SOFTWARE ARCHITECTURES FOR ROBOTICS

SOFAR Project Report
(Mobile Robot Navigation & Mapping)
Year: 2020 - 2021
Group: Navigation-05

https://github.com/mohammed-eldin/SOFAR_NAV_05

Students
Mohamed Alaaeldin Youssef Mahmoud - S4844271
Mohamed Qaoud - S4729321
Naresh Arthimalla – S4927772

Table of Contents:

Abstract

The goal of this project is to investigate the fundamental principles and show how to utilize the ROS Navigation Stack to conduct localization and path planning on a robot in an unfamiliar environment. This is a powerful path planning and simultaneous localization and mapping toolkit (SLAM). We will carry out the project using Unity simulation with the capacity of being carried out on a genuine autonomous vehicle manufactured by the Husqvarna company. This project was written with Python3 and tested using ROS noetic.

1.Introduction

Simultaneous localization and mapping (SLAM) is an important approach that allows the robot to acknowledge the obstacles around it and plan a path to avoid these restrictions. This method is a merge of several approaches that allow robots to navigate on unknown environments. ROS has several packages that perform the desired task, so we can use the following packages for this purpose:

1.1 Gmapping

Gmapping is based on SLAM, a technique that involves mapping an environment while the robot is moving. In other words, while the robot navigates through an area, it receives information from the environment using its sensors and creates a map. Gmapping uses odometry data to build and update the map, as well as the robot's posture.

1.2 Move_Base

The move_base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner that are used to accomplish navigation tasks.

Carrot planner, navfn, and global planner are the three default global planners. The Carrot Planner may be the most basic. navfn employs Dijkstra's algorithm to determine the shortest global path between two points. Global planner is designed to be a more versatile substitute for navfn, with additional choices, including support for A*.

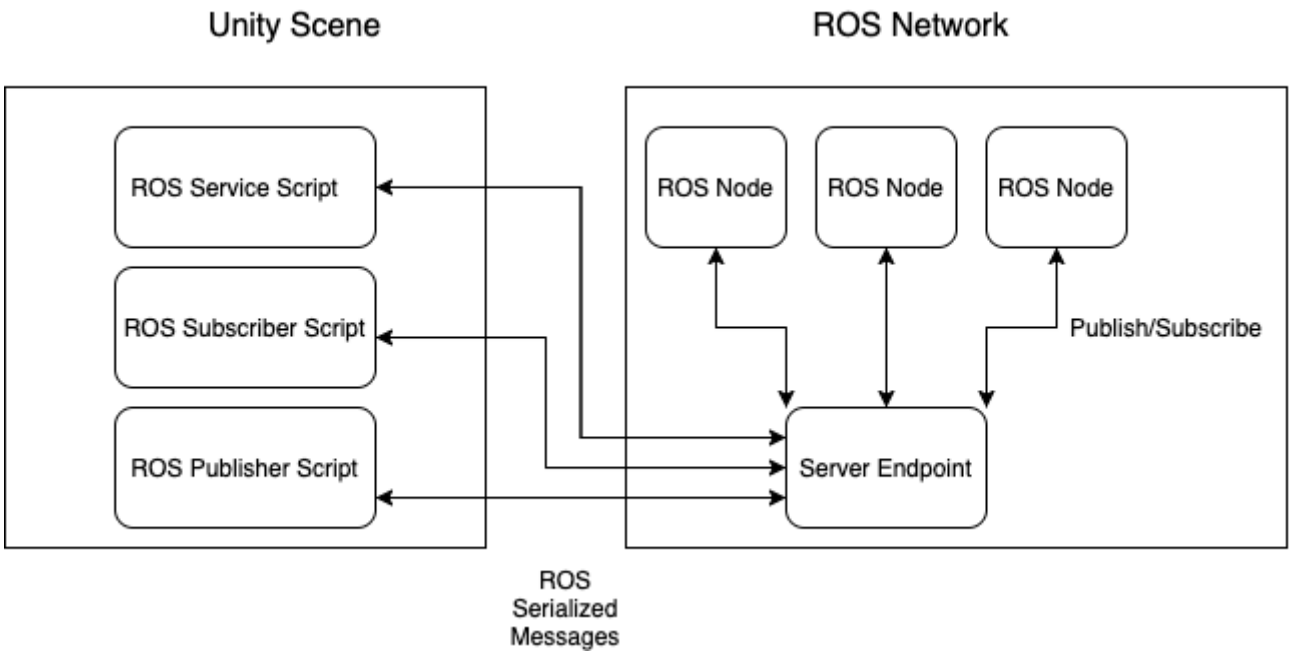
Dwa local planner, eband local planner, and teb local planner are all possible local planners. They produce velocity directives using various algorithms. We utilize dwa local planner in our situation.

