Lab 1: Introduction to Robot Operating System (ROS) [*]

EECS/ME/BIOE C106A/206A Fall 2025

Goals

By the end of this lab you should be able to:

- Set up a new ROS environment, including creating a new workspace and creating a package with the appropriate dependencies specified
- Use the `catkin` tool to build the packages contained in a ROS workspace
- Run nodes using `rosrun`
- Use ROS's built-in tools to examine the topics and services used by a given node

---

If you get stuck at any point in the lab you may submit a help request during your lab section at tinyurl.com/ee106a-fa25lab.

Note: We encourage you to type out all of the commands listed in this document to better remember them. Additionally, there are some formatting things that may break when you copy and paste.

Note: Much of this lab is borrowed from the official ROS website and tutorials. We picked out the material you will find most useful in this class, but feel free to explore other resources if you are interested in learning more.

Contents

1 Lab Checkoffs on Gradescope **2**

2 Creating an Instructional Account **3**

3 What is ROS? **3**
- 3.1 High-level overview and terminology 3
- 3.2 Application to a sample system 3

4 Initial configuration **5**
- 4.1 Running ROS 2 in a Container 5
- 4.2 Workspace setup 6

5 Navigating the ROS file system **6**
- 5.1 File system 6
- 5.2 Anatomy of a package 6
- 5.3 File system tools 8

[*] Rewritten and adapted for ROS2 in Summer 2025 by Jaeyun Stella Seo. Rewritten in Fall 2023 by Mingyang Wang and Eric Berndt. Minor edits for new protocols in Fall 2022 by Emma Stephan, and in Fall 2021 by Josephine Koe and Jaeyun Stella Seo. Developed by Aaron Bestick and Austin Buchan, Fall 2014.

6 Creating ROS Workspaces and Packages **8**
6.1 Creating a workspace . . . 8
6.2 Creating a new package . . . 9
6.3 Building a package . . . 9

7 Understanding ROS nodes **11**
7.1 Running Turtlesim . . . 11

8 Understanding ROS topics **11**
8.1 Using rqt_graph . . . 12
8.2 Using ros2 topic . . . 12

9 Understanding ROS services **12**
9.1 Using rosservice . . . 13
9.2 Calling services . . . 13

10 Publisher/Subscriber Pair Example **14**

11 Writing a publisher/subscriber pair **15**
11.1 What you'll be creating . . . 16
11.2 Steps to follow . . . 16
11.3 Generating a custom message type . . . 17
11.3.1 Defining a new message type . . . 17
11.3.2 Creating an interface/message package and new message types . . . 18
11.3.3 Writing your new talker and listener nodes . . . 19

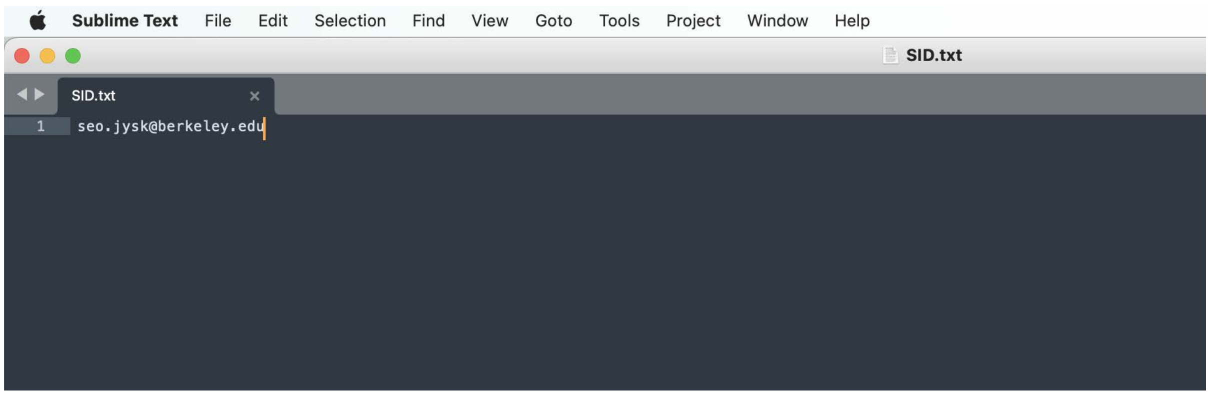
12 Pushing to Github **19**

13 Additional Lab Etiquette **21**

1 Lab Checkoffs on Gradescope

Your lab checkoff progress for the semester will be posted to the `Lab Grades` assignment on Gradescope. The autograder for this assignment will typically be re-run daily; if you do not receive credit for a completed lab within a few days, please reach out to lab staff.

In order to see your lab grades, to the `Lab Grades` assignment, you must submit a file titled **SID.txt** that contains, counterintuitively, your email address. For example, Stella's submission would look like this.



```
seo.jysk@berkeley.edu
```

Figure 1: Example SID.txt file

2 Creating an Instructional Account

To create a lab account, visit `https://acropolis.cs.berkeley.edu/~account/webacct/` and click "Login using your Berkeley CalNet ID." On the next page, next to ee106a click "Create a new Account" and save this login information to use for your 106A labs! Make sure both partners do this so everyone has an account.

3 What is ROS?

3.1 High-level overview and terminology

The ROS website says:

> The Robot Operating System (ROS 2) is an open-source set of software libraries and tools for building robot applications. From hardware drivers and state-of-the-art algorithms to powerful developer utilities, ROS 2 provides everything you need for your next robotics project.

You can understand ROS 2 to be a meta-operating system for your robot. It handles a lot of things: hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

ROS 2 runs as a collection of small programs called **"nodes"** that talk to each other without a central controller. This network of nodes is called the "graph." You'll use three basic ways for nodes to share information:

- **Services (RPC, or Remote Procedure Calls)**: Think about ordering at a restaurant: you place one order, wait, then receive your meal. Remote Procedure Calls let one node invoke a function on another node and wait synchronously for a response.
- **Topics (QoS, or Quality of Service)**: Think about tuning into a news broadcast: the publishers stream updates, and any number of listeners can tune in. Topics are asynchronous streaming data channels where publishers push **messages** and subscribers receive them. There are different policies (reliability, durability, history, deadlines, liveliness, etc.) that let you tune delivery guarantees and performance.
- **Parameters**: Think about the settings in your apps, such as volume, brightness, etc. But now apply them to nodes. Parameters are node-local key–value settings for configuration. Each node exposes its own parameters, which you can query and modify at runtime.

