

7. Navigation and obstacle avoidance

7. Navigation and obstacle avoidance

7.1. Introduction

7.2 Usage

7.2.1 Pre-use Configuration

7.2.2 Starting chassis and lidar related nodes

7.2.3. Starting rviz to display maps

7.2.4 Starting a Navigation Node

7.2.5 Single point navigation

7.2.6 Multi-point navigation

7.3 Node resolution

7.3.1. Displaying computational graphs

7.3.2 Navigating the details of each node

7.3.3 TF transformations

7.4. navigation2 details

7.4.1 amcl

7.4.2 costmaps and layers

7.4.3 planner_server

7.4.4 controller_server

7.4.5 recoveries_server

7.4.6 waypoint following

7.4.7 bt_navigator

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

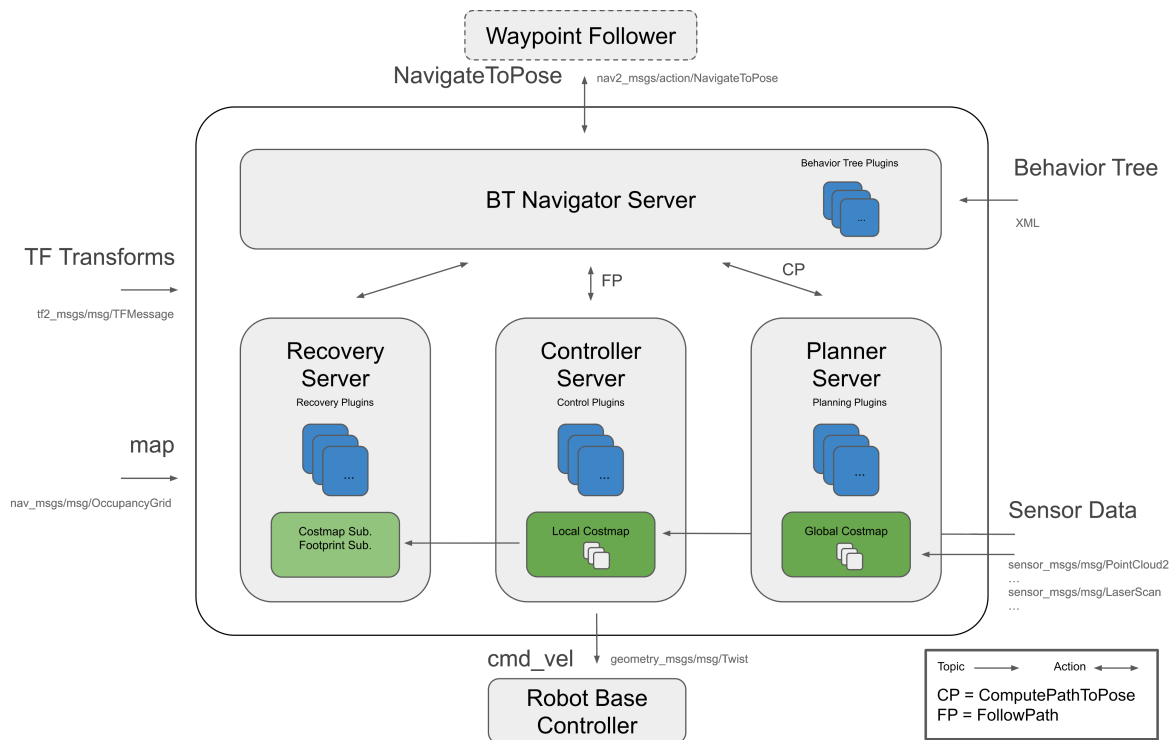
Plugins available for Navigation2: <https://navigation.ros.org/plugins/index.html#plugins>

The operating environment and hardware and software reference configuration are as follows:

- Reference model: ROSMASTER R2
- Robot hardware configuration: Arm series main control, Silan A1 LiDAR, AstraPro Plus depth camera.
- Robot system: Ubuntu (version not required) + docker (version 20.10.21 and above)
- PC virtual machine: Ubuntu (20.04) + ROS2 (Foxy)
- Usage scenario: use on a relatively clean 2D plane

7.1. Introduction

Navigation2 overall architecture diagram



Navigation2 has the following tools:

- Tools for loading, serving, and storing maps (Map Server)
- A tool to locate the robot on a map (AMCL)
- A path planning tool to move from point A to point B avoiding obstacles (Nav2 Planner)
- Tool for controlling the robot during following a path (Nav2 Controller)
- A tool for converting sensor data into a costmap representation of the robot world (Nav2 Costmap 2D)
- Tools for building complex robot behaviors using behavior trees (Nav2 Behavior Tree and BT Navigator)
- A tool to compute recovery behaviors in case of failure (Nav2 Recoveries)
- Tools for following sequential waypoints (Nav2 Waypoint Follower)
- Tools and watchdogs for managing server lifecycles (Nav2 Lifecycle Manager)
- Plug-ins to enable user-defined algorithms and behaviors (Nav2 Core)

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

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

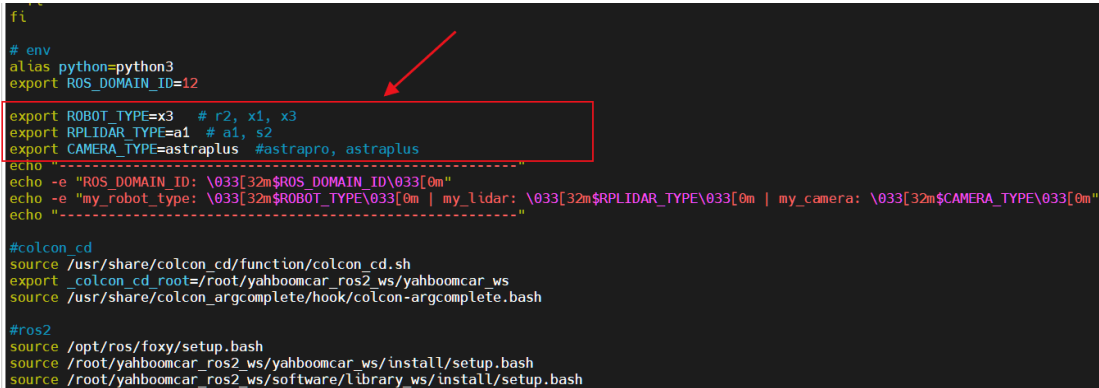
7.2 Usage

7.2.1 Pre-use Configuration

Note: Since ROSMASTER series robots are divided into multiple robots as well as multiple devices, the factory system has been configured with routines for multiple devices, but since it is not possible to automatically recognize the products, you need to manually set the machine type and lidar model.

After entering the container: according to the model of the trolley, the type of lidar and the type of camera make the following changes:

```
root@ubuntu:/# cd
root@ubuntu:~# vim .bashrc
```



```
fl
# env
alias python=python3
export ROS_DOMAIN_ID=12
export ROBOT_TYPE=x3 # r2, x1, x3
export RPLIDAR_TYPE=a1 # a1, s2
export CAMERA_TYPE=astraplus #astrapro, astraplus
echo "-----"
echo -e "ROS_DOMAIN_ID: \033[32m${ROS_DOMAIN_ID}\033[0m"
echo -e "my_robot_type: \033[32m${ROBOT_TYPE}\033[0m | my_lidar: \033[32m${RPLIDAR_TYPE}\033[0m | my_camera: \033[32m${CAMERA_TYPE}\033[0m"
echo "-----"

#colcon cd
source /usr/share/colcon_cd/function/colcon_cd.sh
export _colcon_cd_root=/root/yahboomcar_ros2_ws/yahboomcar_ws
source /usr/share/colcon_argcomplete/hook/colcon_argcomplete.bash

#ros2
source /opt/ros/foxy/setup.bash
source /root/yahboomcar_ros2_ws/yahboomcar_ws/install/setup.bash
source /root/yahboomcar_ros2_ws/software/library_ws/install/setup.bash
```

When the changes are complete, save and exit vim, then execute:

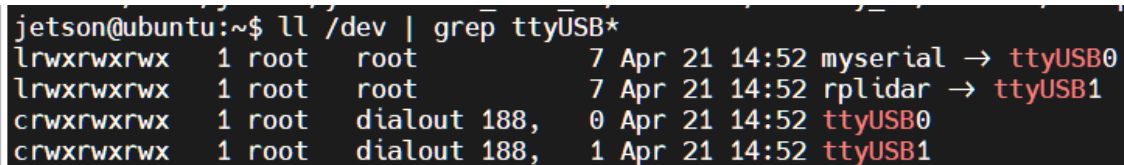
```
root@ubuntu:~# source .bashrc
-----
ROS_DOMAIN_ID: 12
my_robot_type: r2 | my_lidar: a1 | my_camera: astraplus
-----
root@ubuntu:~#
```

You can see the current model of the modified cart, the type of lidar and the type of camera.

