UNIVERSIDAD DE MÁLAGA | uma.es

S2 – PLAYING WITH ROS2

José Raúl Ruiz Sarmiento
Departamento de Ingeniería de Sistemas y Automática

LECTURE CONTENTS

1. PLAYING WITH THE TERMINAL

   Checking the installation, turtlesim, rqt, rqt_graph.

2. DEVELOPING SOFTWARE

   Colcon, Workspaces, Creating packages and nodes (CMake and python).

1. PLAYING WITH ROS 2

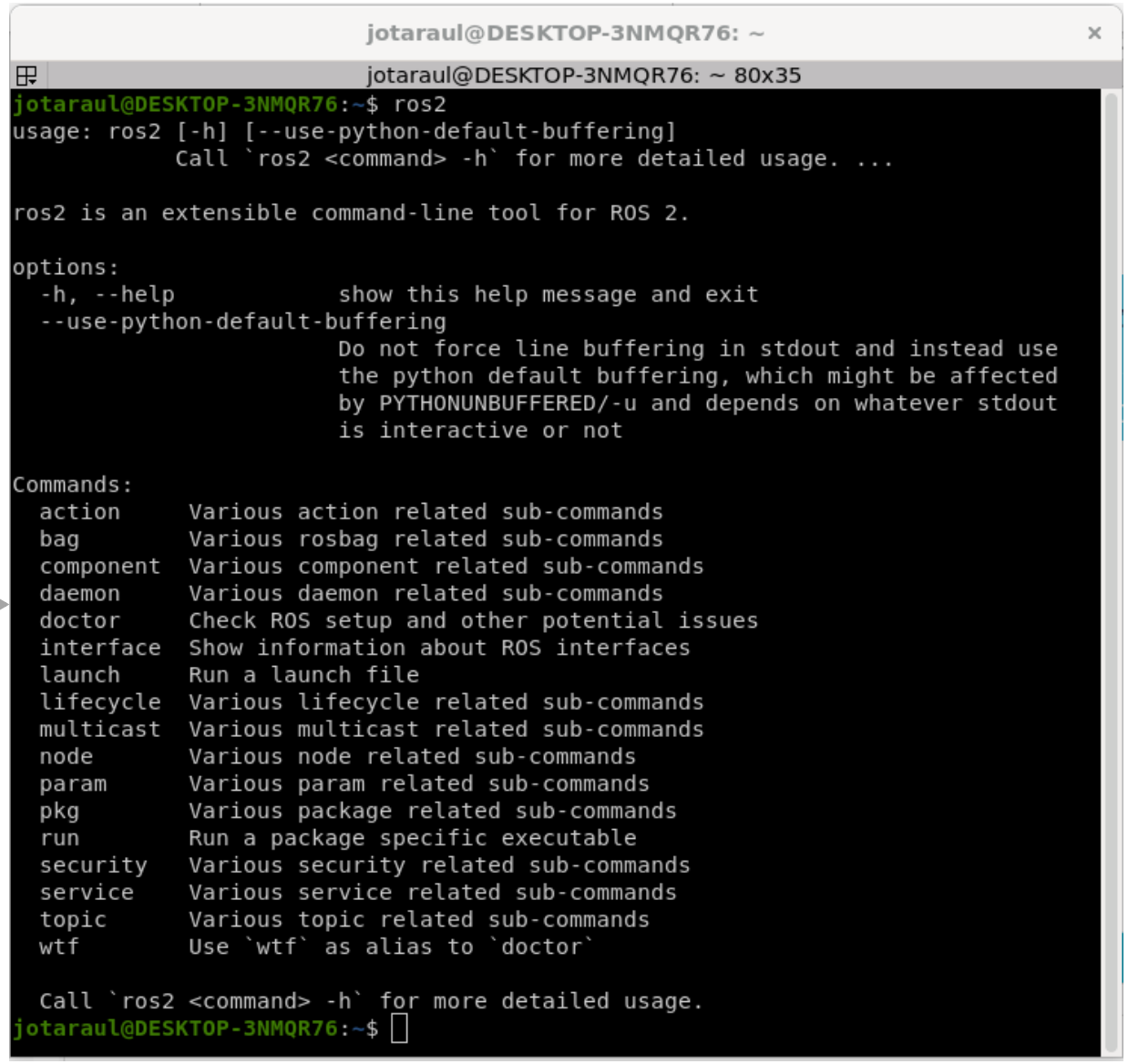
1. PLAYING WITH THE TERMINAL
CHECKING THE INSTALLATION

- Let's start by checking that ROS2 exists...

  $ ros2

  If this doesn't work, your in troubles!

ROS 2 includes a suite of command-line tools for introspecting a ROS 2 system, where the main entry point for the tools is the command *ros2*, which itself has various sub-commands for introspecting and working with nodes, topics, services, and more.



1. PLAYING WITH THE TERMINAL
CHECKING THE INSTALLATION

- Now check that ROS2 environment variables are loaded:

```
$ printenv | grep ROS
```

This should display ROS 2-related environment variables, such as:

Note: If nothing is displayed, then something happened during the installation. These variables are loaded automatically each time a new terminal is created, since you added this line to your *.bashrc* file:

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

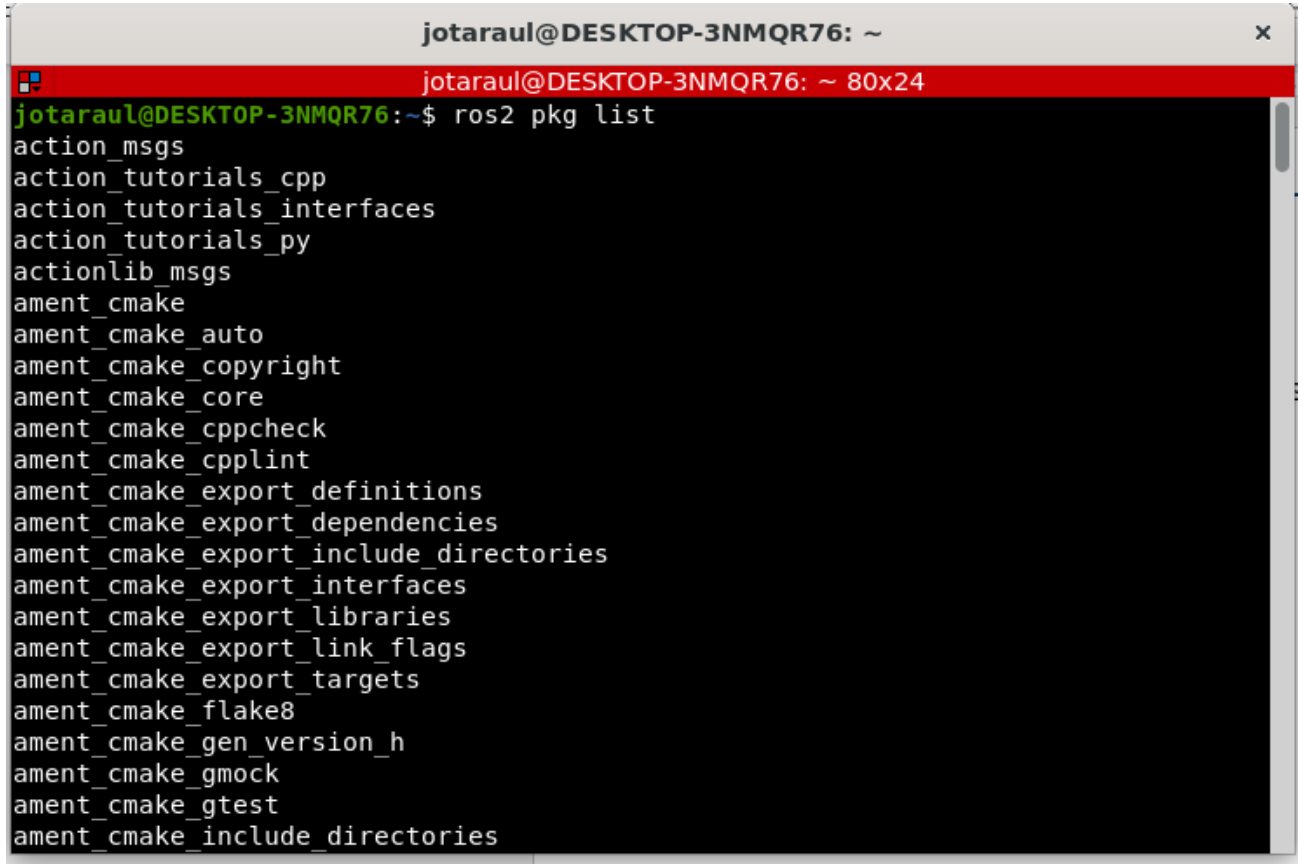
To load these variables manually:

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

1. PLAYING WITH THE TERMINAL
CHECKING THE INSTALLATION

- Run the following command to check basic ROS 2 CLI (Command Line Interface) functionality:

```
$ ros2 pkg list
```



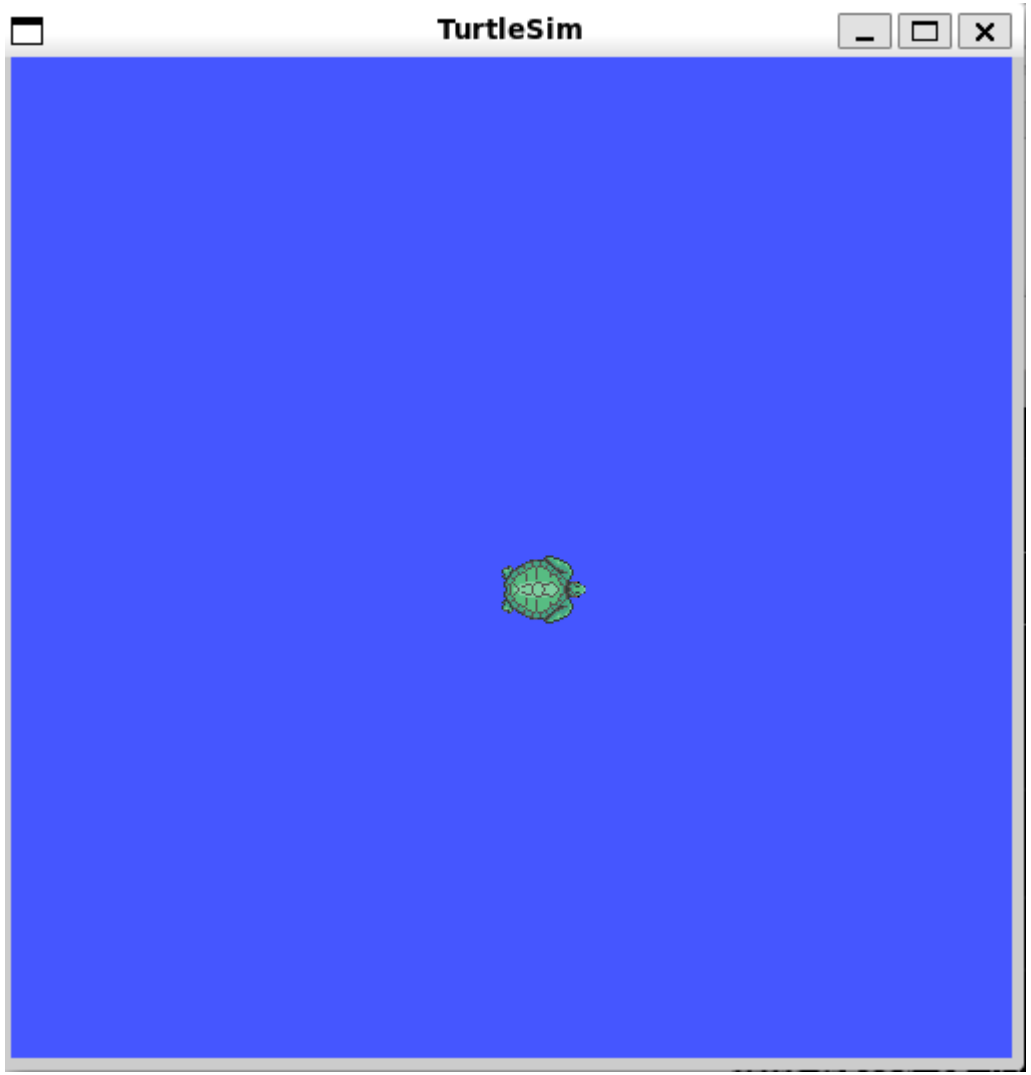
1. PLAYING WITH THE TERMINAL
TURTLESIM

- Now run Turtlesim, a lightweight simulator for learning ROS 2. It illustrates what ROS 2 does at the most basic level, to give you an idea of what you will do with a real robot or robot simulation later on.

```
$ ros2 run turtlesim
    turtlesim_node
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 run turtlesim turtlesim_node
[INFO] [1733068966.289708926] [turtlesim]: Starting turtlesim with
node name /turtlesim
[INFO] [1733068966.297258044] [turtlesim]: Spawning turtle [turtle1
] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

Default turtle's name



1. PLAYING WITH THE TERMINAL
TURTLESIM

This terminal should be active to control the turtle!

- We are going to teleoperate the turtle, so launch the following node:

```
$ ros2 run turtlesim turtle_teleop_key
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 run turtlesim turtle_teleop_
key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations.
'F' to cancel a rotation.
'Q' to quit.
```

- Ok we now have two nodes running. Which will be the available topics? Run in a new terminal:

```
$ ros2 topic list
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

- Move the robot and check the motion commands that are being sent with:

```
$ ros2 topic echo /turtle1/cmd_vel
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 topic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

1. PLAYING WITH THE TERMINAL
TURTLESIM

- As you know, in ROS2 the basic communication mechanism are topics. Let's publish a motion command directly using CLI tools:

```
$ ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0}, angular: {z: 1.0}}"
```

Explanation of the Command:
- `/turtle1/cmd_vel`: The topic controlling the turtle's motion.
- `geometry_msgs/msg/Twist`: The message type used to define linear and angular velocity.
- `{linear: {x: 2.0}, angular: {z: 1.0}}`: The specific velocity values being sent:
  - `linear.x = 2.0`: Moves the turtle forward at a speed of 2 units/sec.
  - `angular.z = 1.0`: Rotates the turtle counterclockwise at a speed of 1 radian/sec.
- You can adjust `linear.x` and `angular.z` to control the turtle's speed and direction.

- Try different velocities and, finally, **stop the robot!**

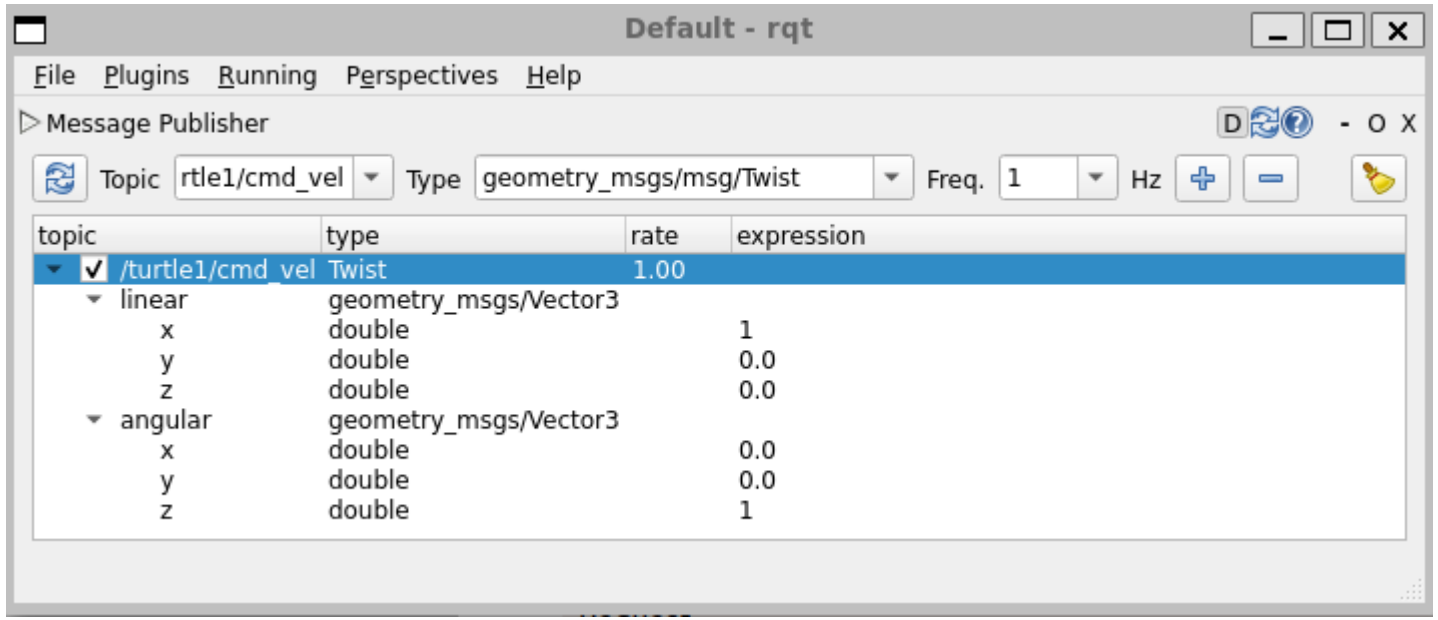
1. PLAYING WITH THE TERMINAL
RQT

It may take some time for rqt to locate all the plugins. If you click on Plugins but don't see Services or any other options, you should close rqt and enter the command `rqt --force-discover` in your terminal.

- To be publishing commands, calling services and actions, etc,, through CLI... is a pain on the neck!

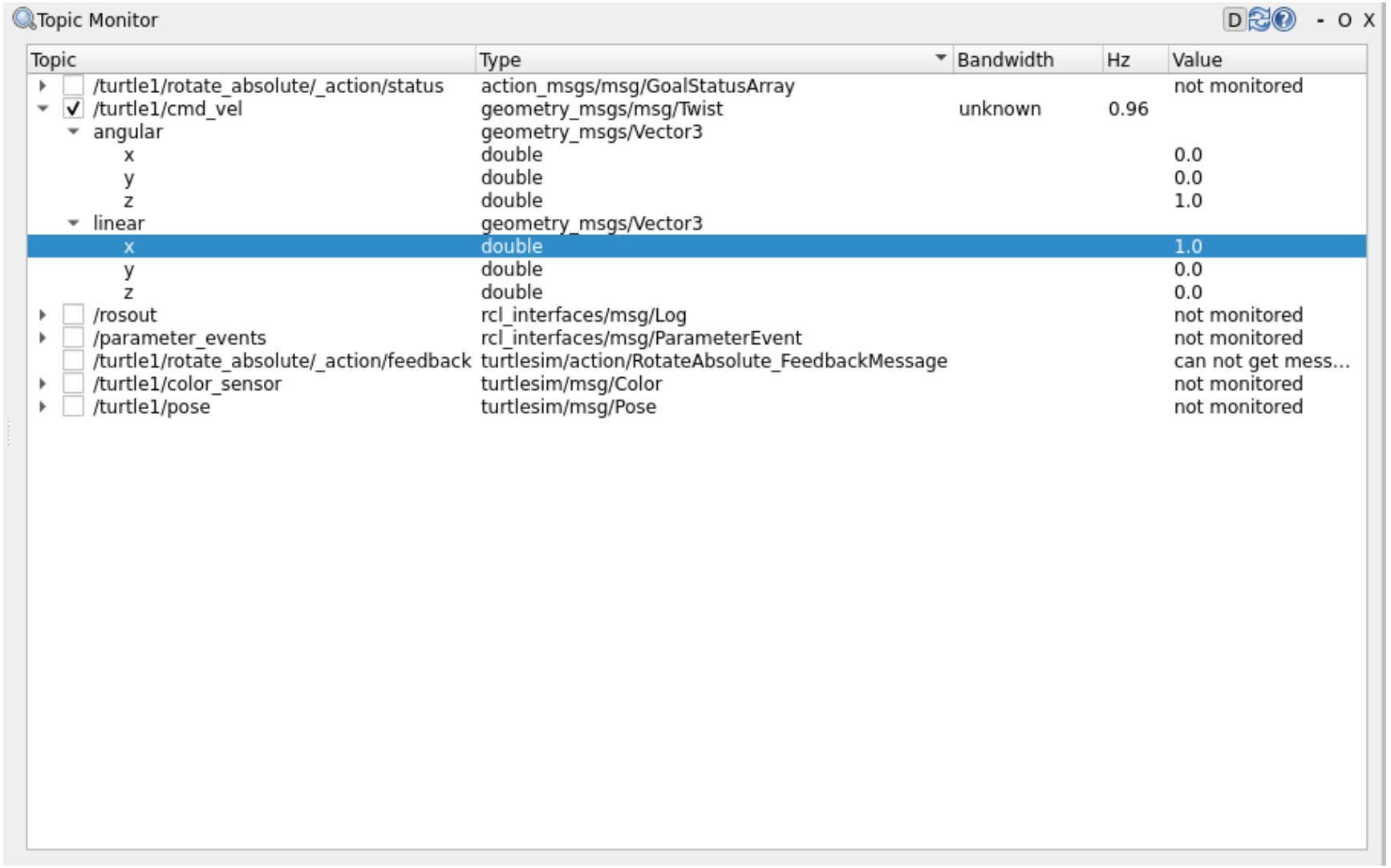
- Fortunatelly ROS2 provides rqt, a graphical user interface (GUI) tool. Everything done in rqt can be done on the command line, but rqt provides a more user-friendly way to manipulate ROS 2 elements. Do the following:

1. Run `$ rqt`.
2. Click on `Plugins > Topics > Message publisher`.
3. Select `/turtle1/cmd_vel`.
4. Click the "+" icon (Add new publisher) and edit the message.
5. Right click `Public selected once`.
6. Mark the box to keep publishing the message.



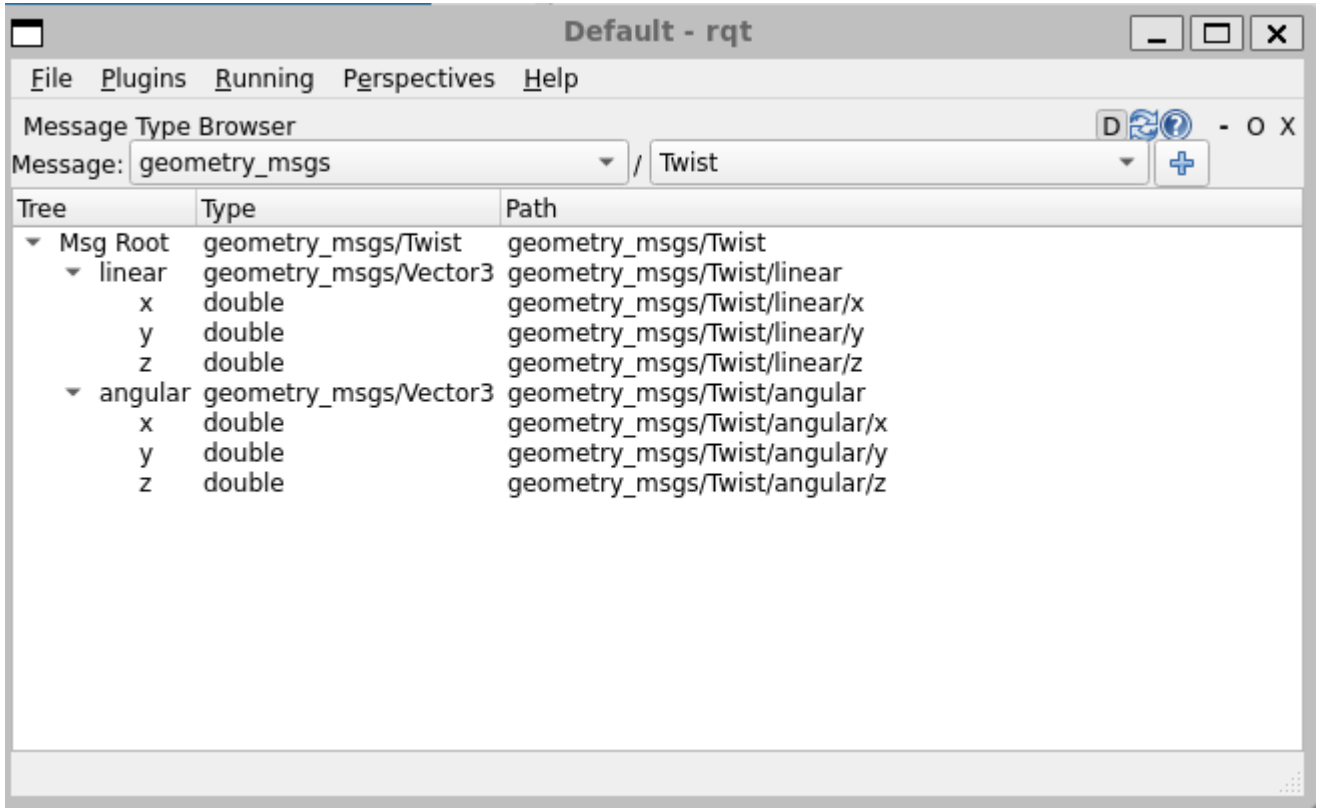
1. PLAYING WITH THE TERMINAL
RQT

- There is another plugin called `Topic monitor`! Open it and explore the values of published messages on topics and statistics like bandwidth, frequency of publication (Hz), etc.



1. PLAYING WITH THE TERMINAL
RQT

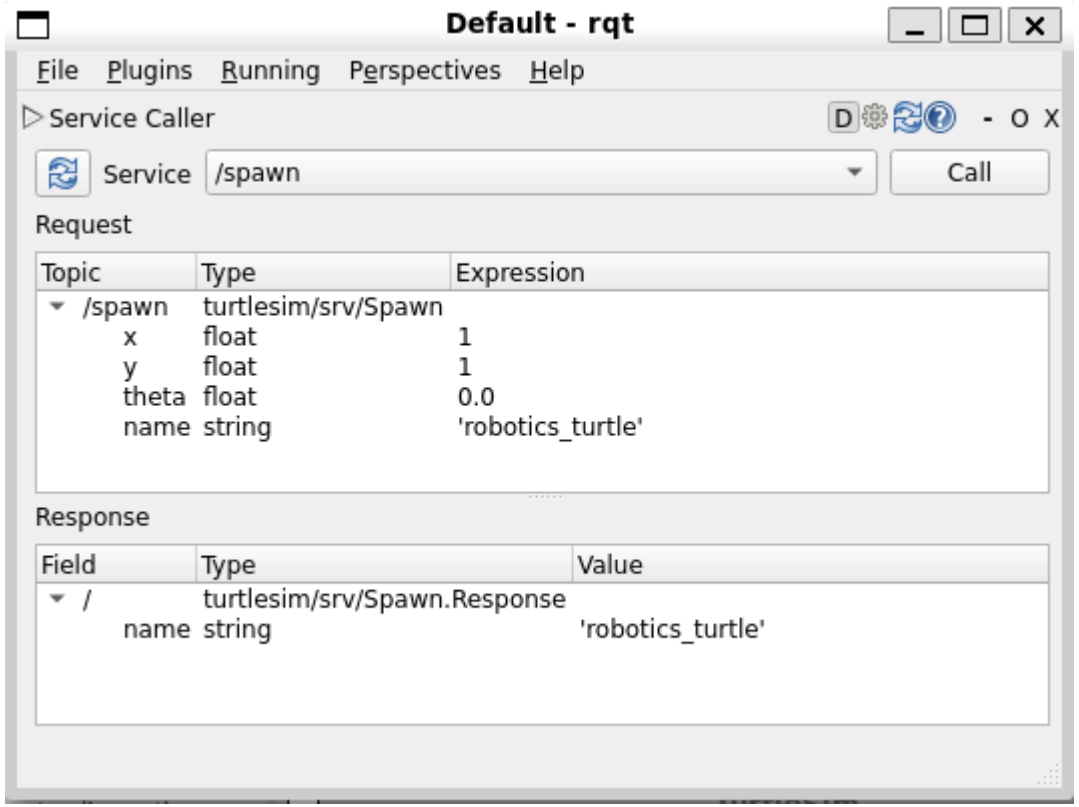
- And finally, play a bit with the `Message Type Browser` one, which permits you to inspect the messages' definition. So useful!



1. PLAYING WITH THE TERMINAL
RQT

- Now, let's call a service from this nice GUI. For that:
  1. Open `Plugins > Services > Service Caller`.
  2. Select the `/spawm` service.
  3. Edit the message with a unique name and position.
  4. Click on "Call".

- If you refresh the services' list, you will see now some appearing with the name of your turtle.



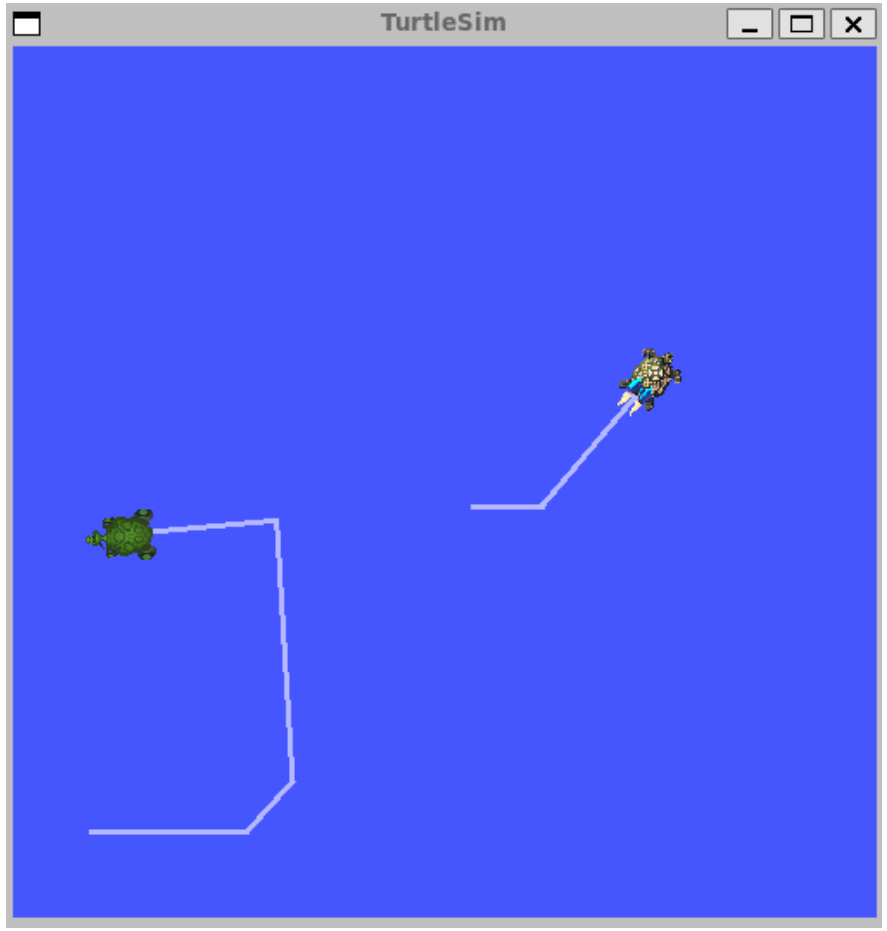
1. PLAYING WITH THE TERMINAL
TURTLESIM

- You need a second teleop node in order to control turtle2 (or the name you used). However, if you try to run the same command as before, you will notice that this one also controls turtle1. The way to change this behavior is by remapping the `cmd_vel` topic.

  In a new terminal, run:

```
ros2 run turtlesim turtle_teleop_key --ros-
  args --remap
  turtle1/cmd_vel:=turtle2/cmd_vel
```

  Now, you can move turtle2 when this terminal is active, and turtle1 when the other terminal running `turtle_teleop_key` is active.

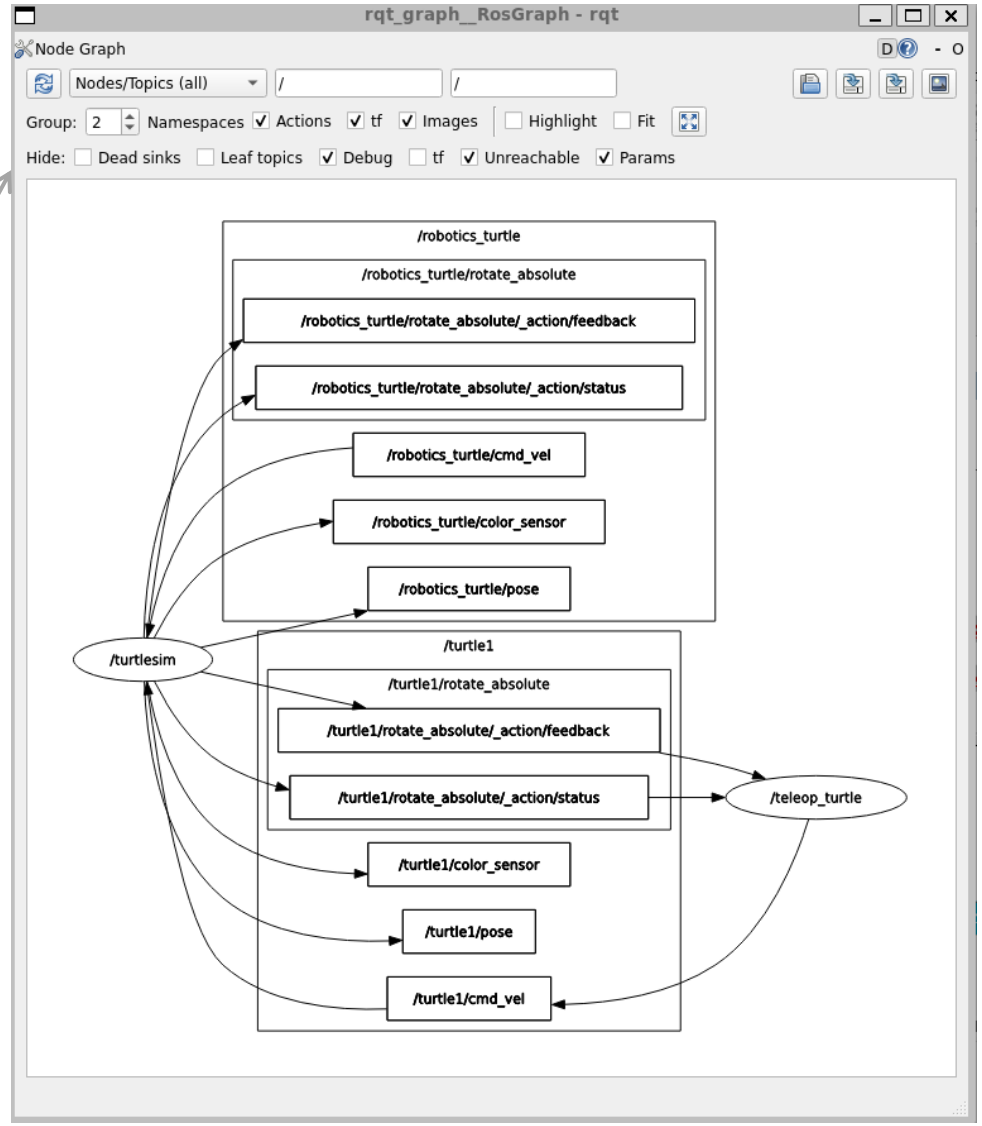


1. PLAYING WITH THE TERMINAL
RQT GRAPH

- Finally, let's launch rqt_graph to visually check how these nodes interact.

