Multi-vehicle platoon

1. Description

This function arranges three robots in a program-defined position. When the lead robot moves, the other two robots follow suit, maintaining a line at the destination. The program offers three possible line configurations:

- Vertical Line: The lead robot is at the front, with the other two robots following behind it, forming a line similar to the Arabic numeral 1;
- Horizontal Line: The lead robot is in the center, with the other two robots to its left and right, forming a line similar to the Chinese character "一";
- Left and Right Guard: The lead robot is in the center, with the other two robots to its left and right, forming a line similar to a "tank" formation.

For ease of explanation, this tutorial only demonstrates the effects of two robots, with robot1 representing the lead robot and robot2 as the secondary robot.

1.1. Functional Requirements

This feature requires two or three vehicles, and each of them must have its namespace and ROS_DOMAIN_ID configured. For setup instructions, refer to [12.Multi-vehicle Course] -- [1.Multi-vehicle control] -- [1.1. Functional Requirements].

1.2. Site Requirements

To run this feature, choose a spacious site. Although the vehicles have obstacle avoidance, a narrow site can result in poor navigation performance or even navigation planning failure.

1.3. Navigation Map

Before starting a multi-car platoon, you need to place the map file in the directory on the board or in the virtual machine.

```
~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/yahboomcar.yaml
```

The map file includes a .yaml parameter file and a .pgm image file.

Note: Both cars require the same map file

2. Program Startup

Using the Raspberry Pi 5 as an example, due to performance issues, a virtual machine is required for visualization.

The virtual machine must be on the same LAN as the two cars, and the ROS_DOMAIN_ID must be set to the same for both cars. The modification method can be referenced above for setting the car's ros_domain_id. All steps involve modifying the contents of ~/.bashrc and refreshing the environment variables after the modification.

For Raspberry Pi and Jetson-Nano boards, open a terminal on the host computer and enter the command to enter the Docker container. Once inside the Docker container, enter the commands mentioned in this section in the terminal. For instructions on entering the Docker container from the host computer, refer to **[01. Robot Configuration and Operation Guide] -- [5.Enter Docker**

(For JETSON Nano and RPi 5)].

For Orin boards, simply open a terminal and enter the commands mentioned in this section.

2.1 Implementation Principle

This function relies on nav2 navigation when activated. When robot1 is given a target point, it publishes two TF transforms: one from robot1 to point2 and one from robot1 to point3. Point2 and point3 are the target points that robot2 and robot3 need to reach. As robot1 moves, it publishes these two TF transforms. Robot2 and robot3 only need to obtain the locations of point2 and point3 on the map and then navigate to them, respectively.

This example uses two robots, Robot1 and Robot2. Please place them nearby in advance to ensure smooth network connectivity. This example requires a high data transfer rate.

2.2. Starting Chassis Data

Enter the following commands on Robot1 and Robot2 to start the robots and chassis data fusion.

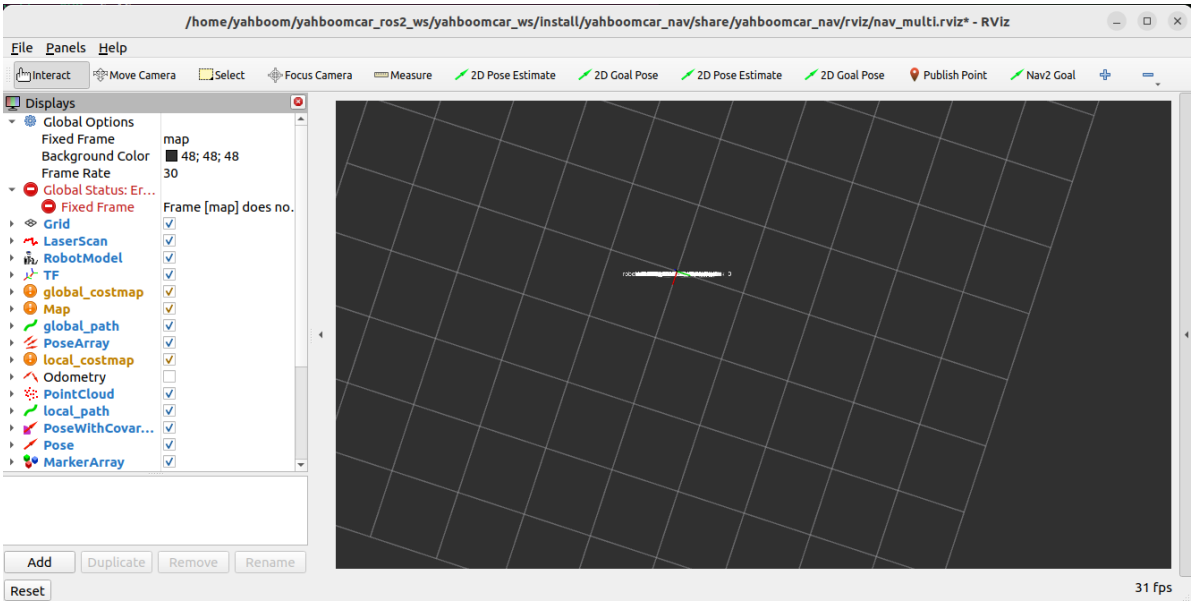
```
#robot1
ros2 launch yahboomcar_multi A1_laser_bringup_multi_launch.xml
robot_name:=robot1
#robot2
ros2 launch yahboomcar_multi A1_laser_bringup_multi_launch.xml
robot_name:=robot2
```

