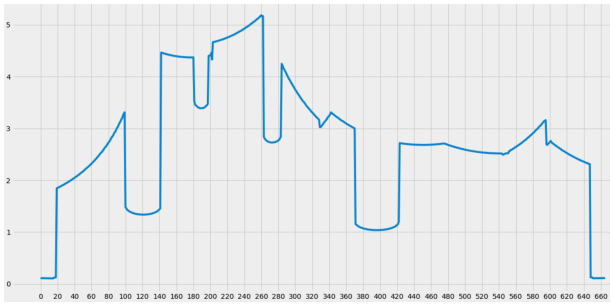

Figure 2: Laser data in polar coordinates from pose_B = {x=11, y=0, yaw=-43°}.

2. the client now connects to the proxy server and sends the pose_B goal, while periodically printing, through a feedback callback, the robot's current status. A custom ClientProxy.action is used to communicate between the client and proxy.

3. the proxy acts now as server w.r.t. to the client node and as client w.r.t. to move_base server. It receives the pose_B goal as 3 floats and constructs the goal as required by move_base filling MoveBase.action goal. Through a feedback callback checks the current positions returned by move_base, makes an internal check with internal variables representing the previous robot pose and determines if the robot is stopped or moving. It publishes through the feedback field of ClientProxy.action the robot current status. If the goal sent to move_base ends with success it also publishes to current status that the robot has reached pose_B goal. We can see how the proxy takes care of constructing the goal to send towards move_base and translating from position values received from move_base to current status strings, all without any overhead in the client node.

4. only if the robot has reached pose_B goal the proxy sends a request to laserScan node to perform movable obstacles detection.

5. the laser scan node computes the obstacles centers through a simple yet effective algorithm that will be described later and sends the locations back to the proxy through the response service fields. The pose_B values are required to be sent in the service request as we decided to compute the obstacles centers with respect to both the robot reference frame and the world reference frame (the r.f. depicted in Gazebo). These values are required to fill in the transformation matrix from robot r.f. to world r.f..

6. while detection is being performed the proxy informs the client about robot's current status publishing on current status "robot started the detection of the obstacles" and "detection is finished". Once the request is completed sends back the positions to the client through positions result fields of ClientProxy.action and sets the client goal to succeeded.

7. only now the goal sent by the client has finished and the client prints at terminal the movable obstacle centers with respect the 2 reference frames.

3 Movable obstacle centers detection algorithm

We implemented a really simple yet effective algorithm to perform movable obstacle centers detection. The algorithm relies on laser data obtained from subscribing to "scan" topic. In order to not cycle for an infinite time the laserScan node does not subsbribe to scan topic and use messageCallback as usual, but obtains only 1 message using waitForMessage function. The laser data is in polar coordinates and contains 666 values. The values in it are the ρ values, while the $agle\_increment * the\_position\_in\_the\_range\_vector + angle\_min$ gives the angle θ. The scan is anti-clock wise. As we can see from Fig. 2 the movable obstacles are characterized by a big discontinuity. We use this discontinuity to localize the start and end of the obstacles, which we call "depressions". By trying at different pose_B we found that a threshold of eps=0.8 generalizes well. Once the depressions are found we localize the argmin positions that corresponds in practice to the angle of the laser hitting the most close to the robot part of the obstacles. How can we now find the obstacle centers? If we assume that the movable obstacles have all the same dimension, the width of the depressions do not depend on the object size but on the distance between the obstacle and the robot. We can extend the ρ value at the argmin angle of each depression by a fixed amount delta to move from the obstacle frontier to its center. This method may appear quite raw but turns out to compute quite correct centers, and the error may mostly come from the laser accuracy (see how the walls appear not perfectly straight). We finally convert the polar coordinates into cartesian ones and compute the movable obstacles centers both w.r.t. to the robot reference frame and the world r.f. (the one depicted in Gazebo). The latter conversion is done through a transformation matrix. Last remark: we remove the first and last 20 values as they appear to be mapping the robot itself.

4 Metrics and conclusions

Computed movable obstacle centers w.r.t. world reference frame, computed at pose_B = {x=11, y=0, yaw=-45°}: {(3.82, -0.31), (4.54, -2.49), (5.99, -1.44), (5.86, 0.84)}. Ground truth movable obstacle centers w.r.t. world reference frame: {(4.16, -0.30), (4.87, -2.42), (6.16, -1.35), (5.81, 0.76)}. Mean Squared Error = 0.2754

All the code was written by 6 hands using Simone's PC and all members contributed equally to the work. We implemented also the extra points. The video in the Google Drive folder shows the robot moving using our defined motion control law through the narrow corridor, starting then navigation through move_base and finally requesting to detect movable obstacles centers.