7.2.2 Starting chassis and lidar related nodes

First of all, you need to do the port binding operation in the host [i.e. the jetson of the car] [see the port binding tutorial chapter], and the lidar and serial port devices are mainly used here;

Then check whether the lidar and serial port devices are in the port binding state: in the host [that is, on the jetson of the car], refer to the following commands to view the successful binding is the following state:



```
jetson@ubuntu:~$ ll /dev | grep ttyUSB*
lrwxrwxrwx 1 root root 7 Apr 21 14:52 myserial -> ttyUSB0
lrwxrwxrwx 1 root root 7 Apr 21 14:52 rplidar -> ttyUSB1
crwxrwxrwx 1 root dialout 188, 0 Apr 21 14:52 ttyUSB0
crwxrwxrwx 1 root dialout 188, 1 Apr 21 14:52 ttyUSB1
jetson@ubuntu:~$
```

If it shows that the lidar or serial device is not bound, you can plug and unplug the USB cable to check again.

Enter the docker container (for steps, see [docker course chapter ----- 5. Entering the docker container of the robot]), and execute it in a sub-terminal:

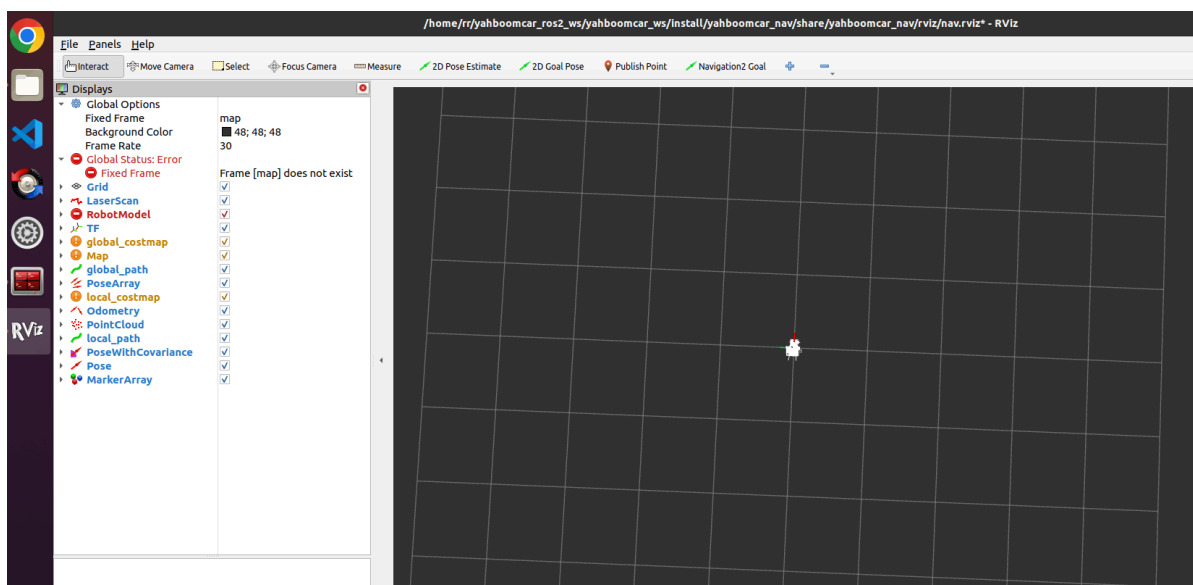
```
ros2 launch yahboomcar_nav laser_bringup_launch.py
```

7.2.3. Starting rviz to display maps

Ubuntu virtual machine and docker container to configure multiple machine communication, this step is recommended to start in the virtual machine: to keep time synchronization and reduce resource consumption, because if you use vnc is very dependent on the network, may lead to navigation failure.

[Note that here you must start the node that displays the map first, and then start the navigation node in step 3, this is due to navigation2 terminal map topic is only published once, how to start the navigation node first, and then start the rviz display, there is a possibility that you will not be able to subscribe to the map topic that was published that one and only time, which will lead to not displaying the map]

```
ros2 launch yahboomcar_nav display_nav_launch.py
```



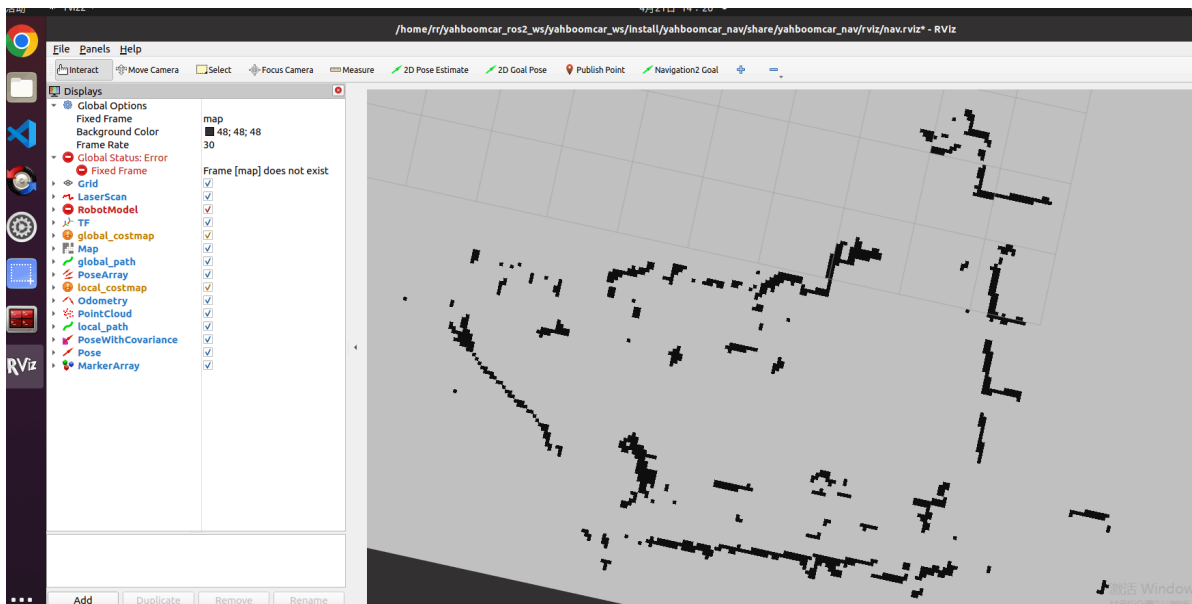
At this time, the map will not be displayed on the screen, and the node topics on the left will be red, so don't worry about it, because you haven't started the navigation node yet.

7.2.4 Starting a Navigation Node

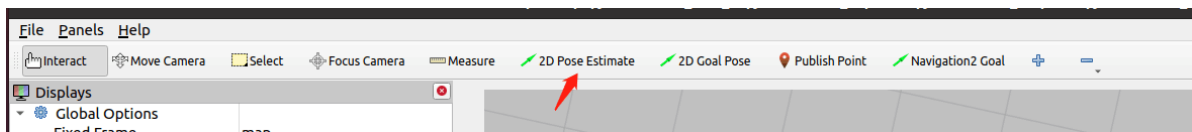
- There are two navigation algorithms: DWA and TEB.
- Navigation can be categorized into single-point navigation and multi-point navigation, as described below.