Unlike ROS 1, there is no central "roscore" master node in ROS 2. Instead, every node discovers and talks directly to its peers over the DDS (Data Distribution Service) middleware.

3.2 Application to a sample system

Now let's think about how this can be applied to a robotic system. Consider a two-joint manipulator arm for a pick-and place task.

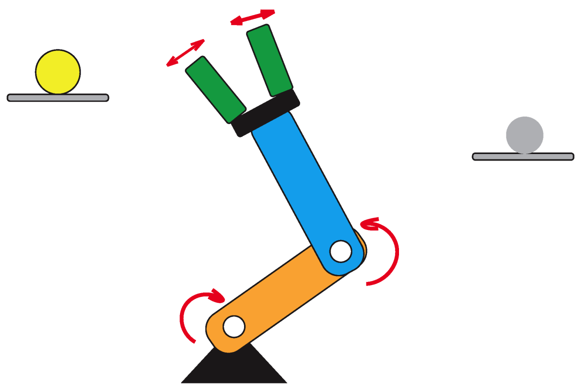


Figure 2: Example two-joint manipulator arm.

A typical robotic system has numerous sensing, actuation, and computing components. Suppose our system has:

- Two motors, each connected to a revolute joint (orange and blue)
- A motorized gripper on the end of the arm (green)
- A stationary camera that observes the robot's workspace

To pick up an object, the robot might:

- Use the camera to measure the position of the object
- Command the arm to move toward the object's position
- Once properly positioned, command the gripper to close around the object.

Given this sequence of tasks, how should we structure the robot's control software?

A useful abstraction for many robotic systems (and computer science in general) is to divide the control software into various low-level, independent control loops, each controlling a single task on the robot. In our example system above, we might divide the control software into:

- A control loop for each joint that, given a position or velocity command, controls the power applied to the joint motor based on position sensor measurements at the joint
- Another control loop that receives commands to open or close the gripper, then switches the gripper motor on and off while controlling the power applied to it to avoid crushing objects
- A sensing loop that reads individual images from the camera

Given this structure for the robot's software, we then couple these low-level loops together via a single high-level module that performs supervisory control of the whole system:

- Query the camera sensing loop for a single image.
- Use a vision algorithm to compute the location of the object to grasp
- Compute the joint angles necessary to move the manipulator arm to this location
- Sends position commands to each of the joint control loops telling them to move to this position
- Signal the gripper control loop to close the gripper to grab the object

An important feature of this design is that the supervisor need not know the implementation details of any of the low-level control loops: it interacts with each only through simple control messages. This encapsulation of functionality makes the system modular, making it easier to reuse code across robotic platforms.

We can conceptualize each individual control loop as a **node**: an individual software process that performs a specific task. Recall that nodes can exchange information to each other. Such information could include control messages or sensor readings. Nodes can exchange this information by publishing or subscribing to **topics**. They can also send requests to other nodes via **services**. Very conveniently, nodes can be written in a variety of languages (including Python and C++), and ROS transparently handles the details of converting between different datatypes, exchanging messages between nodes, etc.

Now, let's visualize the communication and interaction between different software components via a **computation graph**, where:

- Nodes are represented by ovals (ie `/usb_cam` or `/ar_track_alvar)`).
- **Topics** are represented by rectangles (ie `/usb_cam/camera_info` and `/usb_cam/image_raw`).
- The flow of information to and from topics and represented by arrows. In the above example, `/usb_cam` publishes to the topics `/usb_cam/camera_info` and `/usb_cam/image_raw`, which are subscribed to by `/ar_track_alvar`.
- While not shown here, **services** would be represented by dotted arrows.
- Also not shown here, **parameters** are key–value pairs that each node exposes for configuration at runtime. For example, your `/usb_cam` might have parameters specifying image width, image height, or refresh rate.

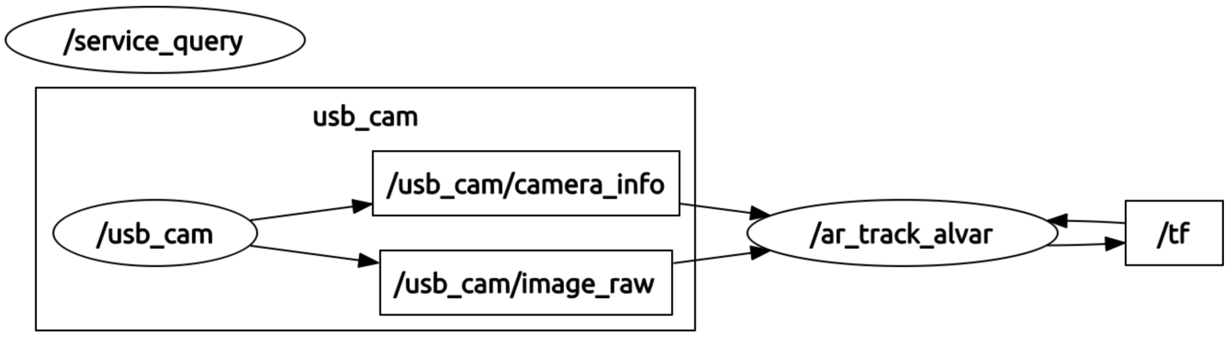


Figure 3: Example computation graph.

4 Initial configuration

Whew! That was a lot of information about ROS. Now let's go about setting up our work stations and class accounts so we can use ROS in our labs.

4.1 Running ROS 2 in a Container

Instead of installing ROS 2 directly on your computer, we'll run it inside a *container*. Think of a container as a sealed lunchbox for software: everything you need goes inside, and it won't spill out or make a mess on your computer. We have packaged Ubuntu, ROS 2, and other tools you'll need into this "box." When you open this box, it will work exactly the same way every time. And when you're done and close the box, the rest of your computer is untouched.

