Navigation and obstacle avoidance

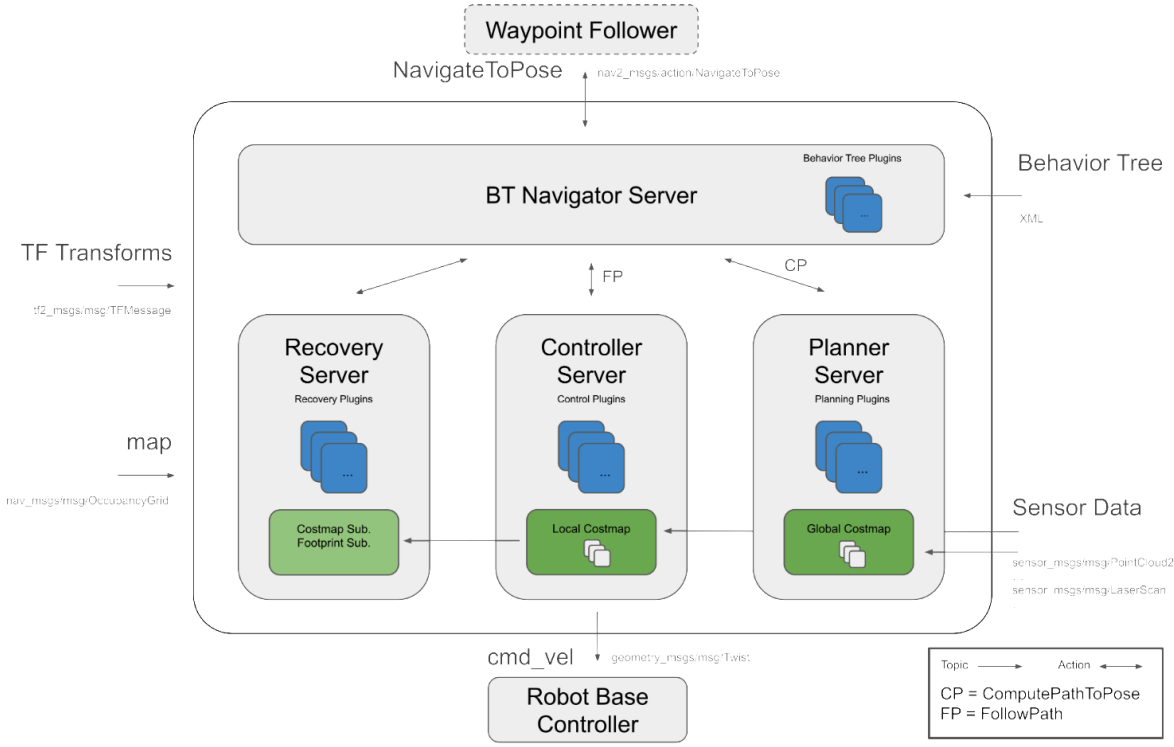
8.1. Program Function Description

After the program on the virtual machine side and the car side is started, rviz will be opened to display the map. By setting the target point, the car can achieve single-point and multi-point navigation.

8.2, Navigation2 Introduction

> Navigation2 Documentation: https://navigation.ros.org/index.html
>
> Navigation2 github: https://github.com/ros-planning/navigation2
>
> Navigation2 Corresponding paper: https://arxiv.org/pdf/2003.00368.pdf
>
> teb_local_planner: https://github.com/rst-tu-dortmund/teb_local_planner/tree/foxy-devel
>
> Navigation2 Plug-ins: https://navigation.ros.org/plugins/index.html#plugins

Navigation2 Overall Architecture Diagram



Navigation2 has the following tools:

● Tools for loading, providing and storing maps (Map Server)

● Tools for locating robots on a map (AMCL)

● Tools for planning paths to move from point A to point B while avoiding obstacles (Nav2 Planner)

● Tools for controlling the robot while following a path (Nav2 Controller)

● Tools for converting sensor data into a cost map representation of the robot's world (Nav2 Costmap 2D)

● Tools for building complex robot behaviors using behavior trees (Nav2 BT Navigator)

● Tools for calculating recovery behaviors in the event of a failure (Nav2 Recoveries)

● Tools for following sequential waypoints (Nav2 Waypoint Follower)

● Tools and watchdogs for managing the server lifecycle (Nav2 Lifecycle Manager)

● Plugins for enabling user-defined algorithms and behaviors (Nav2 Core)

Navigation 2 (Nav 2) is the navigation framework that comes with ROS 2. Its purpose is to enable a mobile robot to move from point A to point B in a safe way. Therefore, Nav 2 can perform behaviors such as dynamic path planning, calculating motor speeds, avoiding obstacles, and recovering structures.

Nav 2 uses Behavior Trees (BT) to call modular servers to complete an action. Actions can be calculated paths, control efforts, recovery, or other navigation-related actions. These actions are independent nodes that communicate with the Behavior Tree (BT) through the action server.

8.3, Program reference path

After SSH connects to the car, the location of the function source code is,

```
/home/sunrise/yahboomcar_ws/src/yahboomcar_nav/launch/map_gmapping_launch.py
```

The source code for virtual machine visualization is located at,

```
/home/yahboom/yahboomcar_ws/src/yahboomcar_rviz/launch/yahboomcar_nav_launch.py
```

8.4, Program startup

8.4.1, Start the virtual machine program

In the virtual machine terminal, enter,

```
ros2 launch yahboomcar_rviz yahboomcar_nav_launch.py
```

At this time, the map will not be displayed in the screen, and the topics of the nodes on the left will not be reported in red, because the navigation node has not been started yet.

