ME5400 Project Report

A User Friendly Human-AMR Interface APP
Based on VUE3-ROS

Name: Huang Zhenbiao
ID: A0285316B

April 30, 2024

Contents

1 Introduction

1.1 ROS in AMRs

ROS, or the Robot Operating System, is an open-source, meta-operating system for robot. It is used by hundreds of companies and techies across the globe in the field of robotics and automation. **AMR, or the Autonomous Mobile Robot**, is a type of robot that is designed to move and operate within an environment without the need for human guidance. On the software side, an AMR typically runs on ROS because it provides the libraries and tools necessary for developing robot applications, including capabilities such as motion planning, perception, and control.

To help developers better understand the robot system and manage data within the robot, ROS provides many GUI tools for data visualization and interaction. Rviz (ROS Visualization) is one of the most widely used tools because it can visualize most of the data in the robot system, including the LiDAR point cloud, the transformation relationship between frames, the robot trajectory, and more. Rviz also provides an intuitive way to interact with the robot. For example, the user can set a navigation goal on the current map loaded by the robot.

1.2 Intergrating ROS with Other Software

Rviz, while powerful, has constraints. It's a desktop app, unsuitable for screenless small robots. Remote desktop software can be used, but it burdens the robot's resources and doesn't support touchscreens. Also, Rviz requires ROS knowledge, which is unrealistic for ordinary users in commercial scenarios.

In order to integrate ROS with other software, the ROS official and community have provided many open-source tools:

- rosjava: An implementation of ROS in pure Java with Android support
- Rosbridge: A web server provides JSON API to ROS functionality for non-ROS programs
- ROS Web Tools: An open source group provides many basic libraries for developers who wish to visualize their robotic data on the web, including roslibjs, ros2djs and ros3djs.

By using these libraries, many developers have already create excellent None-ROS app for ROS, they not only can visualize 2D map of the robot, but also 3D Pointcloud and 3D model for robot arm, like Webviz, Foxglove studio and ROSboard.

While many libraries exist for remote ROS interaction and robot-controlling apps are common in commercial robots, finding an open-source app for ROS beginners to quickly control robots via mobiles is tough. Many libraries, like rosjava, are outdated. Many programs focus on data display but lack effective touchscreen support. Also, many are project-specific and lack universality.

1.3 Developing a Mobile-Compatible ROS Control App

This paper presents a new web app for controlling AMR. Unlike previous roslibjs-based projects, this one is designed for web developers, treating ROS as a back-end server. The result is a lightweight, user-friendly, and easy-to-install robot control app that works well on touch devices.

The development process began with deploying a ROS simulation program via the Windows Subsystem for Linux (WSL). This served as a testing platform for the web app. A ROS-side program was written to add advanced features to the app, with the potential for porting to other AMR projects.

After successful simulation testing, an existing lab AMR was used for real-world testing. All app functionalities were executed successfully on this AMR. The app was then packaged into a Docker image for easy deployment, eliminating the need for a web development environment.

The app was also deployed on another lab AMR via Docker, demonstrating the potential for seamless integration and deployment of web apps in AMR projects, thus expanding the possibilities for innovative robotics solutions.

2 Web APP Development

2.1 Simulation Environment Setup

Prior to initiating the coding process for the web application, a simulated AMR was launched within the WSL, a virtual machine provided by Microsoft. WSL facilitates the creation of an Ubuntu 20.04 environment within the Windows 11 system via the Microsoft Store, offering a more convenient alternative to installing an additional operating system on the laptop.

Within this sub-linux, ROS Noetic was installed, and the Turtlebot3 Simulation tutorial was followed to prepare a simulation for the Turtlebot3 robot. Turtlebot3, a beginner project for ROS learners, utilizes the Gazebo simulator to provide a virtual robot and environment. The tutorial not only instructs on how to launch the simulation, but also includes guidance on running Simultaneous Localization and Mapping (SLAM) and navigation programs within the simulation, thereby providing a convenient environment for testing the web application.

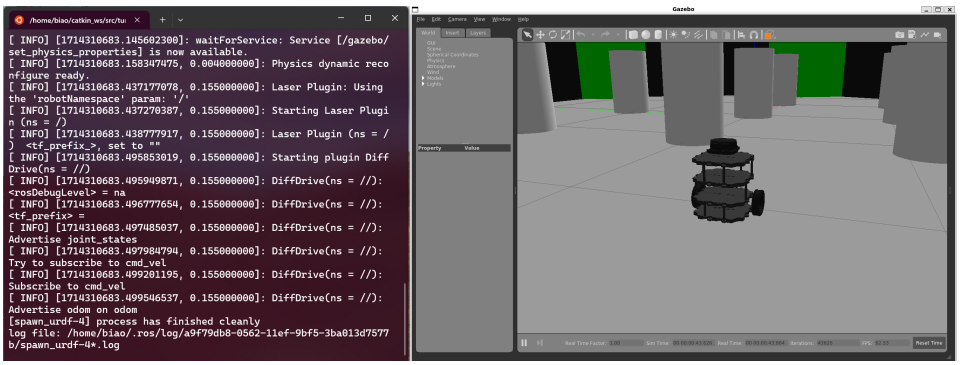


Figure 1: Turtlebot3 simulation running on WSL

2.2 Data Communication

After setting up the Turtlebot3 simulation, Rosbridge was installed in the WSL for ROS access. The roslibjs source code shows it uses a WebSocket client to connect to Rosbridge. WebSockets allow continuous, two-way communication between a client (usually a web browser) and a server (Rosbridge here).