So why use a container? There are a few reasons:

- **Same Setup for Everyone:** No matter which lab machine you use, you'll have the exact same ROS 2 environment.
- **No Risk to Your Computer:** You can't accidentally break anything on the lab PC.
- **Easy to Start Over:** If something goes wrong, just throw away the container and start a fresh one—no complicated fixes needed.
- **No Admin Rights Needed:** You don't need special permissions to install or update anything.

Now let's get this container on your account! Once you are logged into your class, account open up the Terminal and initialize the container system for your account by running

```
inst-containers-setup
```

You'll see "Container setup complete!"

Next, we'll create a ROS 2 container in Distrobox with the following command.

```
~ee106a/create-ros2-container
```

This should happen pretty quickly. Now, we'll enter our ROS 2 container using

```
distrobox enter ros2
```

The first time you do this, it will install a bunch of different things before telling you setup is complete. Look at the first part of your command line interface. You'll note that the hostname has changed from the computer you're using ("c105-n") to "ros2." This shows that you've now switched to being inside the container.

Note that the container doesn't mean your lab account has a "split brain" where what happens inside and outside the container are independent of each other. That's more akin to having a dual-boot computer, or a virtual machine within your computer. Instead of that, our container can interact with everything you would hope it could. So you can continue to write and edit files normally, and they will still be saved to your lab account.

Moving forward, if you want to use ROS in a Terminal, you will need to run first enter the container so that the Terminal has opened the lunchbox with ROS 2 in it.

4.2 Workspace setup

Throughout the semester, course staff will publish the starter code to fix bugs or release new labs. Clone the starter code repository by running the following command from your home folder (~):

```
cd ~
git clone https://github.com/ucb-ee106/106a-fa25-labs-starter.git ros_workspaces
```

This clones the directory as "`ros_workspaces`" (we'll explain what this name means later), which currently should only contain a `/lab0` workspace and a `/lab1` workspace. We will be working in both workspaces for this lab. You will note that `lab0` is empty other than a PDF of this document. This is intentional so that in the first part of this lab, you can see how to build a workspace package from scratch. In `lab1` there is an existing package that you will play around with. Note that `ros_workspaces` will eventually contain several lab-specific workspaces (`/lab2`, `/lab3`, etc.).

Run the following command to rename the remote from "`origin`" to "`starter`".

```
cd ~/ros_workspaces
git remote rename origin starter
```

Now, if you ever want to pull updated starter code, you'd execute the following command:

```
git pull starter main
```

Instructions to publish your lab code to your personal github repository are discussed in Section 12.

Two other useful commands to know are `rmdir` to remove an empty directory and `rm -r` to remove a non-empty directory.

5 Navigating the ROS file system

5.1 File system

Large software systems written using the ROS node, topic, and service model can quickly become quite complex (nodes written in different languages, nodes that depend on third-party libraries and drivers, nodes that depend on other nodes, etc.). To help with this situation, ROS provides a system for organizing your ROS code into logical units and managing dependencies between these units.

The basic unit of software organization in ROS is the **package**, a folder containing executables, source code, libraries, and other resources. In the following subsections, we will have you explore an example package for the Baxter robot SDK.

5.2 Anatomy of a package

`cd` into `/opt/ros/humble/share/turtlesim`. The `turtlesim` package is used as an introduction to ROS 2. Here, we are specifically interested in :

- `package.xml` - the package's metadata, configuration, and dependencies; included in the root directory of every package.
- `/launch` - launch files that start and configure ROS nodes with specific parameters and configurations
- `/scripts` - executable scripts, usually used to interact with ROS nodes

Other packages might contain some additional items:

- `/src` - source code for ROS nodes provided by the package. In fact, any package you build **must** contain this directory for you to put your source code. The `turtlesim` package doesn't have it because it is a default package.
- `/lib` - extra libraries used by the package
- `/msg` and `/srv` - message and service definitions nodes use to exchange data

Open the `package.xml` file. It should look something like this:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/
<package format="3">
  <name>turtlesim</name>
  <version>1.4.2</version>
  <description>
    turtlesim is a tool made for teaching ROS and ROS packages.
  </description>

  <maintainer email="audrow@openrobotics.org">Audrow Nash</maintainer>
  <maintainer email="michael.jeronimo@openrobotics.org">Michael Jeronimo</maintainer>

  <license>BSD</license>

  <url type="website">http://www.ros.org/wiki/turtlesim</url>
  <url type="bugtracker">https://github.com/ros/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ros/ros_tutorials</url>

  <author email="dthomas@osrfoundation.org">Dirk Thomas</author>
  <author>Josh Faust</author>
  <author email="mabel@openrobotics.org">Mabel Zhang</author>
  <author email="sloretz@openrobotics.org">Shane Loretz</author>

  <build_depend>qt5-qmake</build_depend>
  <build_depend>qtbase5-dev</build_depend>

  <buildtool_depend>ament_cmake</buildtool_depend>
  <buildtool_depend>rosidl_default_generators</buildtool_depend>

  <exec_depend>libqt5-core</exec_depend>
  <exec_depend>libqt5-gui</exec_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>

  <depend>ament_index_cpp</depend>
  <depend>geometry_msgs</depend>
  <depend>rclcpp</depend>
  <depend>rclcpp_action</depend>
  <depend>std_msgs</depend>
  <depend>std_srvs</depend>

  <member_of_group>rosidl_interface_packages</member_of_group>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

This `package.xml` file is used by ROS tools to understand and manage the package during build, installation, and runtime. Along with some metadata about the package, the `package.xml` specifies packages on which `turtlesim` depends. The packages with `<build_depend>` are the packages used during the build phase, and the ones with `<exec_depend>` are used during the runtime phase.

The `rclcpp` dependency is important - `rclcpp` is the ROS library that C++ nodes use to communicate with other nodes in the computation graph. The corresponding library for C++ nodes is `rclpy`. We will primarily be using `rclpy` in this course.

5.3 File system tools

ROS provides a collection of tools to create, edit, and manage packages. One of the most useful is `ros2 pkg prefix`, which returns information about a specific package. Let's project ourselves into the future when we will be using the UR7e. We may need some built-in packages. In a new terminal, try running the command

```
ros2 pkg prefix ur_client_library
```

which should return the directory that that particular package lives in.

Note: To get info on the options and functionality of many ROS command line utilities, run the utility plus "`--help`" (e.g., "`ros2 pkg prefix --help`").