1. Start the navigation node, enter the docker container, and execute it in a sub-terminal:

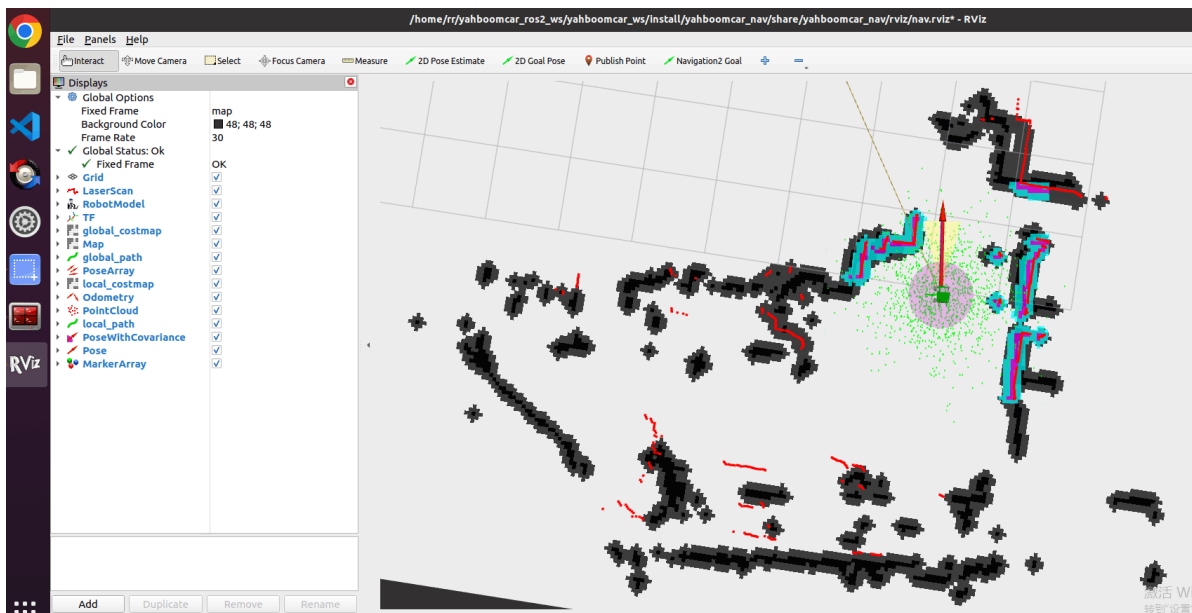
```
ros2 launch yahboomcar_nav navigation_dwa_launch.py # dwa导航 # dwa navigation
or
ros2 launch yahboomcar_nav navigation_teb_launch.py # teb导航 # teb navigation
```



2. Click [2D Pose Estimate] on rviz, and then compare the position of the cart to mark an initial position for the cart on the map;



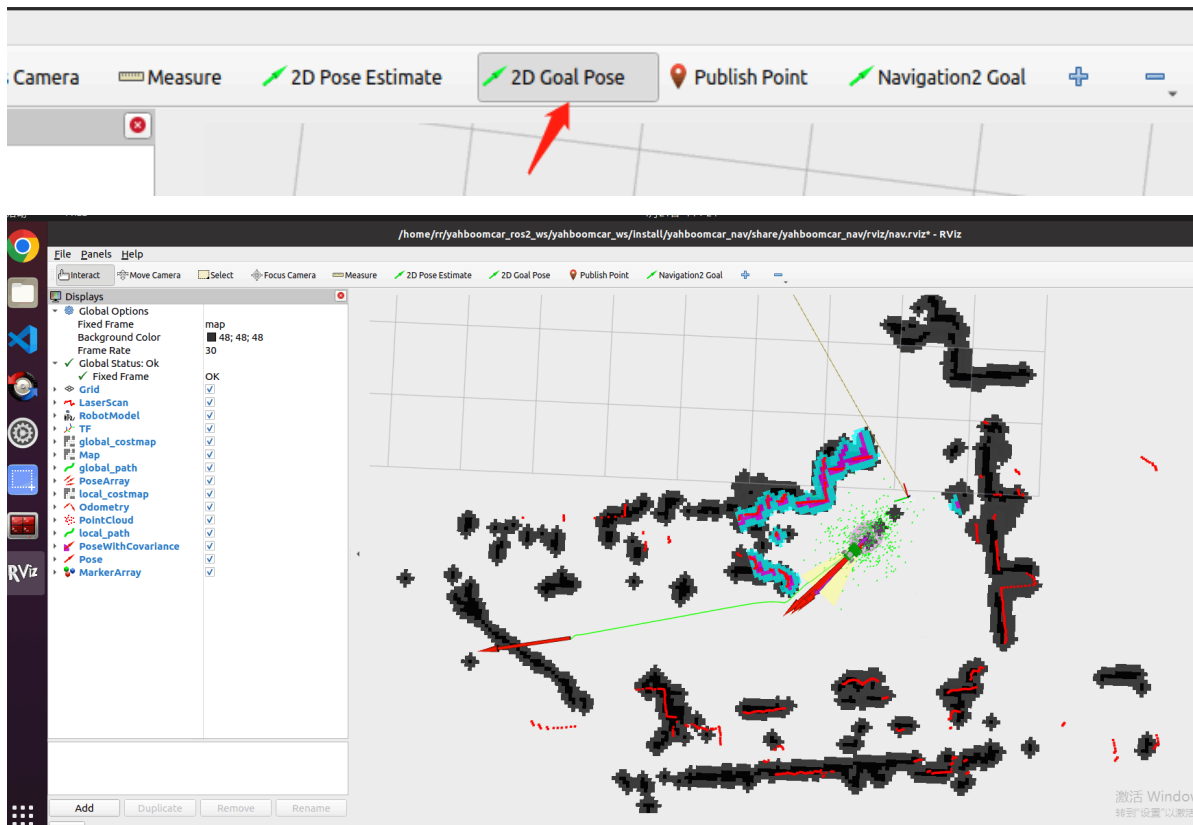
The display after labeling is as follows:



3. Compare the overlap between the lidar scanning point and the obstacle, you can set the initial position for the cart several times until the lidar scanning point and the obstacle roughly overlap;