```
$ rqt_graph
```

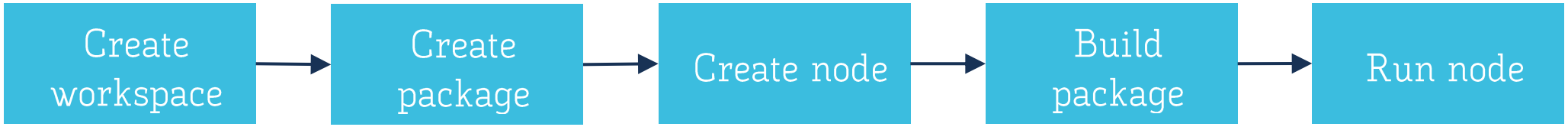
Within the rqt_graph interface you can play with the items to show/hide



2. ROS 2 – DEVELOPING SOFTWARE

2. ROS 2 – DEVELOPING SOFTWARE
COLCON

- When you become a ROS developer, you aim to implement new nodes, which are organized into packages.
- colcon: official build tool of ROS based on Cmake and python. It is in charge of:
  - managing dependences among nodes,
  - organizing and making packages accessible, and
  - generating executables from source code.
- Workflow (from workspace to node):



2. ROS 2 – DEVELOPING SOFTWARE
COLCON WORKSPACE

- A **workspace** is a directory where colcon operates to build packages, with the following directories:
  - src: place to locate the source code of ROS 2 packages.
  - build: intermediate files are stored here.
  - install: contains the installation of each package.
  - log: contains diverse login information.
- The following commands create a workspace called `dev_ws` and its `/src` directory, move to it, calls to colcon build (this creates and initial set of directories and files), and list the resulting directories. With this we get a workspace ready to work with!

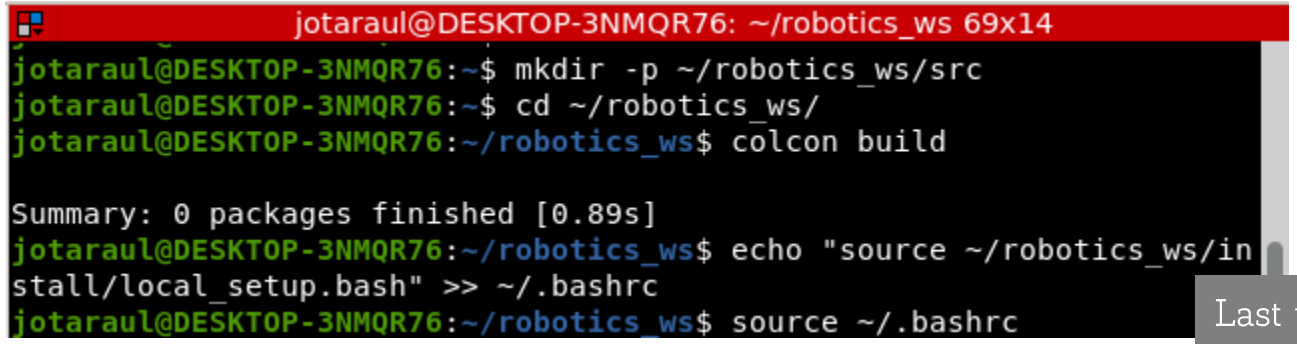
```
File  Edit  View  Search  Terminal  Tabs  Help
robotics@humbl...  robotics@humbl...  robotics@humbl...  robotics@humbl...