Next, let's test out a couple more convenient commands for working with packages. Run

```
ros2 pkg executables turtlesim
```

This shows us all the executables in the `turtlesim` package.

```
ros2 interface show std_msgs/msg/String
```

This shows us the details of a message type.

6 Creating ROS Workspaces and Packages

Now let's create our own ROS package.

6.1 Creating a workspace

First, we first have to create a ROS workspace – a directory used to organize and manage ROS packages. Since all ROS code must be contained within a package in a workspace, this is something you'll do each time you start a new lab or final project.

In Section 4.2, you created the `ros_workspaces` directory, which contains the folder `lab1`. As the name suggests, this will be your workspace for lab 1. In it, there is a `\src` folder. Navigate to that directory with the following:

```
cd lab1/src
```

ROS uses the `colcon` tool to build all code in a workspace and does some bookkeeping to easily run code in packages. Right now, the `lab1/src` directory contains the package `chatter`. Since the directory contains a package, let's build it by running `colcon build` from the workspace directory.

You should notice three new directories alongside `src`

- `build`: This is where your code is compiled into intermediate build artifacts and object files.
- `install`: This is the"runtime" version of your code with things like your executables and message/service code. You'll need to source this folder so ROS can find all your packages.

- `log`: This holds your build and test logs so you can review if something goes wrong.

You can also envision this as a kitchen workflow.

- `build`: This is our prep station, where our intermediate steps like cutting, kneading, and mixing has happened. All the temporary parts of our dish live here.
- `install`: This is the our serving window, where our fully-cooked dishes (nodes, libraries, message files, etc) sit. We source this folder because we want ROS to find the fully cooked/prepared items.
- `log`: This is kind of like the recipe book where the chef may leave notes. Every time you build or test, ROS 2 writes what happened—errors, warnings, test results—into time-stamped notes in this folder. If something goes wrong, you peek here to see exactly which step in the recipe failed.

6.2 Creating a new package

You're now ready to create a Python-based package. From the `src` directory, run

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

As a note, you'll see `ament` show up a lot, because it's ROS's underlying build and package management system. The alternate build type is `ament_cmake` for C++ builds

Now let's examine the contents of our new package `foo`. There are quite a few things here:

- **package.xml**: This contains metadata and the dependencies that your package requires. You'll note that for now, your only dependencies are ament and Python related.
- **setup.py**: When you run `colcon build`, this file tells the Python build plugin how to install your code into the workspace's install/ area. This is where all the cooking gets done in our kitchen.
- **resource/foo**: This is an empty "marker file" so Ament knows that this is a package called `foo`. It does not get edited.
- **foo/ directory**: This is the actual Python module for your package. Any .py files you add here become importable modules. Note that there is an empty `__init__.py` file here.

Next, we'll try creating a new package while specifying a few dependencies. Return to the `src` directory and run the following command:

```
    ros2 pkg create bar --build-type ament_python --dependencies rclpy std_msgs
        geometry_msgs turtlesim
```

Examine the `package.xml` file for the new package `bar` and verify that the dependencies have been added. Note that you need to include the dependency `rclpy` even though you're setting the build type to `ament_python`. This is because the build type tells **how** to build the package (a Python build), and the dependency says **what** you need (Python).

You're now ready to add source code, message and service definitions, and other resources to your project.

6.3 Building a package

Once you've added all your resources to the new package, the last step before you can use the package with ROS is to *build* it. Run the `colcon build` command again from the `lab1` directory.

```
colcon build
```

`colcon build` builds all the packages and their dependencies in the correct order. If everything worked, `colcon build` should print a summary of how many packages were built with no error messages

Note that the `install` directory contains the script `setup.bash`, generated by `colcon build`. "Sourcing" this script will prepare your shell environment for using the ROS packages contained in this workspace (among other actions, it adds "`~/ros_workspaces/lab1/src`" to the `$ROS_PACKAGE_PATH`). Run the commands

```
echo $AMENT_PREFIX_PATH
source install/setup.bash
echo $AMENT_PREFIX_PATH
```

and note the difference between the output of the first and second `echo`.
Note: *Any time that you want to use a non-built-in package, such as one that you have created, you will need to source the* `install/setup.bash` *file for that package's workspace.*

Checkpoint 1

Submit a checkoff request at tinyurl.com/fa24-106alab for a staff member to come and check off your work. At this point you should be able to:

- Explain the contents of your `~/ros_workspaces` directory. Be prepared to draw out the different directory levels and explain each item.
- Demonstrate the use of the `colcon build` command
- Explain the contents of a `package.xml` file
- Use ROS's utility functions to find the path of a package

Following Checkpoint 1, you may close all previous terminal windows.

7 Understanding ROS nodes

A quick review of some computation graph concepts:

- **Node**: an executable representing an individual ROS software process
- **Message**: a ROS datatype used to exchange data between nodes
- **Topic**: nodes can *publish* messages to a topic and/or *subscribe* to a topic to receive messages

We're now ready to test out some actual software running on ROS.

7.1 Running Turtlesim

The ROS equivalent of a "hello world" program is turtlesim, a simulated environment to interact with a virtual turtle. To run Turtlesim, we want to run the `turtlesim_node` executable, which is located in the `turtlesim` package. Open a new terminal window and run

```
ros2 run turtlesim turtlesim_node
```

In general, to start a node, we use the `ros2 run` command. The syntax is

```
ros2 run [package_name] [executable_name]
```

As with packages, ROS provides a collection of tools we can use to get information about the nodes and topics that make up the current computation graph. In a new terminal window, try running

```
ros2 node list
```

You should see that the only node currently running is `/turtlesim`, which is the node that we just started up. We can get more information on the `/turtlesim` node by running

```
ros2 node info /turtlesim
```

We can see that the output shows that `/turtlesim` has multiple subscribers, publishers, service servers, and an action server. Now let's explore how we can utilize this information to make things happen in Turtlesim!

8 Understanding ROS topics

Let's make our turtle do something. Leave the `turtlesim_node` window open from the previous section.