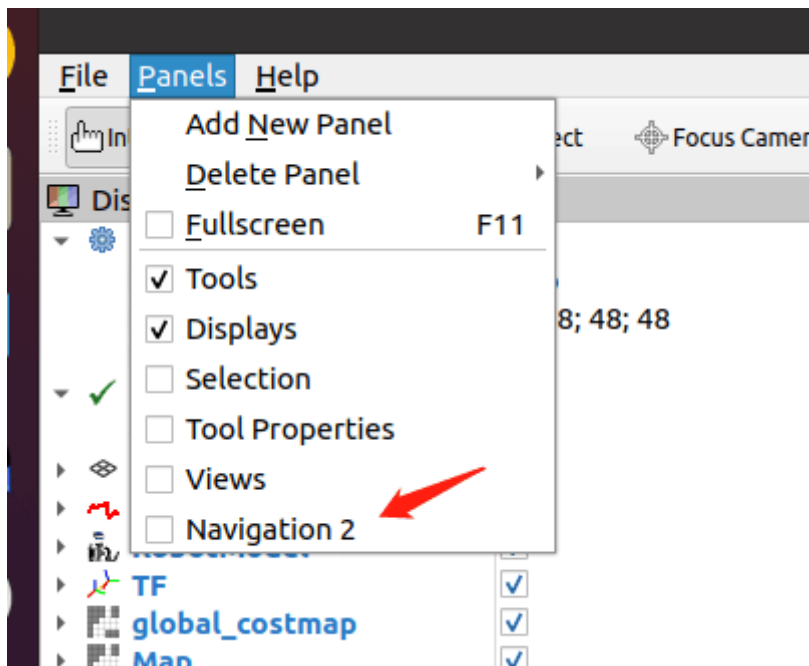
7.2.5 Single point navigation

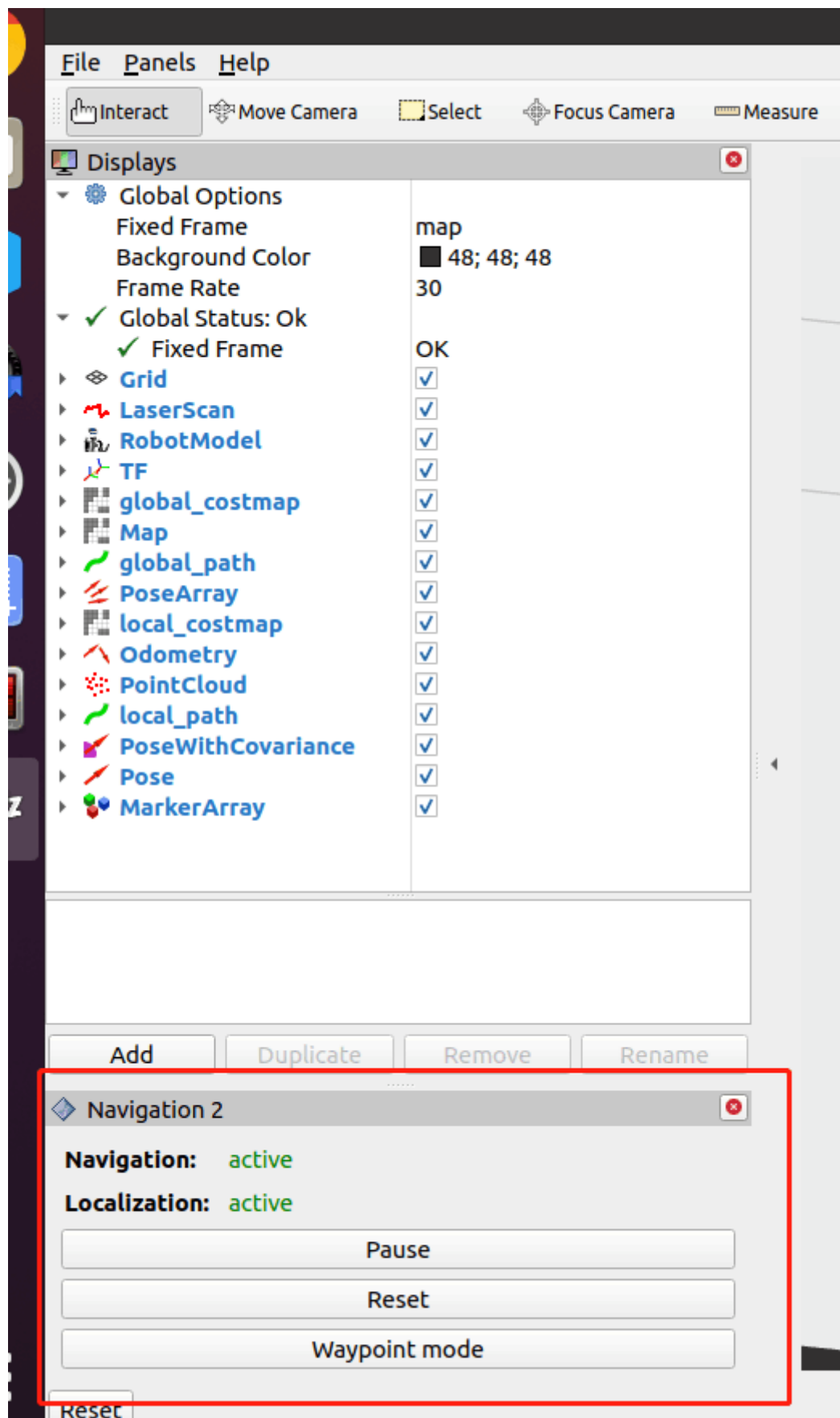
After the initial position is set, you can click [2D Goal Pose] to set a navigation target point, and the cart will start single-point navigation;



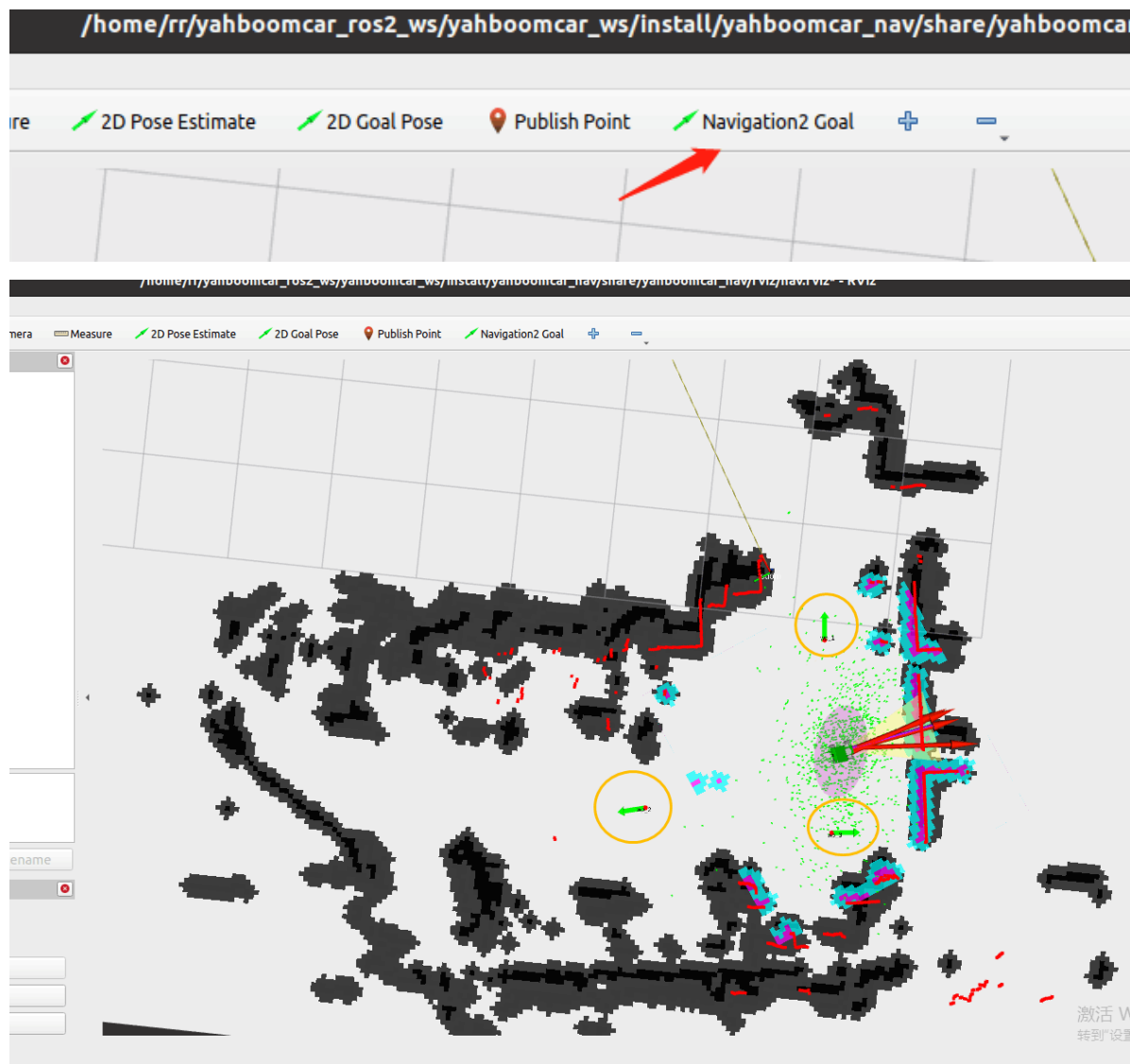
7.2.6 Multi-point navigation

1. After the initial position is set, you can click [Panels] in the upper left corner of the rviz --- check [Navigation 2], then the [Navigation 2] panel will be displayed.