8.4.2, Start chassis and radar related nodes

After SSH connects to the car, enter in the terminal,

```
ros2 launch yahboomcar_nav laser_bringup_launch.py
```

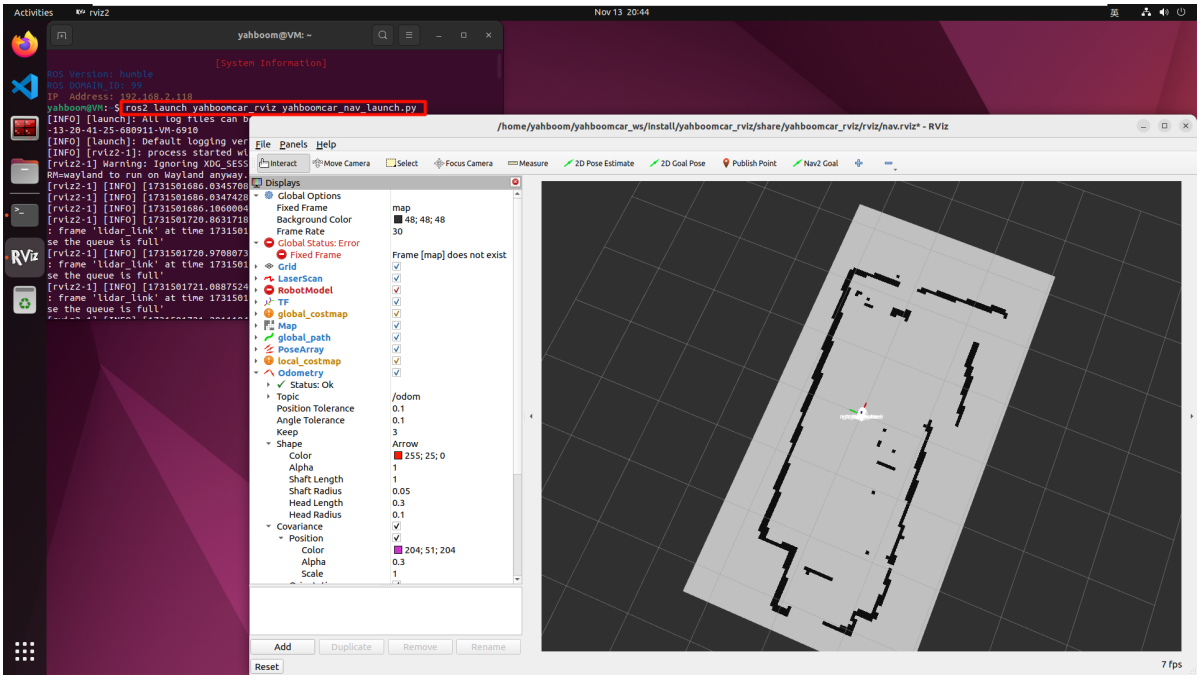
8.4.3, Start navigation node

【Note that you must first start the node that displays the map in step 1, and then start the navigation node. This is because the navigation2 terminal map topic is only published once. If you start the navigation node first and then start rviz display, you may not be able to subscribe to the only published map topic, resulting in no map display】

- There are two navigation algorithms: DWB and TEB
- Navigation can be divided into single-point navigation and multi-point navigation, which will be introduced below.

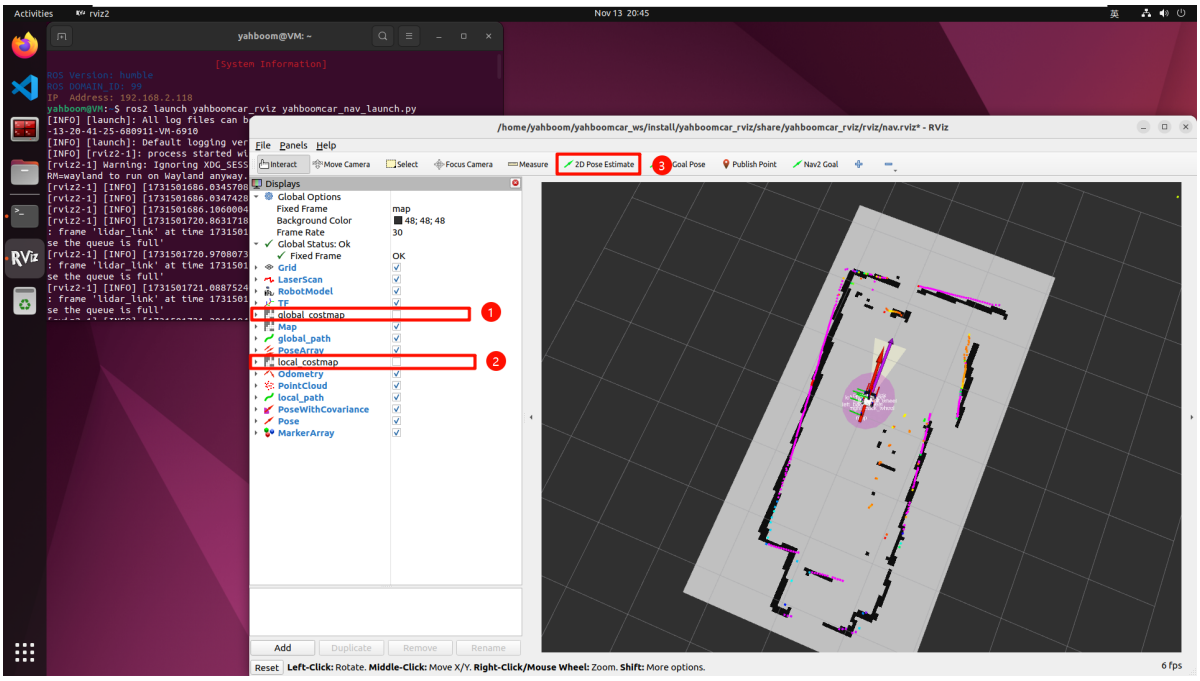
After SSH connects to the car, enter the following in the terminal: **If the map is not the one saved before, you need to recompile the yahboomcar_nav function package (enter the workspace and use: colcon build --packages-select yahboomcar_nav)**

```
#dwb controller
ros2 launch yahboomcar_nav navigation_dwb_launch.py
```



The map appears in rviz.