In yet another new terminal window, use `ros2 run` to run the `turtle_teleop_key` executable in the `turtlesim` package:

```
ros2 run turtlesim turtle_teleop_key
```

This starts a node to take keyboard inputs to tell turtlesim what to do. You should now be able to drive your turtle around the screen with the arrow keys when in this terminal window.

Take note of how the terminal window running Turtlesim changes when you use the given letter keys to rotate to absolute orientations or cancel those rotations. Clearly the `turtle_teleop_key` node is talking to the `turtlesim` node.

8.1 Using rqt_graph

Let's take a closer look at what's going on here. We'll use the tool `rqt_graph` to visualize the current computation graph. Open a new terminal window and run

```
rqt_graph
```

This should produce an illustration like Figure 4. In this example, the `tele_op` node is capturing your keystrokes and publishing them as control messages on the `/turtle1/cmd_vel` topic. The `/turtlesim` node then subscribes to this topic to receive the control messages.

For now, you can ignore the other arrows that include the word `\_action`. These are part of yet another type of communication called **actions**, what we will get more to in later labs.

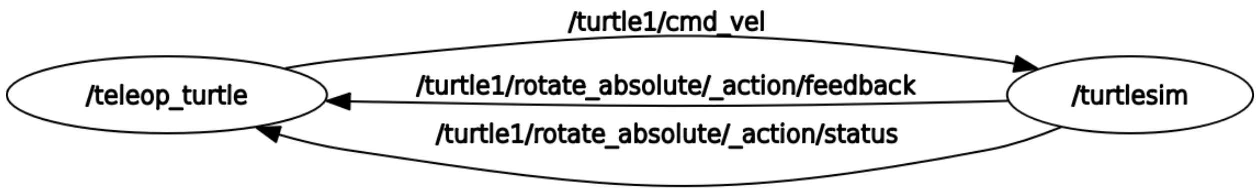

/turtle1/cmd_vel
/teleop_turtle
/turtle1/rotate_absolute/_action/feedback
/turtlesim
/turtle1/rotate_absolute/_action/status

Figure 4: Output of rqt_graph when running Turtlesim.

8.2 Using ros2 topic

Let's take a closer look at the `/turtle1/cmd_vel` topic using the rostopic echo tool. Let's use `rostopic echo` to look at individual messages that `/teleop_turtle` is publishing to the topic. Open a new terminal window and run

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

Move the turtle with the arrow keys and observe the messages published on the topic. Return to your `rqt_graph` window, and click the refresh button (blue circular arrow icon in the top left corner). You might expect to see a new node that is subscribed to the `/turtle1/cmd_vel`. And you would be right to expect that! But it doesn't appear to be there.

Now, in the GUI, be sure to uncheck the **Debug** box under **Hide**. Now you should see your "Listener Node" show up under the name of something like `/ros2cli_NUMBER`. It turns out, this is actually quite a handy feature of `rqt_graph`, as you can imagine the graph would get quite busy if you included everything like terminals listening in on topics or this infrastructural necessity Daemon running in the background.

While `rqt_graph` only shows topics with at least one publisher and subscriber, we can view all the topics published or subscribed to by any node by running

```
ros2 topic list
```

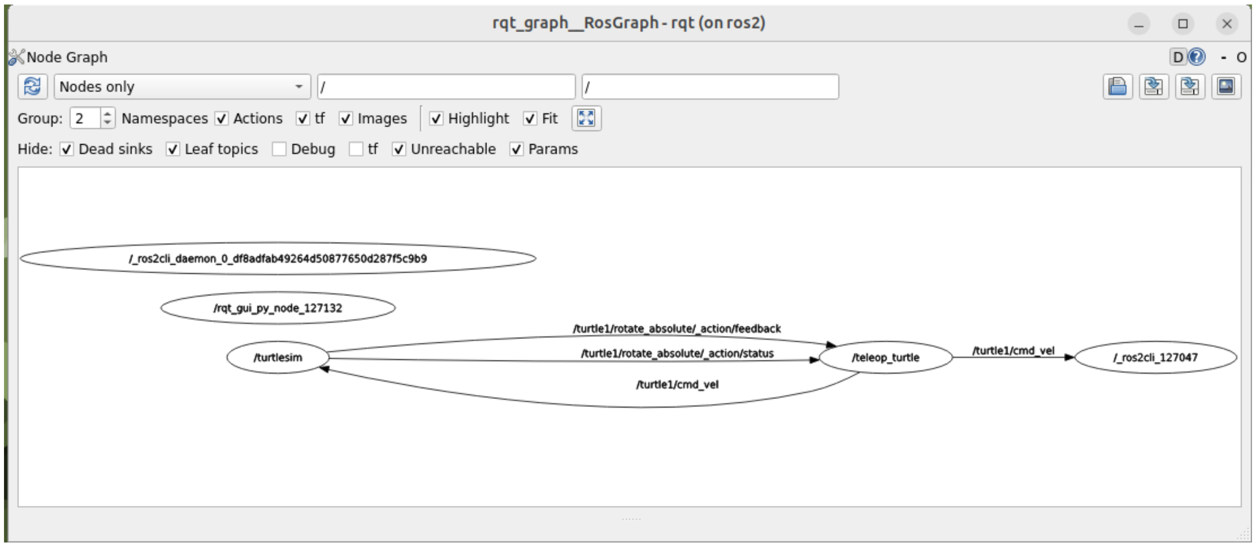
For even more information, including the message type used for each topic, we can use the verbose option:

```
ros2 topic list -v
```

Keep Turtlesim running for use in the next section.

9 Understanding ROS services

Services are another method nodes can use to pass data between each other. While topics are typically used to exchange a continuous stream of data, a service allows one node to *request* data from another node, and receive a *response*. Requests and responses are to services as messages are to topics: that is, they are containers of relevant information for their associated service or topic.


rqt_graph__RosGraph - rqt (on ros2)

Node Graph

Nodes only

Group: 2 Namespaces [✓] Actions [✓] tf [✓] Images [✓] Highlight [✓] Fit

Hide: [✓] Dead sinks [✓] Leaf topics [ ] Debug [ ] tf [✓] Unreachable [✓] Params

/_ros2cli_daemon_0_df8adfab49264d50877650d287f5c9b9
/rqt_gui_py_node_127132
/turtle1/rotate_absolute/_action/feedback
/turtle1/rotate_absolute/_action/status
/turtlesim
/teleop_turtle
/turtle1/cmd_vel
/_ros2cli_127047
/turtle1/cmd_vel

Figure 5: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

9.1 Using rosservice

The `ros2 service` tool is analogous to `ros2 topic`, but for services rather than topics. We can call

```
ros2 service list
```

to show all the services offered by currently running nodes.