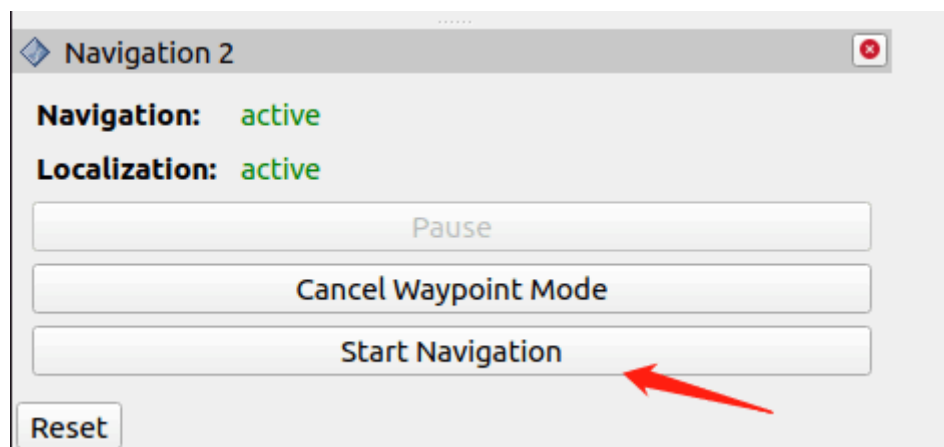




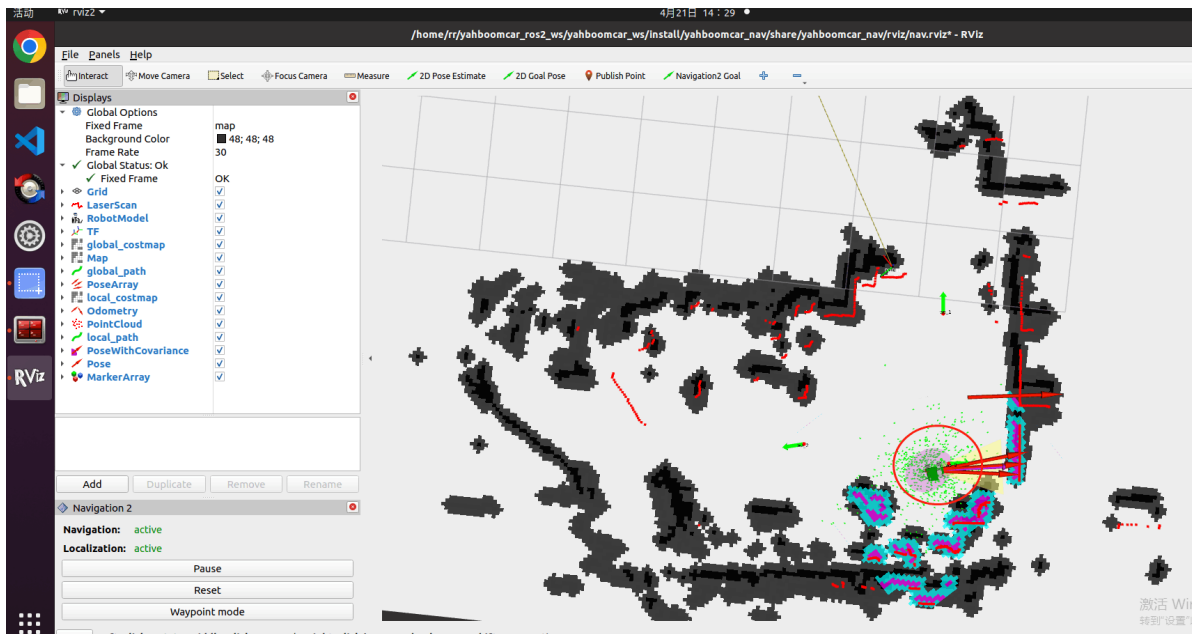
2. Click [Waypoint mode] in the above picture, then click [Navigation2 Goal] on the rviz, you can mark a target point in the map, click [Navigation2 Goal] again, you can mark the second target point in the map, and the cycle goes on, so you can mark more than one target point at a time;



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



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



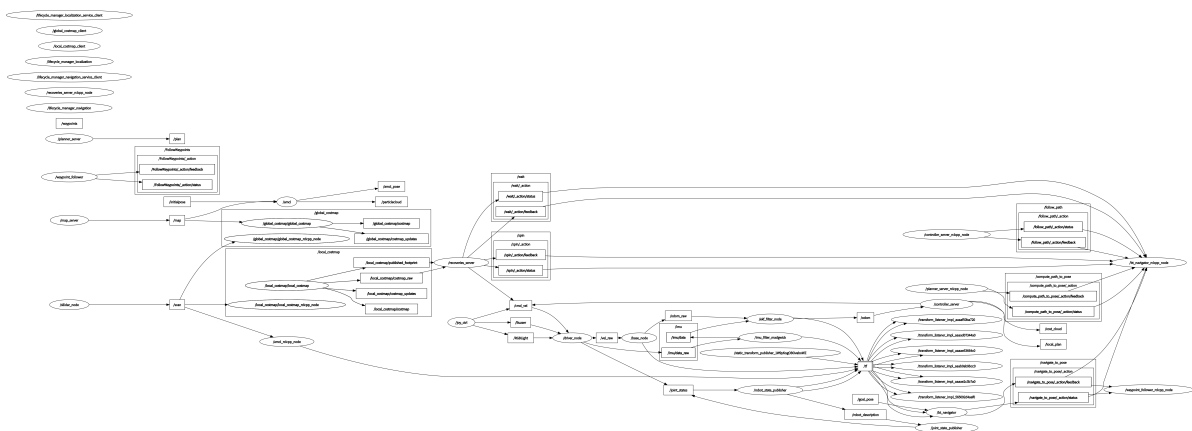
5. navigation process may appear, this is due to the ros-foxy version of navigation2 itself, the subsequent ros2 version has been fixed!

```
[bt_navigator-7] [INFO] [1682061491.700842174] [bt_navigator]: Begin navigating from current location to (-6.95, -1.41)
[bt_navigator-7] [ERROR] [1682061491.727441591] [bt_navigator]: Action server failed while executing action callback: "send_goal failed"
[bt_navigator-7] [WARN] [1682061491.727594875] [bt_navigator]: [navigate_to_pose] [ActionServer] Aborting handle.
```

7.3 Node resolution

7.3.1. Displaying computational graphs

rqt_graph



7.3.2 Navigating the details of each node

/amcl

```
root@ubuntu:/# ros2 node info /amcl
/amcl
Subscribers:
  /initialpose: geometry_msgs/msg/PoseWithCovarianceStamped
  /map: nav_msgs/msg/OccupancyGrid
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /amcl/transition_event: lifecycle_msgs/msg/TransitionEvent
  /amcl_pose: geometry_msgs/msg/PoseWithCovarianceStamped
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /particle_cloud: nav2_msgs/msg/ParticleCloud
  /particlecloud: geometry_msgs/msg/PoseArray
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /amcl/change_state: lifecycle_msgs/srv/ChangeState
  /amcl/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /amcl/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /amcl/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /amcl/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /amcl/get_parameters: rcl_interfaces/srv/GetParameters
  /amcl/get_state: lifecycle_msgs/srv/GetState
  /amcl/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /amcl/list_parameters: rcl_interfaces/srv/ListParameters
  /amcl/set_parameters: rcl_interfaces/srv/SetParameters
  /amcl/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  /reinitialize_global_localization: std_srvs/srv/Empty
  /request_nomotion_update: std_srvs/srv/Empty
Service Clients:

Action Servers:

Action Clients:
```

/bt_navigator

```
root@ubuntu:/# ros2 node info /bt_navigator
/bt_navigator
Subscribers:
  /goal_pose: geometry_msgs/msg/PoseStamped
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /tf: tf2_msgs/msg/TFMessage
  /tf_static: tf2_msgs/msg/TFMessage
Publishers:
  /bt_navigator/transition_event: lifecycle_msgs/msg/TransitionEvent
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /bt_navigator/change_state: lifecycle_msgs/srv/ChangeState
  /bt_navigator/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /bt_navigator/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /bt_navigator/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /bt_navigator/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /bt_navigator/get_parameters: rcl_interfaces/srv/GetParameters
  /bt_navigator/get_state: lifecycle_msgs/srv/GetState
  /bt_navigator/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /bt_navigator/list_parameters: rcl_interfaces/srv/ListParameters
  /bt_navigator/set_parameters: rcl_interfaces/srv/SetParameters
  /bt_navigator/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:
  /navigate_to_pose: nav2_msgs/action/NavigateToPose
Action Clients:
```