1.3 Unity Simulation

Because developing and testing applications with a real robot is expensive and time-consuming, simulation is becoming an increasingly essential element of robotic application development. Validating the application in simulation before deploying it to the robot helps save iteration time by identifying possible problems early on. Simulating also allows for the testing of edge situations or scenarios that would be too risky to test in the actual world.

1.4 ROS_TCP_endpoint

ROS package used to create an endpoint to accept ROS messages sent from a Unity scene using the ROS TCP Connector scripts.



1.5 Husky

A medium-sized robotic development platform. The UGV may be outfitted with stereo cameras, LIDAR, GPS, IMUs, manipulators, and other sensors. Husky is fully supported in ROS thanks to open source code contributed by the community.

1.6 TF

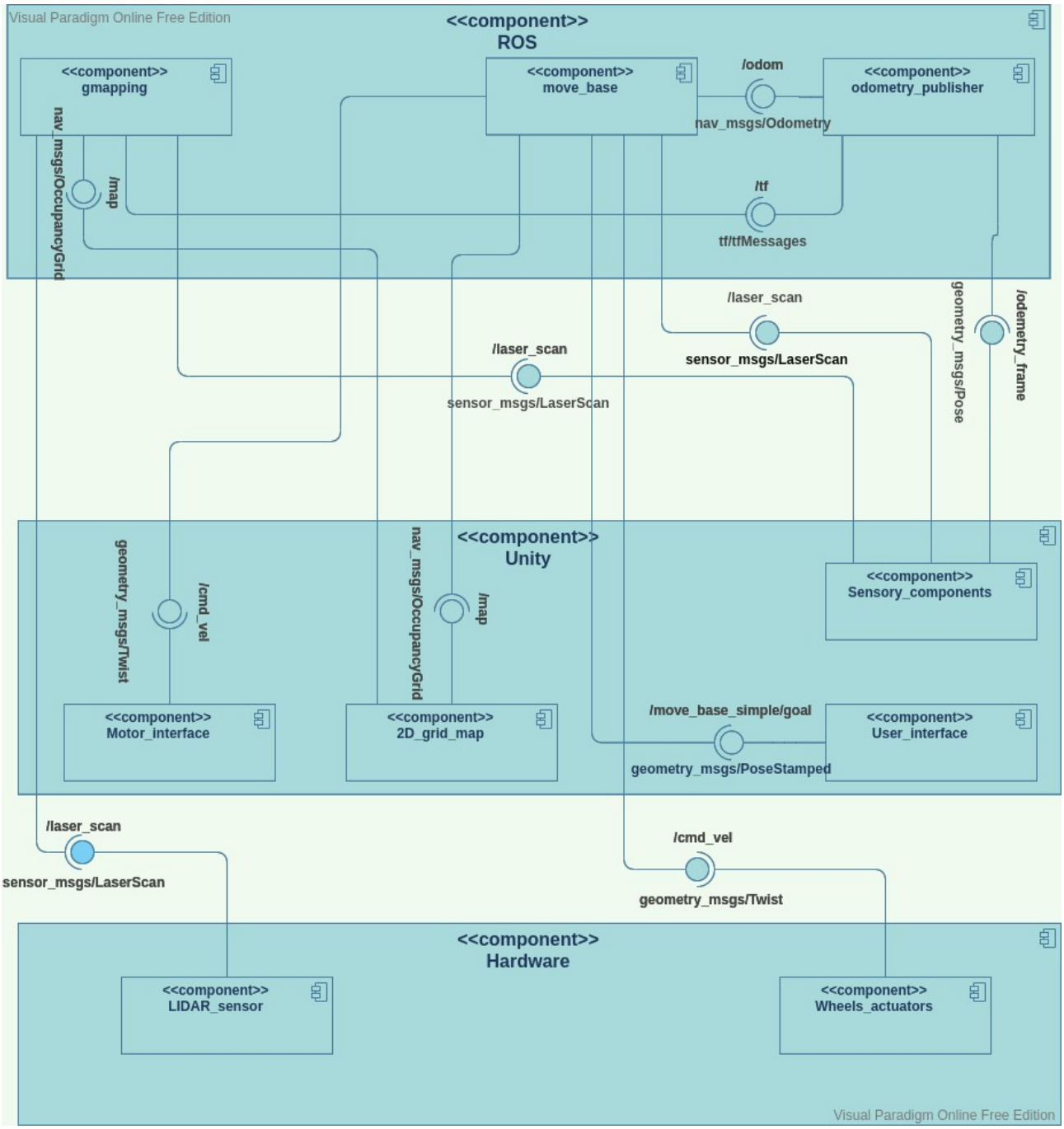
TF is a package that allows the user to track the position of several coordinate frames across time. tf stores the connection between coordinate frames in a time-buffering tree structure and allows the user to convert points, vectors, and so on between any two coordinate frames at any desired moment in time.