When Rosbridge is launched in WebSocket mode, it creates a WebSocket server on port 9090. An online WebSocket client tool was used to connect to this server and send commands to ROS. As shown in Figure 2, the top left browser shows the WebSocket server by Rosbridge on port 9090. The bottom left shows the WSL terminal running Rosbridge and the Turtlebot3 simulation (with "client connected" in the log), and the right side shows the online tool (with received odometer data in the logs).

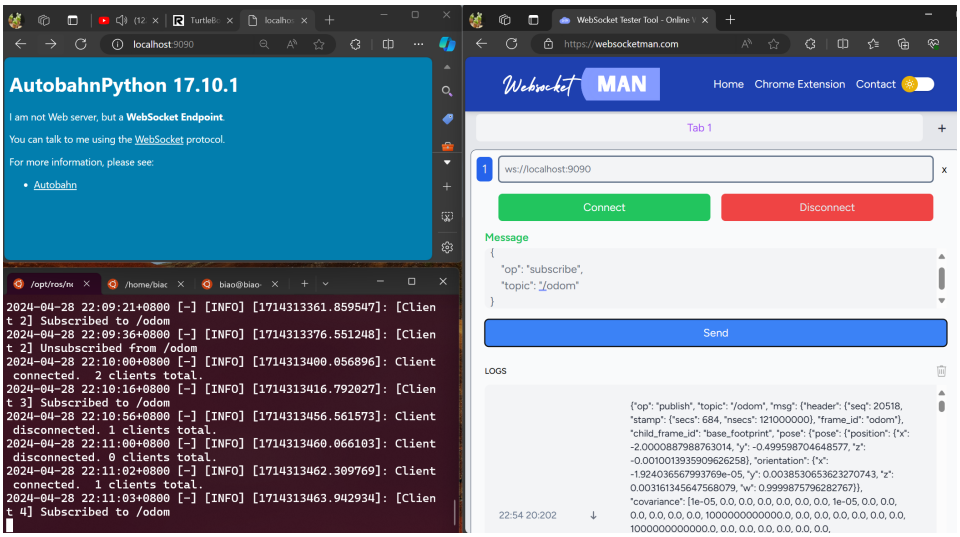


Figure 2: Using online websocket tool to get odometer data from ROS

Further experimentation was conducted with various ROS commands utilizing the online WebSocket tool. This process culminated in the successful deciphering of the protocols required to access ROS via WebSocket. Consequently, the necessity for `roslibjs` in the web application was obviated. This facilitated the direct writing of code based on a WebSocket connection with Rosbridge, thereby streamlining the development process and enhancing the efficiency of the web application.

2.3 Visualization

After proving the feasibility of using Websocket with ROS, I started developing my web app. The fundamental web front-end framework employed in this project is Vue.js, a front-end framework that extends HTML, CSS, and JavaScript. It helps in creating user interface (UI) efficiently. Numerous UI frameworks are built upon Vue.js. For the initialization of my project, I opted for the Quasar Framework as it provides a modern UI following Google's Material Guidelines, supporting both desktop and mobile devices.

In the development of a WebSocket client within the web application, the WebSocket server URL (here, ws://localhost:9090) is utilized to instantiate a WebSocket handler object. Subsequently, the callback function of the WebSocket handler is defined.

The ws.onmessage function is invoked upon the receipt of a new message from the server. A switch statement is employed to process the received message, contingent on their type and ROS Topic. For instance, when a message from the `/odom` topic is received, the message should resemble the following structure:

```
{
    "op":"publish",
    "topic":"/odom",
    "msg": { data: Odometry }
}
```

Indeed, upon establishing the WebSocket connection, data is not automatically received. Instead, a command must be sent to Rosbridge specifying the required data. For instance, to subscribe to the `/map` data, the ws.send function is used to transmit the following message to the WebSocket server:

```
{
    "op": "subscribe",
    "topic": "/map"
}
```

Indeed, it is only after the `subscribe` command is dispatched to Rosbridge, and the target data has been published in ROS, that data can be received in the ws.onmessage function.

2.3.1 Visualization of Map

In the visualization of AMR, the robot's map is crucial. In the Turtlebot3 simulation, launching SLAM or navigation publishes map data in the OccupancyGrid format to the `/map` topic. This data encapsulates a 1-D array for each grid on the map, along with metadata such as height, width, resolution, and origin. Utilizing this data, a map image can be rendered, resized based on resolution, and positioned according to the origin.

PixiJS, a robust 2D rendering engine for web applications, is employed to render the map image and adjust its position. The visualization process commences with the creation of a 2D pixel array, which is dimensioned according to the map's height and width. Each pixel value is then assigned based on the map's data array, converting from an occupied probability range of [0,100] to a grayscale pixel value range of [0,255]. The map image is subsequently scaled and its position adjusted according to the origin data. Finally, the map image is inserted into a canvas created by PixiJS. Figure 3 illustrates the map image as visualized in Rviz and the web application.

To augment the user experience, additional functionalities have been incorporated into the map. Users have the flexibility to change the ROS Topic for map data, catering to

the possibility of different algorithms publishing to different ROS Topics. Leveraging the features of PixiJS, users can smoothly adjust the zoom and position of the map image using a mouse or touch.

