

# Tutorial on ROS

**Olivier Aycard**

Professor

Grenoble INP - PHELMA

GIPSA Lab

<https://www.gipsa-lab.grenoble-inp.fr/user/olivier.aycard>

*olivier.aycard@grenoble-inp.fr*



# Outline

---

1. Our first node in ROS;
2. Compile and run nodes.

# Our first node

---

- Open a terminal to download the source files needed for tutorial\_ros:

1. `cd ~/ros2_ws/src`
2. `git clone https://gricad-gitlab.univ-grenoble-alpes.fr/tp_ros2/tutorial_ros.git`
3. You should see a folder called tutorial\_ros, containing the source files.

# Our first node

---

In order to demonstrate how a ROS node is implemented, let's look at a simple example which does the following :

- Interfaces with a sensor :
  - receives laser data from the LIDAR scanner
  - stores it in some data structures
  
- Does something with the sensor data:
  - reads the range values
  - displays them as text on the terminal
  
- Sends some data from this node to other nodes :
  - sends graphical display points so that we can display them on the RVIZ graphical interface

# Our first node

---

Let's see how a simple node can be implemented !

- Right click on the folder  
Home/ros2\_ws/src/tutorial\_ros  
and open it with VSCode
- Open src/laser\_graphical\_display\_node.cpp

```
class laser_graphical_display_node : public rclcpp::Node
{
protected:
    // ROS subscriptions
    rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr sub_scan_;

    // Visualization markers
    visualization_msgs::msg::Marker marker_field_of_view_, marker_laser_;

    // ROS publishers
    rclcpp::Publisher<visualization_msgs::msg::Marker>::SharedPtr pub_field_of_view_marker_;
    rclcpp::Publisher<visualization_msgs::msg::Marker>::SharedPtr pub_laser_graphical_display_marker_;

    // Laser scan data
    int nb_beams_;
    bool init_laser_, new_laser_; //to check if new data of laser is available or not
    float range_min_, range_max_;
    float angle_min_, angle_max_, angle_inc_;
    float r_[tab_size], theta_[tab_size];
    geometry_msgs::msg::Point current_scan_[tab_size];
}
```

Creation of a node subclass

Declaring subscribers/publishers to receive/send data between nodes

Data structures to store laser data

# Our first node

---

- Our first node will subscribe to the topic called « scan »;
- « scan » is the topic on which messages are published by the laser, containing the laser data;
- Each time a new message is published on the topic called « scan », our first node will receive and store this message with the method « scanCallback »

```
// Subscribers
// Preparing a subscriber to the "scan" topic, in order to receive data from the laser scanner.
sub_scan = this->create_subscription<sensor_msgs::msg::LaserScan>(
    "scan", qos, std::bind(&laser_graphical_display_node::scan_callback, this, _1));
```

# Our first node

---

- Our first node will publish to the topic called «laser\_graphical\_display\_marker»;
- It is a topic on which messages are published by our node so that other ROS nodes can receive them;
- The type of the message is visualization\_msgs::msg::Marker, which represents visual marker points with colours.

```
// Publishers
// Preparing a topic to publish our results. This will be used by the visualization tool rviz
pub_laser_graphical_display_marker_ = this->create_publisher<visualization_msgs::msg::Marker>("laser_graphical_display_marker", 1);
pub_field_of_view_marker_ = this->create_publisher<visualization_msgs::msg::Marker>("field_of_view_marker", 1);
```

# Our first node

```
void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan) {  
  
    new_laser = true;  
    // store the important data related to laserscanner  
    range_min = scan->range_min;  
    range_max = scan->range_max;  
    angle_min = scan->angle_min;  
    angle_max = scan->angle_max;  
    angle_inc = scan->angle_increment;  
    nb_beams = ((-1 * angle_min) + angle_max)/angle_inc;  
  
    // store the range and the coordinates in cartesian framework of each hit  
    float beam_angle = angle_min;  
    for ( int loop=0 ; loop < nb_beams; loop++, beam_angle += angle_inc ) {  
        if ( ( scan->ranges[loop] < range_max ) && ( scan->ranges[loop] > range_min ) )  
            r[loop] = scan->ranges[loop];  
        else  
            r[loop] = range_max;  
        theta[loop] = beam_angle;  
  
        //transform the scan in cartesian framework  
        current_scan[loop].x = r[loop] * cos(beam_angle);  
        current_scan[loop].y = r[loop] * sin(beam_angle);  
        current_scan[loop].z = 0.0;  
    }  
}  
  
} //scanCallback
```