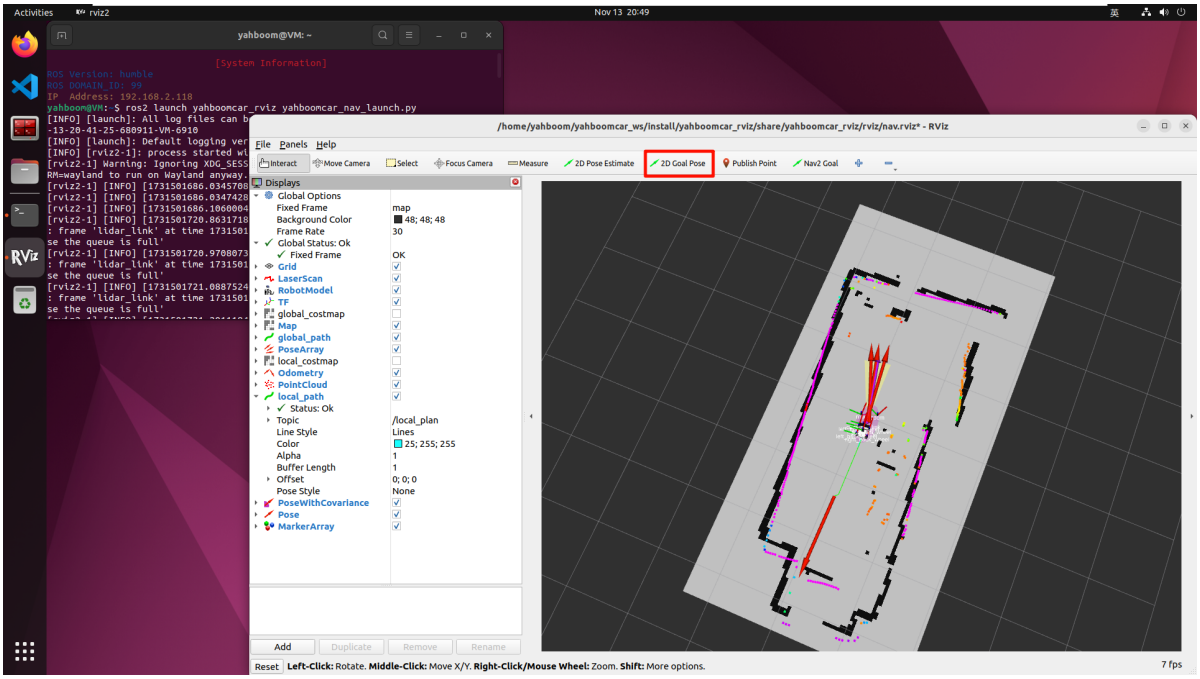
Click [2D Pose Estimate] on rviz, and then compare the car's pose to mark an initial pose on the map: we can uncheck the global cost map and the local cost map to facilitate the calibration of the initial pose



Compare the overlap of the radar scanning point and the obstacle, and set the initial pose of the car multiple times until the radar scanning point and the obstacle roughly overlap;

