8.ROS2_Cartographer mapping algorithm

8.ROS2_Cartographer mapping algorithm
- 1. Introduction
- 2. Use
  - 2.1. Configuration before use
  - 2.2. Specific use
- 3. Node analysis
  - 3.1. Display calculation graph
  - 3.2. Cartographer_node node details
  - 3.3. TF transformation

Cartographer：https://google-cartographer.readthedocs.io/en/latest/

Cartographer ROS2：https://github.com/ros2/cartographer_ros

1. Introduction

- Cartographer

Cartographer is a 2D and 3D SLAM (simultaneous localization and mapping) library supported by Google's open source ROS system. A graph-building algorithm based on graph optimization (multi-threaded backend optimization, problem optimization of ceiling construction). Data from multiple sensors (such as LIDAR, IMU, and cameras) can be combined to simultaneously calculate the sensor's position and map the environment around the sensor.

The source code of cartographer mainly includes three parts: cartographer, cartographer_ros and ceres-solver (back-end optimization).

Cartographer uses the mainstream SLAM framework, which is a three-stage structure of feature extraction, closed-loop detection, and back-end optimization. A certain number of LaserScans form a submap, and a series of submaps constitute the global map. The cumulative error in the short-term process of using LaserScan to construct a submap is not large, but the long-term process of using a submap to construct a global map will have a large cumulative error. Therefore, closed-loop detection is needed to correct the positions of these submaps. The basic unit of closed-loop detection is submap, closed-loop detection uses the scan_match strategy. The focus of cartographer is the creation of submap subgraphs that integrate multi-sensor data (odometry, IMU, LaserScan, etc.) and the implementation of the scan_match strategy for closed-loop detection.

- cartographer_ros

cartographer_ros runs under ROS and can accept various sensor data in the form of ROS messages.

After processing, it is published in the form of a message for easy debugging and visualization.

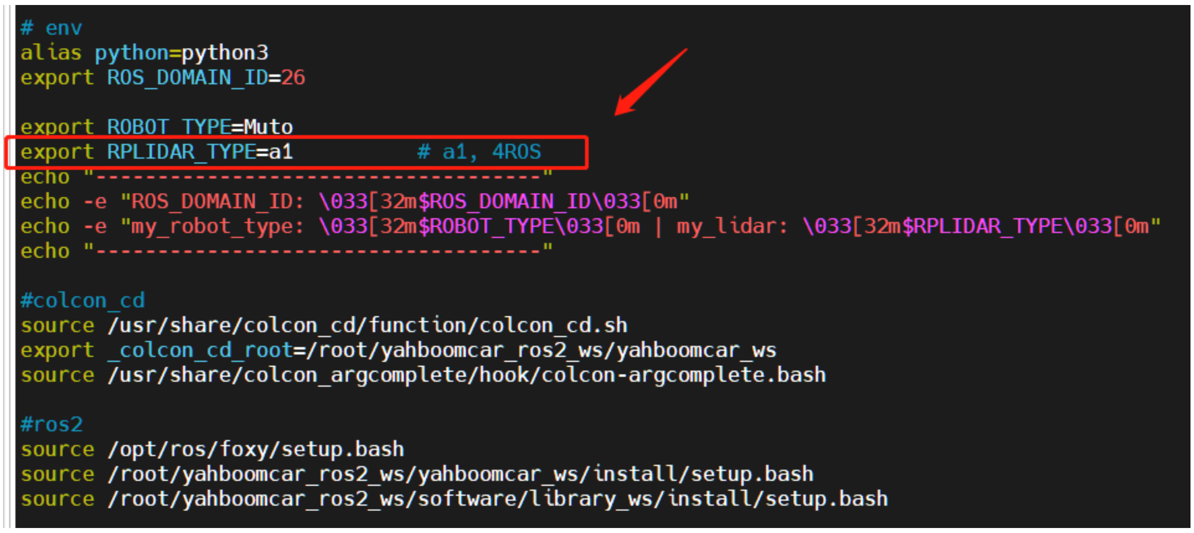
2. Use

2.1. Configuration before use

Note: Since the Muto series robots are equipped with multiple lidar devices, the factory system has been configured with routines for multiple devices. However, since the product cannot be automatically recognized, the lidar model needs to be manually set.