The type of  
messages  
published by the  
laser



# Our first node

---

- Our first node is an infinite loop that will run at 10 hz
  1. It will check if a new message from the laser has been published (`spin_some`);
  2. If a new message has been received on the scan topic, it will call the method `scanCallback` to collect the data of the laser;
  3. The method « `update` » will process the data of the laser

```
// 4) Taux de boucle à 10Hz
rclcpp::Rate rate(10 /*Hz*/);

// 5) Boucle principale : on tourne les callbacks et on appelle update()
while (rclcpp::ok()) {
    // Exécute une itération de callback sans bloquer
    rclcpp::spin_some(node);
    // Votre méthode d'update périodique
    node->update();
    rate.sleep();
}
```

# Our first node

- The method « update » will check if new message of the laser has arrived with the boolean `new_laser`;
- It will loop over the beams, display their data in the terminal, and publish a marker corresponding to the hits;

```
void laser_graphical_display_node::update() {  
    // If we have received a new laser scan at this node cycle, process it.  
    if ( new_laser_ )  
    {  
        new_laser_ = false;  
  
        RCLCPP_INFO(this->get_logger(), "\n\n New data of laser received");  
  
        std_msgs::msg::ColorRGBA color;  
        marker_laser_.points.clear();  
        marker_laser_.colors.clear();  
  
        // Process the laser data  
        for (int loop_hit = 0; loop_hit < nb_beams_ ; loop_hit++)  
        {  
            // Display laser-hit information in the terminal to understand the data structure and types  
            RCLCPP_INFO(this->get_logger(), "r[%i] = %f, theta[%i] (in degrees) = %f, x[%i] = %f, y[%i] = %f",  
                loop_hit, r_[loop_hit], loop_hit, theta_[loop_hit]*180/M_PI, loop_hit, current_scan_[loop_hit].x, loop_hit, current_scan_[loop_hit].y);  
  
            // Add a marker based on the position of the laser hits in current_scan_  
            marker_laser_.points.push_back(current_scan_[loop_hit]);  
  
            // Determine the color of the marker  
            color.r = 0; color.g = 0; color.b = 1.0; color.a = 1.0;  
            marker_laser_.colors.push_back(color);  
        }  
  
        { ...  
  
        // Publish the marker_message using the Publisher, making it available to other nodes, such as Rviz  
        pub_laser_graphical_display_marker_->publish(marker_laser_);  
  
        // Publish markers showing the lidar's field of view  
        display_field_of_view();  
    }  
}
```

Function to print  
information on the terminal

Add a new point to  
the marker array

Publish the  
the marker array

# Outline

---

1. Our first node in ROS;
2. **Compile and run nodes.**

# Compile and run nodes

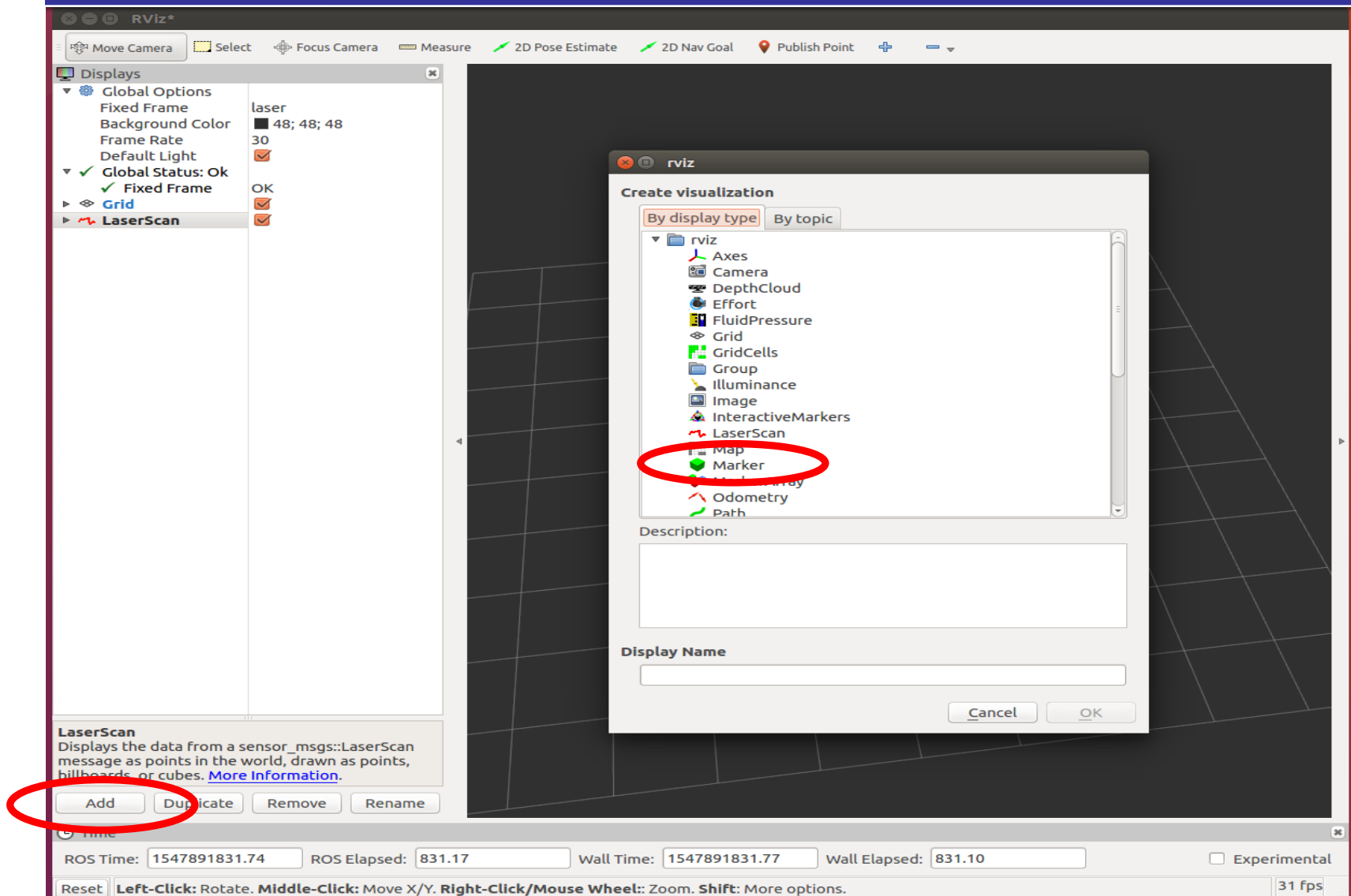
---

Now we've seen what a ROS node looks like, how do we compile and run them ?

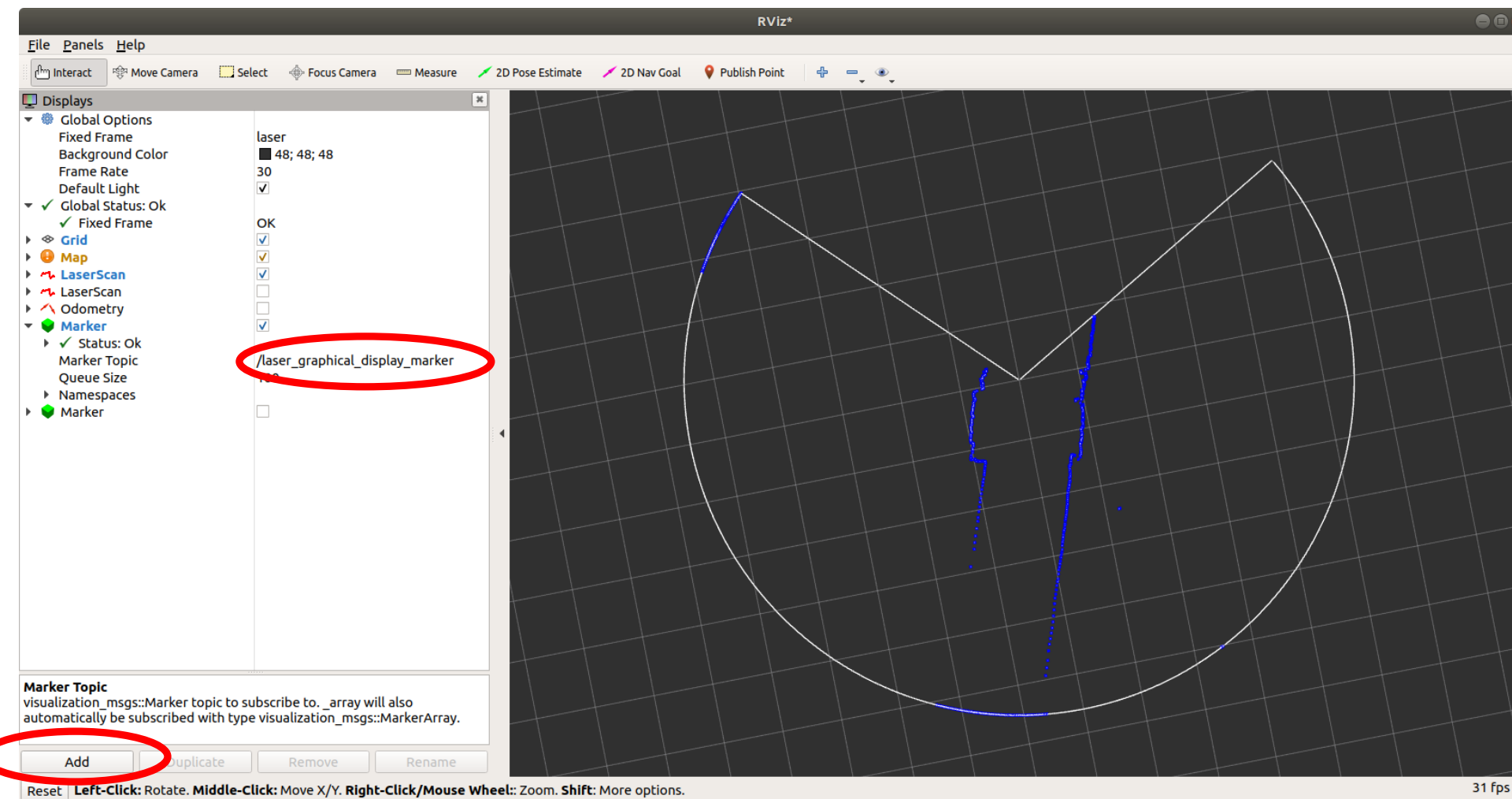
- To compile : open a terminal\*
- `cd ~/ros2_ws`
- `colcon build`
  
- To run : same as any ros2 node : open a terminal
- `cd ~/ros2_ws`
- `source /install/local_setup.bash`  
`ros2 run tutorial_ros laser_graphical_display_node`
  
- You also need two more terminals : one for Rviz and one to play recorded laser data from a rosbag file.

*\*note: the terminal in which you compile using `colcon build` should NOT be one where you have previously executed `source /install/local_setup.bash`. We recommend you always keep the **same terminal** open for compiling, and use it **ONLY** for compiling.*

# Rviz configuration



# Rviz configuration



# Compile and run nodes

---

More general information :

- A folder in `~/ros2_ws/src` is called a package;
- A package contains some source files;
- These source files will be compiled to create nodes;
- To compile packages: `colcon build` in `~/ros2_ws`  
For instance, in the package `tutorial_ros`, there are 2 source files that will generate 2 nodes

- To run a node: `ros2 run <package_name> <node_name>` in `~/ros2_ws`

For instance: `ros2 run tutorial_ros laser_text_display_node`  
runs the node `laser_text_display_node` located in the package `tutorial_ros`