We can also see what type of data is included in a request/response for a service. Check the service type for the `/clear` service by running

```
ros2 service type /clear
```

This tells us that the service is of type `std_srvs/srv/Empty`, which means that the service does not require any data as part of its request, and does not return any data in its response.

9.2 Calling services

Let's try calling the the `/clear` service. While this would usually be done programmatically from inside a node, we can do it manually using the `ros2 service call` command. The syntax is

```
ros2 service call [service] [service type] [JSON request message]
```

Because we saw what type `/clear` service has, we can call

```
ros2 service call /clear std_srvs/srv/Empty "{}"
```

If we look back at the `turtlesim` window, we see that our `/clear` call has cleared the background (you can no longer see the path the turtle traveled).

Next, we will call services that require arguments. Use `ros2 service type` to find the datatype for the `/spawn` service. The query should return `turtlesim/srv/Spawn`, which tells us that the service is of type `Spawn`, and that this service type is defined in the `turtlesim` package. Use `ros2 pkg prefix` to get the location of the `turtlesim` package. You'll find that you'll need to append `/share` to the end of that file path to find the actual package. This is an artifact of `ament`'s filesystem layout.

Now that you're in the right package, open the `Spawn.srv` service definition, located in the package's `/srv` subfolder. The file should look like

```
float32 x
float32 y
float32 theta
string name
---
string name
```

The first portion of `Spawn.srv` tells us that the `/spawn` service takes four arguments in its request: three decimal numbers giving the position (`x`, `y`) and orientation (`theta`) of the new turtle, and a single string (`name`) specifying the new turtle's name. The second portion tells us that the service returns one data item: a string with the name we specified in the request.

Call the `/spawn` service to create a new turtle, specifying the values for each of the four arguments (in order):

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 2.0, theta: 1.2, name: 'new_turtle'}"
```

The service call returns the name of the newly created turtle, and you should see the second turtle appear in the `turtlesim` window.

Checkpoint 2

Submit a checkoff request at tinyurl.com/fa24-106alab for a staff member to come and check off your work. At this point you should be able to:

- Explain what a **node**, **topic**, and **message** are
- Drive your turtle around the screen using arrow keys
- Use ROS's utility functions to view data on topics and messages

---

10 Publisher/Subscriber Pair Example

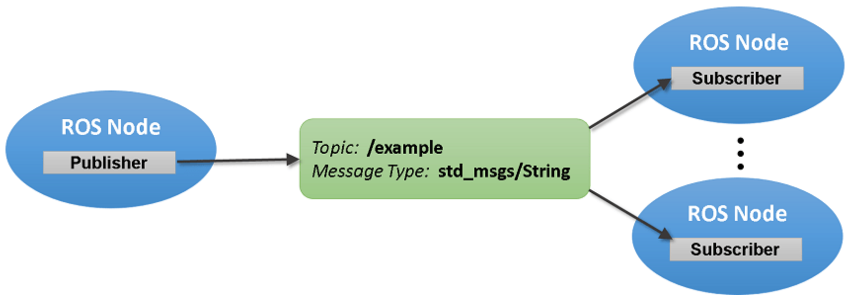


Figure 6: Message-based publishers and subscribers. Source: MathWorks

Your `ros_workspaces` folder should contain the new (unbuilt) workspace `lab1`, containing two packages: `chatter` and `turtle_patrol`. Let's examine the pre-written publisher/subscriber pair in `chatter` to better understand how they work together.

1. Build the `lab1` workspace using `colcon build`.
2. Source the appropriate `setup.bash` file ("`source install/setup.bash`") so that ROS will be able to locate the `lab1` packages.

3. Verify that ROS can find the newly unzipped package by running `ros2 pkg prefix chatter`.

The `/chatter` directory of the `chatter` package should contain two files: `publisher_member_function.py` and `subscriber_member_function.py`, which are Python programs that run a ROS node. The `publisher_member_function.py` program generates text messages and publishes them on the `/chatter_talk` topic, while the `subscriber_member_function.py` program subscribes to this topic and prints the received messages to the terminal.

Now, from your `chatter` package, open up the file `setup.py` file. Towards the bottom of the file, you'll find the following

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main',
        ],
},
```

This is the section of our `setup.py` file where we define our `entry points`. When you build a Python-based ROS 2 package, you use this section to tell ROS which functions should be exposed as runnable nodes. ROS 2 needs a clear registry of "executable names" → "module:function" mappings for the following reasons:

- **Discoverability:** ROS can know exactly what to launch and where to find it when we use the command `ros2 run package_name executable_name`
- **Decoupling:** You can name your script files without breaking the launch command. For example, you will note that one of our Python files is called `publisher_member_function.py`, but we will execute it by calling it `talker`.

Without `entry_points`, ROS 2 wouldn't know which of your Python modules to run as a node, so declaring them is a one-time step that enables the usual `ros2 run` workflow.

You can always check what executables exist in your current path by running `ros2 pkg executables package_name`. Run this command in an appropriately sourced terminal, and you will find that it lists `chatter listener` and `chatter talker` as the executables available. Now let's see them in action!

1. In an appropriately sourced terminal, try running the publisher by executing

   ```
   ros2 run chatter talker
   ```

2. Now that the publisher is running, run the subscriber by opening a new terminal window and following a similar process to run `example_sub.py`. Examine the behavior of the publisher and subscriber.
3. Look over each of the files to understand how they work. Take advantage of the provided comments to understand their contents. *What happens if you start multiple instances of the same publisher or subscriber file in different terminal windows?*

11 Writing a publisher/subscriber pair

Note: Please read through **all** of Section 11 before beginning coding!