2.3. Starting RVIZ and Publishing Map Data

In the virtual machine configured for multi-machine communication, open a terminal and enter the following command to start RVIZ.

```
ros2 launch yahboomcar_multi display_multi_nav_launch.py
```

After launching, the image below appears.



Then launch the map loading program. The default map launched is yahboomcar.yaml, and the file path is:

```
~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/yahboomcar.yaml
```

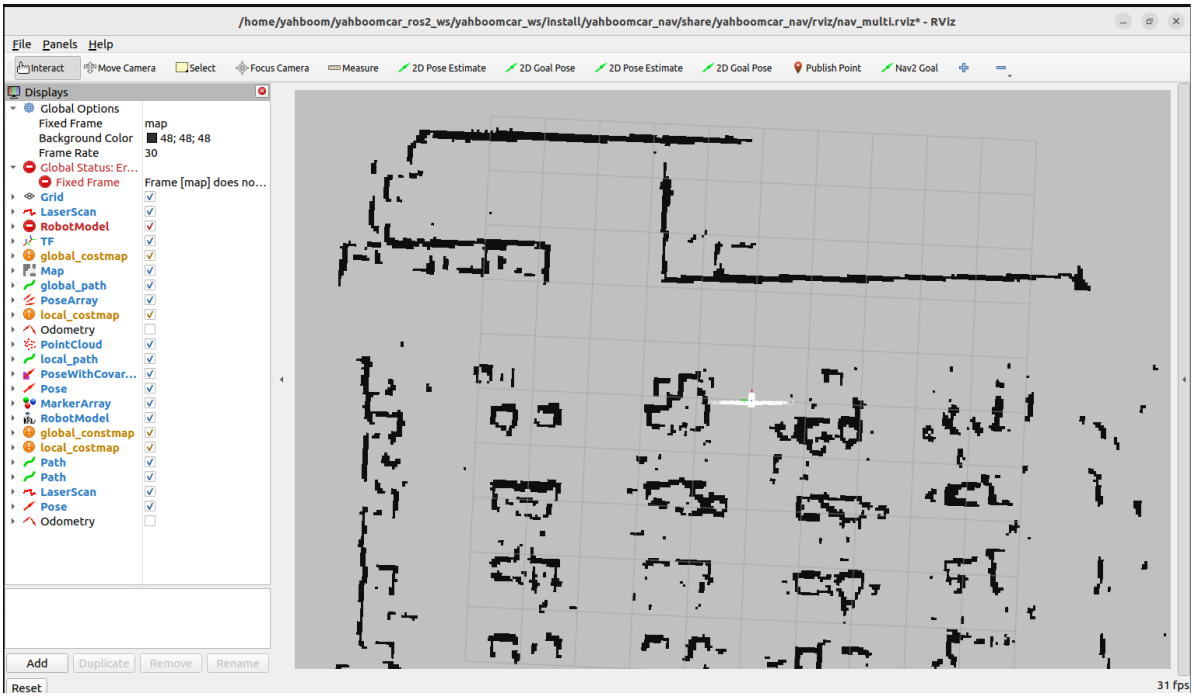
Enter the following command in the terminal to launch (you only need to load the map service in one robot's terminal).

```
ros2 launch yahboomcar_multi map_server_launch.py
```

```
jetson@yahboom:~$ ros2 launch yahboomcar_multi map_server_launch.py
[INFO] [launch]: All log files can be found below /home/jetson/.ros/log/2025-08-22-10-36-56-811488-yahboom-35304
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [map_server-1]: process started with pid [35353]
[INFO] [lifecycle_manager-2]: process started with pid [35355]
[lifecycle_manager-2] [INFO] [1755830217.082366334] [map_lifecycle_manager]: Creating
[lifecycle_manager-2] [INFO] [1755830217.102205472] [map_lifecycle_manager]: Creating and initializing lifecycle service clients
[lifecycle_manager-2] [INFO] [1755830217.142624264] [map_lifecycle_manager]: Starting managed nodes bringup...
[lifecycle_manager-2] [INFO] [1755830217.143903718] [map_lifecycle_manager]: Configuring map_server_node
[map_server-1] [INFO] [1755830217.148328764] [map_server_node]:
[map_server-1]  map_server_node lifecycle node launched.
[map_server-1]  Waiting on external lifecycle transitions to activate
[map_server-1]  See https://design.ros2.org/articles/node_lifecycle.html for more information.
[map_server-1] [INFO] [1755830217.148639115] [map_server_node]: Creating
[map_server-1] [INFO] [1755830217.248611386] [map_server_node]: Configuring
[map_server-1] [INFO] [1755830217.248791491] [map_io]: Loading yaml file: /home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomc
ar_nav/maps/yahboomcar.yaml