/controller_server

```

root@ubuntu:/# ros2 node info /controller_server
/controller_server
Subscribers:
  /odom: nav_msgs/msg/Odometry
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /cmd_vel: geometry_msgs/msg/Twist
  /controller_server/transition_event: lifecycle_msgs/msg/TransitionEvent
  /cost_cloud: sensor_msgs/msg/PointCloud
  /evaluation: dwb_msgs/msg/LocalPlanEvaluation
  /local_plan: nav_msgs/msg/Path
  /marker: visualization_msgs/msg/MarkerArray
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /received_global_plan: nav_msgs/msg/Path
  /rosout: rcl_interfaces/msg/Log
  /transformed_global_plan: nav_msgs/msg/Path
Service Servers:
  /controller_server/change_state: lifecycle_msgs/srv/ChangeState
  /controller_server/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /controller_server/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /controller_server/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /controller_server/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /controller_server/get_parameters: rcl_interfaces/srv/GetParameters
  /controller_server/get_state: lifecycle_msgs/srv/GetState
  /controller_server/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /controller_server/list_parameters: rcl_interfaces/srv/ListParameters
  /controller_server/set_parameters: rcl_interfaces/srv/SetParameters
  /controller_server/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:
  /controller_server/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /controller_server/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /controller_server/get_parameters: rcl_interfaces/srv/GetParameters
  /controller_server/list_parameters: rcl_interfaces/srv/ListParameters
  /controller_server/set_parameters: rcl_interfaces/srv/SetParameters
  /controller_server/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Action Servers:

Action Clients:

```

/global_costmap/global_costmap

```

root@ubuntu:/# ros2 node info /global_costmap/global_costmap
/global_costmap/global_costmap
Subscribers:
  /global_costmap/footprint: geometry_msgs/msg/Polygon
  /map: nav_msgs/msg/OccupancyGrid
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /global_costmap/costmap: nav_msgs/msg/OccupancyGrid
  /global_costmap/costmap_raw: nav2_msgs/msg/Costmap
  /global_costmap/costmap_updates: map_msgs/msg/OccupancyGridUpdate
  /global_costmap/global_costmap/transition_event: lifecycle_msgs/msg/TransitionEvent
  /global_costmap/published_footprint: geometry_msgs/msg/PolygonStamped
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /global_costmap/clear_around_global_costmap: nav2_msgs/srv/ClearCostmapAroundRobot
  /global_costmap/clear_entirely_global_costmap: nav2_msgs/srv/ClearEntireCostmap
  /global_costmap/clear_except_global_costmap: nav2_msgs/srv/ClearCostmapExceptRegion
  /global_costmap/get_costmap: nav2_msgs/srv/GetCostmap
  /global_costmap/global_costmap/change_state: lifecycle_msgs/srv/ChangeState
  /global_costmap/global_costmap/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /global_costmap/global_costmap/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /global_costmap/global_costmap/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /global_costmap/global_costmap/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /global_costmap/global_costmap/get_parameters: rcl_interfaces/srv/GetParameters
  /global_costmap/global_costmap/get_state: lifecycle_msgs/srv/GetState
  /global_costmap/global_costmap/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /global_costmap/global_costmap/list_parameters: rcl_interfaces/srv/ListParameters
  /global_costmap/global_costmap/set_parameters: rcl_interfaces/srv/SetParameters
  /global_costmap/global_costmap/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:

Action Clients:

```

/lifecycle_manager_localization

```

root@ubuntu:/# ros2 node info /lifecycle_manager_localization
/lifecycle_manager_localization
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /lifecycle_manager_localization/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /lifecycle_manager_localization/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /lifecycle_manager_localization/get_parameters: rcl_interfaces/srv/GetParameters
  /lifecycle_manager_localization/is_active: std_srvs/srv/Trigger
  /lifecycle_manager_localization/list_parameters: rcl_interfaces/srv/ListParameters
  /lifecycle_manager_localization/manage_nodes: nav2_msgs/srv/ManageLifecycleNodes
  /lifecycle_manager_localization/set_parameters: rcl_interfaces/srv/SetParameters
  /lifecycle_manager_localization/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:

Action Clients:

```

/local_costmap/local_costmap

```

root@ubuntu:/# ros2 node info /local_costmap/local_costmap
/local_costmap/local_costmap
Subscribers:
  /local_costmap/footprint: geometry_msgs/msg/Polygon
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /local_costmap/costmap: nav_msgs/msg/OccupancyGrid
  /local_costmap/costmap_raw: nav2_msgs/msg/Costmap
  /local_costmap/costmap_updates: map_msgs/msg/OccupancyGridUpdate
  /local_costmap/local_costmap/transition_event: lifecycle_msgs/msg/TransitionEvent
  /local_costmap/published_footprint: geometry_msgs/msg/PolygonStamped
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /local_costmap/clear_around_local_costmap: nav2_msgs/srv/ClearCostmapAroundRobot
  /local_costmap/clear_entirely_local_costmap: nav2_msgs/srv/ClearEntireCostmap
  /local_costmap/clear_except_local_costmap: nav2_msgs/srv/ClearCostmapExceptRegion
  /local_costmap/get_costmap: nav2_msgs/srv/GetCostmap
  /local_costmap/local_costmap/change_state: lifecycle_msgs/srv/ChangeState
  /local_costmap/local_costmap/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /local_costmap/local_costmap/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /local_costmap/local_costmap/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /local_costmap/local_costmap/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /local_costmap/local_costmap/get_parameters: rcl_interfaces/srv/GetParameters
  /local_costmap/local_costmap/get_state: lifecycle_msgs/srv/GetState
  /local_costmap/local_costmap/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /local_costmap/local_costmap/list_parameters: rcl_interfaces/srv/ListParameters
  /local_costmap/local_costmap/set_parameters: rcl_interfaces/srv/SetParameters
  /local_costmap/local_costmap/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:

Action Clients:

```

/map_server


```

root@ubuntu:/# ros2 node info /map_server
/map_server
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /map: nav_msgs/msg/OccupancyGrid
  /map_server/transition_event: lifecycle_msgs/msg/TransitionEvent
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /map_server/change_state: lifecycle_msgs/srv/ChangeState
  /map_server/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /map_server/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /map_server/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /map_server/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /map_server/get_parameters: rcl_interfaces/srv/GetParameters
  /map_server/get_state: lifecycle_msgs/srv/GetState
  /map_server/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /map_server/list_parameters: rcl_interfaces/srv/ListParameters
  /map_server/load_map: nav2_msgs/srv/LoadMap
  /map_server/map: nav_msgs/srv/GetMap
  /map_server/set_parameters: rcl_interfaces/srv/SetParameters
  /map_server/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:

Action Clients:

```

/planner_server

```

root@ubuntu:/# ros2 node info /planner_server
/planner_server
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /plan: nav_msgs/msg/Path
  /planner_server/transition_event: lifecycle_msgs/msg/TransitionEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /planner_server/change_state: lifecycle_msgs/srv/ChangeState
  /planner_server/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /planner_server/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /planner_server/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /planner_server/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /planner_server/get_parameters: rcl_interfaces/srv/GetParameters
  /planner_server/get_state: lifecycle_msgs/srv/GetState
  /planner_server/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /planner_server/list_parameters: rcl_interfaces/srv/ListParameters
  /planner_server/set_parameters: rcl_interfaces/srv/SetParameters
  /planner_server/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:
  /planner_server/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /planner_server/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /planner_server/get_parameters: rcl_interfaces/srv/GetParameters
  /planner_server/list_parameters: rcl_interfaces/srv/ListParameters
  /planner_server/set_parameters: rcl_interfaces/srv/SetParameters
  /planner_server/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Action Servers:

Action Clients:

```

/recoveries_server

```

root@ubuntu:/# ros2 node info /recoveries_server
/recoveries_server
Subscribers:
  /local_costmap/costmap_raw: nav2_msgs/msg/Costmap
  /local_costmap/published_footprint: geometry_msgs/msg/PolygonStamped
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /cmd_vel: geometry_msgs/msg/Twist
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /recoveries_server/transition_event: lifecycle_msgs/msg/TransitionEvent
  /rosout: rcl_interfaces/msg/Log
Service Servers:
  /recoveries_server/change_state: lifecycle_msgs/srv/ChangeState
  /recoveries_server/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /recoveries_server/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /recoveries_server/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /recoveries_server/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /recoveries_server/get_parameters: rcl_interfaces/srv/GetParameters
  /recoveries_server/get_state: lifecycle_msgs/srv/GetState
  /recoveries_server/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /recoveries_server/list_parameters: rcl_interfaces/srv/ListParameters
  /recoveries_server/set_parameters: rcl_interfaces/srv/SetParameters
  /recoveries_server/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:
  /backup: nav2_msgs/action/BackUp
  /spin: nav2_msgs/action/Spin
  /wait: nav2_msgs/action/Wait
Action Clients:

```

/waypoint_follower

```

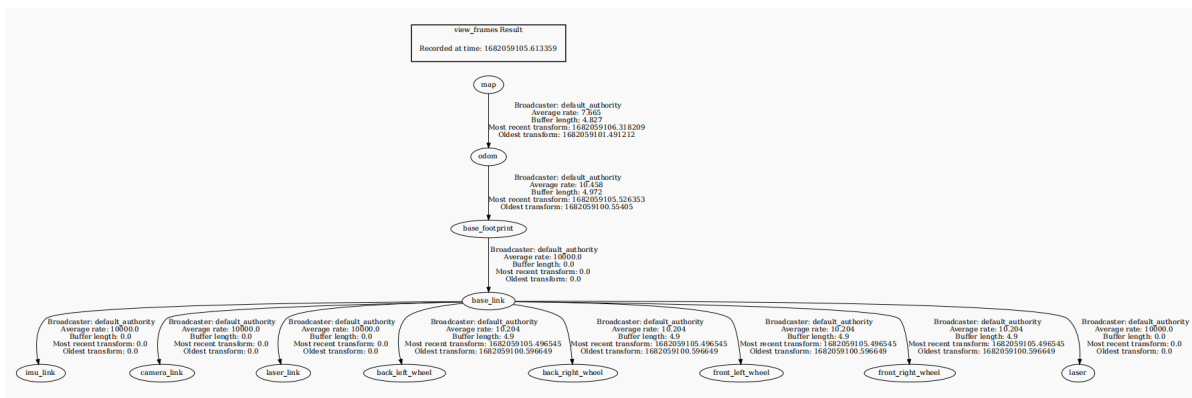
root@ubuntu:/# ros2 node info /waypoint_follower
/waypoint_follower
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /waypoint_follower/transition_event: lifecycle_msgs/msg/TransitionEvent
Service Servers:
  /waypoint_follower/change_state: lifecycle_msgs/srv/ChangeState
  /waypoint_follower/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /waypoint_follower/get_available_states: lifecycle_msgs/srv/GetAvailableStates
  /waypoint_follower/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
  /waypoint_follower/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /waypoint_follower/get_parameters: rcl_interfaces/srv/GetParameters
  /waypoint_follower/get_state: lifecycle_msgs/srv/GetState
  /waypoint_follower/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
  /waypoint_follower/list_parameters: rcl_interfaces/srv/ListParameters
  /waypoint_follower/set_parameters: rcl_interfaces/srv/SetParameters
  /waypoint_follower/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:
  /FollowWaypoints: nav2_msgs/action/FollowWaypoints
Action Clients:

```

7.3.3 TF transformations

```
ros2 run tf2_tools view_frames.py
```



7.4. navigation2 details

• Comparison of the navigation framework in ROS 1 vs. ROS 2

In Nav 2, move_base is split into multiple components. Unlike a single state machine (ROS 1), Nav 2 utilizes low-latency, reliable communication between the action server and ROS 2 to separate ideas. Behavior trees are used to choreographically organize these tasks, which allows Nav 2 to have highly configurable navigation behaviors without having to programmatically arrange the tasks through behavior tree xml files.

nav2_bt_navigator replaces move_base at the top level, which uses an Action interface to accomplish a tree-based action model navigation task. A more complex state machine is implemented via BT, and recovery behaviors are added as additional Action Servers. these behavior trees are configurable xml.

The path planning, recovery, and controller servers are also Action Servers that the BT navigator can call to perform computations. All 3 servers can host many plugins for many algorithms, each of which can individually invoke specific behaviors from the navigation behavior tree. The default plugins provided are ported from ROS 1, i.e. DWB, NavFn and similar recovery behaviors such as rotate and clear costmaps, plus a new recovery mechanism that waits a fixed amount of time has been added. These servers are invoked from the BT Navigator via their action servers to compute 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.

Porting package:

- amcl: ported to nav2_amcl
- map_server: ported to nav2_map_server
- nav2_planner: replaces global_planner, manages **N** planner plugins
- nav2_controllere: replaces local_planner, manages **N** controller plugins
- Navfn: port to nav2_navfn_planner
- DWB: replaces DWA and is ported to ROS 2 under the nav2_dwb_controller metapackage
- nav_core: ported to nav2_core with interface updates
- costmap_2d: ported to nav2_costmap_2d

New package:

- Nav2_bt_navigator: replace move_base state machine
- nav2_lifecycle_manager: handle server program lifecycle

- nav2_waypoint_follower: can perform complex tasks with many waypoints
- nav2_system_tests: a set of basic tutorials for CI integration testing and simulation
- nav2_rviz_plugins: a rviz plugin to control Navigation2 servers, commands, cancel and navigation
- nav2_experimental: experimental (and incomplete) work on deep reinforcement learning controllers
- navigation2_behavior_trees: a wrapper for the behavior tree library used to call the ROS action server

7.4.1 amcl

The full English name of amcl is adaptive Monte Carlo localization, which is a probabilistic localization system for 2D mobile robots. It is actually an upgraded version of the Monte Carlo localization method that uses an adaptive KLD method to update the particles and uses a particle filter to track the pose of the robot based on a known map. As currently implemented, the node is only suitable for laser scanning and laser maps. It can be extended to handle other sensor data. amcl receives laser-based maps, laser scans, and transform information and outputs pose estimates. On startup, amcl initializes its particle filters based on the parameters provided. 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

- The Monte Carlo method, also known as statistical simulation and statistical experimentation is an idea or method. It is a numerical simulation method that takes probabilistic phenomena as the object of study. Monte Carlo is a method of calculating an unknown characteristic quantity by obtaining statistical values according to the sampling method.
- Example: A rectangle has an irregular shape inside, how to calculate the area of the irregular shape? It is not easy to calculate. But we can approximate. Take a bunch of beans, evenly spread on the rectangle, and 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. In terms of robot localization, it could be at any location in the map, how do we express the confidence of a location in this case? We also use particles, where there are more particles, it means the robot where the probability is high.

The biggest advantage of the Monte Carlo method

- The error of the method is independent of the dimension of the problem.
- It is straightforward to solve problems of a statistical nature.
- Discretization is not necessary for continuous problems.

Disadvantages of the Monte Carlo method

- For deterministic problems it needs to be transformed into stochastic problems.
- Errors are probabilistic.
- Usually requires more computational steps N .

Particle filtering

- The number of particles represents how likely something is. The distribution of particles is changed by some evaluation method (evaluating the likelihood of the thing). For example, in robot localization, if a particle A, I think it is highly probable that this particle is at this coordinate (e.g., this coordinate belongs to the previously mentioned "this thing"), then I give it a high score. The next time I rearrange the positions of all the particles, I'll place more of

them near this position. After a few more rounds of this, the particles will be concentrated in the high probability positions.

Adaptive Monte Carlo

- Solves the robot abduction problem by globally re-scattering particles when it realizes that the average score of the particles has suddenly decreased (meaning that the correct particles have been discarded in a certain iteration).
- Solved the problem of fixed number of particles, because sometimes when the robot positioning is almost obtained, for example, these particles are concentrated in one piece, it is not necessary to maintain so many particles, this time the number of particles can be a little less.