After entering the container: Make the following modifications according to the lidar type:

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



```
# env
alias python=python3
export ROS_DOMAIN_ID=26

export ROBOT_TYPE=Muto
export RPLIDAR_TYPE=a1          # a1, 4ROS
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"
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
```

After the modification is completed, save and exit vim, and then execute:

```
root@jetson-desktop:~# source .bashrc
------------------------------------
ROS_DOMAIN_ID: 26
my_robot_type: Muto | my_lidar: a1
------------------------------------
root@jetson-desktop:~#
```

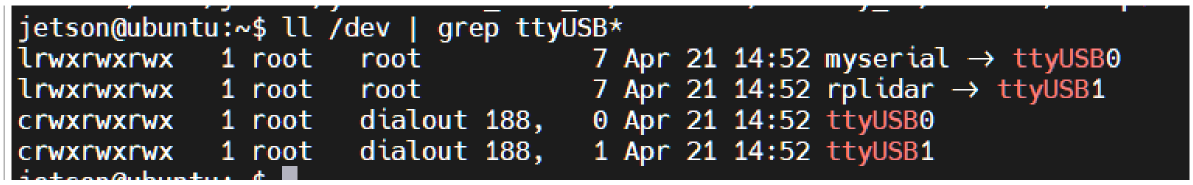
You can see the current modified lidar type.

2.2. Specific use

Note: When building a map, the slower the speed, the better the effect (note that the rotation speed should be slower). If the speed is too fast, the effect will be poor.

First, the port binding operation needs to be performed on the host machine [that is, Muto's jetson]. The two devices of lidar and serial port are mainly used here;

Then check whether the lidar and serial device are in the port binding state: on the host machine [that is, Muto's jetson] refer to the following command to execute the check. The successful binding is as follows:



```
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
```

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

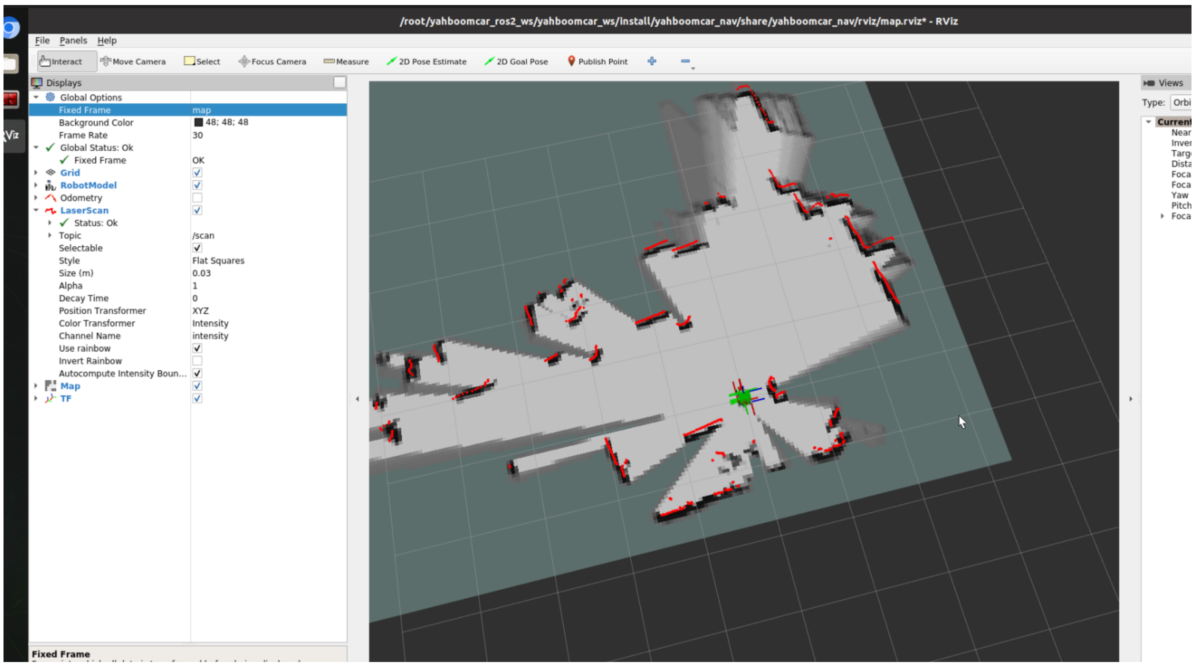
Enter the docker container and execute the following launch file in a terminal:

1. Start mapping

```
ros2 launch yahboomcar_nav map_cartographer_launch.py
```

2. Start rviz to display the map. It is recommended to perform this step in a virtual machine. Multi-machine communication needs to be configured in the virtual machine.

```
ros2 launch yahboomcar_nav display_map_launch.py
```



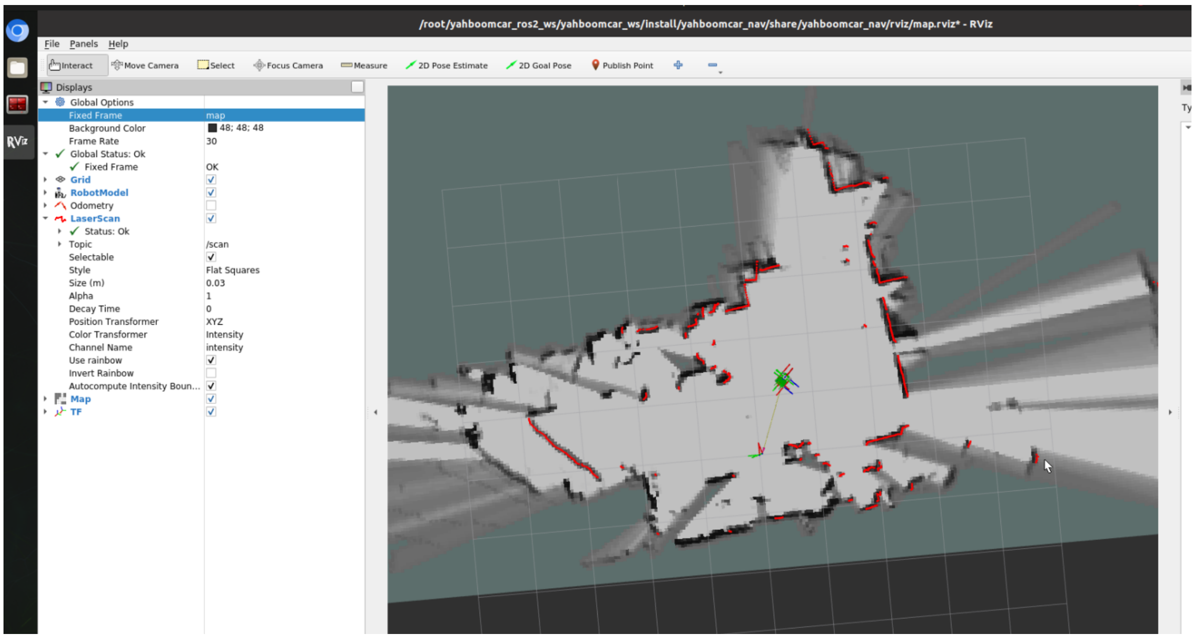
/root/yahboomcar_ros2_ws/yahboomcar_ws/install/yahboomcar_nav/share/yahboomcar_nav/rviz/map.rviz* - RViz

```
Displays
▾ Global Options
    Fixed Frame                        map
    Background Color                   ■ 48; 48; 48
    Frame Rate                         30
▾ ✓ Global Status: Ok
    ✓ Fixed Frame                      OK
▸ Grid                                 ✓
▸ RobotModel                           ✓
▸ Odometry                             ☐
▾ LaserScan                            ✓
  ▸ ✓ Status: Ok
  ▸ Topic                              /scan
    Selectable                         ✓
    Style                              Flat Squares
    Size (m)                           0.03
    Alpha                              1
    Decay Time                         0
    Position Transformer               XYZ
    Color Transformer                  Intensity
    Channel Name                       intensity
    Use rainbow                        ✓
    Invert Rainbow                     ☐
    Autocompute Intensity Boun...      ✓
▸ Map                                  ✓
▸ TF                                   ✓
```

3. Start the keyboard control node. It is recommended to perform this step in a virtual machine. Multi-machine communication needs to be configured in the virtual machine.