[map_server-1] [INFO] [1755830217.250262954] [map_io]: resolution: 0.05
[map_server-1] [INFO] [1755830217.250301068] [map_io]: origin[0]: -9.76
[map_server-1] [INFO] [1755830217.250314445] [map_io]: origin[1]: -12.7
[map_server-1] [INFO] [1755830217.250323661] [map_io]: origin[2]: 0
[map_server-1] [INFO] [1755830217.250332270] [map_io]: free_thresh: 0.25
[map_server-1] [INFO] [1755830217.250340878] [map_io]: occupied_thresh: 0.65
[map_server-1] [INFO] [1755830217.250350863] [map_io]: mode: trinary
[map_server-1] [INFO] [1755830217.250362159] [map_io]: negate: 0
[map_server-1] [INFO] [1755830217.254317615] [map_io]: Loading image_file: /home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboom
car_nav/maps/yahboomcar.pgm
[map_server-1] [INFO] [1755830217.293899982] [map_io]: Read map /home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/map
s/yahboomcar.pgm: 351 X 559 map @ 0.05 m/cell
[lifecycle_manager-2] [INFO] [1755830217.305750541] [map_lifecycle_manager]: Activating map_server_node
[map_server-1] [INFO] [1755830217.306148096] [map_server_node]: Activating
[map_server-1] [INFO] [1755830217.306623991] [map_server_node]: Creating bond (map_server_node) to lifecycle manager.
[lifecycle_manager-2] [INFO] [1755830217.417856136] [map_lifecycle_manager]: Server map_server_node connected with bond.
[lifecycle_manager-2] [INFO] [1755830217.417968941] [map_lifecycle_manager]: Managed nodes are active
[lifecycle_manager-2] [INFO] [1755830217.417990575] [map_lifecycle_manager]: Creating bond timer...
```

Next, on the virtual machine, our rviz visualization loads the created map.

(Due to network issues, the map may not be transferred to the virtual machine. Press Ctrl+C to shut it down and then restart the map loading service.)



2.4. Start AMD positioning

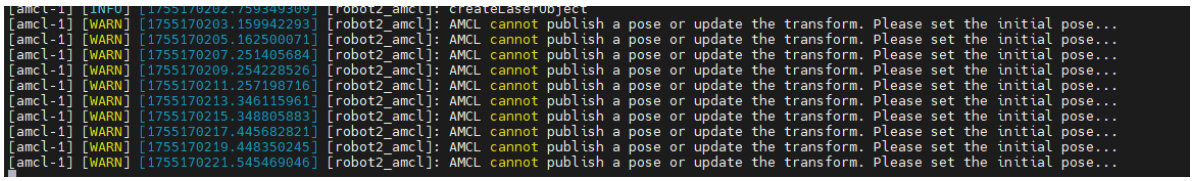
Open the terminal on robot1 and enter the following command to start AMD positioning.

```
ros2 launch yahboomcar_multi A1_amcl_robot1_launch.py
```

Open the terminal on the robot2 car and enter the following command to start AMCL positioning.