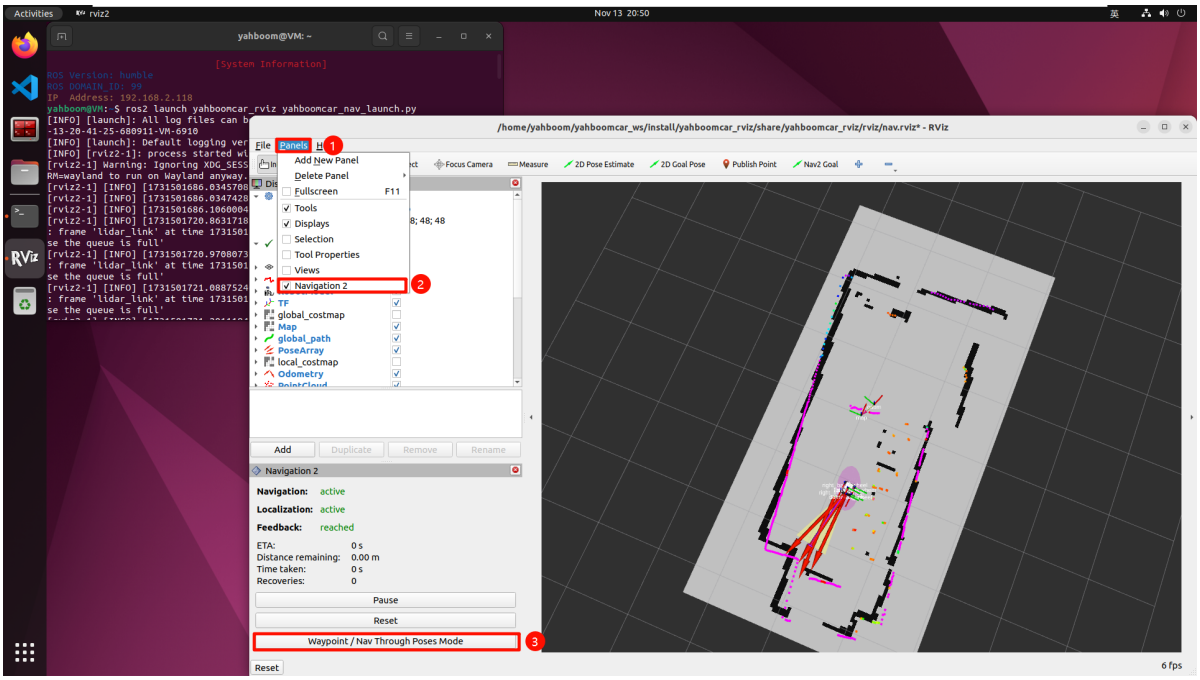
8.4.4, Single-point navigation

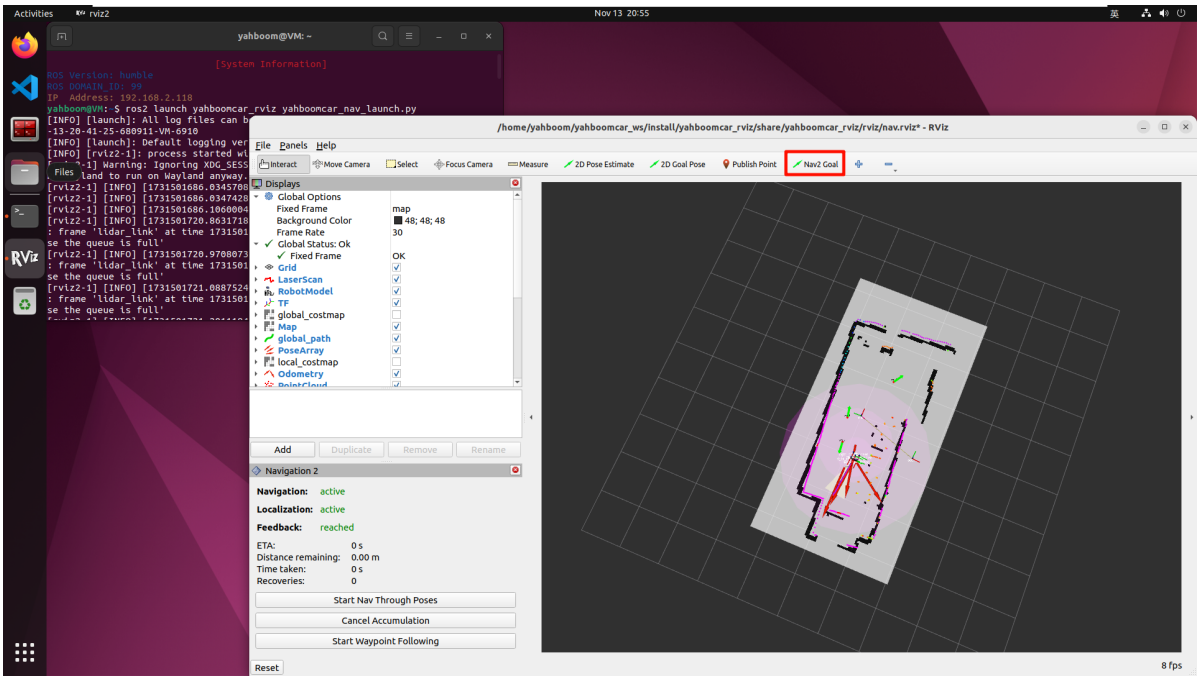
After the initial pose is set, you can click [2D Goal Pose] to set a navigation target point, and the car will start single-point navigation;



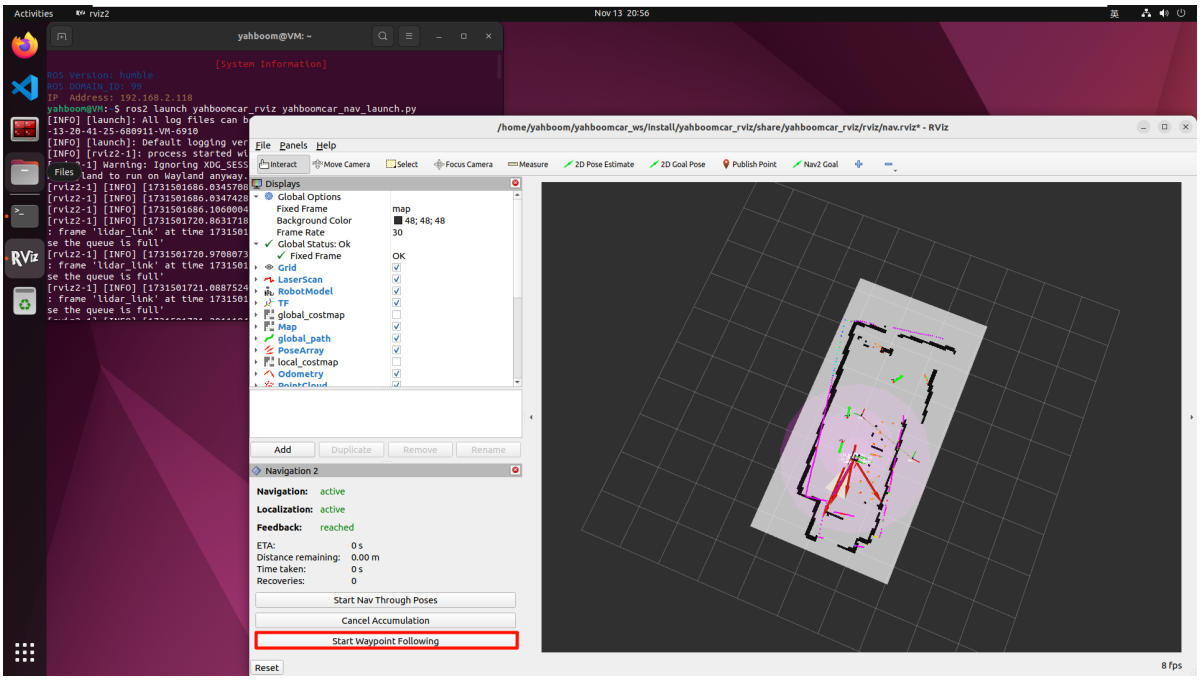
8.4.5. Multi-point navigation

1. After setting the initial position, click [Panels] in the upper left corner, then click [Navigation2] → [Waypoint / Nav Through Pose Mode]: You can click [Nav2 Goal] in the upper right corner to set multiple target points.

