Lab01

Understanding ROS 2: basic tools, topics, nodes

Objectives

The aim of this lab consists of understanding the basic behaviors of ROS 2 nodes and topics.
The main objectives can be summarized as:

- Get to know basic Command Line Interface (CLI) commands.
- Getting familiar with the ROS 2 workspace organization, building, and sourcing packages.
- Using visualization tools (rqt and PlotJuggler).
- Analyzing two simple publisher-subscriber nodes.
- Writing two simple publisher-subscriber nodes.

Requirements

Read and run at least once the following tutorials if you do not have any prior knowledge. These topics are **fundamental** for all the activities that will be done from now on.

- Knowing the basic Linux terminal commands (here).
- Environment configuration (here).
- Getting familiar with git basic commands (here)

The main source of information for any doubt related to ROS should be the official documentation.

Using `turtlesim` and `rqt`

`turtlesim` is a basic **simulator tool** used for learning and demonstrating fundamental ROS concepts. It simulates a simple environment with a turtle that can be controlled through command-line or programming interfaces. The turtle moves based on **velocity commands** and gives **feedback** on its **position**.

`rqt` is a collection of graphical tools for visualizing and interacting with various aspects of the ROS 2 environment. It is a plugin-based interface that allows users to extend its

functionality with different plugins, such as monitoring node activity, inspecting topics, plotting data, and debugging issues within the ROS 2 system.

ACTION: Please follow the official tutorial before going on.

Understanding nodes

In ROS 2, a **node** is a fundamental building block of the system. A node is an executable or process that performs computation and communicates with other nodes. Each node can publish data to topics, subscribe to data from other topics, provide or consume services, and interact with actions.

ACTION: Please follow the official tutorial before going on.

Understanding topics

In ROS 2, a **topic** is a communication channel used by nodes to exchange data asynchronously. Nodes can either **publish** data to a topic or **subscribe** to a topic to receive that data. Topics are used for broadcasting messages to multiple subscribers or receiving information from multiple publishers in a decoupled manner.

ACTION: Please follow the official tutorial before going on.

Create and build your first package

A ROS workspace is a directory with a particular structure. Commonly there is a `src` subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically the directory starts otherwise empty.

`colcon` is the tool commonly used to "**build**" the entire workspace or a specific package. To **"build"** a ROS 2 package means to turn its **source code** (C++, Python, etc.) into **executable** binaries and prepare the package for execution within the ROS 2 ecosystem. By default it will create the following directories as peers of the `src` directory:

- `build` directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.
- `install` directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.
- `log` directory contains various logging information about each colcon invocation.

This is the typical aspect of a workspace after it has been built:

```
ros_ws
  ├── build
  ├── install
  ├── log
  └── src
      └── my_package
```

All ROS 2 Python packages have their own minimum required contents:

- **`package.xml`** file containing meta-information package. The information related to the developer contacts, license and software dependencies are written here.
- **`resource/<package_name>`** marker file for the package (**do not modify**)
- **`setup.cfg`** is required when a package has executables, so **`ros2 run`** can find them (**do not modify**)
- **`setup.py`** containing instructions for how to install the package. Instructions to install new files and executables are written here.
- **`<package_name>`** a directory with the same name as your package, used by ROS 2 tools to find your package, contains **`__init__.py`**. **Your ROS 2 nodes (Python scripts) must be stored here.**

ACTION: Follow carefully this tutorial to create a package and write your first publisher and subscriber nodes.

Plotjuggler

PlotJuggler is a powerful open-source tool used for real-time **data visualization** and analysis. It is especially popular in robotics, automation, and systems development, where large amounts of data from sensors or simulations need to be visualized quickly.

To get this useful tool run the following command in a terminal:

```
sudo apt install ros-humble-plotjuggler-ros
```

Then, to run the program type in a terminal the following command:

```
ros2 run plotjuggler plotjuggler
```

You can find a description of the program functionalities here and how to load ROS 2 topics here.

Exercise

In this exercise, you will create a ROS 2 **package** and build a ROS 2 system with **three nodes**. These nodes will interact to **simulate a robot's movement and localization**, with the ability to **reset** their states. You will also visualize the relevant data using PlotJuggler and analyze the system's behavior using **`rqt_graph`**.

Task 1

Create a package named **`lab01_pkg`**.

Task 2

Create a node called **`controller`**, which publishes velocity commands on a topic called **`/cmd_vel`** of type **`geometry_msgs/msg/Twist`** at a frequency of 1 Hz. The robot always moves at 1 m/s, and its movement follows this rule:

1. N seconds along the X-axis
2. N seconds along the Y-axis
3. N seconds opposite the X-axis
4. N seconds opposite the Y-axis

N starts from 1 and increases by 1 after each set of movements.

Each time you publish a new message, you should print on the shell that you have published a message and its content with the logger (see all logging details and syntax here):

```
self.get_logger().info('Insert here a string to log')
```

You can check all the logging messages using the **`rqt_console`** with the following command:

```
ros2 run rqt_console rqt_console
```

Task 3

Create a node called **`localization`**, which subscribes to the topic **`/cmd_vel`** and estimates the robot's position starting from the axis's origin, considering the topic's period of 1 s and the velocity of 1 m/s. Publish the obtained pose on a topic called **`/pose`** of type **`geometry_msg/msg/Pose`**. Each time you publish a new message, print it on the shell with the logger as in Task 2.

Task 4

- Plot the following data with plotjuggler:
  - velocity along X-axis
  - velocity along Y-axis
  - position along X-axis
  - position along Y- axis
  - position on XY plane
- Analyze the ROS 2 graph using **`rqt_graph`**.

Task 5

Create a node called **`reset_node`** that subscribes to **`/pose`**. When the distance from the origin of the reference frame is larger than 6.0 m, publish a boolean value **`std_msgs/msg/Bool`** (True or False, as you wish) on the topic **`/reset`** to take care of this limit condition and reset the node.

Task 6

Create a copy of the **`controller`** and the **`localization`** nodes, called **`controller_reset`** and **`localization_reset`**, and modify them.
When the controller receives a message on the **`/reset`** topic, reset itself and start the control sequence again with N = 1. When **`localization_reset`** receives a message on the **`/reset`** topic, reset the position to the origin.
Both nodes must publish a message on the terminal when such an operation is performed.

Task 7

- Plot the following data with plotjuggler:
    - velocity along X-axis
    - velocity along Y-axis
    - position along X-axis
    - position along Y- axis
    - position on XY plane
- Analyze the ROS 2 graph using **`rqt_graph`**.