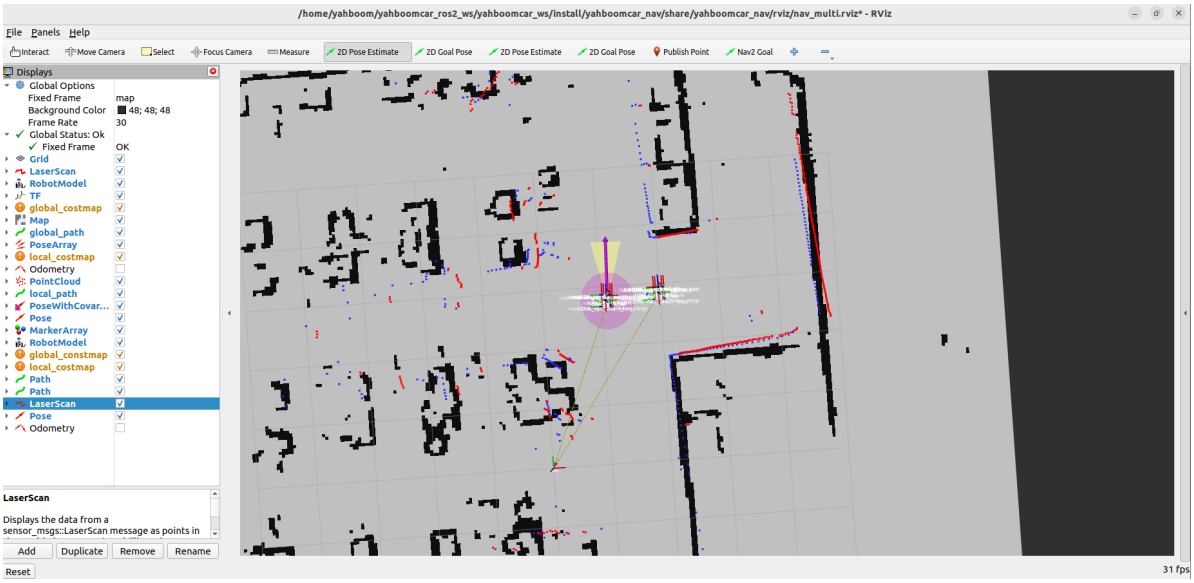
```
ros2 launch yahboomcar_multi A1_amcl_robot2_launch.py
```

As shown in the image below, the message "**AMCL cannot publish a pose or update the transform. Please set the initial pose..." appears, indicating that the program is running the AMCL positioning program.



Next, based on the car's position on the actual map, we use the [2D Pose Estimate] tool in rviz2 to give the car an initial pose. Here, rviz has two [2D Pose Estimate], the first tool is used to set the initial pose of robot 1, and the second tool is used to set the initial pose of robot 2. Use these two tools to set the initial poses for both robots.

As shown in the figure below, the areas scanned by the two radars overlap with the black area on the map. The blue point cloud represents the radar scan of robot 2, and the red point cloud represents the radar scan of robot 1.