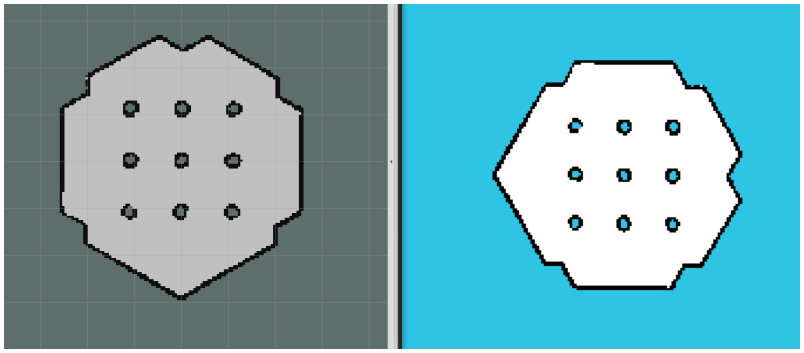


Figure 3: Map image in Rviz and web app

2.3.2 Visualization of Robot Pose

In Rviz, the robot's pose is visualized by calculating the transformation from the robot frame to the map frame. However, for web application visualization, only xy coordinates and a θ direction in the xy-plane are required. Thus, a C++ ROS node was developed to calculate the robot pose and publish it to a ROS Topic, which is then used directly in the web application.

The ROS node employs a `TransformListener` to look up the transformation from the `map` frame to the `base_link` frame (representing the robot), and publishes a PoseStamped format data to the `/robot_pose` ROS Topic every 10ms. The web application subscribes to this topic and uses the data to adjust the robot's pose in the PixiJS canvas. This approach, which involves creating the robot image in the initial stage and adjusting the pose according to the ROS Topic data, reduces the rendering load on the web application. Figure 4 illustrates the robot pose in Rviz and in the web application, with the blue arrow representing the robot pose in the web application.

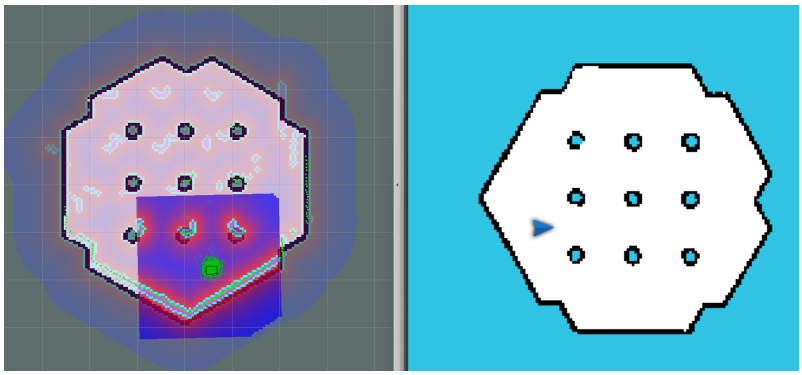


Figure 4: Robot pose in Rviz and web app

2.4 Basic Robot Control

To control the robot via the web application, it's necessary to send commands via ROS Topics. These commands are similar to topic JSON received from Websocket server, with the op field set to "publish", the topic field specifying the target ROS Topic, and the msg field containing the data to be published. For instance, to send a data to the `/cmd_vel` topic, the following JSON structure would be used:

```
{
    "op": "publish",
    "topic": "/cmd_vel",
    "msg": {
        // Message structure here
    }
}
```

2.4.1 Virtual Joystick Controller

Most ROS AMRs have a ROS Topic to receive twist commands (including linear and angular velocity) and move accordingly. For the Turtlebot3 simulation, this ROS Topic is `/cmd_vel`. A ROS program provided in the Turtlebot3 tutorial sends twist commands using a Linux terminal keyboard.

NippleJS, a JavaScript library, creates a virtual joystick for touch-capable interfaces. It's commonly used in games, apps, or web pages controlling hardware devices. An empty page with two joysticks was created. The Y-value of the left joystick controls linear movement, and the X-value of the right joystick controls rotation. A timer was added to read these values, convert them to twist data, and send them to the ROS `/cmd_vel` Topic every 25ms. To prevent the joystick from continuously sending zero-speed commands when not touched, a boolean was added to check if the last command was zero speed. If the previous and current commands are both zero speed, no command is sent to ROS. Figure 12 shows the virtual joystick. The left terminal runs `rostopic echo /cmd_vel`, and the right side displays the virtual joystick.