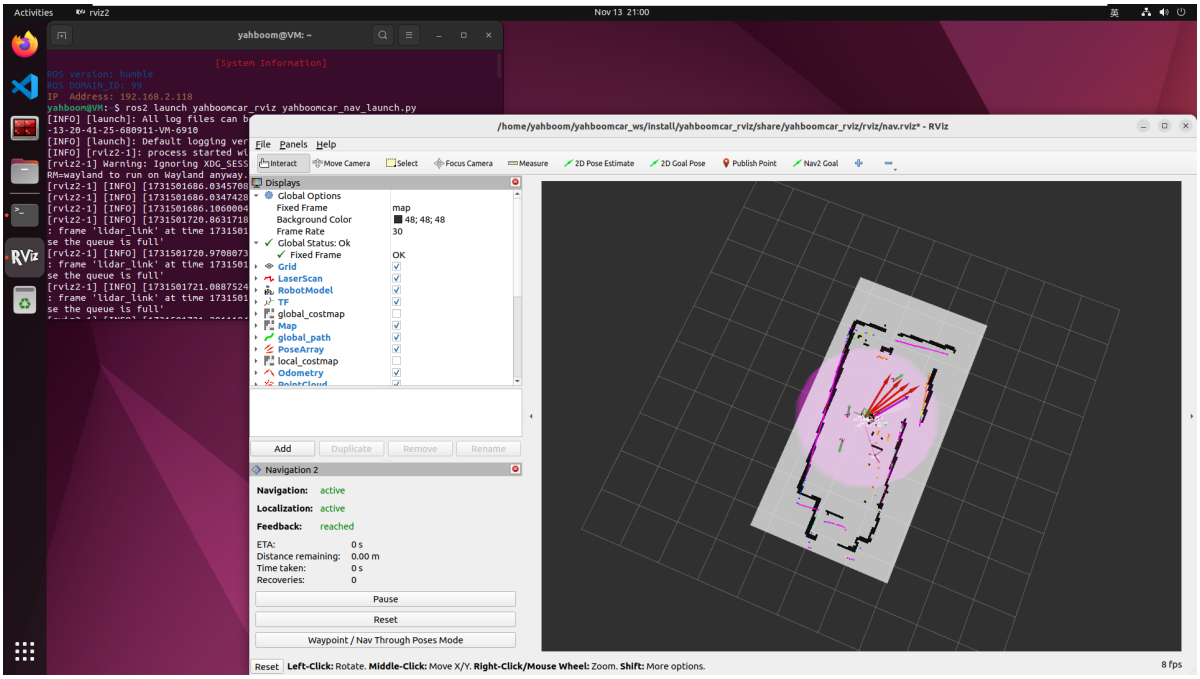




3. After marking multiple target points, click [Start Waypoint Following] to start multi-point navigation;



4. After multi-point navigation is completed, the car will stay at the position of the last target point.



For complete parameter configuration, please refer to:

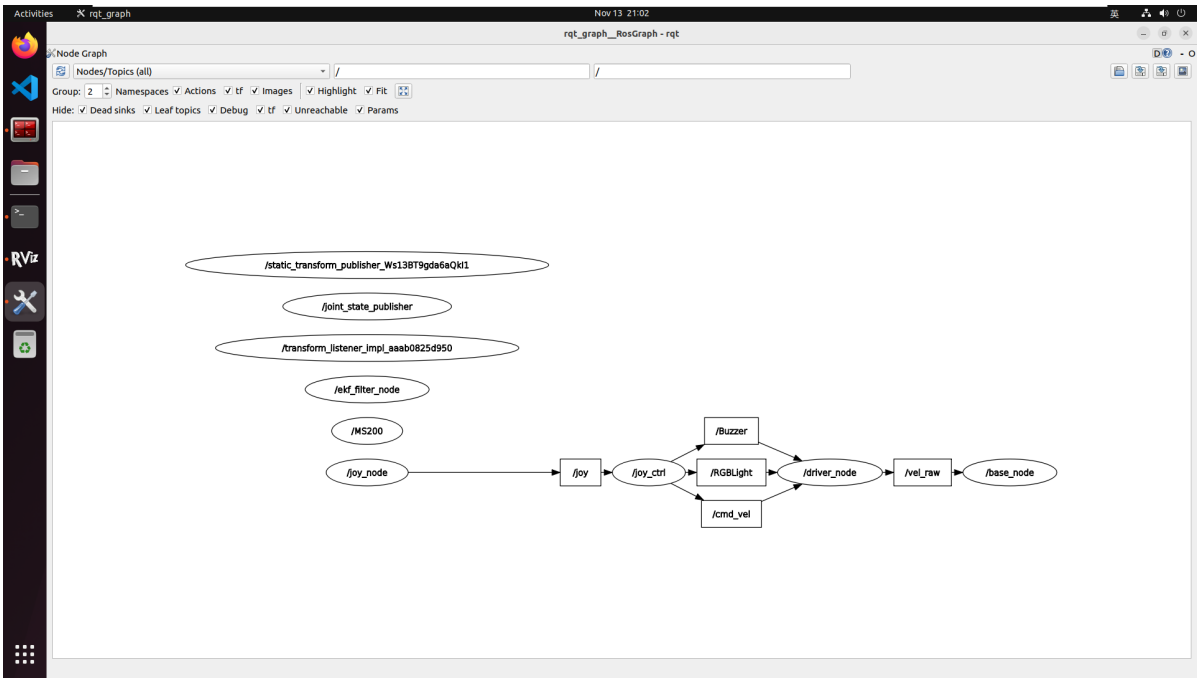
```
/home/yahboom/yahboomcar_ws/src/yahboomcar_nav/params/dwb_nav_params.yaml
```