Or use the remote control [slowly moving car] to start mapping until a complete map is created

```
ros2 run yahboomcar_ctrl yahboom_keyboard
Or
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```



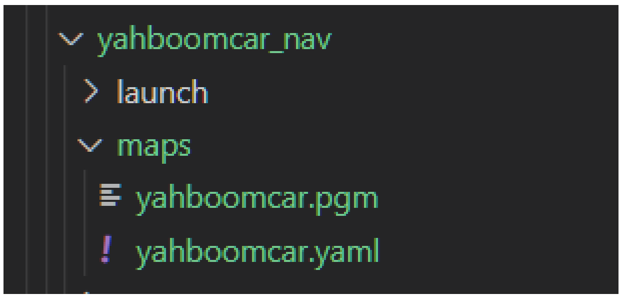
Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
Odometry	☐
LaserScan	✓
✓ Status: Ok	
Topic	/scan
Selectable	✓
Style	Flat Squares
Size (m)	0.03
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	Intensity
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Autocompute Intensity Boun...	✓
Map	✓
TF	✓

4. Save the map with the following path:

```
ros2 launch yahboomcar_nav save_map_launch.py
```

Save path is as follows:

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/
```



A pgm picture, a yaml file yahboomcar.yaml

```
image:
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/yahboomcar.pgm
mode: trinary
resolution: 0.05
origin: [-10, -10, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

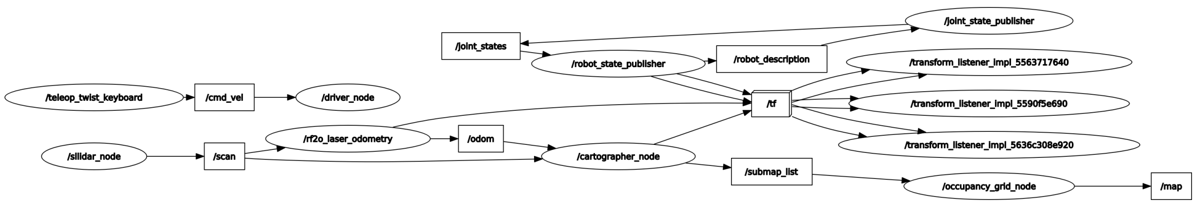
Parameter analysis:

- image: The path of the map file, which can be an absolute path or a relative path.
- mode: This attribute can be one of trinary, scale or raw, depending on the selected mode. trinary mode is the default mode
- resolution: resolution of the map, meters/pixel
- Origin: 2D pose (x, y, yaw) in the lower left corner of the map. The yaw here is rotated counterclockwise (yaw=0 means no rotation). Many parts of the current system ignore the yaw value.
- negate: whether to reverse the meaning of white/black and free/occupied (the interpretation of the threshold is not affected)
- occupied_thresh: Pixels with an occupation probability greater than this threshold will be considered fully occupied.
- free_thresh: Pixels with occupancy probability less than this threshold will be considered completely free.

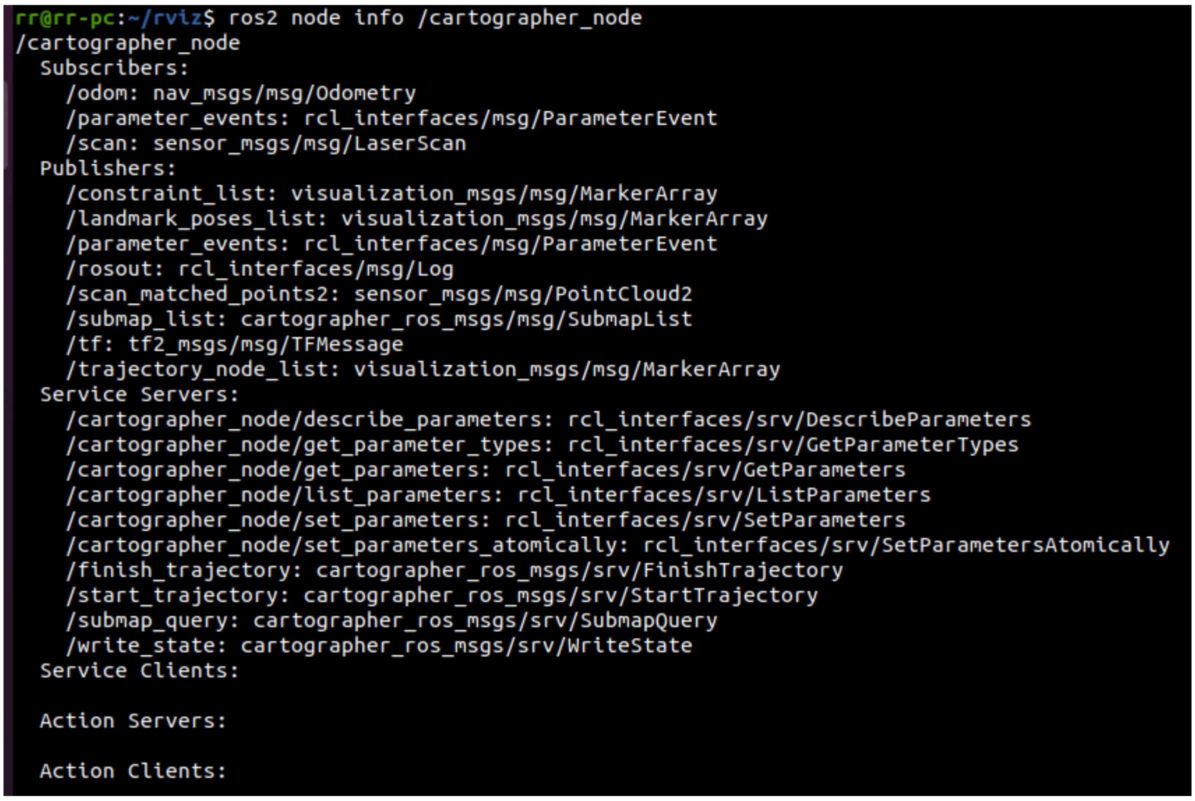
3. Node analysis

3.1. Display calculation graph