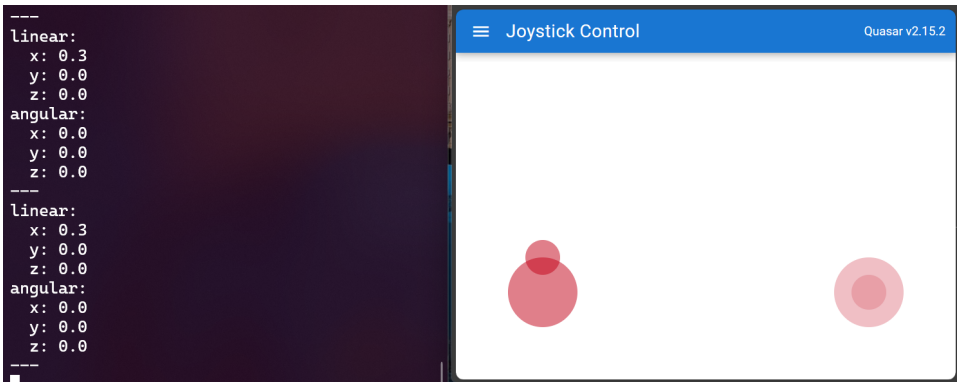


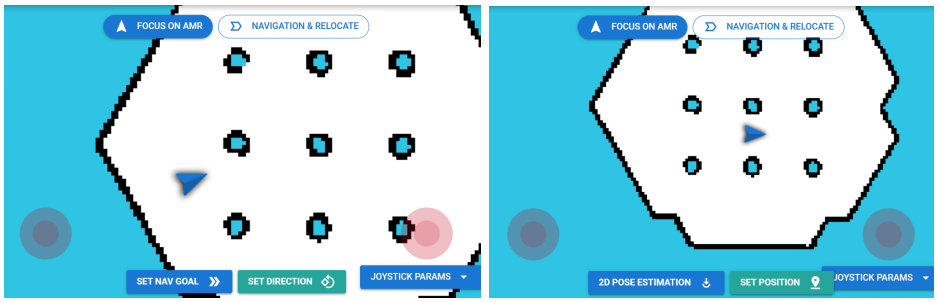
Figure 5: Virtual Joystick and /cmd_vel topic

Upon validating the functionality of the virtual joystick with the Turtlebot3 simulation, additional features were incorporated. These include control via arrow keys, setting maximum velocity, and changing the target ROS Topic. Consequently, this joystick can control any AMR capable of receiving twist commands. To facilitate simultaneous robot control and mapping, the joystick was also integrated into the map visualization page of the web application.

2.4.2 Pose Estimation and Navigation Setup

In Rviz, `2D Pose Estimate` and `2D Pose Estimate` are commonly used functions for AMRs. `2D Pose Estimate` sets an initial guess of the robot's pose in the map for sample-based localization algorithms like AMCL, while `2D Nav Goal` sets a navigation goal for the robot. These functions work by sending pose data in the map frame to specific ROS Topics.

To calculate pose data in the web and send commands, an `pointerdown` EventListener is added to the canvas hosting the map image. Clicking a position on the canvas returns the position value relative to the browser page. This position value is then converted to the map frame by calculating the relative position between this position and the origin of the map in the browser page. Thus, the position in the map frame can be obtained using the web app. Two different positions in an xy-plane can determine a direction. When setting the pose, the user first chooses a point in the canvas as the position value, then chooses another point, and the vector from the previous point to this point is used to calculate the direction value. Figure 6 shows how this is achieved in the web app. A switch button allows the user to decide whether to set the position or direction of the target pose. Another switch button is used to decide whether it is `2D Pose Estimate` or `2D Nav Goal`. Once the user chooses the pose and function, the web app sends a command to the target ROS Topic, `/initialpose` or `/move_base_simple/goal`.



(a) Set Direction (b) Set Position

Figure 6: 2D pose estimate and nav goal

2.5 Advanced Robot Control

The preceding sections detail the development of a web application capable of basic AMR visualization and control, effectively replacing Rviz for simple tasks. This section delves into the development of advanced controls, such as map selection for navigation launch or SLAM launch for mapping, all through the web application. These controls are commonplace in

commercial robots, as it is impractical to expect users without a ROS background to launch ROS programs.

Typically, commercial robots have a program that manages other ROS programs (like SLAM and navigation), which starts upon robot boot-up. This allows users to operate the robot simply by connecting to it via an app and sending commands.

To implement these functions, a ROS program named AMR Remote Control Toolkit was developed. Figure 7 illustrates the main components of this package. The following sections will provide an overview of how this program operates.

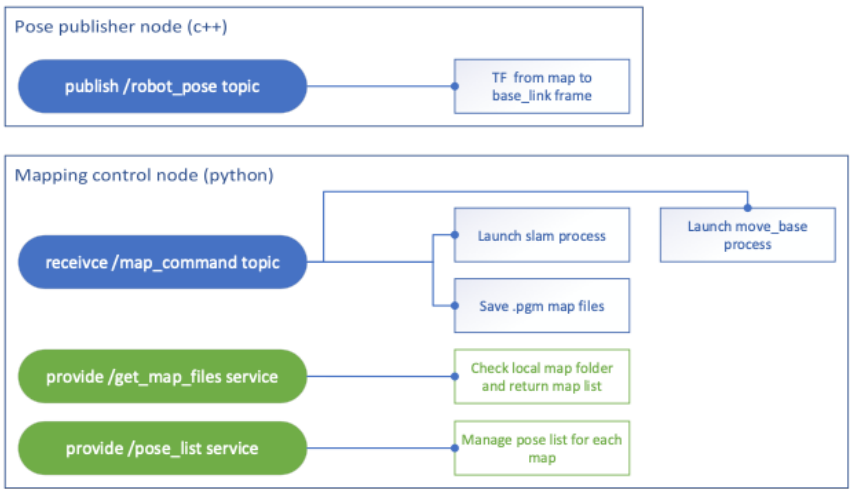


Figure 7: Overall structure of AMR Remote Control Toolkit

2.5.1 Remote Start of SLAM and Navigation

The Python `subprocess` module enables the execution of Linux commands within a Python program. This functionality is leveraged to create a Python ROS node, mapping_control.py, which receives commands and launches other ROS programs, including SLAM and navigation.

A robot first uses SLAM to generate an environmental map, which is then utilized for navigation. Consequently, SLAM and navigation programs should not run concurrently. To manage this, a state machine is implemented in mapping_control.py with three states: `idle`, `navigation`, and `mapping`. Initially, the program is in the `idle` state. It receives commands from the `/map_command` ROS Topic and changes state upon command execution.