2.5. Start nav2 Navigation

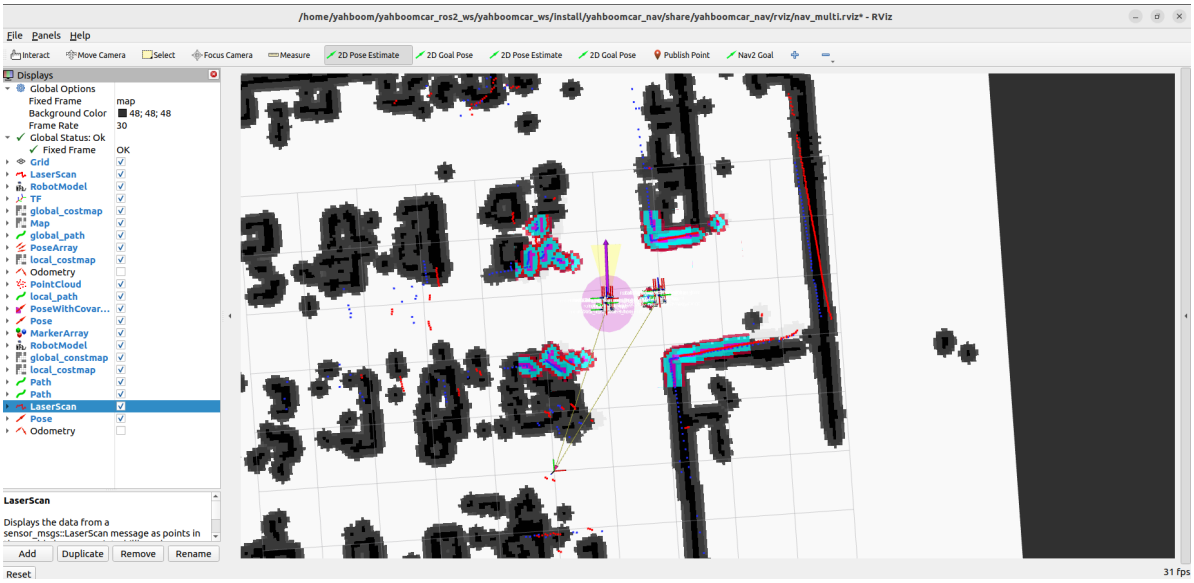
Open the terminal on robot 1 and enter the following command to start nav2 navigation for robot 1.

```
ros2 launch yahboomcar_multi A1_nav_robot1_launch.py
```

Open the terminal on robot 2 and enter the following command to start nav2 navigation for robot 2.

```
ros2 launch yahboomcar_multi A1_nav_robot2_launch.py
```

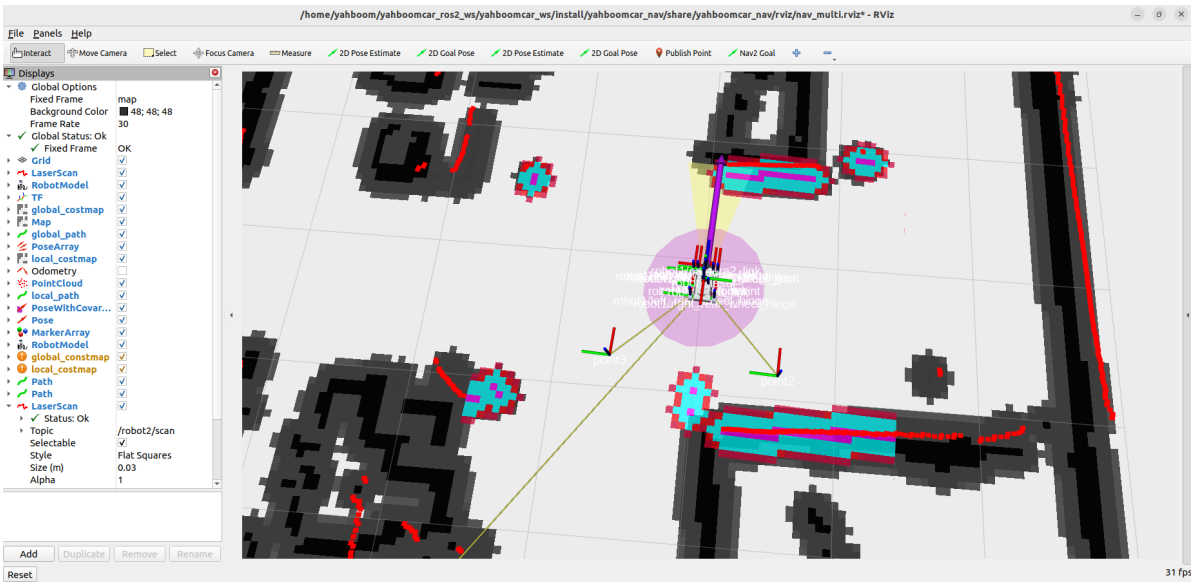
If both launch successfully, the expansion area shown in the figure below will appear.