8.5、Node parsing

8.5.1、Display calculation graph

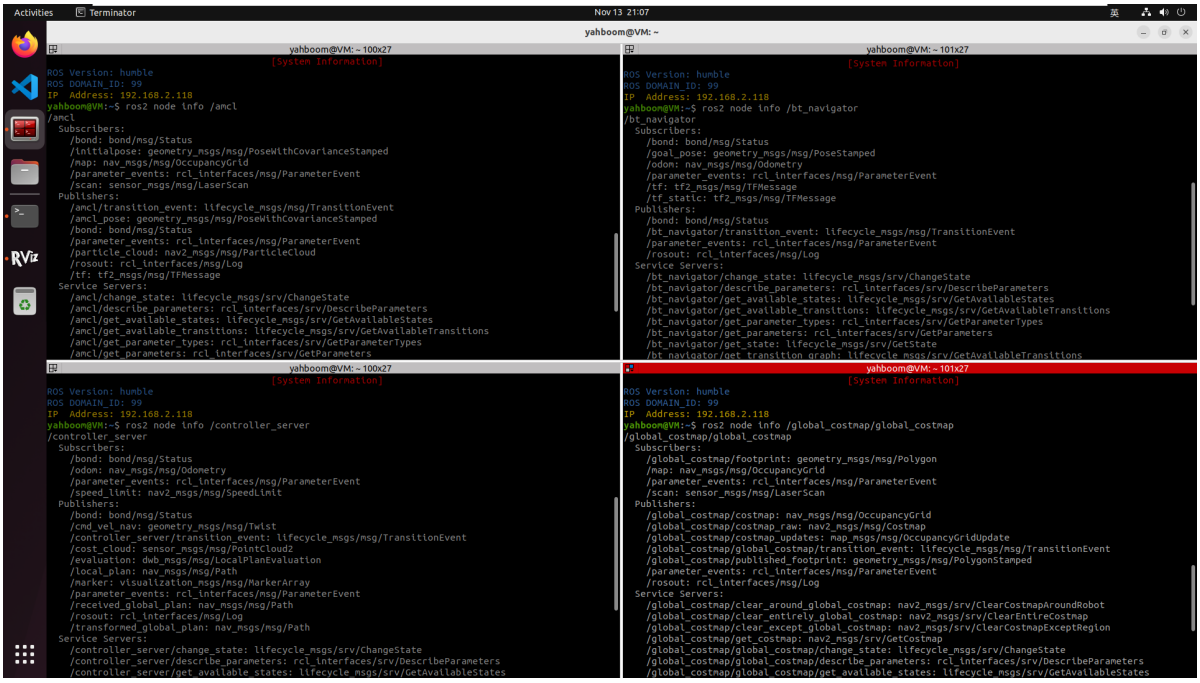
Virtual machine terminal input,

```
ros2 run rqt_graph rqt_graph
```



8.5.2、Navigation details of each node

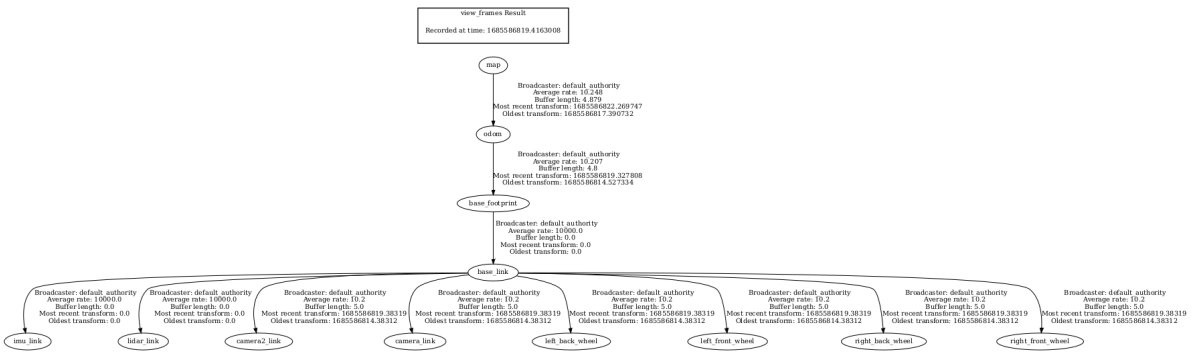
```
ros2 node info /amcl
ros2 node info /bt_navigator
ros2 node info /controller_server
ros2 node info /global_costmap
......
```



8.5.3, TF transformation

Run in the virtual machine terminal,

```
#Save tf tree
ros2 run tf2_tools view_frames
#View tf tree
evince frames.pdf
```



8.6, navigation2 details

- **Comparison of navigation frameworks in ROS 1 and ROS 2**

In Nav 2, move_base is split into multiple components. Unlike a single state machine (ROS1), Nav 2 uses action servers and ROS 2's low-latency, reliable communication to separate ideas. Behavior trees are used to orchestrate these tasks, which allows Nav 2 to have highly configurable navigation behaviors through behavior tree xml files without programming to schedule tasks.

nav2_bt_navigator replaces move_base at the top level, which uses an Action interface to complete a tree-based action model navigation task. More complex state machines are implemented through BT, and recovery behaviors are added as additional Action Servers. These behavior trees are configurable xml.

Path planning, recovery, and controller servers are also action servers, and BT navigators can call these servers for calculations. All 3 servers can host many plugins for many algorithms, each of which can individually call specific behaviors from the navigation behavior tree. The default plugins provided are ported from ROS 1, namely: DWB, NavFn and similar recovery behaviors like rotation and clearing costmaps, in addition a new recovery mechanism that waits for a fixed time is added. These servers are called from the BT Navigator via their action servers to calculate results or complete tasks. The state is maintained by the BT Navigator behavior tree. All these changes make it possible to replace these nodes at startup/runtime by any algorithm that implements the same interface.