- When the node receives a "load" command (to load an existing map), the command should also contain the map name. The node terminates other sub-processes, such as old navigation or mapping processes, then launches the predefined launch file in a sub-process, and sets the current state as `navigation`. If the node is already in the `navigation` state and receives a new navigation command, it implies that the user wants to change the map used in navigation. If the node is in the `mapping` state and receives a navigation command, it signifies that the node will abandon mapping and revert to the `navigation` state. The default navigation launch files in Turtlebot3 include the map file path, which is hard-coded in the file. This was modified to make

the map path file a configurable item, enabling the program to launch navigation using a specific map file according to the command.

- When the node receives a "start" command (to start mapping), if the node is not in the `mapping` state, it uses the predefined mapping launch file to start a sub-process and terminates other sub-processes, like the navigation sub-process. If the node is already in the `mapping` state, it ignores the command. The default SLAM ROS launch file in the Turtlebot3 project is sufficient for launching the sub-process, so it is used directly.
- When the node receives a "save" command, the command should also contain the map name. If the node is in the `mapping` state, it launches the save map command to save the map files (.pgm and .yaml) to the map folder, and uses the newly created map to launch the navigation sub-process. If the node is not in the `mapping` state, it simply ignores the command.

The state transitions are depicted in Figure 8. A subscriber in the web application tracks the state and determines the visibility of buttons. For instance, the create map button appears when the node is not in the `mapping` state, and the save map button is hidden when the node is in the `navigation` state. To ensure compatibility with other robots, the launch file paths for SLAM and navigation, as well as the map folder path, are configurable.

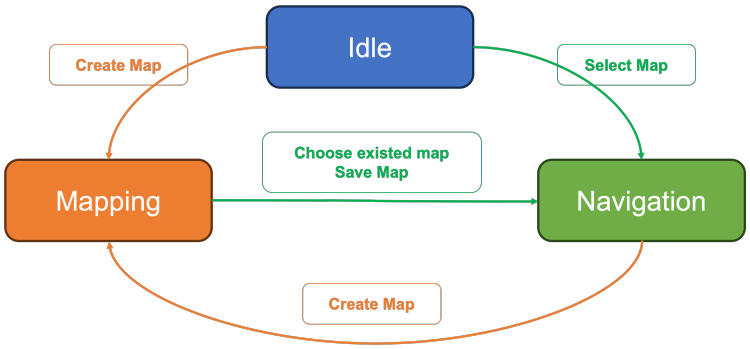


Figure 8: States in mapping_control.py

2.5.2 Map Selector

To facilitate the "load" command via the `/map_command` ROS Topic, which requires the name of the map file to launch the navigation sub-process, an API was added to query existing map files in the map folder. In mapping_control.py, the `/get_map_files` ROS Service queries the names of all .pgm images in the map folder, concatenates them into a comma-separated string, and returns this string.

In the web application, the JSON command structure for a ROS Service differs from that of a ROS Topic. A Service command must contain an id field, and only the message with a matching id from the WebSocket server is the correct response. To implement this feature, all service responses from Rosbridge are stored in a hashmap in the web application, where the key is the id. After the web application sends a Service command to Rosbridge, it checks the hashmap every 100 milliseconds until the matching id is found.

```
{
    op:"call_service",
    id: uuid
    service: '/get_map_files'
    args: []
}
```

Following the successful validation of the Service function, a map selector dialog was developed. This dialog displays available maps, eliminating the need for manual input of map names. Figure 9 illustrates this feature in the web application and Linux environment.

Figure 9: Map selector dialog and map folder

2.5.3 Pose Manager

The 2D navigation goal function, previously introduced, sets the pose by screen-clicking. This can be imprecise for pose selection or pre-set target pose movement. To mitigate this, a `/pose_list` ROS Service was integrated into mapping_control.py. This service accesses a pose list, creating a .yaml file for each map to store `PoseStamp` formatted pose data, enabling creation, modification, and deletion of poses in the current map.

A Pose Manager dialog was added to the web application to display current map poses, which also appear on the map canvas when the dialog is activated. This allows the robot's current map pose to be used to add a new pose to the map pose list, or an existing pose to be selected as the navigation goal. As depicted in Figure 10, users can select a pose from the list, and the web application will publish a navigation goal command using this pose. Additional functionalities include adding map poses or saving the current list.

This function is particularly useful for testing the robot's motion precision. By marking two poses on the floor, controlling the robot to the pose, recording the map pose, and then having the robot move between these two map poses multiple times, the precision of the robot's stop pose relative to the floor-marked pose can be validated.