These adjustments are needed because the sensors measure the environment in relation to themselves, not in relation to the robot, in other words, a geometric conversion is needed. To make this conversion simpler, TF tool, which makes it possible to adjust the sensors positions in relation to the robot and, this way, adequate the measures to the robot‘s navigation.

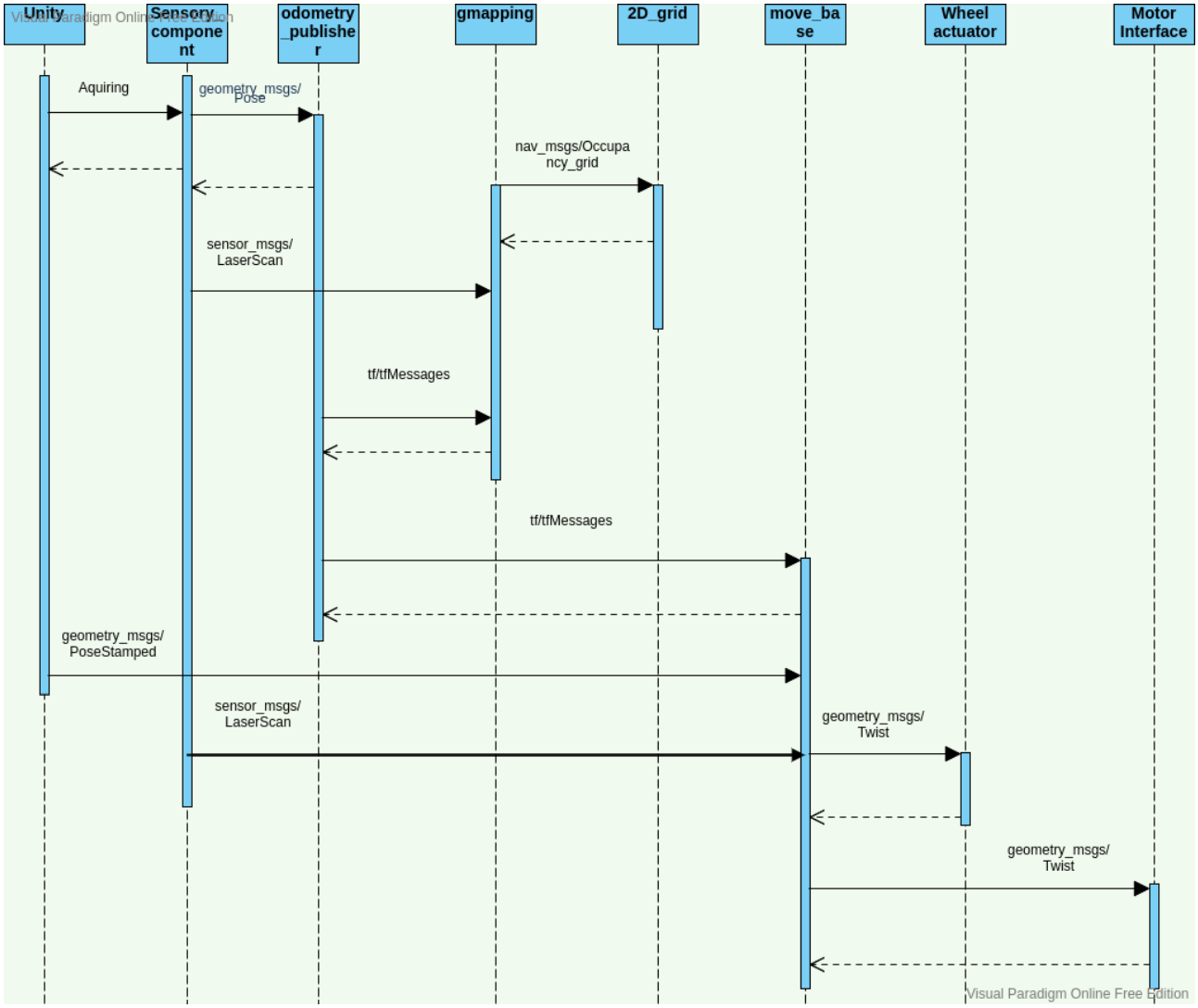
2.Architecture of the System:

In this part we are going to discuss the whole software architecture of the system, which included the data flow among the different packages that will be used for the desired purpose. For investigating this phenomena we are going to use two types of diagrams which are based on unified modeling language to visualize the way of a software architecture is designed. That would be as following:

2.1 Component Diagram



2.2 Temporal Diagram



As it has been shown in the previous diagrams, we can describe the sequence of the data flow as the following:

- Data is collected from the LIDAR sensor (Unity) and transmitted to Gmapping package using the publisher subscribe design pattern via the /laser_scan (ROS) topic.

- The robot's location is acquired and transmitted to ROS through the publish subscribe design pattern via the /odometry_frame topic.

- The base link frame is broadcast to ROS from Unity via the /tf topic.

- The position is published in ROS via a transformation on the /odom subject.

- The Gmapping package utilizes data from /tf and /laser_scan to publish a map on the /map topic.

- The Move_base package accepts as input the map created by Gmapping and information from Unity simulation: the robot's location and the target position, which is published on the topic /move_base_simple/goal.

3 Description of the System’s Architecture:

In this part we will investigate each component in the system’s architecture in detail which included:

3.1 Gmapping

The primary goal of this package is to support localization and mapping in the ROS environment. This node has the ability to generate a map that will be published on the (/map) topic by sending (nav_msgs/OccupancyGrid) messages. This package subscribes to the (/tf) topic, which supplies it with the (tf/tfMessages) message, which includes the relationship between the laser link and the base link, indicating the laser scanner's posture in reference to the robot base. If this transform is incorrect, the localization will most likely act oddly. It also receives the LiDAR sensor, which publishes (sensor_msgs/LaserScan) messages on the (/laser_scan) topic.

3.2 Navigation

Its job is to allow the robot to reach the desired position while avoiding obstacles utilizing the navigation stack. The move base ROS package is used to implement it. The move base package comprises nodes that collaborate to achieve the navigation goal, which may be categorized as follows:

3.2.1 Local and Global costmaps:

The topics containing the information that represents the projection of the obstacles in a 2D plane, as well as a security inflation radius, an area around the obstacles that guarantees that the robot will not collide with any objects, regardless of its orientation, are the local and global 2D costmaps. The global costmap depicts the whole environment, whereas the local costmap is a scrolling window that moves in the global costmap in proportion to the robot's current position.

3.2.2 Local and Global planners:

Local and global planners do not operate in the same manner. The global planner takes the current robot location and the objective and plots a lower-cost route in relation to the global costmap. The local planner, on the other hand, has a more fascinating task: it operates over the local costmap, and because the local costmap is smaller, it generally has more definition, and therefore can identify more barriers than the global costmap. As a result, the local planner is in charge of developing a trajectory rollout across the global trajectory that can return to the original trajectory at the lowest cost. To clarify, move base is a package that comprises the local and global planners and is in charge of connecting them to fulfill the navigation aim.

3.3 Odometry_publisher

The major job of this component is to preserve the relationship between the various coordinates by subscribing to the /odometry frame topic and republishing it on the /odom topic, which may be used for the local planner node. In addition, publishing in the /tf topic that was utilized by gmappig to generate the necessary map. As a result, we may classify this node as an adapter design pattern.

3.4 User_interface

This module's responsibility is to publish the target location at the start of the simulation. When the user presses the Move! button in the Unity graphical interface, the goal's location as geometry_msgs/PoseStamped message is published on the /move base simple/goal topic. A red circle marks the objective in the Unity scenario.