Some general tips:

- You can create a new file from the terminal by using `nano <filename>` or `vim <filename>`. Alternatively, you can boot up VS Code with `code .`
- Make sure to source the workspace `setup.bash` file before running any programs

11.1 What you'll be creating

Now you're ready to write your own publisher/subscriber pair! Instead of sending messages you generate in your Python script, let's send messages that the user inputs into the Terminal. In addition to the user-generated message, we also want to send the timestamp of that message. At a high level, you expect your nodes to do the following:

Publisher

1. Prompt the user to enter a line of text (you might find the Python function `input()` helpful)

   ```
   Please enter a line of text and press <Enter>:
   ```

2. Generate a message containing the user's text and a timestamp of when the message was entered. You should look into the documentation for the Node's `get_clock()` function here. You are encouraged to use the Internet look up how you can then get the nanoseconds from the Clock datatype to use as your timestamp.
3. Publish the message on the `/user_messages` topic
4. Continue prompting the user for input until the node is killed

Subscriber

1. Subscribe to the `/user_messages` topic and wait to receive messages
2. When a message is received, print it to the command line using the format

   ```
   Message: <message>, Sent at: <timestamp>, Received at: <timestamp>
   ```

   Note: The received `<timestamp>` is NOT part of the received message, which should contain only a single message and timestamp. Where does it come from?

3. Wait for more messages until the node is killed

11.2 Steps to follow

We recommend using the example code in `chatter` to help you get started. You'll need to complete the following steps:

1. Create a new package named `my_chatter_msgs` that will hold your new custom message type. In it, define a new message type that holds the user input (a string) and message timestamp (a number). More in 11.3.2
2. Create a new package `my_chatter` with the appropriate dependencies, including the `my_chatter_msgs` package. You can refer to the `chatter` package. If you have difficulty, refer to Lab 1, Section 5.2.
3. Inside of your `my_chatter` package's `my_chatter` directory, write Python code for your two new publisher and subscriber nodes.
4. Create entry points for your new Python files to be run as executables associated with your `my_chatter` package.
5. Build the new package
6. Run and test both nodes

It might be interesting to see if you can detect any discrepancy between when the messages are created in the publisher and when they are received by the subscriber; this is why we ask you to print both timestamps!

11.3 Generating a custom message type

The sample publisher/subscriber in `chatter` uses the primitive message type `string`, found in the `std_msgs` package. However, to send both the message text and its timestamp, we need to create a new custom message type with both data structures in it.

A ROS message definition is a text file of the form:

```
<< data_type1 >>  << name_1 >>
<< data_type2 >>  << name_2 >>
<< data_type3 >>  << name_3 >>
...
<< data_typen >>  << name_n >>
```

(Don't include the << and >> in the message file.)
Examples of `data_type` include

- `int8, int16, int32, int64`
- `float32, float64`
- `string`
- other msg types specified as `package/MessageName`
- variable-length `array[]` and fixed-length `array[N]`

You may find the complete list of buit-in types currently supported in the official ROS2 documentation.
Each `name` identifies one of the data fields contained in the message and must be unique.

11.3.1 Defining a new message type

The sample publisher/subscriber in `chatter` uses the primitive message type `string`, found in the `std_msgs` package. However, to send both the message text and its timestamp, we need to create a new custom message type with both data structures in it.

A ROS message definition is a text file of the form:

```
<< data_type1 >>  << name_1 >>
<< data_type2 >>  << name_2 >>
<< data_type3 >>  << name_3 >>
...
<< data_typen >>  << name_n >>
```

Examples of `data_type` include

- `int8, int16, int32, int64`
- `float32, float64`
- `string`
- other msg types specified as `package/MessageName`
- variable-length `array[]` and fixed-length `array[N]`

You may find the complete list of buit-in types currently supported in the official ROS2 documentation.
Each `name` identifies one of the data fields contained in the message and must be unique.

11.3.2 Creating an interface/message package and new message types

In ROS 2, it is quite inconvenient to create custom message types in any package built with `ament_python` (we'll share more on why later). Consequently, it is common practice to use `ament_cmake` to create an **interface package** to hold any custom messages, services, or actions that you may need to create. These are sometimes called **message packages** if all they hold are custom message types.

In your `lab1/src` directory, let's create this message package. Do so by running

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

Now in your new `my_chatter_msgs` directory, create a new directory called `msg`. This is the name that ROS will look for when searching for your custom message type.

In the `msg` folder, create a new message description file called `TimestampString.msg`. Ensure that it contains the two data types we desire based on the information above.

Next, we need to tell `colcon build` that we have a new message type that needs to be built. To do this we must modify both the `package.xml` and `CMakeLists.txt` in our `my_chatter_msgs` package.

Let's modify our `my_chatter_msgs/package.xml` file first. In this file, add the following:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

The first line ensures that your message files get built properly. The second makes sure they can be run properly. The third makes sure that your new package's custom types can be considered for tools that care about custom messages.

Now, we must update our `my_chatter_msgs/CMakeLists.txt` file. You may have noticed that you do not have access to a `CMakeLists.txt` for any packages you created using `ament_python` (and you can go back and check `foo` and `bar`). There are necessary edits we make to this file in order to use our custom message type, so you can see why we created an interface package.

In `my_chatter_msgs/CMakeLists.txt`, add the following:

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
    "msg/TimestampString.msg"
)
find_package(rosidl_default_runtime REQUIRED)
ament_export_dependencies(rosidl_default_runtime)
```

The first line brings in the code generators for messages, services, and actions you've defined. The second line generates the code for the custom files you've defined. The two last lines export the dependencies so other packages can use these custom types.

Now, you can build the new message type using `colcon build`. Verify that your message is being built correctly by confirming that ROS can find the `my_chatter_msgs` by running `ros2 pkg prefix my_chatter_msgs`.
Keep in mind that `TimestampString` is a class, and the input message must be instantiated as an object of this class. To use the new message, you will add a corresponding `import` statement to any Python programs that use it:

```
from my_chatter_msgs.msg import TimestampString
```

Do this for both the publisher and subscriber you are writing, integrating the new `TimestampString` type you just created. Think about how you would create a new instance of this type when sending the message from your publisher

11.3.3 Writing your new talker and listener nodes

Use the files from the `chatter` package to base your files on in `my_chatter`. Don't forget to create the appropriate entry points so you can call your nodes with