robotics@humble:~$ mkdir -p ~/dev_ws/src
robotics@humble:~$ cd ~/dev_ws/
robotics@humble:~/dev_ws$ colcon build

Summary: 0 packages finished [0.30s]
robotics@humble:~/dev_ws$ ls
build  install  log  src
robotics@humble:~/dev_ws$ echo "source ~/dev_ws/install/local_setup.bash" >> ~/.bashrc
robotics@humble:~/dev_ws$ source ~/.bashrc
```

Last two lines add useful ROS 2 environment variables (e.g. where to look for installed executables) to bashrc. **MUST BE DONE ONLY ONCE!**

2. ROS 2 – DEVELOPING SOFTWARE
COLCON WORKSPACE

- Let's create our first workspace! For that use the following commands:

```
$ mkdir -p ~/robotics_ws/src
$ cd ~/robotics_ws/
$ colcon build
$ ls
$ echo "source ~/robotics_ws/install/local_setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```



Last two lines add useful ROS 2 environment variables (e.g. where to look for installed executables) to bashrc. **MUST BE DONE ONLY ONCE!**

2. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

- Typical structure of the workspace:

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt       -- CMakeLists.txt file for package_1
          package.xml          -- Package manifest for package_1

      package_2/
          setup.py             -- Installation instructions for package_2
          package.xml          -- Package manifest for package_2
          resource/package_2
      ...
      package_n/
          CMakeLists.txt       -- CMakeLists.txt file for package_1
          package.xml          -- Package manifest for package_n
```

Remind:
- A single workspace can contain as many packages as you want.
- A package can have different build types (e.g. CMake or Python).
- No nested packages are allowed.

Remind:
- To build a package, just go to the worksapce root directory and call colcon with `colcon build`.
- To build only package_1: `coclon build --packages-select package_1`

2. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

Run it only inside the `src` folder!!

- ros2 pkg create command, with **CMake:**

More convenient when developing C++ nodes

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

```
File  Edit  View  Search  Terminal  Help
robotics@humble:~/ros2_ws/src$ ros2 pkg create --build-type
ament_cmake --node-name my_node my_package_cmake
going to create a new package
package name: my_package_cmake
destination directory: /home/robotics/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['robotics <humble@example.com>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: []
node_name: my_node
creating folder ./my_package_cmake
creating ./my_package_cmake/package.xml
creating source and include folder
creating folder ./my_package_cmake/src
creating folder ./my_package_cmake/include/my_package_cmake
creating ./my_package_cmake/CMakeLists.txt
creating ./my_package_cmake/src/my_node.cpp

[WARNING]: Unknown license 'TODO: License declaration'.  Thi
s has been set in the package.xml, but no LICENSE file has b
een created.
It is recommended to use one of the ament license identitife
rs:
Apache-2.0
BSL-1.0
```

This creates a directory with the same name and some initial content:

```
File  Edit  View  Search  Terminal  Help
robotics@humble:~/ros2_ws/src/my_package_cmake$ ls
CMakeLists.txt  include  package.xml  src
robotics@humble:~/ros2_ws/src/my_package_cmake$
```

In this case, as we called *ros2 pkg create* with the *–node-name* option, a node called my_node with a simple Hello World executable is created.

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (CMAKE)

1. Create a new source file in the src folder of the package. For example, `my_node.cpp`
2. Modify the CMakeList.txt at `/src/my_package_cmake/` to include a reference to a such file. This tells the ROS 2 system we want to have a new executable (node).

```
 8 # find dependencies
 9 find_package(ament_cmake REQUIRED)
10 find_package(rclcpp REQUIRED)
11 find_package(std_msgs REQUIRED)
12
13 add_executable(my_node src/my_node.cpp)
14 ament_target_dependencies(my_node rclcpp std_msgs)
15
16 install(TARGETS my_node
17   DESTINATION lib/${PROJECT_NAME})
```

3. Update package.xml. Add the necessary build, execution and testing dependencies
4. Develop the node itself. In our example, implement in `my_node.cpp` the desiserd functionality. You can use a simple text editor or an IDE (e.g. Visual Studio Code).
5. Compile the source code and generate the node executable.

```
$ ~/ros2_ws
$ colcon build
```

5. Run your node! `ros2 run my_package_cmake my_node`

Depending on your configuration, you may previously need to go to your workspace root directory and source the setup files:
`. install/setup.bash`

2. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

Run it only inside the `src` folder!!

- ros2 pkg create command, with **Python:**

More convenient when developing python nodes

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

```
File  Edit  View  Search  Terminal  Help
Either remove the directory or choose a different destinatio
n directory or package name
robotics@humble:~/ros2_ws/src$ ros2 pkg create --build-type
ament_python --node-name my_node my_package_python
going to create a new package
package name: my_package_python
destination directory: /home/robotics/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['robotics <humble@example.com>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
node_name: my_node
creating folder ./my_package_python
creating ./my_package_python/package.xml
creating source folder
creating folder ./my_package_python/my_package_python
creating ./my_package_python/setup.py
creating ./my_package_python/setup.cfg
creating folder ./my_package_python/resource
creating ./my_package_python/resource/my_package_python
creating ./my_package_python/my_package_python/__init__.py
creating folder ./my_package_python/test
creating ./my_package_python/test/test_copyright.py
creating ./my_package_python/test/test_flake8.py
creating ./my_package_python/test/test_pep257.py
creating ./my_package_python/my_package_python/my_node.py
```

This creates a directory with the same name and some initial content:

```
File  Edit  View  Search  Terminal  Help
robotics@humble:~/ros2_ws/src/my_package_python$ ls
my_package_python  resource    setup.py
package.xml        setup.cfg   test
robotics@humble:~/ros2_ws/src/my_package_python$
```

In this case, as we called `ros2 pkg create` with the *–node-name* option, a node called my_node with a simple Hello World executable is created.

2. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

- Ok, let's create our first package! We will call it turtlesim_control.
- For that, in the `src` folder, execute the following command:

```
$ ros2 pkg create --build-type
  ament_python turtlesim_control
```

- You must now have a directory with the package name and this initial content:

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src$ ls turtlesim_control/
package.xml  resource  setup.cfg  setup.py  test  turtlesim_control
```

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src$ ros2 pkg create --build-type
ament_python turtlesim_control
going to create a new package
package name: turtlesim_control
destination directory: /home/jotaraul/robotics_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['jotaraul <jotaraul@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
creating folder ./turtlesim_control
creating ./turtlesim_control/package.xml
creating source folder
creating folder ./turtlesim_control/turtlesim_control
creating ./turtlesim_control/setup.py
creating ./turtlesim_control/setup.cfg
creating folder ./turtlesim_control/resource
creating ./turtlesim_control/resource/turtlesim_control
creating ./turtlesim_control/turtlesim_control/__init__.py
creating folder ./turtlesim_control/test
creating ./turtlesim_control/test/test_copyright.py
creating ./turtlesim_control/test/test_flake8.py
creating ./turtlesim_control/test/test_pep257.py

[WARNING]: Unknown license 'TODO: License declaration'.  This has been se
t in the package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

1. Create a new python file in the `turtlesim_control` folder of the package called `turtlesim_control_node.py`

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src/turtlesim_control/turtlesim_control$ ls
__init__.py
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src/turtlesim_control/turtlesim_control$ touch turtlesim_control_node.py
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src/turtlesim_control/turtlesim_control$ ls
__init__.py  turtlesim_control_node.py
```

2. Modify setup.py at `/src/turtlesim_control/` by updating the maintainer, email, description, and license (put the same as in package.xml) and add a new entry telling ROS 2 about the existence of your node:

```
16      maintainer='jotaraul',
17      maintainer_email='jotaraul@todo.todo',
18      description='TODO: Package description',
19      license='TODO: License declaration',
20      tests_require=['pytest'],
21      entry_points={
22          'console_scripts': [
23                  'turtlesim_control_node = turtlesim_control.turtlesim_control_node:main',
24          ],
25      },
```

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

3. Update package.xml. Add the necessary dependencies for geometry_msgs and turtlesim.
   - Dependency types:
     - <build_depend>,
     - <exec_depend>,
     - <test_depend>
     - <depend> (includes all the above)

We need to include these three

Open | *package.xml ~/robotics_ws/src/turtlesim_control | Save

```
1  <?xml version="1.0"?>
2  <?xml-model href="http://download.ros.org/schema/
   package_format3.xsd" schematypens="http://www.w3.org/2001/
   XMLSchema"?>
3  <package format="3">
4    <name>turtlesim_control</name>
5    <version>0.0.0</version>
6    <description>TODO: Package description</description>
7    <maintainer email="jotaraul@todo.todo">jotaraul</maintainer>
8    <license>TODO: License declaration</license>
9
10   <!-- Build and Execution Dependencies -->
11   <depend>rclpy</depend>
12   <depend>geometry_msgs</depend>
13   <exec_depend>turtlesim</exec_depend>
14
15   <!-- Test Dependencies -->
16   <test_depend>ament_copyright</test_depend>
17   <test_depend>ament_flake8</test_depend>
18   <test_depend>ament_pep257</test_depend>
19   <test_depend>python3-pytest</test_depend>
20
21   <!-- Export Section -->
22   <export>
23     <build_type>ament_python</build_type>
24   </export>
25 </package>
26
```

XML | Tab Width: 8 | Ln 4, Col 33 | INS

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

4. Develop the node itself. In our example, implement in `turtlesim_control_node.py` the desired functionality. You can use a simple text editor or an IDE (e.g. Visual Studio Code). Copy this code:

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose

class TurtlesimControlNode(Node):
    def __init__(self):
        super().__init__('turtlesim_control_node')

        # Log message indicating the node has started
        self.get_logger().info("Turtlesim Control Node is now running...")

        # Subscriber to the /turtle1/pose topic
        self.pose_subscription = self.create_subscription(
            Pose,  # Message type for the /turtle1/pose topic
            '/turtle1/pose',  # Topic name to subscribe to
            self.pose_callback,  # Callback function that processes the message
            10  # QoS queue size
        )
        self.pose_subscription  # Prevent unused variable warning

        # Publisher to the /turtle1/cmd_vel topic
        self.velocity_publisher = self.create_publisher(
            Twist,  # Message type for the /turtle1/cmd_vel topic
            '/turtle1/cmd_vel',  # Topic name to publish to
            10  # QoS queue size
        )

        # Timer to publish velocity commands periodically
        self.timer = self.create_timer(0.5, self.publish_velocity_command)

        # Store the latest pose received from the subscriber
        self.current_pose = None
```

Class constructor

Subscriber

Publisher

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

```python
    def pose_callback(self, msg):
        """
        Callback function for the /turtle1/pose topic.
        Called whenever a new Pose message is received.
        Parameters:
        - msg (Pose): The message containing the turtle's position, orientation, and velocity.
        """
        self.current_pose = msg  # Store the received pose in the node's state
        # Print the pose information to the terminal
        self.get_logger().info(
            f'Turtle Pose - x: {msg.x:.2f}, y: {msg.y:.2f}, theta: {msg.theta:.2f}'
        )

    def publish_velocity_command(self):
        """
        Publishes velocity commands to the /turtle1/cmd_vel topic.
        This method is periodically called by the timer.
        """
        if self.current_pose is not None:  # Ensure we have a valid pose before publishing
            # Create a Twist message to specify linear and angular velocities
            twist = Twist()
            twist.linear.x = 2.0  # Move forward at a speed of 2.0 units/sec
            twist.angular.z = 1.0  # Rotate counterclockwise at a speed of 1.0 rad/sec

            # Publish the velocity command to the /turtle1/cmd_vel topic
            self.velocity_publisher.publish(twist)

            # Log the command for debugging
            self.get_logger().info(f'Published velocity command: {twist}')

```

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

```python
def main(args=None):
    """
    Main function to start the node.
    """
    rclpy.init(args=args)  # Initialize the ROS 2 system
    turtlesim_control_node = TurtlesimControlNode()  # Create an instance of the node
    rclpy.spin(turtlesim_control_node)  # Keep the node running until interrupted
    turtlesim_control_node.destroy_node()  # Clean up the node
    rclpy.shutdown()  # Shut down the ROS 2 system


if __name__ == '__main__':
    main()
```

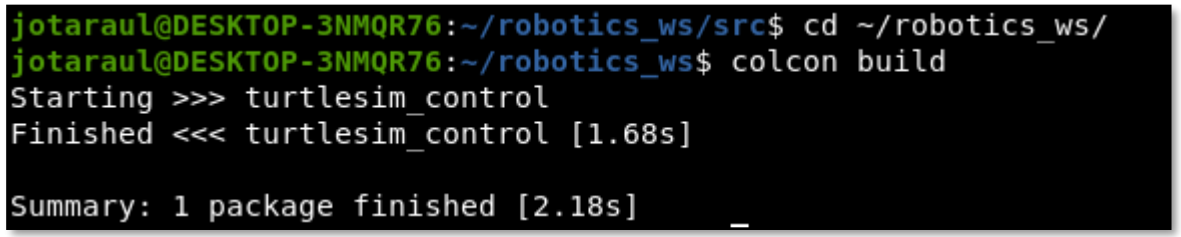
2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

5. Install python packages and set environment scripts.. This is done with the *colcon build* command, which carry out the following actions:

   - **Installation Setup**: Python packages defined with ament_python (in setup.py) are installed into the install/ directory. For example:
     - The setup.py entry points (e.g., console_scripts) are registered as executables in the workspace.
     - The package files are copied or symbolically linked into the install/ directory.
   - **Environment Scripts**: ROS 2 environment setup scripts are generated in the install/ directory, which ensure the workspace dependencies and paths are properly configured.

   Run:

   ```
   $ ~/ros2_ws
   $ colcon build
   ```

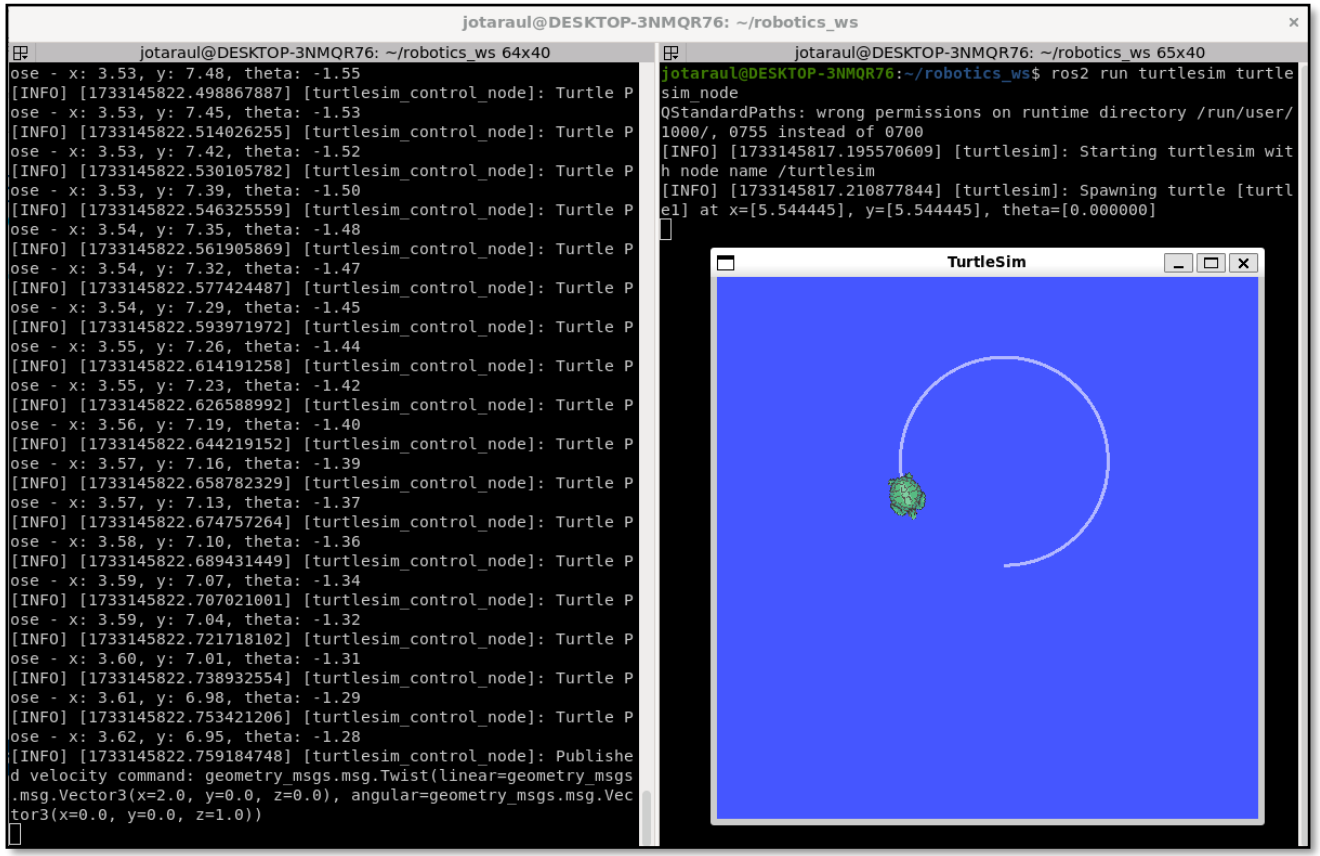


   You may need to run *$ source ~/robotics_ws/install/setup.bash* after this to update all the dependencies chain.

2. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

6. Run your node! For that you must specify the package and node names:

```
$ ros2 run turtlesim_control turtlesim_control_node
```



REFERENCES
WHERE TO FIND MORE KNOWLEDGE

[1] ROS 2 Documentation: https://docs.ros.org/en/humble/index.html

[2] Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." *ICRA workshop on open source software*. Vol. 3. No. 3.2. 2009.

[3] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," Science Robotics vol. 7, May 2022.

SEE YOU IN THE NEXT LECTURE!