3.5 Sensor_interface

This system component may be thought of in terms of hardware-software interaction and, as previously said, platform specific nodes. This component is made up of two parts:

- Simulated implementation of a real-world LIDAR sensor. Laser scan does not accept input and instead publishes on the /laser scan topic.
- Odometry data published with the use of wheel encoders.

3.6 Motor_interface

This component is a simulated version of the real robot that is subscribed to the cmd_vel topic and receives velocity commands as geomety_msgs/Twist messages. It is responsible for controlling the robot to accomplish the specified objective while avoiding obstacles.

3.7 LiDAR_sensor

This component is thought to be the genuine light detection and ranging sensor that would be installed on the actual robot to scan the surroundings for navigation and mapping.

3.8 Wheels_actuators

This operates the real robot by subscribing to the /cmd_vel topic in real time and adjusting the rotation velocity of the wheels based on the published messages.

4 Installation

This project is built with:

- ROS noetic (http://wiki.ros.org/noetic/Installation)
- Python3 (https://www.python.org/downloads/)
- Gmapping (http://wiki.ros.org/gmapping)
- DWA local planner (http://wiki.ros.org/dwa_local_planner)
- Husky (https://github.com/typicode/husky)

We put the project through two rounds of testing.
1 - Implementing the project on the same computer (Linux).
2 - Applying to several computers (Linux for ROS) (MacOS for Unity).

4.1 Installing

To install the system using this approach and run the code flawlessly, we must perform the following:
1 - Install ROS-Noetic

2 - Install Unity 2020.x.x: We used both Unity 2020.2.2/2020.3.13.

3 - Clone our Repo into your ROS Workspace By running this command in the terminal

“ git clone https://github.com/ mohammed-eldin/SOFAR_NAV_05 “

After you've cloned the repository, go to the config folder and open the params.yawl file to change the IP address of the ROS/Unity.

4 - Installing the Navigation and gmapping libraries by using the following commands:

“ sudo apt-get install ros-noetic-navigation ”

“ sudo apt-get install ros-noetic-openslam-gmapping ”

5 - As the Husky robot is the robot that would scan the environment, we have to include it to our workspace:

To do so, we must use the following command to clone the repository to our ROS workspace.

“ git clone https://github.com/husky/husky.git “

6 - Also, we have to install the LiDAR sensor by running the following command:

“ sudo apt-get install ros-noetic-lms1xx ”

The following is the second way for preparing the system to run the project:

We utilized the identical processes as in the first way, but instead of utilizing the same computer, we used two distinct machines "remotely" by using middleware to establish a virtual network. We utilized a program named LogMeIn Hamachi. After installing the program on both devices, one machine creates a network and invites the other machine to join it. After entering the network, the program assigns each computer an IP address, so we use these IP addresses instead of the real ones to be on the same network "virtually."

4.2 Running The Code

After finishing all the required to install, we can run the project as follows:
In the ROS side:

1 - Launch the gmapping package by the command:

“ roslaunch mobile_robot_navigation_project gmapping.launch “

2 - Launch the move_base package by the command:

“ roslaunch mobile_robot_navigation_project move_base.launch “

3 - Launch the model on Rviz by the command:

“ roslaunch mobile_robot_navigation_project view_model.launch “

Also, uploading the model of the robot in the Rviz but opening it from Rviz from the repository config folder.

4 - Launch the Navigation node to start the communications between ROS and Unity by the following command line:

“ roslaunch mobile_robot_navigation_project navigation.launch “

On the Unity side, we need to navigate to the previously created environment and enter the IP address of the ROS machine into the Scene area. If we simply used one computer, the IP address would be the same, but if we used two different computers connected via a virtual network, we would use the virtual IP address. Wait until all of the nodes on the ROS side have been launched before hitting the play button in the Unity in the center top window to avoid issues. If the connection is good, we should see a window on the left with the date from the ROS and the data from the Unity.

5 System Testing and Results:

As previously said, we tested the project in two ways: one using the same machine to run ROS and Unity and the other using a virtual network to utilize two machines and link both machines to the same network to execute the project.
The benefits of having a remote control include the ability to utilize various machines in completely different locations, but the downside is that scanning the map takes some time because it is dependent on the speed of the internet.
The advantage of having them placed on one machine is that you can scan in less time than if you use two separate machines, but the downside is that it is not practical, especially if you use it in a real robot.
The time it takes to scan the map using the same machine technique may take up to 20 minutes, however utilizing two machines could take twice as long. The image below depicts the outcome of scaling the Unity map.