Ported packages:

- amcl: ported to nav2_amcl
- map_server: ported to nav2_map_server
- nav2_planner: replaces global_planner, manages `N` planner plugins
- nav2_controlle: replaces local_planner, manages `N` controller plugins
- Navfn: ported to nav2_navfn_planner
- DWB: replaces DWA and ported to nav2_dwb_controller metapackage in ROS 2
- nav_core: ported to nav2_core, and updated the interface
- costmap_2d: ported to nav2_costmap_2d

New packages:

- Nav2_bt_navigator: replaces move_base state machine
- nav2_lifecycle_manager: handles the life cycle of server programs
- nav2_waypoint_follower: can perform complex tasks with many waypoints
- nav2_system_tests: A set of integration tests for CI and basic tutorials in simulation
- nav2_rviz_plugins: An rviz plugin to control the Navigation2 server, commands, cancel and navigation
- nav2_experimental: Experimental (and incomplete) work on a deep reinforcement learning controller
- navigation2_behavior_trees: A wrapper around the behavior tree library for calling the ROS action server

8.6.1, amcl

amcl stands for adaptive Monte Carlo localization, a probabilistic localization system for two-dimensional mobile robots. It is actually an upgraded version of the Monte Carlo localization method, using an adaptive KLD method to update particles and a particle filter to track the robot's posture based on a known map. As currently implemented, this node is only applicable to laser scanning and laser maps. It can be extended to process other sensor data. amcl receives laser-based maps, laser scanning, and transformation information, and outputs a posture estimate. At startup, amcl initializes its particle filter based on the provided parameters. Note that due to the

default settings, if no parameters are set, the initial filter state will be a medium-sized particle cloud centered at (0,0,0).

Monte Carlo

- Monte Carlo method, also known as statistical simulation method or statistical test method, is an idea or method. It is a numerical simulation method that takes probabilistic phenomena as the research object. It is a calculation method that obtains statistical values according to the sampling survey method to infer unknown characteristic quantities.
- For example: There is an irregular shape inside a rectangle. How to calculate the area of the irregular shape? It is not easy to calculate. But we can approximate it. Take a bunch of beans and spread them evenly on the rectangle, then count the number of beans in the irregular shape and the number of beans in the remaining places. The area of the rectangle is known, so the area of the irregular shape is estimated. Take the robot positioning as an example. It is possible that it is in any position in the map. In this case, how do we express the confidence of a position? We also use particles. Where there are more particles, it means that the robot is more likely to be there.

The biggest advantage of the Monte Carlo method

- The error of the method is independent of the dimension of the problem.
- It can be solved directly for problems with statistical properties.
- There is no need to discretize continuous problems

Disadvantages of the Monte Carlo method

- Deterministic problems need to be converted into random problems.
- The error is a probabilistic error.
- Usually a large number of calculation steps N are required.

Particle filtering

- The number of particles represents the probability of something. Through a certain evaluation method (evaluating the probability of this thing), the distribution of particles is changed. For example, in robot positioning, for a particle A, I think it is very likely that this particle is at this coordinate (for example, this coordinate belongs to the "this thing" mentioned before), so I give it a high score. Next time when all the particles are rearranged, arrange more near this position. In this way, after a few more rounds, the particles will be concentrated in the position with high probability.

Adaptive Monte Carlo

- Solve the robot kidnapping problem. When it finds that the average score of particles suddenly decreases (meaning that the correct particles are abandoned in a certain iteration), it will re-scatter some particles globally.
- Solve the problem of fixed number of particles, because sometimes when the robot positioning is almost obtained, for example, these particles are concentrated together, it is unnecessary to maintain so many particles, and the number of particles can be reduced at this time.

8.6.2, costmaps and layers

The current environment is expressed as a cost map. The cost map is a regular 2D cell grid containing cells from unknown, idle, occupied or inflated costs. The cost map is then sought to calculate the global plan or sampled to calculate the local control work.
Various costmap layers are implemented as pluginlib plugins to buffer information into the costmap. This includes information from LIDAR, RADAR, sonar, depth sensors, image sensors, etc. It is best to process the sensor data before it is fed into the layer-local map, but this is up to the developer.
Costmap layers can be created using cameras or depth sensors to detect and track obstacles in the scene to avoid collisions. In addition, layers can be created to change the base costmap based on some rules or heuristics. Finally, they can be used to buffer real-time data into a 2D or 3D world for binary marking of obstacles.

8.6.3, planner_server

The task of the planner is to calculate a path according to a specific objective function. Depending on the chosen nomenclature and algorithm, a path is also called a route. Two typical examples here are to calculate a path plan to a target location (e.g., from the current location to a target pose) or complete coverage (e.g., a path plan covering all free space). The planner has access to the global environment representation and cached data of the sensors. A planner can have the following capabilities:

- Compute the shortest path
- Compute a fully covered path
- Compute a path along a sparse or predefined route

The general task of a planner in Nav2 is to compute a valid and possibly optimal path from the current position to the target pose.

Algorithm plugins in the planning server use the environmental information captured by different sensors to search for a path for the robot. Some of these algorithms search the environment grid to obtain a path, others expand the possible states of the robot based on the feasibility of the path.

As mentioned earlier, the planning server uses plugins that work on a grid space, such as NavFn Planner , Smac Planner 2D and Theta Star Planner . NavFn planner is a navigation function planner using Dijkstra or A algorithm. Smac 2D planner implements the 2D A algorithm with 4 or 8 connected neighbors and smoothers and multi-resolution features. Theta Star planner (smooth arbitrary-angle path planning in a continuous environment, Theta is a variant of A that propagates information along the edges of the graph but does not restrict the path to the edges of the graph (finding "arbitrary-angle" paths)) uses a line-of-sight algorithm to create non-discrete directed path segments. Commonly used planners are as follows:

8.6.4, controller_server

Controllers, also called local planners in ROS 1, follow a globally computed path or complete a local task. Controllers have access to a local representation of the environment and try to compute a feasible path to follow from a baseline path. Many controllers project the robot forward into space and compute a local feasible path at each update iteration. Controllers have the following capabilities:

- Follow a path

- Use a probe to dock with a charging station in the mileage coordinate system
- Board an elevator
- Interface with a tool

In Nav2, the general task of a controller is to compute an effective control effort to follow a globally planned path. However, there are many controller classes and local planner classes. The goal of the Nav2 project is to make all controller algorithms available as plugins in this server for general research and industrial tasks.