```
rqt_graph
```



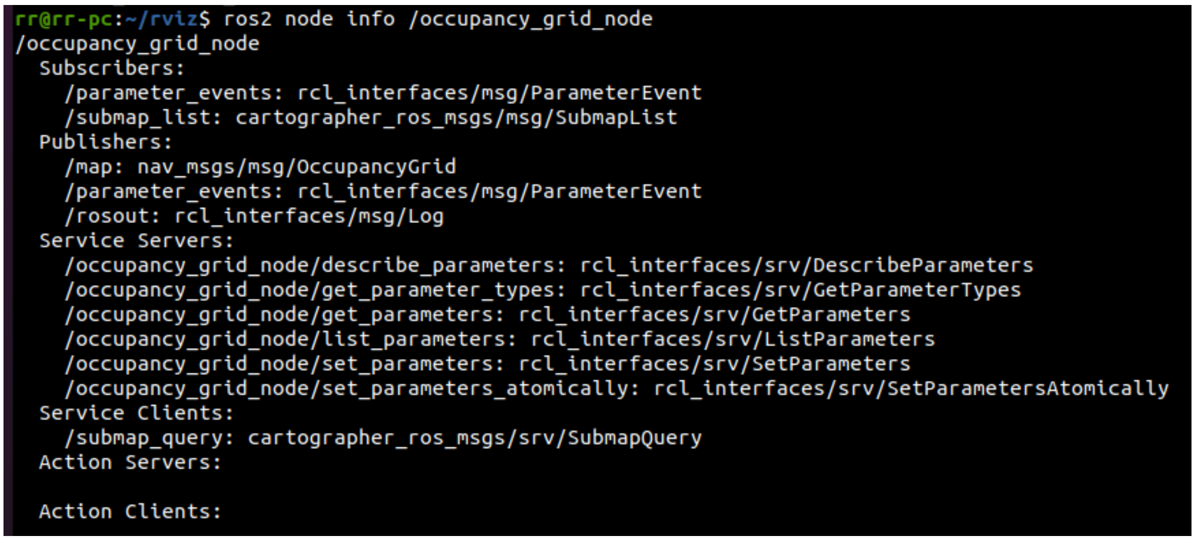
3.2. Cartographer_node node details