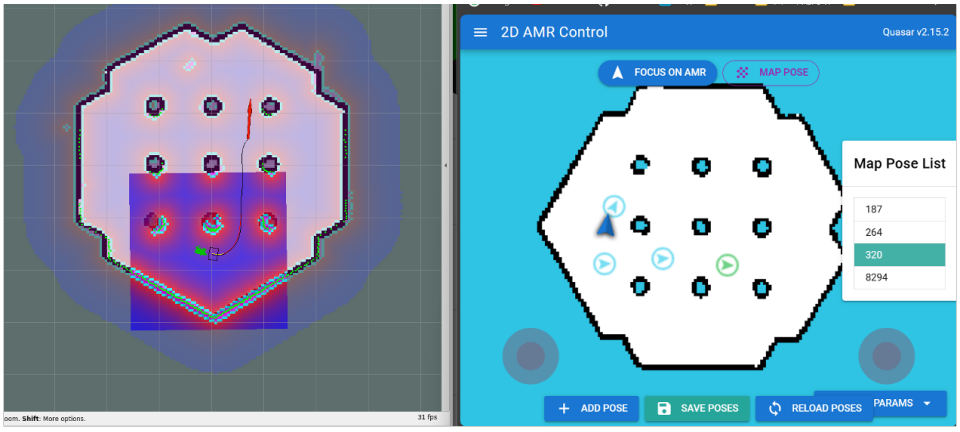


Figure 10: Map pose list dialog in web app

3 Implementation on Real Robot

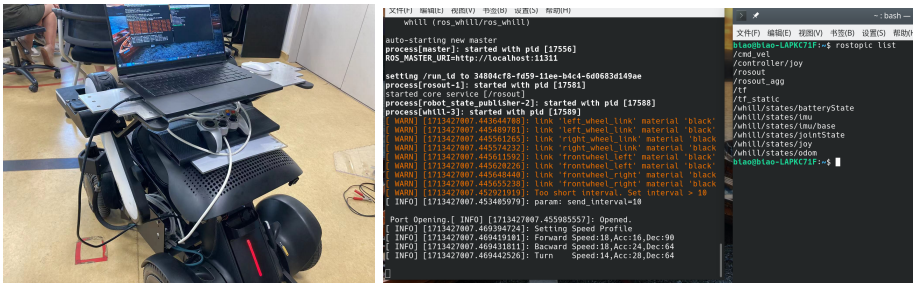
Building upon the capability to control the Turtlebot3 robot in Gazebo via the web application, the next phase involves testing on a real robot, specifically the WHILL Model CR wheelchair robot. This robot, an upgrade from the original electric wheelchair, is equipped with two sick tim551 2D LiDARs for environmental scanning, a Nvidia Jetson for executing ROS programs, and a compatible power supply system. The Jetson operates on Ubuntu 18.04 with the Melodic version of ROS, and hosts ROS programs capable of launching the robot, including basic drivers, SLAM, and navigation functions.

However, deploying the web application directly on the Jetson is not feasible due to potential interference with existing programs and the robot's use in other projects. Therefore, Ubuntu 20.04 was installed on a laptop, which was then connected to the wheelchair's hardware and LiDARs. The ROS program was launched on this laptop, which also hosts the web application environment. Consequently, the web application can be used to control the wheelchair AMR.

3.1 Driver Packages for Wheelchair AMR

3.1.1 Connect to Wheelchair

The ros_whill driver package, provided by the wheelchair's manufacturer, establishes a connection to the wheelchair via a USB serial port upon launch. It continuously publishes the wheelchair's state, including odometer data and battery status, and listens for control input such as joystick controller or twist command. While the official GitHub page states that the latest supported ROS version is ROS Melodic, running on Ubuntu 18.04, successful compilation was achieved on a laptop running Ubuntu 20.04 and ROS Noetic. Further functionality testing involved connecting the wheelchair to the laptop via a USB cable, launching the driver program as per the instruction file, and performing some configuration steps. The successful operation is depicted in Figure 11.



(a) Laptop connect to wheelchair (b) ROS_WHILL package output

Figure 11: Launch basic wheelchair driver on a laptop

3.1.2 Modify Wheelchair Control Input Function

The wheelchair driver program's default code for receiving control commands exhibited certain deficiencies. Specifically, the joystick input callback function would send commands to the wheelchair based on the joystick controller's current state. This meant that even when the joystick controller was connected but not in use, it would continuously send zero-speed commands, which could interfere with the navigation program. Additionally, the twist input callback function, which purportedly used experimental features, failed to function as expected during testing.

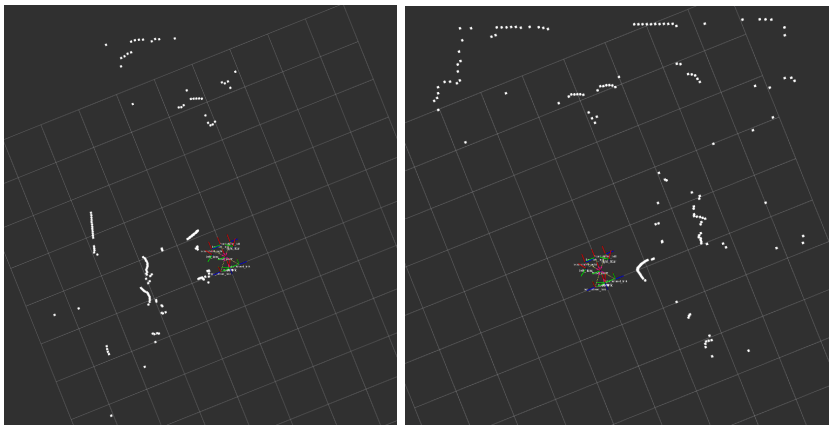
To rectify the joystick callback function issue, a safety button mechanism was implemented. This mechanism ensures that the node sends commands to the wheelchair only when the joystick's left bumper is pressed, as illustrated in Figure 12. Consequently, when the joystick is connected but not in use, and the left bumper is not pressed, the node refrains from sending zero-speed commands to the wheelchair. This enhancement significantly improves the system's efficiency and user-friendliness.

Figure 12: Operation method of the joystick controller

For the twist input function, modifications were made to align it with the joystick input function's interfaces. To ensure the accuracy of the speed value in the command transmission, a converter was incorporated. This converter adjusts the command in accordance with the wheelchair's speed profile configurations, thereby enhancing the precision and reliability of the system.