7.4.2 costmaps and layers

The current environment representation is a costmap. A costmap is a regular 2D grid of cells containing cells from unknown, idle, occupied, or inflated costs. This costmap is then searched for to compute a global plan or sampled to compute local control work.

Various cost map layers are implemented as pluginlib plugins to buffer information into the cost map. This includes information from LIDAR, lidar, sonar, depth sensors, image sensors, etc. It is best to process the sensor data before it is fed into the layer-based map, but this is up to the developer.

Layers of costal maps 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 cost map based on some rules or heuristic algorithms. Finally, they can be used to buffer real-time data into a 2D or 3D world for binarized marking of obstacles.

7.4.3 planner_server

The task of the planner is to compute paths based on a specific goal function. Depending on the chosen nomenclature and algorithm, a path is also called a route. here are two typical examples, a path planner that computes a path to a target location (e.g. to reach a target pose from the current location) or a path planner with complete coverage (e.g. a path planner that covers all free space). The planner has access to global environment representations and cached data from sensors. The planner can have the following capabilities:

- Calculate shortest paths
- Calculate complete coverage paths
- Calculate paths along sparse or predefined routes

The general task of the 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 environmental information captured by different sensors to search for paths for the robot. Some of these algorithms obtain paths by searching the environment raster, others extend the possible states of the robot taking into account the feasibility of the path.

As mentioned earlier, the planning server uses plugins that work on the grid space such as NavFn Planner, Smac Planner 2D and Theta Star Planner. NavFn planner is a navigationally functional planner that uses Dijkstra or A algorithm. Smac 2D planner implements the 2D A algorithm with 4- or 8-connected neighbor nodes and has smoothing and multi-resolution features. theta Star

planner (smooth path planning at arbitrary angles in continuous environments, theta is a variant of A that propagates information along the edges of the graph without restricting the paths to the edges (finding the "arbitrary angles")). "arbitrary angle" paths)) is implemented using a line-of-sight algorithm by creating non-discrete directed path segments. Commonly used planners are listed below:

Plugin Name	Supported robot types
NavFn Planner	Circular Differential, Circular Omnidirectional
Smac2D Planner	Circular Differential, Circular Omnidirectional
Theta Star Planner	Circular Differential, Circular Omnidirectional
Smac Hybrid A* Planner	Non-circular or rounded akerman, non-circular or rounded legged
Smac Lattice Planner	Non-circular micro speed, non-circular omnidirectional

7.4.4 controller_server

A controller, also known as a local planner in ROS 1, accomplishes the global computation of a path to follow or completes a local task. The controller has access to the local environment representation and tries to compute a feasible path for the base path to follow. Many controllers project the robot forward into space and compute the local feasible path at each update iteration. The controller has the following capabilities:

- Follow a path
- Follow a path
- Follow a path
- Follow a path
- Follow a path
- Follow a path
- Follow a path
- Follow a path
- Board an elevator
- Interface with a tool

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

Common controller plugins are DWA Controller and TEB Controller

1. DWA Controller

The idea of DWA algorithm is that by sampling different possible speeds and then simulating different trajectories, using the distance to the target point, obstacle distance and time etc. to evaluate, select the trajectory with the best score, and issue a speed command to control the cart to move forward.

The sampling of speeds simulated by the DWA algorithm is not randomly selected but a range exists, the magnitude of the speeds is limited by the maximum and minimum speeds of the trolley, the magnitude of the acceleration is affected by the performance of the trolley motors, and also the distance of the obstacles affects the value of the speeds in order to ensure safety.

Differential speed trolley, McWheeler in the adjacent moment, the movement distance is short, you can see the trajectory between two adjacent points, as a straight line, and the other models of the trajectory as an arc.

Three evaluation functions:

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

Physical meaning: make the car toward the target point, avoid obstacles, fast driving

2. TEB Controller

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

- Follow the path + obstacle avoidance: the constraints have two main goals, to follow a consistent global path planning and obstacle avoidance. Both objective functions are very similar. The following path applies an external force to pull the local path towards the global path, and the obstacle avoidance constraint applies a force to keep the local path away from the obstacle.
- Velocity/acceleration constraints: velocity and acceleration should be within a certain range.
- Kinematic constraints: smooth trajectory consisting of several arc segments, control quantities are only linear and angular velocities, Ackermann structure has a minimum turning radius, and wheels/all directions/differential speeds are all 0.
- Fastest path constraint: the objective function enables the robot to obtain the fastest path, where the attitude points on the path are uniformly separated in time, instead of the traditional spatial shortest path.

TEB can be formulated as a multi-objective optimization problem, where most of the objectives are local and related to only a small number of parameters, since they depend on only a few consecutive robot states. After determining the positions of N control points, the paths are optimized using the open source framework g2o (General Graph Optimization). In the cart scenario, all the position points, time intervals, and obstacles are described as points, and the constraints are described as edges. The minimum distance from an obstacle connects an obstacle to a bit-position point, and the velocity constraint connects two neighboring bit-position points and their time differences. In summary, the local trajectory generated by the TEB consists of a series of discrete poses with time information. The optimization goal of the g2o algorithm is to optimize the final trajectory composed of these discrete poses to achieve the goals of minimum time, minimum distance, and distance from obstacles, while limiting the velocity and acceleration so that the trajectory satisfies the robot dynamics.

7.4.5 recoveries_server

Recovery behaviors are the backbone of a fault-tolerant system. The goal of the recoveries is to deal with unknown or faulty conditions in the system and handle them autonomously. Examples include faults in the sensing system that would cause the environmental representation to be filled with false obstacles. This triggers a clear cost map recovery to allow the robot to move. Another example is when the robot gets stuck due to dynamic obstacles or poor control. Where permitted, backtracking or rotating in place would allow the robot to move from the stuck position into free space where navigation can be successfully performed. Finally, in the case of a complete failure, recovery can be implemented to draw the attention of the operator for assistance. This can be done via email, SMS, Slack, Matrix, etc.

7.4.6 waypoint following

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

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

In the first school of thought, the `nav2_waypoint_follower` package is sufficient to create a product-level robotics solution. Since the autonomous system/scheduler takes into account factors such as the robot's posture, battery level, current task, etc. when assigning tasks, the application on the robot only needs to be concerned with the task at hand and not with the other complexities of completing the tasks required by the system. In this case, a request sent to a waypoint follower should be treated as 1 unit of work (e.g., 1 pick in a warehouse, 1 safety patrol cycle, 1 aisle, etc.) to perform the task and then returned to the scheduler for the next task or request for recharging. In this school of thought, the waypoint following application is simply a step above the navigation software stack and below the system autonomy application.

In the second school of thought, the `nav2_waypoint_follower` package is a good example application/proof-of-concept, but it does require a waypoint-following/autonomous system on the robot to take on more tasks to develop a robust solution. In this case, the `nav2_behavior_tree` package should be used to create a custom application level behavior tree to accomplish tasks using navigation. This can contain subtrees such as checking the state of charge during a task to return to a mooring dock, or handling more than 1 unit of work in a more complex task. Soon there will be a `nav2_bt_waypoint_follower` (name to be adjusted) which will allow users to create this application more easily. In this school of thought, the waypoint following application is more closely linked to the autonomous system, or in many cases, the waypoint following application is the autonomous system itself.

These two schools of thought do not simply say who is better than the other; who is better depends largely on what tasks the robot is accomplishing, what type of environment it is in, and what cloud resources are available. Often, for established business cases, this distinction is very clear.

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/params/dwa_nav_params.  
yaml  
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/params/teb_nav_params.  
yaml
```

7.4.7 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 navigating tasks and to provide a way to easily specify complex robot behaviors (including recovery).

Planer Server, Controller Server and Recovery Server implement their respective functions, i.e. global path planning, local path planning and recovery operations. But their functions are independent and do not interfere with each other. To realize a complete navigation function requires the cooperation of various modules. BT Navigator Server is the one who assembles the Lego blocks.

For a complete parameterization see: