Lab01 - 26/10/2023

Understanding ROS 2: basic tools, topics, ROS 2 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.

Tutorial

In this part of the lab, you must follow **all the steps** to familiarize yourself with the tools needed for the final exercise and the following activities.

For further insights, you should study the official ROS 2 Humble documentation (https://docs.ros.org/en/humble/index.html).

A. Requirements

Read and run at least once the following tutorials to know better the fundamentals of ROS 2 environment configuration, nodes, and topics.

- Knowing the basic Linux terminal commands (here).
- Environment configuration (here).
- Using `turtlesim`, `ros2` and `rqt` (here).
- Basic knowledge of the nodes and how to interact with them with CLI (here).
- Basic knowledge of the topics and how to interact with them with CLI (here).
- Using `colcon` to build packages (here).
- Getting familiar with git basic commands (here)

Before starting the lab, please know that if two or more PCs use ROS with the same ROS_DOMAIN_ID in certain network environments, they can share topics. While this might be fine within PoliTO's network due to its restrictions on user communication, you might encounter this problem when using ROS at home. To avoid potential conflicts, set your ROS_DOMAIN_ID as follows:

```
export ROS_DOMAIN_ID=<your_domain_id>
```

To maintain settings between shell sessions, you can add the command to your shell startup script:

```
echo "export ROS_DOMAIN_ID=<your_domain_id>" >>  ~/.bashrc
```

You can find more details on the ROS domain ID here.

B. Creating a workspace

For the full guide, we recommend reading here.

A workspace is a directory containing ROS 2 packages. Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal.

You can also source an "overlay" - a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending or "underlay". Your underlay must contain the dependencies of all the packages in your overlay. Packages in your overlay will override packages in the underlay.

For example if you want to run the packages you will create you need to source first your ROS 2 installation, which will be your underlay. Then, you need to source the workspace, which will represent your overlay. The packages that populate the workspace will have some dependencies (i.e.: other packages or libraries) which need to be installed in the underlay.

1. Source the ROS 2 environment

In this tutorial and for every exercise you run, your ROS 2 installation will be your underlay. Therefore, you always need to source your ROS 2 when you open a new terminal:

```
source /opt/ros/humble/setup.bash
```

To avoid writing the same command every time you open a new shell, we recommend inserting the sourcing in the `.bashrc` file, which is executed each time a new terminal is started:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

2. Clone a sample repo

If you read and run the previous tutorials, you'll be familiar with the `turtlesim` package. Make sure that you are in the `src` directory and run the following command:

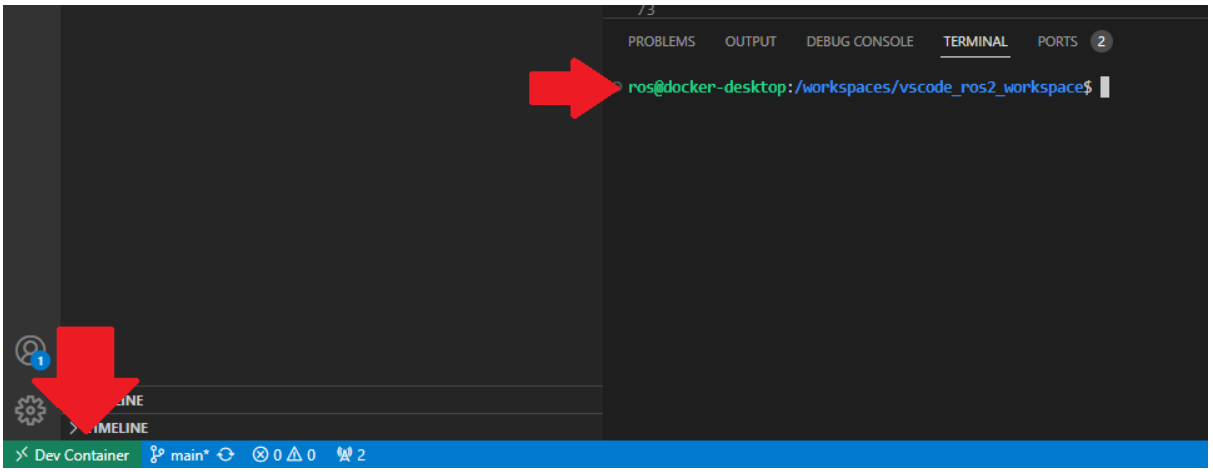
```
git clone https://github.com/ros/ros_tutorials.git -b humble
```

With this command, you cloned a repository from GitHub. Since a repo can have many branches, you need to check out the one that targets your installed ROS 2 distro. To address a particular branch, you add the `-b` argument after the command specifying the name of the branch `humble`.

3. Resolve dependencies

Before building the workspace, you need to resolve the package dependencies. You may have all the dependencies already, but best practice is to check for dependencies every time you clone.

Before running the following commands, make sure to be in the root of your workspace (`vscode_ros2_workspace`) and be sure to be inside the Dev Container if you are using it.



If you haven't initialized yet, run the command:

```
sudo apt update
sudo rosdep init
rosdep update
```

Then, run:

```
rosdep install -i --from-path src --rosdistro humble -y
```

If you already have all your dependencies installed, the console will return the following:

```
#All required rosdeps installed successfully
```

4. Build the workspace with colcon

From the root of your workspace, you can build your packages using:

```
colcon build
```

And the console will return something like this:

```
Starting >>> turtlesim
Finished <<< turtlesim [5.49s]

Summary: 1 package finished [5.58s]
```

There are some useful arguments for `colcon build`:

- `--symlink-install` saves you from having to rebuild every time you tweak Python scripts
- `—-packages-select` allows you to build only a list of selected packages.

If you list the contents of the `vscode_ros2_workspace` folder with the ls command, you'll obtain the following output:

```
build install log src
```

We strongly recommend understanding their purpose.

5. Source the overlay

Before sourcing the overlay, it is very important that you open a new terminal separate from the one where you built the workspace. Sourcing an overlay in the same terminal where you built, or likewise building where an overlay is sourced, may create complex issues.

In the new terminal, move to your ws root folder, source your main ROS 2 environment as the "underlay" (if you did not add the source command to your .bashrc) so you can activate the overlay ws "on top of" it:

```
source install/local_setup.bash
```

Sourcing the `local_setup` of the overlay will only add the packages available in the overlay to your environment. If you source `setup`, you source the overlay and the underlay it was created in, allowing you to use both workspaces.

Now you can run the `turtlesim` package from the overlay:

```
ros2 run turtlesim turtlesim_node
```

In summary, every time you create a new package, you will make a new directory in `src` and then build it. To run a node contained in the package, you must open a new terminal, source the underlay and the overlay and finally run the node.

C. Creating a package

Please read and follow the full guide (here), considering **Python** packages to understand the basic structures of packages and their content.

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

- `package.xml` file containing meta-information package
- `resource/<package_name>` marker file for the package
- `setup.cfg` is required when a package has executables, so `ros2 run` can find them
- `setup.py` containing instructions for how to install the package
- `<package_name>` a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

You must read the documentation and examine `package.xml`, `setup.cfg` and `setup.py` to understand what you need to modify if you create your own package.

D. Writing a simple publisher and subscriber

You will now create a simple Python package containing two nodes that pass the information as a string message to each other. In this case, one node, called 'talker', will publish data and another, called 'listener', will receive them.

1. Create a new package

Every time you need to create a new Python package, you need to navigate to the `src` directory located in the root of your ws and execute the following command:

```
ros2 pkg create --build-type ament_python <package_name>
```

In this case, the name of the package will be `py_pubsub`.

2. Writing the publisher/subscriber code and analysing the code

You should follow the ROS 2 official guide here.
Analyze the code carefully and notice all the modifications applied to all the files. At the end of this step, you're expected to verify the functioning of the two nodes and the communications between them.

E. PlotJuggler

To plot the topic data, you must install and use PlotJuggler.
To install it, write on the terminal:

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

You can run it as a normal ROS 2 node (enjoy the meme):

```
ros2 run plotjuggler plotjuggler
```

Following this tutorial to know its functions and learn how to use it is recommended.
You must increase the buffer value if you need to plot for longer times.

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 velocities on a topic called `/cmd_topic` 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've published a message and its content with the logger (see all logging details and syntax here):

```
self.get_logger().info('<Logger message>')
```

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 `/cmd_topic` 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 system's behavior using `rqt_graph` and discuss the node interaction.

Task 5

Publish all the previous log messages with the command:

```
self.get_logger().debug('<Logger message>')
```

Check for differences in log output and explain their implications.

Task 6

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 (True or False, as you wish) on the topic `/reset` to take care of this limit condition and reset the node.

Task 7

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 with the log level equal to `info` when such an operation is performed. Adjust the logging settings to see all log messages and only report the reset messages.

Task 8

Use PlotJuggler to visualize the following data after implementing the reset functionality:

- velocity along X-axis
- velocity along Y-axis
- position along X-axis
- position along Y-axis
- position on XY plane.

Analyze the system's behavior using `rqt_graph` and discuss the node interaction after the modification.

Report requirements

At the end of LAB03, you must write a comprehensive report of the first 3 lab activities.

Request for each task:

- Task 1: nothing
- Task 2: describe the implementation of the controller node. Show the terminal log and the echo of the published topic.
- Task 3: describe the implementation of the localization node. Show the terminal log and the echo of the published topic.
- Task 4: show the plots and the node graph.
- Task 5: did you find any difference in the terminal output?
- Task 6: describe the design of the reset node.
- Task 7: describe the nodes design and report an example of successful reset.
- Task 8: show the plots and the node graph.

You should also deliver on the Portale della Didattica a zipped folder containing the package that you developed, named:

`sesars_lab<lab_number>_<group_number>.zip`