3.1.3 LiDARs on Wheelchair AMR

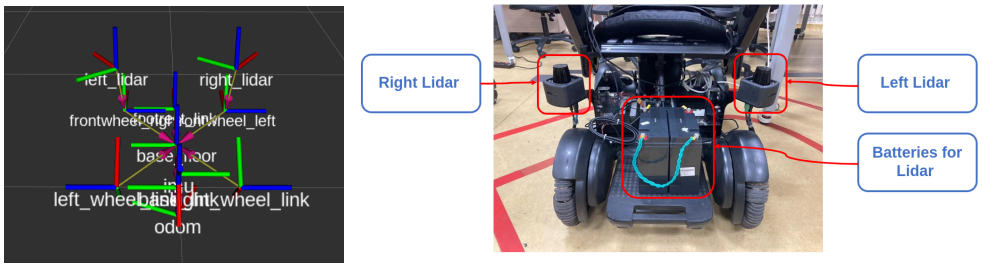
The wheelchair AMR employs two `SICK TIM551` for environmental scanning. Unlike the wheelchair, these LiDARs utilize network cables for data transmission. Both LiDARs are connected to a LAN switch, enabling access via a laptop connection. The official driver package for these LiDARs, sick_tim, can be installed using the apt command in Ubuntu. Following the installation of the LiDAR driver and the launch of configuration files—modified to reflect the actual IP of the LiDARs in the current local network—data was successfully received from both LiDARs, as depicted in Figure 13.



(a) left LiDAR (b) right LiDAR

Figure 13: Point clouds of two LiDARs

Data from both LiDARs are frame-specific, necessitating a transformation from the LiDAR frames to the robot base frame for SLAM and navigation. Given that the LiDARs were absent from the wheelchair's basic model file, they were incorporated into the .xacro model file. Consequently, the `ros_whill` driver facilitates the transformation between the LiDAR frames and the robot base frame, as depicted in Figure 14. Utilizing this transformation information, point cloud data from both LiDARs are converted to the robot base frame and combined to publish a fused LiDAR data via the laser_scan_fusion node.

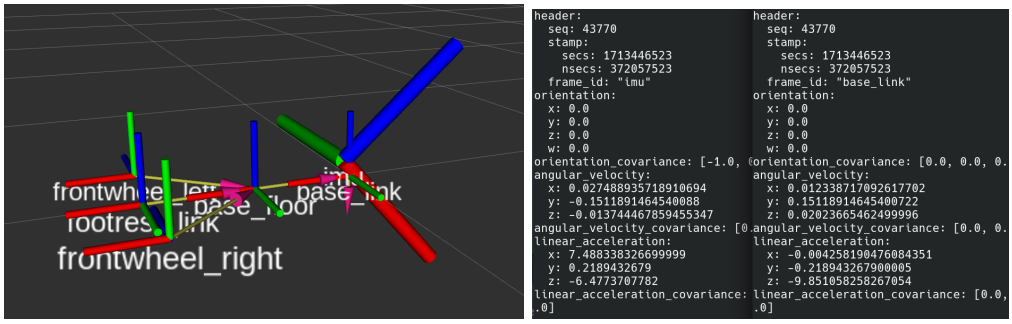


(a) LiDARs TF tree (b) LiDARs on wheelchair

Figure 14: LiDARs layout on wheelchair AMR

3.1.4 Recover IMU Data for Wheelchair

The wheelchair driver's instructions indicate the presence of IMU data. However, the latest `ros_whill` version omits this data, as the official team deemed it unreliable. Examination of the wheelchair's model files revealed that the IMU's installation pose aligns with a tilted support structure, not the wheelchair base (Figure 15(a)). To address this, the IMU data code was reintroduced into the driver program, and a transformer was added to convert the IMU data from the `imu` frame to the `base_link` frame, the wheelchair's rotation center. Figure 15(b) displays the IMU data pre- and post-transformation, with the right side showing the transformed data and a linear acceleration value that aligns with reality.



(a) IMU frame and other frames

```
header:
  seq: 43770
  stamp:
    secs: 1713446523
    nsecs: 372057523
  frame_id: "imu"
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
orientation_covariance: [-1.0, 0.
angular_velocity:
  x: 0.027488935718910694
  y: -0.1511891464540088
  z: -0.013744467859455347
angular_velocity_covariance: [0.
linear_acceleration:
  x: 7.488338326699999
  y: 0.2189432679
  z: -6.4773707782
linear_acceleration_covariance:
.0]
```

```
header:
  seq: 43770
  stamp:
    secs: 1713446523
    nsecs: 372057523
  frame_id: "base_link"
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
orientation_covariance: [0.0, 0.0, 0.
angular_velocity:
  x: 0.012338717092617702
  y: 0.15118914645400722
  z: 0.02023665462499996
angular_velocity_covariance: [0.0, 0.
linear_acceleration:
  x: -0.004258190476084351
  y: -0.218943267900005
  z: -9.851058258267054
linear_acceleration_covariance: [0.0,
.0]
```

(b) rostopic echo imu data

Figure 15: IMU frame and data

3.1.5 Deploy SLAM and navigation algorithm

With the sensor data and control input for the wheelchair AMR prepared, the next step involved deploying the 2D SLAM and navigation algorithms on the robot. In the previous Jetson project, launch files and parameters were available for SLAM (gmapping, cartographer) and navigation (`move_base`). These files were modified and successfully launched on the laptop. During the mapping process, a joystick controller was used to control the wheelchair AMR, with the current map monitored in Rviz. Upon map completion, a navigation goal could be set in Rviz, prompting the robot to move towards the target. The subsequent objective is to implement these operations within the web application.

3.2 Launch Web APP on Wheelchair Robot

As depicted in Figure 16 (a), the ROS program for the wheelchair robot and the web application were launched in the IDE. Subsequently, the wheelchair was connected to the WiFi hotspot of a mobile device. The web application was then accessed via the mobile device's browser using the laptop's IP in the local network.