Common controller plug-ins include DWA Controller and TEB Controller

1. DWA Controller
The idea of the DWA algorithm is to sample different possible speeds, simulate different trajectories, evaluate the distance to the target point, obstacle distance and time, select the trajectory with the best score, and issue speed instructions to control the car to move forward. The speed sampling simulated by the DWA algorithm is not randomly selected, but there is a range. The speed is limited by the maximum and minimum speeds of the car, and the acceleration is affected by the performance of the car motor. In addition, in order to ensure safety, the distance of the obstacle will also affect the value of the speed.

The differential car and the McLennan car have a short movement distance in adjacent moments. The movement trajectory between two adjacent points can be regarded as a straight line, and the movement trajectory of other models can be regarded as an arc.

Three evaluation functions:

- heading(v, w): azimuth evaluation function, the angle difference between the car and the target, the smaller the angle difference, the higher the score
- dist(v, w): the distance between the car and the nearest obstacle, the farther the distance, the higher the score
- velocity(v, w): the speed corresponding to the trajectory, the greater the speed, the higher the score

Physical meaning: make the car head towards the target point, avoid obstacles, and drive quickly

2, TEB Controller
The idea of the TEB algorithm is to regard the path connecting the starting point and the end point as a rubber band that can be deformed, and then use external constraints as external forces to deform the path.

- Follow the path + obstacle avoidance: The constraints have two main goals, following the consistent global path planning and obstacle avoidance. Both objective functions are very similar. Following the path applies external force to pull the local path toward the global path, and obstacle avoidance constraints apply force to keep the local path away from obstacles
- Speed/acceleration constraints: speed and acceleration should be within a certain range
- Kinematic constraints: a smooth trajectory composed of several arc segments, the control quantity is only linear velocity and angular velocity, the Ackerman structure has a minimum turning radius, and the McLennan/omnidirectional/differential speed are all 0
- Fastest path constraint: the objective function enables the robot to obtain the fastest path, and the pose points on the path are evenly separated in time, rather than the shortest path in traditional space

TEB can be expressed as a multi-objective optimization problem, most of which are local and only related to a small number of parameters because they only depend on a few continuous robot states. After determining the positions of N control points, the open source framework g2o (General Graph Optimization) is used to optimize the path. In the case of the car, all pose points, time intervals, and obstacles are described as points, and the constraints are described as edges. The minimum distance from the obstacle connects the obstacle and the pose point, and the velocity constraint connects two adjacent pose points and their time difference. In general, the local trajectory generated by TEB consists of a series of discrete poses with time information. The goal of the g2o algorithm optimization is that the trajectory composed of these discrete poses can achieve the goals of shortest time, shortest distance, and far away from obstacles, while limiting the speed and acceleration so that the trajectory satisfies the robot dynamics.

8.6.5, recoveries_server

Recovery behaviors are the backbone of fault-tolerant systems. The goal of the recoveries is to handle unknown or faulty conditions of the system and handle them autonomously. Examples include failures in the perception system that cause the environment representation to be full of false obstacles. This will trigger the cost map clearing recovery to allow the robot to move.
Another example is that the robot is stuck due to dynamic obstacles or poor control. If allowed, backing up or rotating in place will allow the robot to move from the stuck position to a free space where it can successfully navigate.
Finally, in the case of complete failure, recovery can be implemented to attract the attention of the operator for help. This can be done via email, SMS, Slack, Matrix, etc.

8.6.6, waypoint following

Waypoint following is one of the basic functions of a navigation system. It tells the system how to use the navigation program to reach multiple destinations. The nav2_waypoint_follower package contains a waypoint follower with a plugin interface for task-specific executors. This is useful if you need to have the robot go to a given pose and complete a specific task like taking a picture, picking up a box, or waiting for user input. This is a nice demo application to show how to use Nav2 in a sample application.

However, this package can be used for more than just sample applications. There are two schools of thought about robot fleet managers/schedulers: **dumb robots + smart centralized scheduler**; **smart robots + dumb centralized scheduler**.

In the first school of thought, the nav2_waypoint_follower package is enough to create a production-grade robotics solution. Since the autonomous system/scheduler takes into account the robot's pose, battery level, current mission, etc. when assigning tasks, the application on the robot only needs to worry about the task at hand and not the other complexities of completing the task required by the system. In this case, the request sent to the waypoint follower should be considered as 1 unit of work (e.g. 1 pick in the warehouse, 1 security patrol cycle, 1 aisle, etc.) to perform the task and then return to the scheduler for the next task or request to recharge. In this school of thought, the waypoint following application is just a step above the navigation software stack and below the system's autonomous application.

In the second school of thought, the nav2_waypoint_follower package is a nice example application/proof of concept, but really requires the waypoint tracking/autonomy system on the robot to take on more tasks to make a robust solution. In this case, the nav2_behavior_tree package should be used to create a custom application-level behavior tree to complete the task using navigation. This can contain subtrees, such as checking the charge status in a task to return

to the dock, or handle more than 1 unit of work in a more complex task. Soon there will be a nav2_bt_waypoint_follower (name to be adjusted) that will allow users to create this application more easily. In this school of thought, waypoint following applications are more closely tied to autonomous systems, or in many cases, the waypoint following application is the autonomous system itself.

It is not easy to say which of these two schools of thought is better than the other. Which one is better depends a lot on what task the robot is completing, what type of environment it is in, and what cloud resources are available. Usually, for a given business case, the difference is very clear.

8.6.7, bt_navigator

The BT Navigator (Behavior Tree Navigator) module implements the NavigateToPose task interface. It is a behavior tree-based navigation implementation designed to allow flexibility in navigation tasks and provide a way to easily specify complex robot behaviors (including recovery).

Planer Server, Controller Server, and Recovery Server implement their own functions, namely global path planning, local path planning, and recovery operations. But their functions are independent and do not interfere with each other. To achieve a complete navigation function, the cooperation of each module is required. BT Navigator Server is the guy who assembles the Lego bricks.