```
ros2 run my_chatter my_talker
ros2 run my_chatter my_listener
```

As a reminder, you are encouraged and expected to use the Internet as a resource to look up documentation to complete this task!

Checkpoint 3

Submit a checkoff request at tinyurl.com/fa25-106alab for a staff member to come and check off your work. At this point you should be able to:

- Explain all the contents of your `lab1` workspace
- Discuss the new message type you created for the `user_messages` topic
- Demonstrate that your package builds successfully
- Demonstrate the functionality of your new nodes using `TimestampString`

---

12 Pushing to Github

We highly recommend backing up your labs to Github to share code between partners and have access to your lab code outside of class.

If you're new to git, feel free to reference this guide from CS61B. It has far more detail than we need in this course, but it can be helpful to get a better understanding of such a widely used tool.

The one-time setup instructions to set up git on your instructional machine are borrowed from CS61C.

1. Set your git identity

   ```
   git config --global user.name "your_name"
   git config --global user.email your_email@example.com
   ```

2. Generate a SSH key pair by running the command below. When it asks you what file directory to place the key in, select the default directory by pressing enter. When it asks for a password, we recommend setting a password so no one else can login to your GitHub using your SSH key and destroy your work, but if you like, you can also skip this step by pressing enter.

   ```
   ssh-keygen -t ed25519 -C "your_email@example.com"
   ```

3. Add key to ssh-agent.

   ```
   eval "$(ssh-agent -s)" # Start the SSH agaent
   ssh-add ~/.ssh/id_ed25519 # Add your key to your SSH agent
   ```

4. Print out your public key by running

```
cat ~/.ssh/id_ed25519.pub
```

The output should look similar to the following (length may differ):

```
ssh-ed25519 AAAAC3NzaC1lZDI1N6jpH3Bnbebi7Xz7wMr20LxZCKi3U8UQTE5AAAAIBTc2HwlbOi8T
your_email@example.com
```

5. In your browser, go to GitHub $\implies$ Settings $\implies$ SSH and GPG Keys $\implies$ New SSH key and add your public key. Set the title to something that helps you remember what device the key is on (e.g. `EECS106A Lab Machine`)

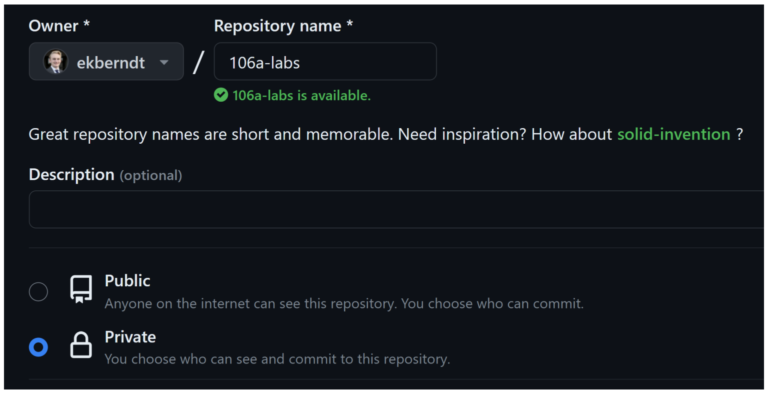
6. Try connecting to GitHub with SSH:

```
ssh -T git@github.com
```

If all went well, you should see something like:

```
Hi USERNAME! You've successfully authenticated, but GitHub does not provide shell access.
```

First, we want to go to `github.com` (create an account if you do not have one) and then create a new repository. Make sure you change the repository from public to private!



Owner * ekberndt / Repository name * 106a-labs

106a-labs is available.

Great repository names are short and memorable. Need inspiration? How about solid-invention ?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

[✓ Private]
You choose who can see and commit to this repository.

Figure 7: GitHub Repository Creation

After creating a new repository on Github, you can link it to your `ros_workspaces` directory. Since we, the lab course staff, already created a git project on `github.com` and had you clone it to your computer, you already have a git project created in `/ros_workspaces` directory. So lets cd into that directory

```
cd ~/ros_workspaces
```

Now, to check that this directory has a git project in it we can run

```
ls -a
```

which lists all the normal files plus the hidden files (files with a . in front of them). We should see there a folder called `.git`. This is a telltale sign that a git project has already been created in this folder.

Now, we want to tell our local git project where our remote git cloud storage (on `github.com`) is located. So first, go to your newly created `github.com` repository. Now make sure that you select SSH and copy your Github repository SSH URL, NOT your HTTPS link, or you will not be able to commit changes to your repository. Now we can add our `github.com` repository as a remote to our local git project as follows

```
git remote add origin git@github.com:{GITHUB_USERNAME}/{YOUR_REPO}.git
```

where `git@github.com:{GITHUB_USERNAME}/{YOUR_REPO}.git` is the link you should copy off of github.com itself in your repository, as mentioned above. Please replace `{GITHUB_USERNAME}` with your github username and `{YOUR_REPO}` with the name of your newly created `github.com` repository.
Git is a version control system that allows you to save snapshots of your code in checkpoints called commits. To add your changes to a git commit, in the root of your git repository (in `/ros_workspaces`) run

```
git add *
```

where * is the wildcard operator specifying to add every file in this directory. Now that git knows what files to add the commit, we can create the commit itself with a custom message

```
git commit -m "Complete lab 1"
```

After which, you can push to your repository by running `git push origin`. If you try to `git push` instead of `git push origin`, you will try to push to the class starter code Github repository, which will fail.

Reminder: You may share your repository with your lab partners, but your repository **must be private**. We would like to maintain academic integrity!

13 Additional Lab Etiquette

Because the lab workstations are shared, we sometimes run into strange bugs. The most common is that someone leaves a process running when they leave the lab; when the next person uses the workstation, ROS can get confused by the additional processes running from the prior user. A particularly insidious example is leaving a `roscore` master node running in the background, causing the next person to be unable to run a master node with proper communications.

To avoid issues like this in the lab, please do the following before leaving:

- `Ctrl+C` out of every terminal when you are done. **Do not Ctrl+Z**. While Ctrl+Z may look like it stops your process, it really only pauses it, and the process will continue to run in the background.
- Don't forget! You're probably still in your `ros2` container. To exit, run th ecommand `exit` from every terminal.

Congratulations on finishing Lab 1!