

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Configuring environment

Goal: This tutorial will show you how to prepare your ROS 2 environment.

Tutorial level: Beginner

Time: 5 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Source the setup files](#)
 - [2 Add sourcing to your shell startup script](#)
 - [3 Check environment variables](#)
- [Summary](#)
- [Next steps](#)

Background

ROS 2 relies on the notion of combining workspaces using the shell environment. “Workspace” is a ROS term for the location on your system where you’re developing with ROS 2. The core ROS 2 workspace is called the underlay. Subsequent local workspaces are called overlays. When developing with ROS 2, you will typically have several workspaces active concurrently.

Combining workspaces makes developing against different versions of ROS 2, or against different sets of packages, easier. It also allows the installation of several ROS 2 distributions (or “distros”, e.g. Dashing and Eloquent) on the same computer and switching between them.

This is accomplished by sourcing setup files every time you open a new shell, or by adding the source command to your shell startup script once. Without sourcing the setup files, you won’t be able to access ROS 2 commands, or find or use ROS 2 packages. In other words, you won’t be able to use ROS 2.

Prerequisites

Before starting these tutorials, install ROS 2 by following the instructions on the [ROS 2 Installation](#) page.

The commands used in this tutorial assume you followed the binary packages installation guide for your operating system (deb packages for Linux). You can still follow along if you built from source, but the path to your setup files will likely be different. You also won't be able to use the `sudo apt install ros-<distro>-<package>` command (used frequently in the beginner level tutorials) if you install from source.

If you are using Linux or macOS, but are not already familiar with the shell, [this tutorial](#) will help.

Tasks

1 Source the setup files

You will need to run this command on every new shell you open to have access to the ROS 2 commands, like so:

Linux

macOS

Windows

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

Replace `.bash` with your shell if you're not using bash. Possible values are: `setup.bash`, `setup.sh`, `setup.zsh`.

ⓘ Note

The exact command depends on where you installed ROS 2. If you're having problems, ensure the file path leads to your installation.

2 Add sourcing to your shell startup script

If you don't want to have to source the setup file every time you open a new shell (skipping task 1), then you can add the command to your shell startup script:

Linux

macOS

Windows

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

To undo this, locate your system's shell startup script and remove the appended source command.

3 Check environment variables

Sourcing ROS 2 setup files will set several environment variables necessary for operating ROS 2. If you ever have problems finding or using your ROS 2 packages, make sure that your environment is properly set up using the following command:

Linux

macOS

Windows

```
$ printenv | grep -i ROS
```

Check that variables like `ROS_DISTRO` and `ROS_VERSION` are set.

```
ROS_VERSION=2
ROS_PYTHON_VERSION=3
ROS_DISTRO=humble
```

If the environment variables are not set correctly, return to the ROS 2 package installation section of the installation guide you followed. If you need more specific help (because environment setup files can come from different places), you can [get answers](#) from the community.

3.1 The `ROS_DOMAIN_ID` variable

See the [domain ID](#) article for details on ROS domain IDs.

Once you have determined a unique integer for your group of ROS 2 nodes, you can set the environment variable with the following command:

Linux

macOS

Windows

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

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

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

3.2 The `ROS_LOCALHOST_ONLY` variable

By default, ROS 2 communication is not limited to localhost. `ROS_LOCALHOST_ONLY` environment variable allows you to limit ROS 2 communication to localhost only. This means your ROS 2 system, and its topics, services, and actions will not be visible to other computers on the local network. Using `ROS_LOCALHOST_ONLY` is helpful in certain settings, such as classrooms, where multiple robots may publish to the same topic causing strange behaviors. You can set the environment variable with the following command:

Linux

macOS

Windows

```
export ROS_LOCALHOST_ONLY=1
```

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

```
echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

Summary

The ROS 2 development environment needs to be correctly configured before use. This can be done in two ways: either sourcing the setup files in every new shell you open, or adding the source command to your startup script.

If you ever face any problems locating or using packages with ROS 2, the first thing you should do is check your environment variables and ensure they are set to the version and distro you intended.

Next steps

Now that you have a working ROS 2 installation and you know how to source its setup files, you can start learning the ins and outs of ROS 2 with the [turtlesim tool](#).

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Using `turtlesim`, `ros2`, and `rqt`

Goal: Install and use the turtlesim package and rqt tools to prepare for upcoming tutorials.

Tutorial level: Beginner

Time: 15 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Install turtlesim](#)
 - [2 Start turtlesim](#)
 - [3 Use turtlesim](#)
 - [4 Install rqt](#)
 - [5 Use rqt](#)
 - [6 Remapping](#)
 - [7 Close turtlesim](#)
- [Summary](#)
- [Next steps](#)
- [Related content](#)

Background

Turtlesim is 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 a robot simulation later on.

The ros2 tool is how the user manages, introspects, and interacts with a ROS system. It supports multiple commands that target different aspects of the system and its operation. One might use it to start a node, set a parameter, listen to a topic, and many more. The ros2 tool is part of the core ROS 2 installation.

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

This tutorial touches upon core ROS 2 concepts, like nodes, topics, and services. All of these concepts will be elaborated on in later tutorials; for now, you will simply set up the tools and get a feel for them.

Prerequisites

The previous tutorial, [Configuring environment](#), will show you how to set up your environment.

Tasks

1 Install turtlesim

As always, start by sourcing your setup files in a new terminal, as described in the [previous tutorial](#).

Install the turtlesim package for your ROS 2 distro:

Linux

macOS

Windows

```
$ sudo apt update
$ sudo apt install ros-humble-turtlesim
```

To check if the package is installed, run the following command, which should return a list of turtlesim's executables:

```
$ ros2 pkg executables turtlesim
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

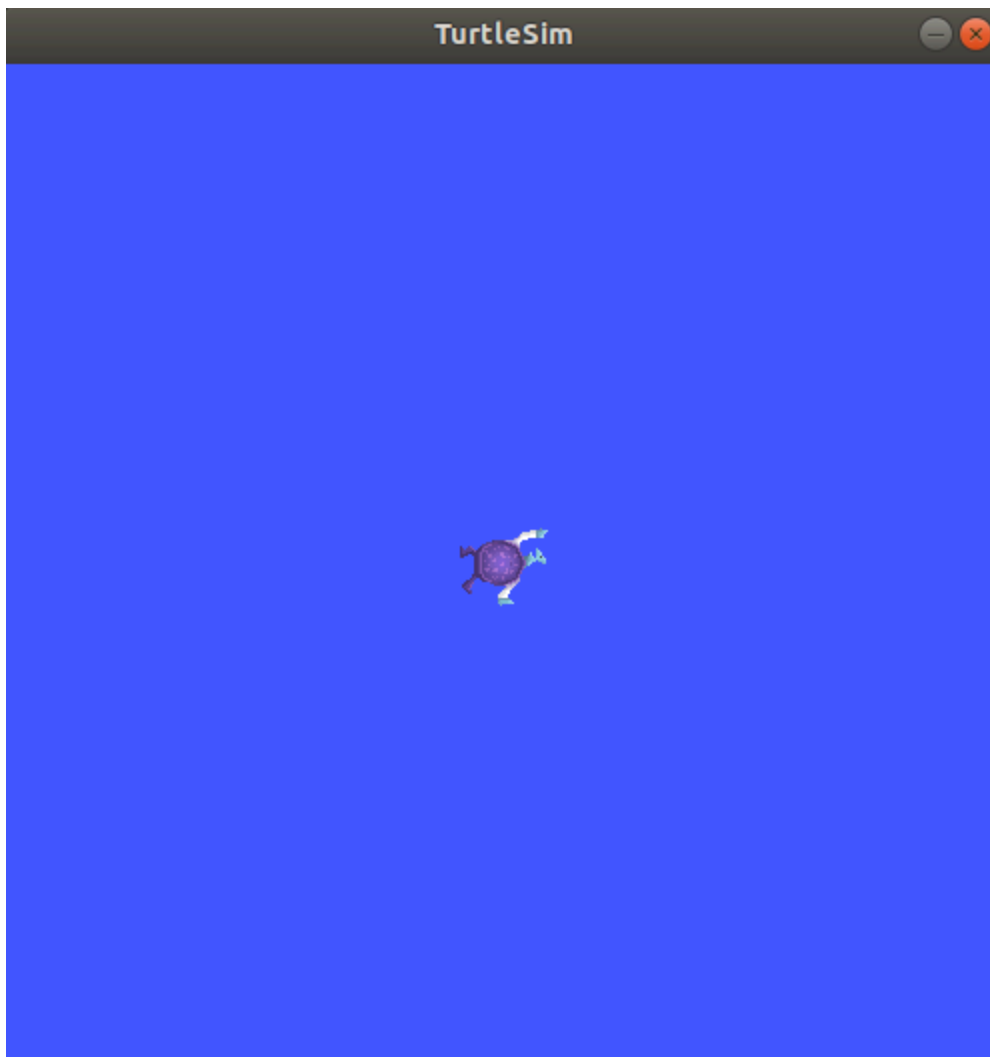
2 Start turtlesim

To start turtlesim, enter the following command in your terminal:

```
$ ros2 run turtlesim turtlesim_node  
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim  
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

Under the command, you will see messages from the node. There you can see the default turtle's name and the coordinates where it spawns.

The simulator window should appear, with a random turtle in the center.



3 Use turtlesim

Open a new terminal and source ROS 2 again.

Now you will run a new node to control the turtle in the first node:

```
$ ros2 run turtlesim turtle_teleop_key
```


At this point you should have three windows open: a terminal running `turtlesim_node`, a terminal running `turtle_teleop_key` and the turtlesim window. Arrange these windows so that you can see the turtlesim window, but also have the terminal running `turtle_teleop_key` active so that you can control the turtle in turtlesim.

Use the arrow keys on your keyboard to control the turtle. It will move around the screen, using its attached “pen” to draw the path it followed so far.

Note

Pressing an arrow key will only cause the turtle to move a short distance and then stop. This is because, realistically, you wouldn’t want a robot to continue carrying on an instruction if, for example, the operator lost the connection to the robot.

You can see the nodes, and their associated topics, services, and actions, using the `list` subcommands of the respective commands:

```
$ ros2 node list
$ ros2 topic list
$ ros2 service list
$ ros2 action list
```

You will learn more about these concepts in the coming tutorials. Since the goal of this tutorial is only to get a general overview of turtlesim, you will use `rqt` to call some of the turtlesim services and interact with `turtlesim_node`.

4 Install `rqt`

Open a new terminal to install `rqt` and its plugins:

Ubuntu Linux

macOS

Windows

```
$ sudo apt update
$ sudo apt install '~nros-humble-rqt*'
```

To run `rqt`:

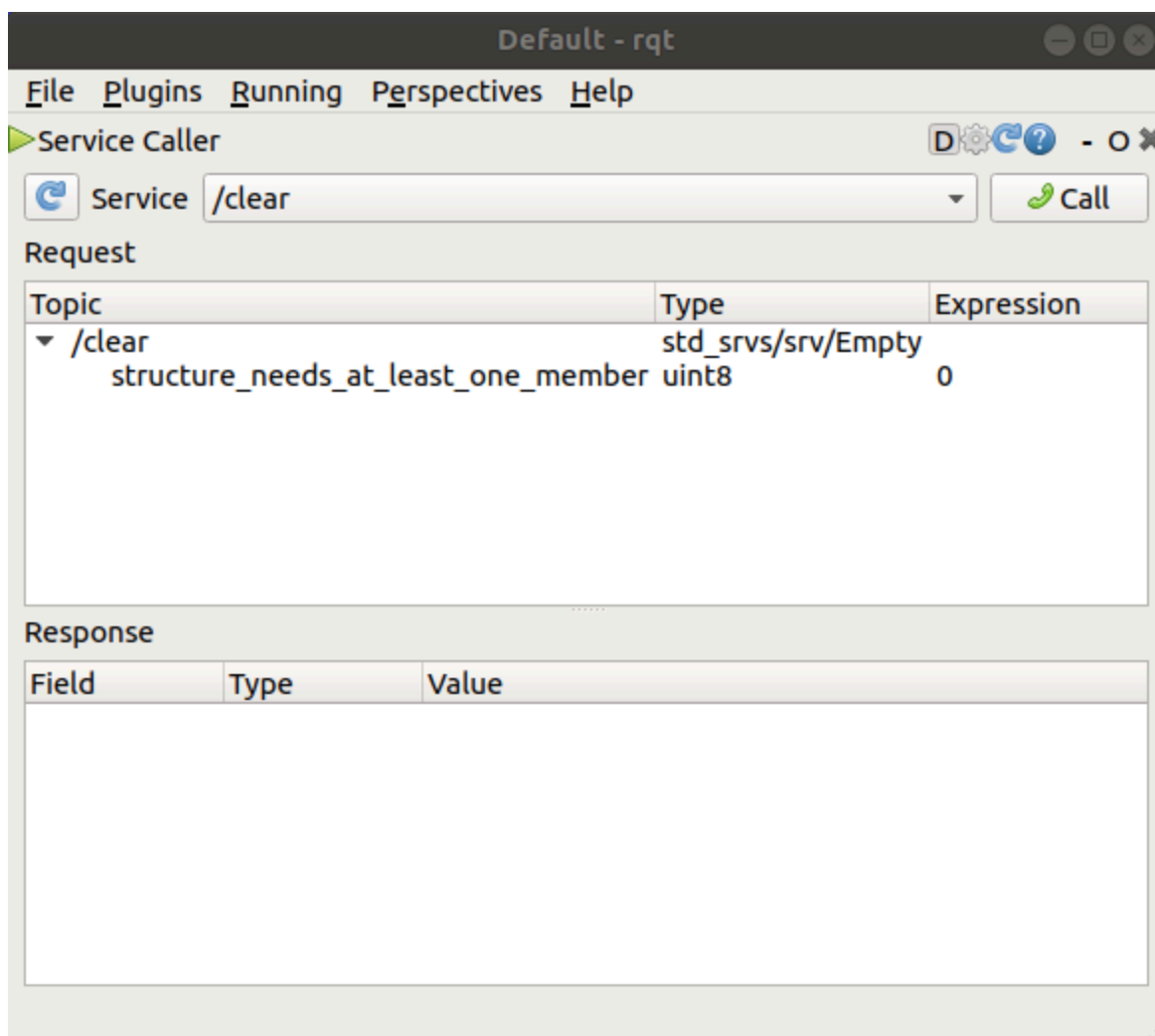
\$ rqt

5 Use rqt

When running rqt for the first time, the window will be blank. No worries; just select **Plugins > Services > Service Caller** from the menu bar at the top.

! Note

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.



Use the refresh button to the left of the **Service** dropdown list to ensure all the services of your turtlesim node are available.

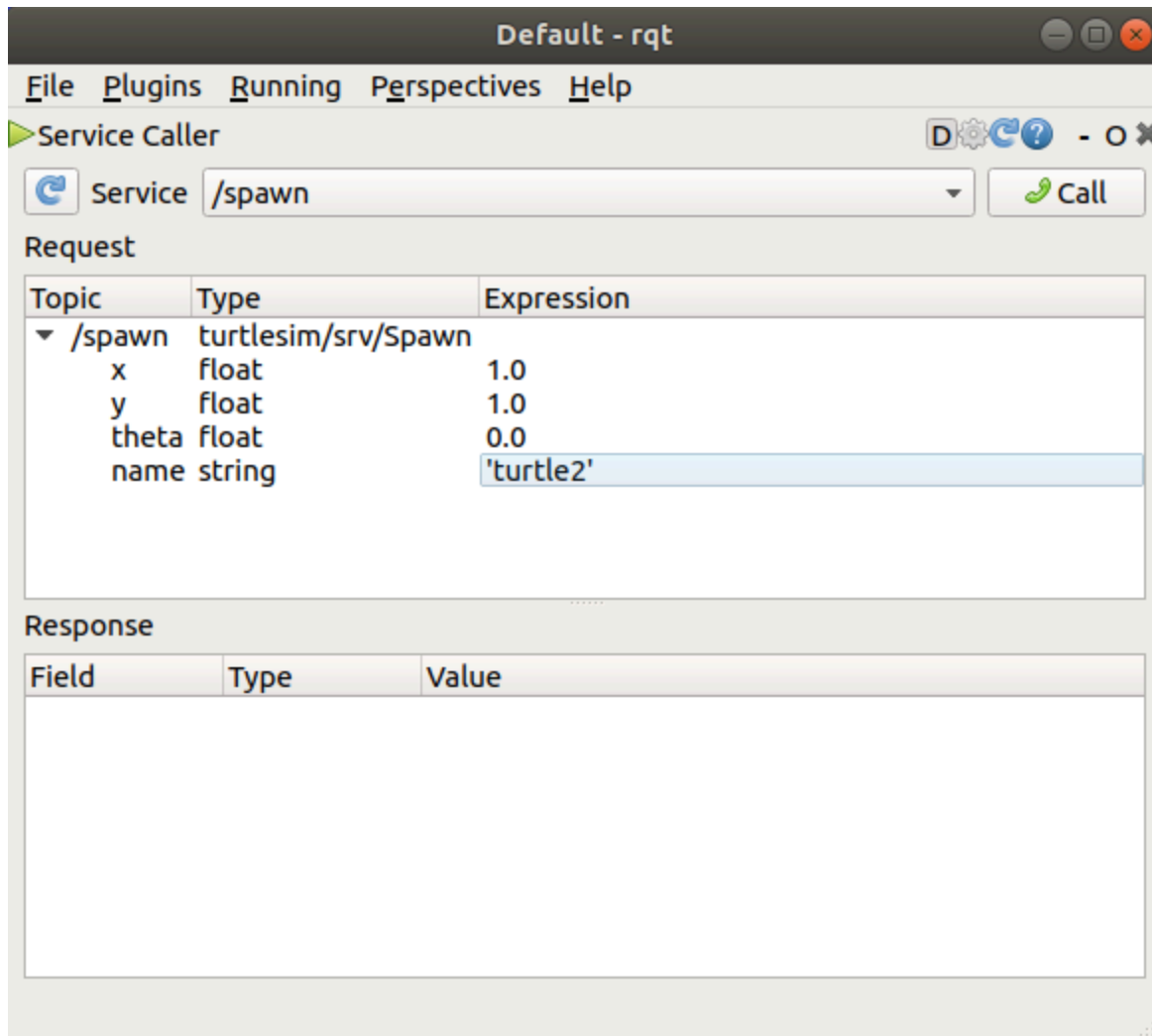
Click on the **Service** dropdown list to see turtlesim's services, and select the `/spawn` service.

5.1 Try the spawn service

Let's use rqt to call the `/spawn` service. You can guess from its name that `/spawn` will create another turtle in the turtlesim window.

Give the new turtle a unique name, like `turtle2`, by double-clicking between the empty single quotes in the **Expression** column. You can see that this expression corresponds to the value of **name** and is of type **string**.

Next enter some valid coordinates at which to spawn the new turtle, like `x = 1.0` and `y = 1.0`.



! Note

If you try to spawn a new turtle with the same name as an existing turtle, like the default `turtle1`, you will get an error message in the terminal running `turtlesim_node`:

```
[ERROR] [turtlesim]: A turtle named [turtle1] already exists
```

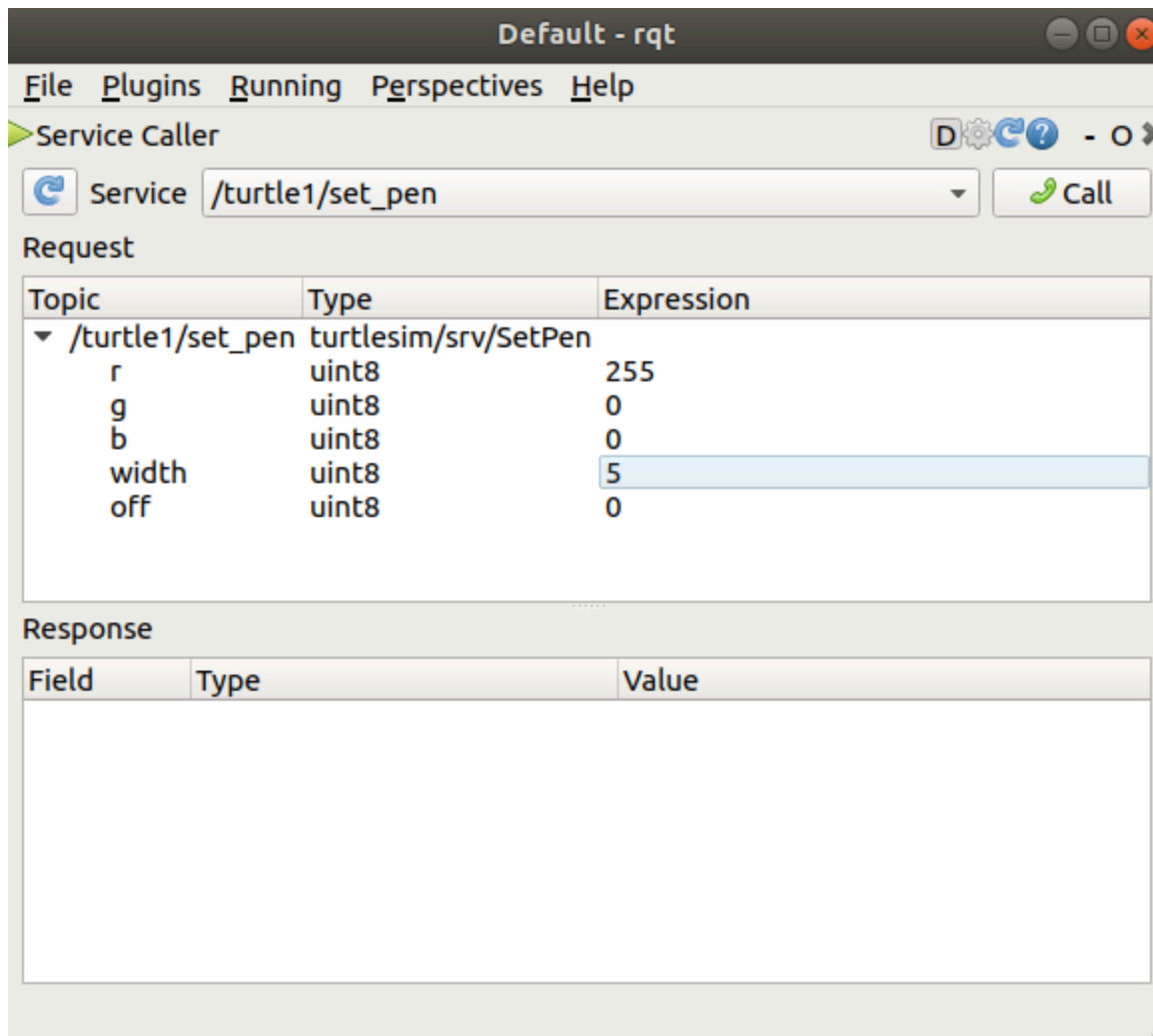
To spawn `turtle2`, you then need to call the service by clicking the **Call** button on the upper right side of the rqt window.

If the service call was successful, you should see a new turtle (again with a random design) spawn at the coordinates you input for **x** and **y**.

If you refresh the service list in rqt, you will also see that now there are services related to the new turtle, `/turtle2/...`, in addition to `/turtle1/...`.

5.2 Try the `set_pen` service

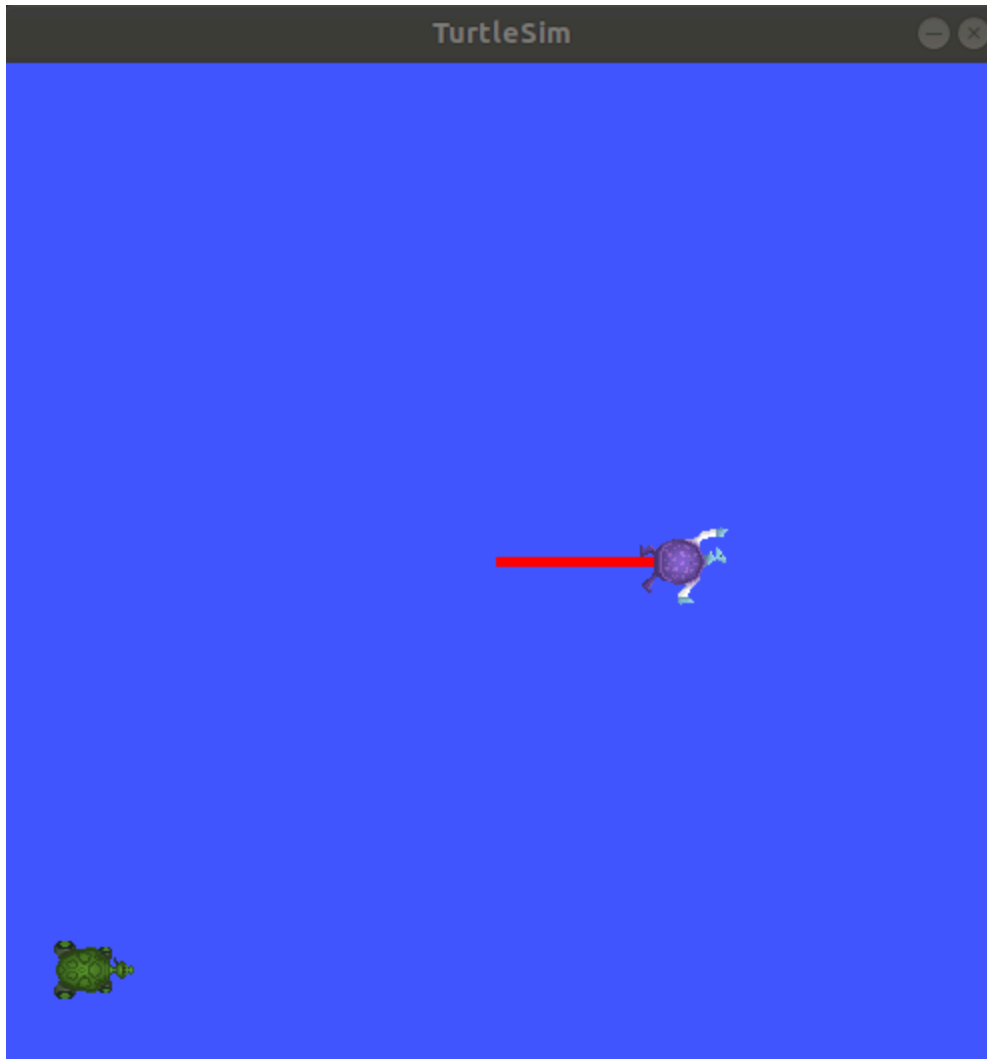
Now let's give `turtle1` a unique pen using the `/set_pen` service:



The values for **r**, **g** and **b**, which are between 0 and 255, set the color of the pen `turtle1` draws with, and **width** sets the thickness of the line.

To have `turtle1` draw with a distinct red line, change the value of **r** to 255, and the value of **width** to 5. Don't forget to call the service after updating the values.

If you return to the terminal where `turtle_teleop_key` is running and press the arrow keys, you will see `turtle1`'s pen has changed.



You've probably also noticed that there's no way to move `turtle2`. That's because there is no teleop node for `turtle2`.

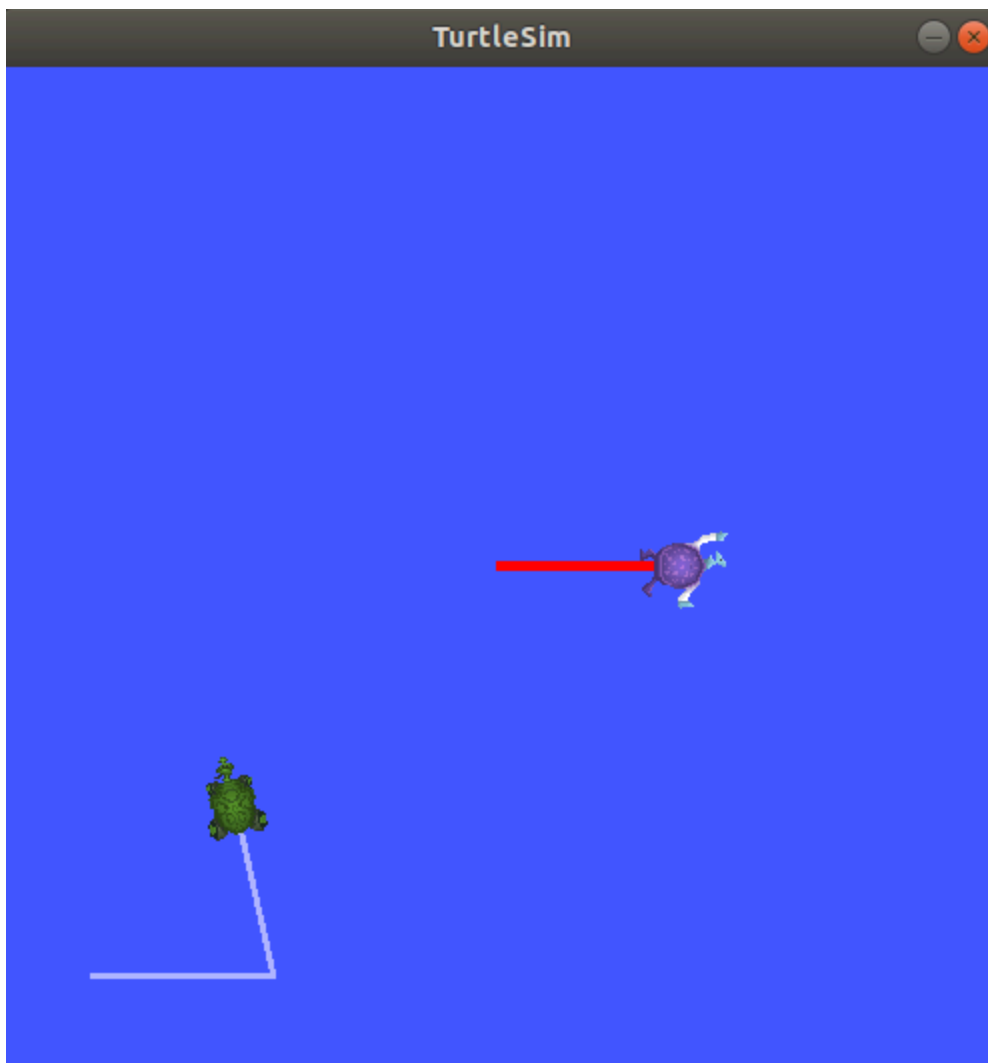
6 Remapping

You need a second teleop node in order to control `turtle2`. 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, source ROS 2, and 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.



7 Close turtlesim

To stop the simulation, you can enter `Ctrl + C` in the `turtlesim_node` terminal, and `q` in the `turtle_teleop_key` terminals.

Summary

Using turtlesim and rqt is a great way to learn the core concepts of ROS 2.

Next steps

Now that you have turtlesim and rqt up and running, and an idea of how they work, let's dive into the first core ROS 2 concept with the next tutorial, [Understanding nodes](#).

Related content

The turtlesim package can be found in the [ros_tutorials](#) repo.

[This community contributed video](#) demonstrates many of the items covered in this tutorial.

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Understanding nodes

Goal: Learn about the function of nodes in ROS 2, and the tools to interact with them.

Tutorial level: Beginner

Time: 10 minutes

Contents

- [Background](#)
 - [1 The ROS 2 graph](#)
 - [2 Nodes in ROS 2](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 ros2 run](#)
 - [2 ros2 node list](#)
 - [3 ros2 node info](#)
- [Summary](#)
- [Next steps](#)
- [Related content](#)

Background

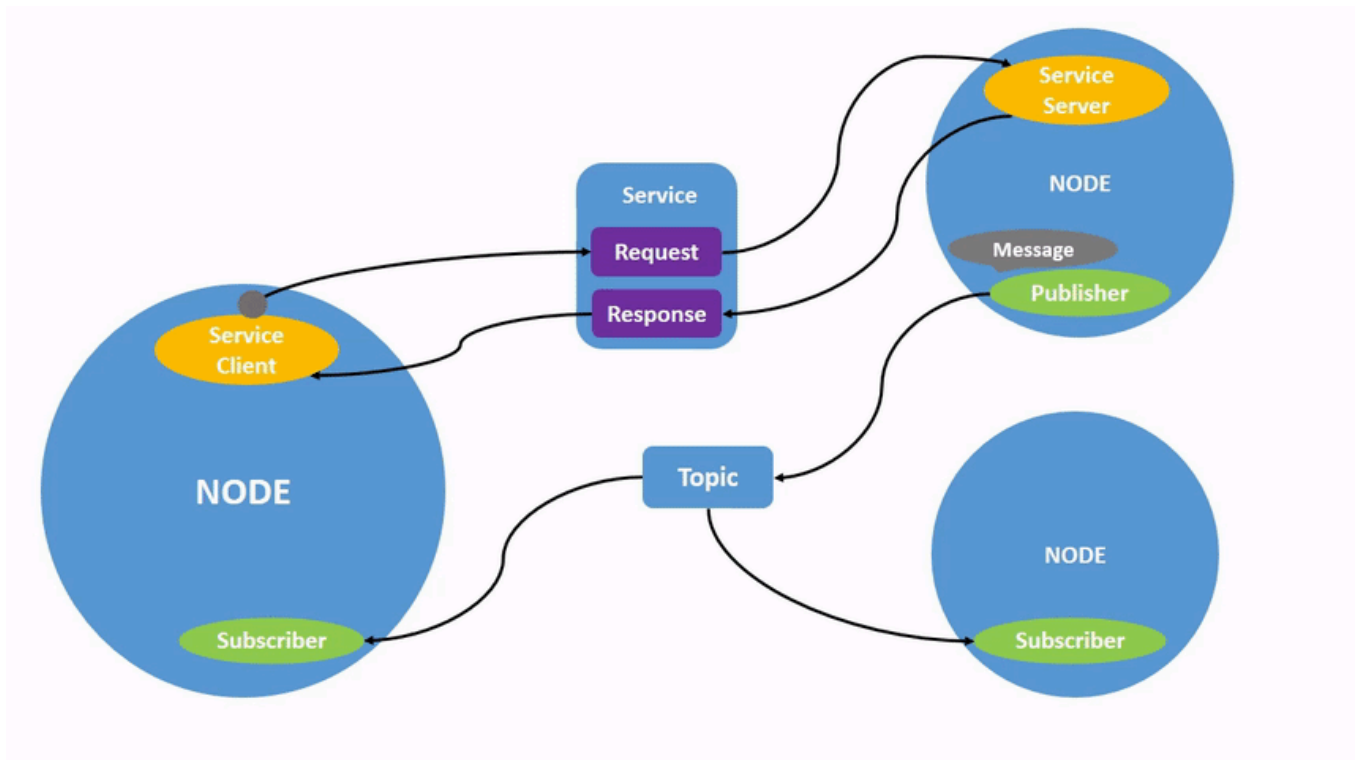
1 The ROS 2 graph

Over the next few tutorials, you will learn about a series of core ROS 2 concepts that make up what is referred to as the “ROS (2) graph”.

The ROS graph is a network of ROS 2 elements processing data together at the same time. It encompasses all executables and the connections between them if you were to map them all out and visualize them.

2 Nodes in ROS 2

Each node in ROS should be responsible for a single, modular purpose, e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder. Each node can send and receive data from other nodes via topics, services, actions, or parameters.



A full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes.

Prerequisites

The [previous tutorial](#) shows you how to install the `turtlesim` package used here.

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

1 `ros2 run`

The command `ros2 run` launches an executable from a package.

```
$ ros2 run <package_name> <executable_name>
```

To run `turtlesim`, open a new terminal, and enter the following command:

```
$ ros2 run turtlesim turtlesim_node
```

The turtlesim window will open, as you saw in the [previous tutorial](#).

Here, the package name is `turtlesim` and the executable name is `turtlesim_node`.

We still don't know the node name, however. You can find node names by using `ros2 node list`

2 ros2 node list

`ros2 node list` will show you the names of all running nodes. This is especially useful when you want to interact with a node, or when you have a system running many nodes and need to keep track of them.

Open a new terminal while turtlesim is still running in the other one, and enter the following command. The terminal will return the node name:

```
$ ros2 node list  
/turtlesim
```

Open another new terminal and start the teleop node with the command:

```
$ ros2 run turtlesim turtle_teleop_key
```

Here, we are referring to the `turtlesim` package again, but this time we target the executable named `turtle_teleop_key`.

Return to the terminal where you ran `ros2 node list` and run it again. You will now see the names of two active nodes:

```
$ ros2 node list  
/turtlesim  
/teleop_turtle
```

2.1 Remapping

Remapping allows you to reassign default node properties, like node name, topic names, service names, etc., to custom values. In the last tutorial, you used remapping on `turtle_teleop_key` to change the `cmd_vel` topic and target **turtle2**.

Now, let's reassign the name of our `/turtlesim` node. In a new terminal, run the following command:

```
$ ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Since you're calling `ros2 run` on turtlesim again, another turtlesim window will open. However, now if you return to the terminal where you ran `ros2 node list`, and run it again, you will see three node names:

```
/my_turtle  
/turtlesim  
/teleop_turtle
```

3 ros2 node info

Now that you know the names of your nodes, you can access more information about them with:

```
$ ros2 node info <node_name>
```

To examine your latest node, `my_turtle`, run the following command:

```
$ ros2 node info /my_turtle
/my_turtle
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/color_sensor: turtlesim/msg/Color
  /turtle1/pose: turtlesim/msg/Pose
Service Servers:
  /clear: std_srvs/srv/Empty
  /kill: turtlesim/srv/Kill
  /my_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /my_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /my_turtle/get_parameters: rcl_interfaces/srv/GetParameters
  /my_turtle/list_parameters: rcl_interfaces/srv/ListParameters
  /my_turtle/set_parameters: rcl_interfaces/srv/SetParameters
  /my_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  /reset: std_srvs/srv/Empty
  /spawn: turtlesim/srv/Spawn
  /turtle1/set_pen: turtlesim/srv/SetPen
  /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
  /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
Service Clients:

Action Servers:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
Action Clients:
```

`ros2 node info` returns a list of subscribers, publishers, services, and actions. i.e. the ROS graph connections that interact with that node.

Now try running the same command on the `/teleop_turtle` node, and see how its connections differ from `my_turtle`.

You will learn more about the ROS graph connection concepts including the message types in the upcoming tutorials.

Summary

A node is a fundamental ROS 2 element that serves a single, modular purpose in a robotics system.

In this tutorial, you utilized nodes created in the `turtlesim` package by running the executables `turtlesim_node` and `turtle_teleop_key`.

You learned how to use `ros2 node list` to discover active node names and `ros2 node info` to introspect a single node. These tools are vital to understanding the flow of data in a complex, real-world robot system.

Next steps

Now that you understand nodes in ROS 2, you can move on to the [topics tutorial](#). Topics are one of the communication types that connects nodes.

Related content

The [Concepts](#) page adds some more detail to the concept of nodes.

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Understanding topics

Goal: Use `rqt_graph` and command line tools to introspect ROS 2 topics.

Tutorial level: Beginner

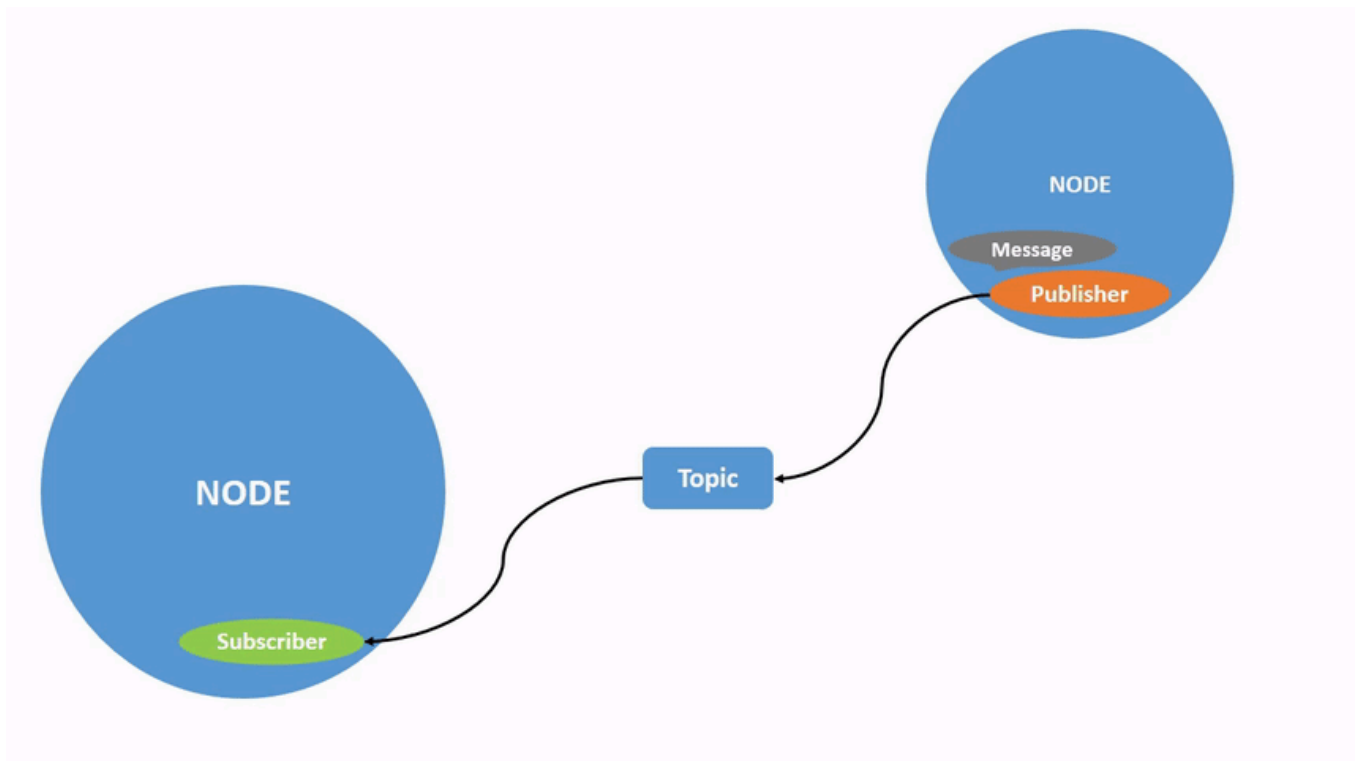
Time: 20 minutes

Contents

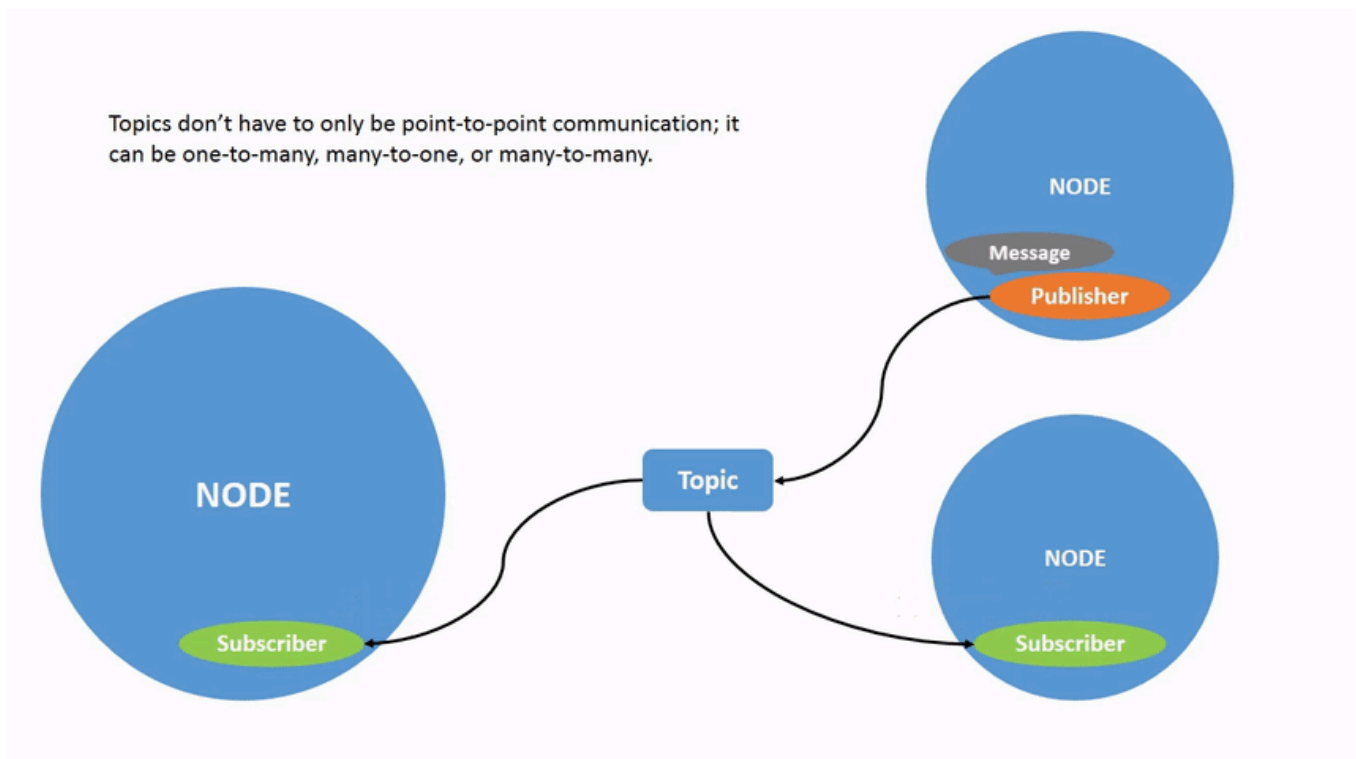
- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Setup](#)
 - [2 rqt_graph](#)
 - [3 ros2 topic list](#)
 - [4 ros2 topic echo](#)
 - [5 ros2 topic info](#)
 - [6 ros2 interface show](#)
 - [7 ros2 topic pub](#)
 - [8 ros2 topic hz](#)
 - [9 ros2 topic bw](#)
 - [10 ros2 topic find](#)
 - [11 Clean up](#)
- [Summary](#)
- [Next steps](#)