2.6. Start the Formation Program

The following command on the master vehicle terminal starts the TF publishing program.

```
ros2 run yahboomcar_multi pub_follower_goal
```

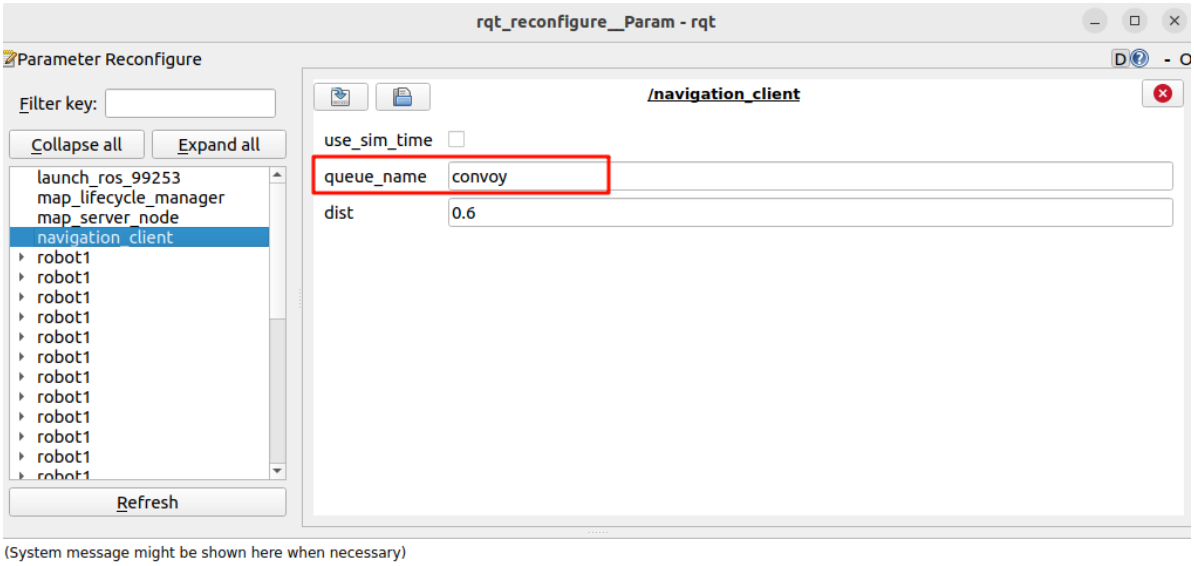


The following command on the slave vehicle 2 terminal starts the formation.

```
ros2 run yahboomcar_multi get_follower_point --ros-args -p robot_id:=robot2
```

Open a terminal in the virtual machine and type in the command to find the main vehicle's published TF node.

```
ros2 run rqt_reconfigure rqt_reconfigure
```



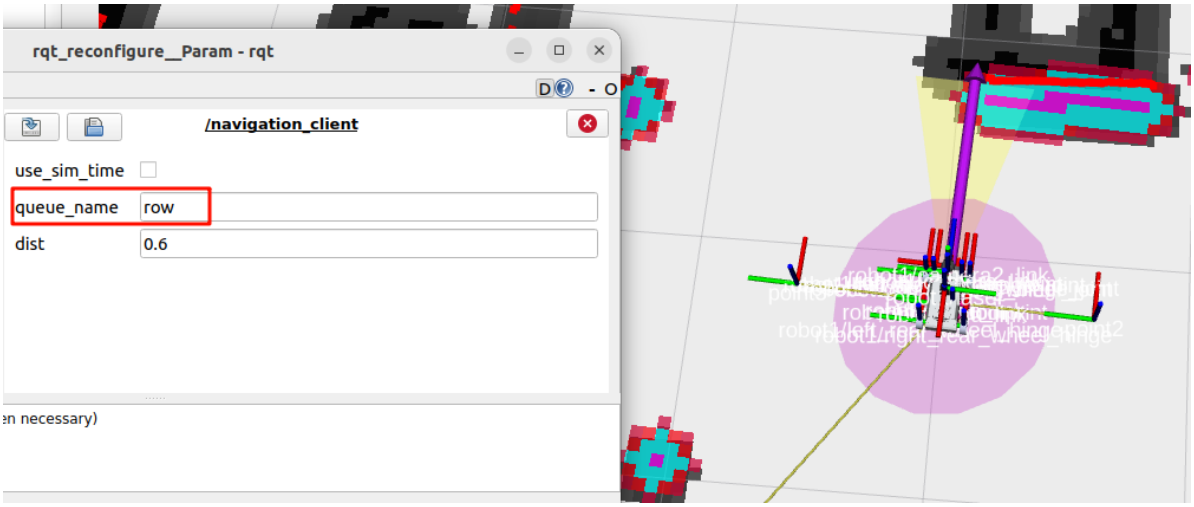
☑ After modifying the parameters, click the blank area in the GUI to enter the parameter value. Note that this will only take effect for the current boot. To permanently effect the parameter, modify it in the source code.