The web application's visualization and basic control components seamlessly integrated with the wheelchair. For advanced control, a modified version of the navigation ROS launch file for the wheelchair robot was created, enabling the map name to be a configurable item. To facilitate program initiation, a ROS launch file was written that includes not only the drivers for the wheelchair robot but also the remote control toolkit program.

(a) Running program on laptop (b) Control wheelchair robot on phone

Figure 16: Using web app to control the wheelchair robot

3.3 Launch Web APP by Docker

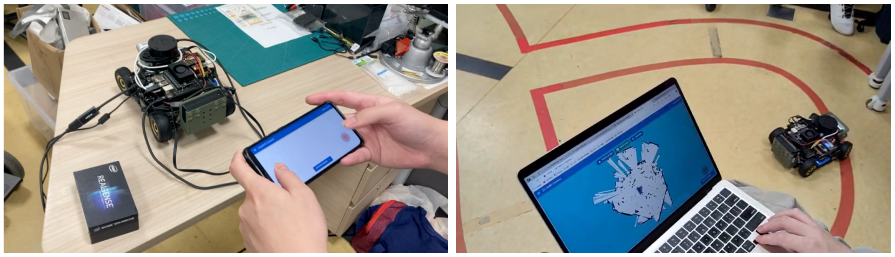
Upon successful testing of the web application on the wheelchair robot, the decision was made to package the application as a Docker image. Docker, a software platform, streamlines the process of constructing, executing, managing, and distributing applications by virtualizing the requisite environment. With the web application packaged as a Docker image, it can be deployed on any device equipped with Docker, thereby enhancing its portability and accessibility.

3.3.1 Docker Pipeline Construction

Initial attempts to package the web application and build a Docker image on an AMD64 architecture laptop were unsuccessful due to incompatibility with ARM64 architecture devices, including Jetson and Raspberry Pi. This challenge was addressed by leveraging Github Actions, a robust feature provided by Github. This tool enables the creation of an auto-packaging pipeline that compiles the web application, packages it as a Docker image for both ARM64 and AMD64 architectures, and uploads it to Dockerhub upon code submission to the main branch. This approach enhances the application's compatibility and distribution efficiency.

3.3.2 Deployment on Other Robot

The Docker image was tested by deploying it on a robot, named `Ackercito`, from another ME5400 group. This robot utilizes a Jetson Orin Nano to host the program and due to its compact size, it cannot accommodate a portable screen. To test the web application, a connection was established with the robot via the same WiFi network. The Docker image was then deployed and accessed via a browser on a device, as depicted in Figure 17. The web application not only facilitated the control part testing of this robot for the group, but also enabled visualization of the robot map on their laptops or phones when the robot was operational on the ground without a screen connection.



(a) Control the robot from phone (b) Control the robot from laptop

Figure 17: Used my web app on ARM64 based AMR with docker

4 Conclusion

In summary, a web application was developed for ROS-based AMRs. This application not only visualizes data but also facilitates robot control. In conjunction with the ROS program developed, it enables users without a ROS background to interact with the robot. The application was tested using a turtlebot3 simulation and a ROS program for a wheelchair robot. The web application is compatible with multiple robot types, ranging from smaller robots like ackercito and turtlebot3 to larger ones like the wheelchair robot. Furthermore, an auto-packaging pipeline was constructed, enabling automatic packaging of the web application for various devices. This comprehensive approach enhances the application's versatility and user-friendliness.

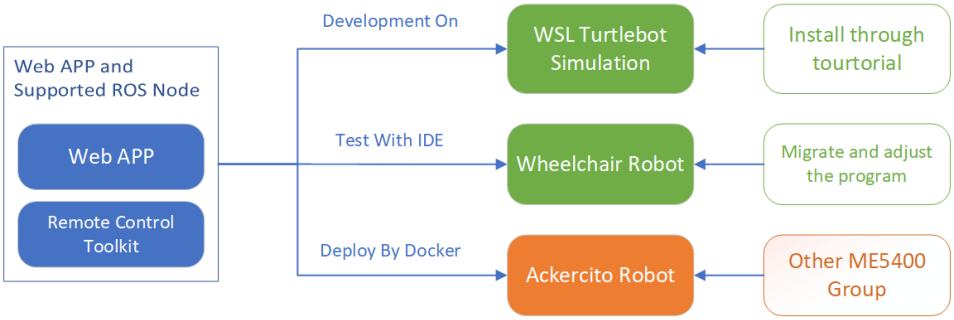


Figure 18: Project component graph

5 Future Works

The current web application is confined to 2D mapping. However, with the integration of a 3D Animation Engine such as three.js, it could potentially visualize 3D data, including point clouds from 3D LiDAR or depth cameras. Given that ROS has ceased new releases and ROS2 has become the primary branch, plans are underway to migrate the remote control toolkit to ROS2. Presently, map files and map pose files are stored on the robot, but future developments could involve storing these files on an online server or database. This would enhance the application's versatility and data accessibility.

6 Appendix

Github repository for web app: ros2d-quasar
Github repository for wheelchair robot: ros whill arc
Github repository for remote control toolkit: AMR Remote Control Toolkit
Demo video for turtlebot in web app: Running ROS2D-Quasar on turtlebot simulation
Demo video for wheelchair in web app: Runing ROS2D-Quasar on wheelchair robot
Dockerhub page for web app: legubiao/ros2d-quasar