ROS 2 Getting Started

Author: Manoj Sharma

Robot Operating System (ROS), www.ros.org, is an open source operating system designed for robots. It is "a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms."

This guide complements the official ROS 2 tutorial . Throughout this guide, the Operating System of choice is Ubuntu (Linux) 22.04 along with Humble version of ROS 2 (or simply ROS). The programming language of choice is Python with a slight mix of C++.

---

1 PREREQUISITE

1. Computer/Machine Setup (Install Ubuntu + ROS 2) [pick one, (a) or (b)]:
   (a) Use Virtual Machine to install Ubuntu and ROS 2.
      i. Virtual Machine (VM): Follow *this link* to setup Ubuntu VM using VirtualBox (refer to Section 5.1 for (M1/M2) Mac).
      ii. Install ROS 2 (Humble): *click here*.
   (b) Import a pre-curated virtual machine.
      i. Install VirtualBox: *click here*.
      ii. Download pre-curated Virtual Machine *(.ova)* file: *click here* [1].
      iii. Deploy VM *(.ova)* file to VirtualBox: *click here*.
2. Linux Fundamentals : *click here*.
3. Introduction to Python : *click here*.
4. Introduction to C++ : *click here*.

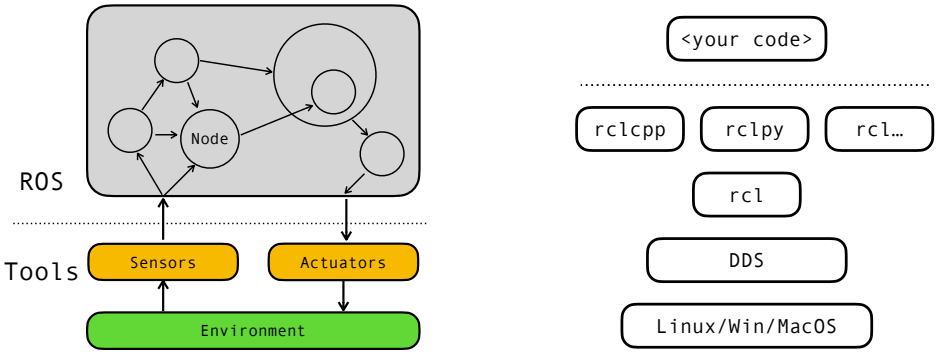
[1] This is a link to SCU Google Drive, so make sure you have SCU credentials to access this *.ova* file.

Affiliation Robotic Systems Lab., Santa Clara University, CA
Correspondence msharma@scu.edu, *Github*, *Homepage*
Draft August 3, 2023

Contents

2 ABOUT ROS 2

ROS 2 is a ground-up reimagining of ROS 1[2]. ROS 2 was started in 2014. As of 2023 ROS 2 is on eighth named release, called *Humble*. ROS 2 Humble uses Ubuntu 22 as a host OS to run. Its End-of-Life (EoL) date is May 2027.



(a) ROS 2 organisation (b) ROS big picture

Figure 1: ROS 2 architecture.

ROS 2 is a middleware designed to run on Linux, Windows or MacOS. It is built on DDS[3], a middleware proven in industry. Figure 1b shows this structure. ROS client library is used to communicate between the nodes.

2.1 WHAT IS ROS

ROS allows a problem to be broken down into smaller pieces that could be developed as a separate entity and can be combined together (similar to *lego* pieces). It provides a framework, tools, and interfaces for distributed development. In essence, ROS is:

2.1.1 PLUMBING Allows data/information between nodes as shown in Figure 1a.

2.1.2 TOOLS Logging, plotting, visualisation, diagnostics.

2.1.3 CAPABILITIES Planning, perception, execution.

2.1.4 ECOSYSTEM Encourages re-use of software pieces and share among the community.

[2]ROS 1 has been around since 2008. It uses custom TCP/IP middleware for the communication network. Its latest release, *Noetic*, has EOL date in May 2025.

[3]Click here for more details on ROS 2 design

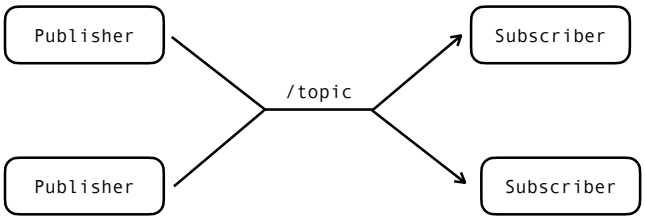


Figure 2: Message: Publisher - Subscriber model.

2.2 Types of Interaction b/w ROS Nodes

ROS offers three core types of interactions between ROS nodes. *Messages* are the simplest type where a publisher continuously broadcasts the message at a user defined rate on a *topic*, refer to Figure 2, (similar to a radio station broadcasts on a specific frequency) to which a subscriber can acquire data by subscribing (similar to tuning a radio to a specific frequency). *Service* and *Actions* are asynchronous means of interact between nodes. Table 1 summarises the strengths and weakness of the three core interactions types.

Type	**Strengths**	**Weaknesses**
Message	Good for most sensors	Easy to overload
	One-to-many	Message drop w/o knowledge
Service	Knowledge of missed call	Blocks until completion
	Well defined feedback	Connection typically re-establishes for each service call (slow activity)
Actions	Monitor long running processes	Complicated
	Handshake	

Table 1: Three core types of interaction between ROS nodes.

2.3 ROS Messages

ROS uses a simplified messages description language for describing the data values (or messages) that ROS nodes publish and/or subscribe:

1. **Standard messages**, *std_msgs*, represents primitive data types and other basic message constructs, such as multiarrays.
2. **Sensor messages**, *sensor_msgs*, represents messages for commonly used sensors, including cameras and scanning laser rangefinders.
3. **Geometry messages**, *geometry_msgs*, represents geometric primitives such as points, vectors, and poses.
4. **Other messages** include *action_msgs*, *diagnostic_msgs*, *nav_msgs*, *shape_msgs*, *stereo_msgs*, *trajectory_msgs*, *visualization_msgs*, and <*custom_msgs*>.
5. **Industrial-core messages**, *industrial_core*, are the messages unique to the industrial manipulator widely used across the industry. Click *here* for more details on ROS Industrial.

2.4 ROS 1 - ROS 2 Bridge

There are several instances where a ROS 1 workspace can't be easily converted or ported over to ROS 2 and therefore unable to communicate with ROS 2 workspaces. A workaround to that is to use a ROS 1 to ROS 2 bridge which, as the name suggests, bridges the communication. Click here for more details.

3 ROS 2 TUTORIAL - BASIC

This section covers four main topics - setup a workspace (refer to Figure 3), create a package, create a simple publisher & subscriber nodes, and lastly a launch file to run multiple nodes using a simple launch file.

For additional details, refer to the official tutorial on ROS 2- *click here*.

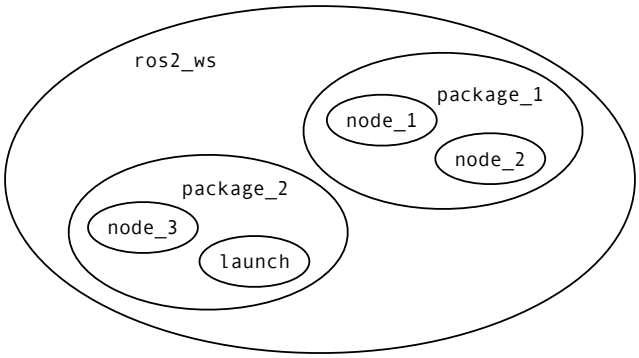


Figure 3: A typical ROS workspace architecture.

3.1 Setup a Workspace

Setting up a workspace refers to creating a workspace directory of choice, resolving the package dependencies (if any), followed by building the workspace.

The first step is to source the ROS 2 environment:

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

Create a workspace in the */home* directory by creating a parent directory of a custom name and a *src* directory within it. Here is how to create a workspace named *ros2_ws*:

```
cd ~/
mkdir -p ~/ros2_ws/src/
```

The directory *ros2_ws* is also known as root directory of this workspace. To move to the root directory from *home* directory:

```
cd ~/ros2_ws/
```

Resolve any package dependencies:

```
cd ~/ros2_ws/
rosdep install -i --from-path src --rosdistro humble -y
```

Finally, build the workspace:

```
colcon build
```

FYI - The console will return the following (no need to run these commands):

```
Summary: 0 package finished [0.16s]
```

3.2 Create a Package

A package may consists of logical programs (or nodes) for a specific task. This package example consists of a simple subscriber and a publisher.

Navigate to your *src* directory:

```
cd ~/ros2_ws/src/
```

Create a package with a name of your choice using *ament python* tools. Syntax to create a package named *'my_package'*:

```
ros2 pkg create --build-type ament_python my_package
```

Next step is to build the package. To do so, move back to the root workspace directory:

```
cd ~/ros2_ws/
```

Build the package:

```
colcon build
```

FYI - The console will return the following (no need to run these commands):

```
Starting >>> my_package
Finished <<< my_package [1.71s]

Summary: 1 package finished [1.80s]
```

3.3 Create/Add Node(s) in a Package

A package may consists of logical programs (or nodes) for a specific task. This example consists of a simple subscriber and a publisher.

Nodes are programs designed for a specific task. A node may consists of subscriber(s) and/or publisher(s). To add a node navigate to the following directory:

```
cd ~/ros2_ws/src/my_package/my_package/
```

3.3.1 Publisher Node: A sample publisher python script is available on Github - click here. Download/duplicate this python script in the directory mentioned above.

To keep it simple, use the same file name, i.e. *publisher_member_function.py*. This script initializes the node as *minimal_publisher*, starts to publish *'Hello World'* followed by a timestamp on a topic aptly-named */topic*, and prints it on the console at every $0.5s$. Be sure to read the documentation to understand more about this code.

3.3.2 Subscriber Node: A sample subscriber python script is available on Github - click here. Download/duplicate this python script in the directory mentioned above.

To keep it simple, use the same file name, i.e. *subscriber_member_function.py*. This script initializes the node as *minimal_subscriber*, subscribes to a topic aptly-named */topic*, and prints it on the console as it receives the message . Be sure to read the documentation to understand more about this code.

FYI - At this point there should be two additional nodes (one publisher and one subscriber) present within *my_package*. To see these files run:

```
ls ~/ros2_ws/src/my_package/my_package/
```

The console will return the following (no need to run these commands):

```
__init__.py publisher_member_function.py
subscriber_member_function.py
```

3.3.3 ADD ENTRY POINTS: This is done by editing *setup.py*, located in:

`/ros2_ws/src/my_package/`

Create two entry points for *publisher_member_function.py* and *subscriber_member_function.py* named *pub* and *sub*, respectively. To do so, open *setup.py* using an editor of your choice, scroll to the bottom of the script, and add the following within *'console_scripts'*:

```
'pub = my_package.publisher_membership_function:main',
'sub = my_package.subscriber_membership_function:main',
```

FYI - The completed *setup.py* file is also available on Guthub - click here.

3.3.4 BUILD & RUN: Navigate to the root of the workspace and then build the package:

```
cd ~/ros2_ws/
colcon build
```

To run the publisher - open a new terminal, navigate to the root of the workspace, source ROS 2 environment, and source the workspace:

```
cd ~/ros2_ws/
source /opt/ros/humble/setup.bash
source install/setup.bash
```

Use *ros2 run* command to run *pub* node which is present in `my_package` package:

```
ros2 run my_package pub
```

The console will start publishing info, something like this:

```
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
...
```

To run the subscriber - open a new terminal, navigate to the root of the workspace, source ROS 2 environment, and source the workspace:

```
cd ~/ros2_ws/
source /opt/ros/humble/setup.bash
source install/setup.bash
```

Use *ros2 run* command to run *pub* node which is present in `my_package` package:

```
ros2 run my_package sub
```

The console will start publishing info, something like this:

```
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
```

3.3.5 Additional Resources/Tools: ROS 2 has several tools often helpful in debugging. A few of them are listed below. For more information, refer to a ROS 2 command guide - click here.

While the two nodes are running (in two separate terminals), open another terminal, source the ROS 2 environment:

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

A GUI tool called *rqt_graph* provides a pictorial representation of all the active nodes and ROS structure, shown in Figure 4. To run *rqt_graph* run:

```
ros2 run rqt_graph rqt_graph
```

This will open up a new window showing the active ROS 2 architecture. This can also be used to verify that all the nodes and active properly connected.

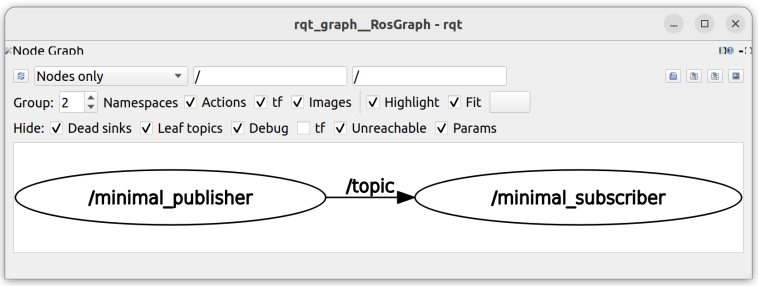


Figure 4: Graphical representation of active nodes and topics.

Similarly there are many more useful tools for eg., *ros2 topic list*, *ros2 topic echo <topic name>*, *ros2 topic info <topic name>*, etc ...

Note - to stop any activity on a terminal press *Ctrl* + *C*.

3.4 Launch File Setup

A typical workspace may contain several packages where each package may contain multiple nodes in it. This means running multiple nodes one-by-one is very time consuming. A launch file is designed to run and/or launch several nodes and/or launch files. One way to architect this is to create a new package and containing the launch file(s).

3.4.1 Setup a Package (cmake) In the case of a package consisting of launch file(s) it is preferred to build a package using *ament cmake* tools (and not *ament python*). To create a package, navigate to your *src* directory:

```
cd ~/ros2_ws/src/
```

Syntax to create a package named *'my_launch_pkg'*:

```
ros2 pkg create --build-type ament_cmake my_launch_pkg
```

Build the package. To do so, move back to the root workspace directory:

```
cd ~/ros2_ws/
colcon build
```

FYI - The console will return the following (no need to run these commands):

```
Starting >>> my_package
Starting >>> my_launch_pkg
Finished <<< my_package [1.27s]
Finished <<< my_launch_pkg [1.57s]

Summary: 2 package finished [1.75s]
```

Next, create a *launch* directory in */ros2_ws/src/my_launch_pkg/*, and change directory to the newly created *launch* directory:

```
cd ~/ros2_ws/src/my_launch_pkg
mkdir launch
cd launch/
```

3.4.2 Create a Launch File: A sample file python launch script is available on Github - click here. Download/duplicate this python script inside the *launch* directory with a *my_first.launch.py* name. Note that the *entry points* for *pub* & *pub* (Section 3.3.3) acts as executable which are referenced in *my_first.launch.py*. This means launching a (single) *launch* file will run the publisher and subscriber node.

3.4.3 Edit CMake list: Lastly the *CMakeLists.txt* in `/ros2_ws/src/rover_start/` needs to be updated by adding the following:

```
install (DIRECTORY launch
    DESTINATION share/${PROJECT_NAME})
```

Refer to an updated *CMakeLists.txt* on Github - click here.

3.4.4 Build & Launch: Navigate to the root of the workspace and then build the package:

```
cd ~/ros2_ws/
colcon build
```

To launch the *my_first.launch.py* file (in the *my_launch_pkg* package) source the workspace and use the launch command:

```
cd ~/ros2_ws/
source install/setup.bash
ros2 launch my_launch_pkg my_first.launch.py
```

If done correctly, the console should output the publisher and subscriber info similar to the ones discussed in Section 3.3.4.

Additionally, the both the nodes can be verified by running *rqt_graph* which is discussed in Section 3.3.5.

Note that, here, (by design) one launch file is able to start two nodes. It is a good practice to run all the logical nodes via one launch file. There may be cases where a single node needs to be run one at a time for the purpose of debugging.

[Optional] To avoid sourcing the ROS2 environment every time you open a new shell, then you can add the command to your shell startup script

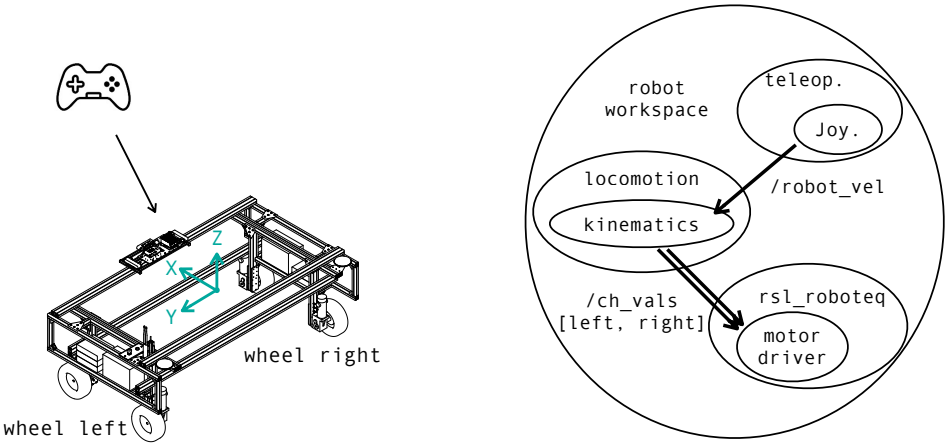
```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

Note - sometimes the computer may have multiple versions of ROS installed and therefore the user may need to decide which version to source. In such scenario the the above step is not recommended as it may cause confusion.

4 DIY - REMOTE CONTROLLED DIFFERENTIAL DRIVE ROBOT

This section requires a basic understanding of ROS (Section 3). Refer to an example workspace, *robot_ws*, available on Github.

4.1 Two Wheel Robot Description



(a) Real robot (b) Robot workspace

Figure 5: Two wheel mobile robot with skid steering.

The two wheel AMR shown in Figure 5a is capable of moving linearly along X - axis and rotating about Z - axis[4] (at the centroid of the robot). For a desired linear and angular velocities, denoted by v_l and v_a respectively, for the robot with left and right wheel radius denoted by r_{wL} and r_{wR}, respectively, and distanced along Y- axis by d, the left and right wheel velocities, denoted by ω_L and ω_R respectively, are given by

$$\omega_L = \left(\frac{1}{r_{wL}}\right) v_l \; - \; \left(\frac{d}{2}\right) r_{wL}\, v_a \tag{1}$$

$$\omega_R = \left(\frac{1}{r_{wR}}\right) v_l \; + \; \left(\frac{d}{2}\right) r_{wR}\, v_a \tag{2}$$

Equations 1 and 2 compute the wheel velocities as a function of linear and angular velocities of the robot.

[4]Complex angular motions are also possible, refer to the Wikipedia page for more details.

4.2 Electrical Wiring

A functional electrical diagram of a simple two wheeled differential drive robot is shown in Figure 6.

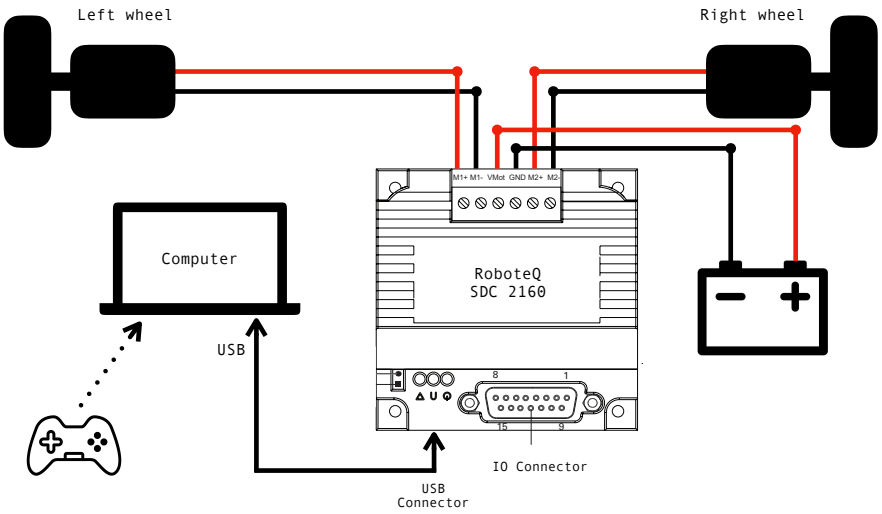


Figure 6: Functional electrical diagram for a simple two wheeled drive robot.

4.3 Construct the Workspace

Set up a new workspace with a name of your choice. The workspace shown in Figure 5b, a teleportation package is used to produce linear and angular velocities for the robot.

4.3.1 Teleoperation Package There exist several teleoperation packages which can be used to publish the velocities. Two of the teleoperation packages are - *teleop_twist_keyboard* and *teleop_twist_joy*. Pick a teleop package and clone it into the `/src` directory of the workspace.
Note that both the packages publish the velocities of type *geometry_msgs/Twist* on a `/cmd_vel` topic by default.

4.3.2 Locomotion Package The locomotion package consists of a *kinematics* node which subscribes to the `/cmd_vel` and publishes the wheel velocities.

4.3.3 Motor-driver Package

The amplifier of choice RoboteQ SDC2160 which is a dual channel brushed DC motor controller. Clone the *rsl_roboteq* package (available on Github) into the `/src` directory of the workspace. The *roboteq_node* subscribes to `/ch_vals` topic of type array). The array accepts two values[5], one for each (left and right) motor.

4.4 Build and Run

Navigate to the root of the workspace and then build the package:

```
cd ~/robot_ws/
colcon build
```

Source the workspace and then launch the appropriate file to run all the needed nodes to run the robot.

```
source install/setup.bash
ros2 launch rover_launch rover.launch.py
```

Note - If the console responds with *"Error opening serial port,"* refer to Section 5.2.

[5]The values depend on the amplifier (or controller) configuration - open or closed loop position or velocity. In its default configuration, by design, the value $\in [-1000, 1000]$; -1000 and 1000 represents full power in CW and CCW direction* respectively, and intermediate value translates proportionately. Refer to the manual for more details.

5 ADDITIONAL RESOURCES

5.1 Install VM on an Apple Silicon (M1/M2) Mac

Step by step guide on how to install[6] VM & Ubuntu on an Apple silicon (M1/M2) Mac.

Install *Multipass* using the installer.

Create an instance using the following command, choosing the username, number of CPUs, amount of diskspace, and amount of memory:

```
multipass launch --name UserName --cpus 4 --disk 20G --memory 8G
```

Run the following command to enter command line for your created instance:

```
multipass shell UserName
```

In the Ubuntu command line, set a username and password using the following commands, following the prompts. Replace *"newUsername"* with your choice of username, and *"pass"* with your choice of password:

```
sudo usermod -l newUsername ubuntu
sudo passwd newUsername
```

Update your instance and install the graphical desktop package using the following commands:

```
sudo apt update
sudo apt install ubuntu-desktop
```

[6] Credits to Harry Clark.

5.2 Issue - "Error Opening Serial Port"

Accessing serial port require additional permission. To set serial port permission, first determine the serial port:

```
ls /dev/tty*
```

The serial port may be `USBx` or `ACMx`. If the port in question is `USBx`, run the following command

```
ls -l /dev/ttyUSB*
```

FYI - The console may return something similar

```
crw-rw---- 1 root dialout 188, 0 5 apr 23.01 ttyUSB0
```

Add the Linux user to the dialout group (replace the <username> with the appropriate username)

```
sudo usermod -a -G dialout <username>
```

Log out and log in again (or reboot) for this change to take effect.