🛠 Parameter Analysis:

[queue_name]: Multi-vehicle array formation, column, row, or convoy

[dist]: Distance between the secondary vehicle and the primary vehicle. A small distance is not recommended.

If set to row, the positions of point2 and point3 will change in real time, and the secondary vehicle will navigate to those points. The effect is shown in the figure.



3. Core Code Analysis

3.1. pub_follower_goal.py

Robot 1 is the primary vehicle, publishing the coordinates of robot 2, robot 3, and TF.

```
import rclpy
from rclpy.action import ActionClient
...
```

```
import math

class NavigationClient(Node):
    def __init__(self):
        super().__init__('navigation_client')
        # 创建导航动作客户端，连接到NavigateThroughPoses动作服务器
        # Create a navigation action client and connect to the
NavigateThroughPoses action server
        self._client = ActionClient(self, NavigateThroughPoses,
'/navigate_through_poses')
        ...
        # 初始化目标位姿，设置坐标系为map
        # Initialize the target pose and set the coordinate system to map
        self.goal_pose = PoseStamped()
        self.goal_pose.header.frame_id = "map"
        ...
        # 声明并获取队列类型和间距参数
        #Declare and get queue type and spacing parameters
        self.declare_parameter("queue_name", "convoy")
        self.queue =
self.get_parameter("queue_name").get_parameter_value().string_value
        self.declare_parameter("dist", 0.6)
        self.dist =
self.get_parameter("dist").get_parameter_value().double_value
        # 创建定时器检查参数变化
        # Create a timer to check parameter changes
        self.timer = self.create_timer(0.5, self.on_timer)

    def get_waypointsPoseCallBack(self, msg):
        # 从MarkerArray中提取倒数第二个标记的位姿作为导航目标
        # Extract the pose of the second-to-last marker from MarkerArray as the
navigation target
        self.goal_pose.pose.position.x =
msg.markers[len(msg.markers)-2].pose.position.x
        ...
        self.send_goal()  # 发送导航目标 Send navigation target

    def on_timer(self):
        # 检查参数是否变化，如有变化则更新TF变换
        # Check if the parameters have changed, if so update the TF transform
        current_queue =
self.get_parameter("queue_name").get_parameter_value().string_value
        ...
        if current_queue != self.prev_queue or current_dist != self.prev_dist:
            self.update_tf(current_queue, current_dist)
            ...

    def update_tf(self, queue_type, dist):
        # 根据队列类型设置机器人间的相对位置TF变换
        # Set the relative position TF transformation between robots according
to the queue type
        self.queue = queue_type
        self.dist = dist
        ...
        if self.queue == "column":  # 列队形：x轴等距排列 Formation: equidistant
arrangement on the x-axis
            robot2_transform.transform.translation.x = -self.dist
            ...
```

```
        elif self.queue == "row":  # 行队形：y轴等距排列 Row formation: equidistant
on the y-axis
            robot2_transform.transform.translation.x = 0.0
            ...
        else:  # 护卫队队形：xy轴错开排列 Guard formation: staggered arrangement on
xy axis
            robot2_transform.transform.translation.x = -self.dist
            ...
        # 发布TF变换
        # Publish TF transformation
        self.robot1_to_point2_broadcaster.sendTransform(robot2_transform)
        ...

    def send_goal(self):
        # 创建并发送导航目标到动作服务器
        # Create and send navigation targets to the action server
        goal_msg = NavigateThroughPoses.Goal()
        goal_msg.poses.append(self.goal_pose)
        self._client.wait_for_server()
        ...

    def goal_response_callback(self, future):
        # 处理动作服务器对目标的响应
        # Process the action server's response to the target
        result = future.result()
        if not result.accepted:
            self.get_logger().error('Goal was rejected!')
            return
        ...

    def result_callback(self, future):
        # 处理导航完成结果，状态4表示目标已到达
        # Process the navigation completion result. Status 4 means the target
has been reached.
        result_msg = future.result()
        if result_msg.status == 4:
            print("Goal reached.")

    def feedback_callback(self, feedback_msg):
        # 监控导航进度，剩余距离小于0.10米时提示完成
        # Monitor navigation progress and indicate completion when the remaining
distance is less than 0.10 meters
        if feedback_msg.feedback.distance_remaining < 0.10:
            print("Done.")

...
```