```
rr@rr-pc:~/rviz$ ros2 node info /cartographer_node
/cartographer_node
  Subscribers:
    /odom: nav_msgs/msg/Odometry
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /scan: sensor_msgs/msg/LaserScan
  Publishers:
    /constraint_list: visualization_msgs/msg/MarkerArray
    /landmark_poses_list: visualization_msgs/msg/MarkerArray
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /scan_matched_points2: sensor_msgs/msg/PointCloud2
    /submap_list: cartographer_ros_msgs/msg/SubmapList
    /tf: tf2_msgs/msg/TFMessage
    /trajectory_node_list: visualization_msgs/msg/MarkerArray
  Service Servers:
    /cartographer_node/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /cartographer_node/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /cartographer_node/get_parameters: rcl_interfaces/srv/GetParameters
    /cartographer_node/list_parameters: rcl_interfaces/srv/ListParameters
    /cartographer_node/set_parameters: rcl_interfaces/srv/SetParameters
    /cartographer_node/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
    /finish_trajectory: cartographer_ros_msgs/srv/FinishTrajectory
    /start_trajectory: cartographer_ros_msgs/srv/StartTrajectory
    /submap_query: cartographer_ros_msgs/srv/SubmapQuery
    /write_state: cartographer_ros_msgs/srv/WriteState
  Service Clients:

  Action Servers:

  Action Clients:
```

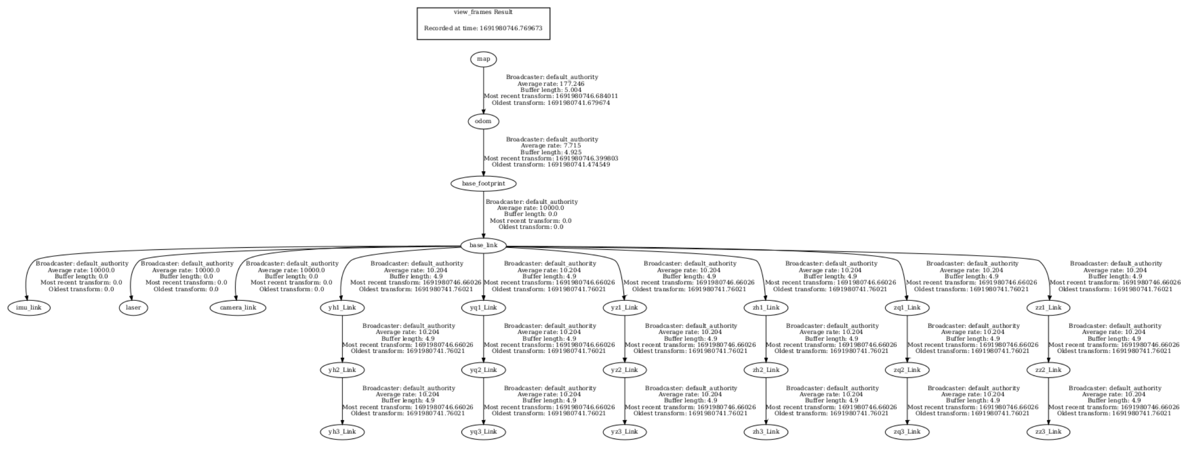


```
rr@rr-pc:~/rviz$ ros2 node info /occupancy_grid_node
/occupancy_grid_node
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /submap_list: cartographer_ros_msgs/msg/SubmapList
  Publishers:
    /map: nav_msgs/msg/OccupancyGrid
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /occupancy_grid_node/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /occupancy_grid_node/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /occupancy_grid_node/get_parameters: rcl_interfaces/srv/GetParameters
    /occupancy_grid_node/list_parameters: rcl_interfaces/srv/ListParameters
    /occupancy_grid_node/set_parameters: rcl_interfaces/srv/SetParameters
    /occupancy_grid_node/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:
    /submap_query: cartographer_ros_msgs/srv/SubmapQuery
  Action Servers:

  Action Clients:
```

3.3. TF transformation

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


view_frames Result
Recorded at time: 1691980746.769673
map
Broadcaster: default_authority
Average rate: 177.246
Buffer length: 5.004
Most recent transform: 1691980746.684011
Oldest transform: 1691980741.679674
odom
Broadcaster: default_authority
Average rate: 7.715
Buffer length: 4.925
Most recent transform: 1691980746.399803
Oldest transform: 1691980741.474549
base_footprint
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
base_link
imu_link
laser
camera_link
yh1_Link
yq1_Link
yz1_Link
zh1_Link
zq1_Link
zz1_Link
yh2_Link
yq2_Link
yz2_Link
zh2_Link
zq2_Link
zz2_Link
yh3_Link
yq3_Link
yz3_Link
zh3_Link
zq3_Link
zz3_Link
Broadcaster: default_authority
Average rate: 10.204
Buffer length: 4.9
Most recent transform: 1691980746.66026
Oldest transform: 1691980741.76021