3.2、get_follower_point.py

Robot2 and robot3, the auxiliary vehicles, subscribe to the target point sent by robot1.,

```
import rclpy
from rclpy.node import Node
import tf2_ros
...

class TfListenerNode(Node):
```

```
    def __init__(self):
        super().__init__('tf_listener_node')
        # 声明并获取机器人ID参数，用于动态设置发布话题
        #Declare and obtain the robot ID parameter for dynamically setting the
publishing topic
        self.declare_parameter('robot_id', 'robot2')
        robot_id =
self.get_parameter('robot_id').get_parameter_value().string_value

        # 创建TF缓冲区和监听器
        # Create TF buffer and listener
        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        # 根据机器人ID动态创建目标位姿发布器
        # Dynamically create a target pose publisher based on the robot ID
        self.pub_robot_pose = self.create_publisher(PoseStamped,
f"/{robot_id}/goal_pose", 10)

        # 初始化目标位姿消息，坐标系设为map
        # Initialize the target pose message, and set the coordinate system to
map
        self.goal_pose = PoseStamped()
        self.goal_pose.header.frame_id = "map"

        # 创建10Hz定时器，定期获取变换信息
        # Create a 10Hz timer to periodically obtain transformation information
        self.timer = self.create_timer(0.1, self.timer_callback)
        ...
        self.get_point()  # 初始获取一次变换 Initially obtain a transformation

    def timer_callback(self):
        # 定时器回调，定期执行坐标变换获取
        # Timer callback, periodically execute coordinate transformation
acquisition
        self.get_point()

    def get_point(self):
        try:
            # 获取从map坐标系到point2坐标系的变换
            # Get the transformation from the map coordinate system to the
point2 coordinate system
            transform = self.tf_buffer.lookup_transform('map', 'point2',
rclpy.time.Time())

            # 提取四元数并转换为欧拉角（用于调试显示）
            # Extract quaternion and convert to Euler angle (for debugging
display)
            quaternion = [0, 0, transform.transform.rotation.z,
transform.transform.rotation.w]
            rotation = R.from_quat(quaternion)
            euler_angles = rotation.as_euler('xyz', degrees=True)

            # 获取当前变换的坐标和旋转值
            # Get the coordinates and rotation values ••of the current
transformation
            current_x = transform.transform.translation.x
            ...
```

```
            # 检查变换是否发生变化（位置变化>0.01m或旋转变化）
            # Check if the transformation has changed (position change > 0.01m
or rotation change)
            if current_z != self.prev_z or current_w != self.prev_w or
abs(current_x - self.prev_x)>0.01 or abs(current_y - self.prev_y)>0.01:
                # 打印调试信息
                # Print debug information
                print("quaternion: ",quaternion)
                ...

                # 更新目标位姿并发布
                # Update target pose and publish
                self.goal_pose.pose.position.x =
transform.transform.translation.x
                ...
                self.pub_robot_pose.publish(self.goal_pose)

                # 保存当前值用于下一次比较
                # Save the current value for next comparison
                self.prev_z = current_z
                ...

        except (tf2_ros.TransformException, KeyError) as e:
            # 处理变换获取异常
            # Handle transformation acquisition exceptions
            self.get_logger().warn(f"Could not transform: {e}")

    def normalize_angle(self,angle):
        # 角度归一化函数，将角度限制在[-180, 180]范围内
        # Angle normalization function, limiting the angle to the range [-180,
180]
        res = angle
        while res > 180:
            res -= 2.0 * 180
        ...
        return res

...
```