Background

ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.



A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.



Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

Prerequisites

The [previous tutorial](#) provides some useful background information on nodes that is built upon here.

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

By now you should be comfortable starting up turtlesim.

Open a new terminal and run:

```
$ ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```
$ ros2 run turtlesim turtle_teleop_key
```

Recall from the [previous tutorial](#) that the names of these nodes are `/turtlesim` and `/teleop_turtle` by default.

2 rqt_graph

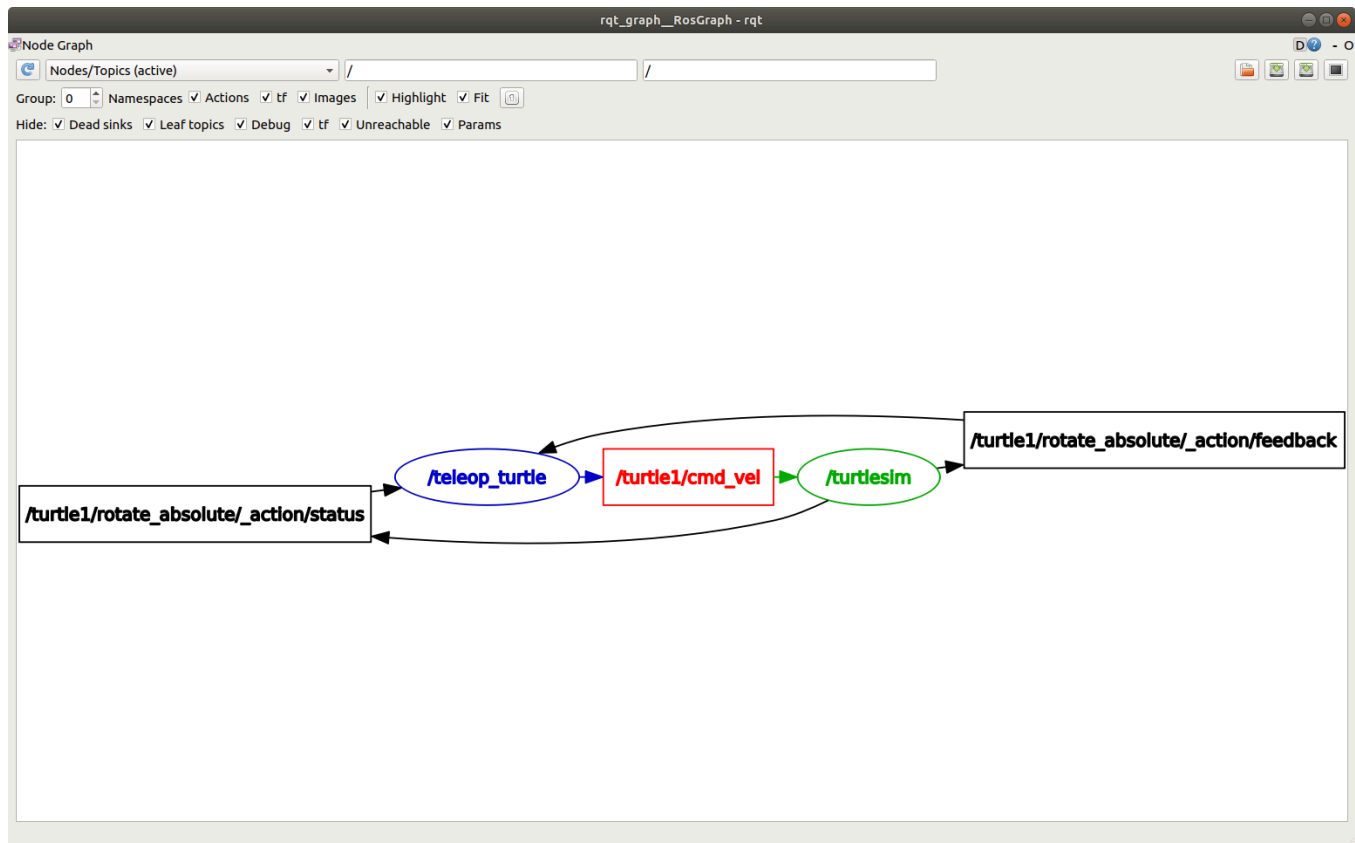
Throughout this tutorial, we will use `rqt_graph` to visualize the changing nodes and topics, as well as the connections between them.

The [turtlesim tutorial](#) tells you how to install rqt and all its plugins, including `rqt_graph`.

To run `rqt_graph`, open a new terminal and enter the command:

```
$ rqt_graph
```


You can also open `rqt_graph` by opening `rqt` and selecting **Plugins > Introspection > Node Graph**.



You should see the above nodes and topic, as well as two actions around the periphery of the graph (let's ignore those for now). If you hover your mouse over the topic in the center, you'll see the color highlighting like in the image above.

The graph is depicting how the `/turtlesim` node and the `/teleop_turtle` node are communicating with each other over a topic. The `/teleop_turtle` node is publishing data (the keystrokes you enter to move the turtle around) to the `/turtle1/cmd_vel` topic, and the `/turtlesim` node is subscribed to that topic to receive the data.

The highlighting feature of `rqt_graph` is very helpful when examining more complex systems with many nodes and topics connected in many different ways.

`rqt_graph` is a graphical introspection tool. Now we'll look at some command line tools for introspecting topics.

3 `ros2 topic list`

Running the `ros2 topic list` command in a new terminal will return a list of all the topics currently active in the system:

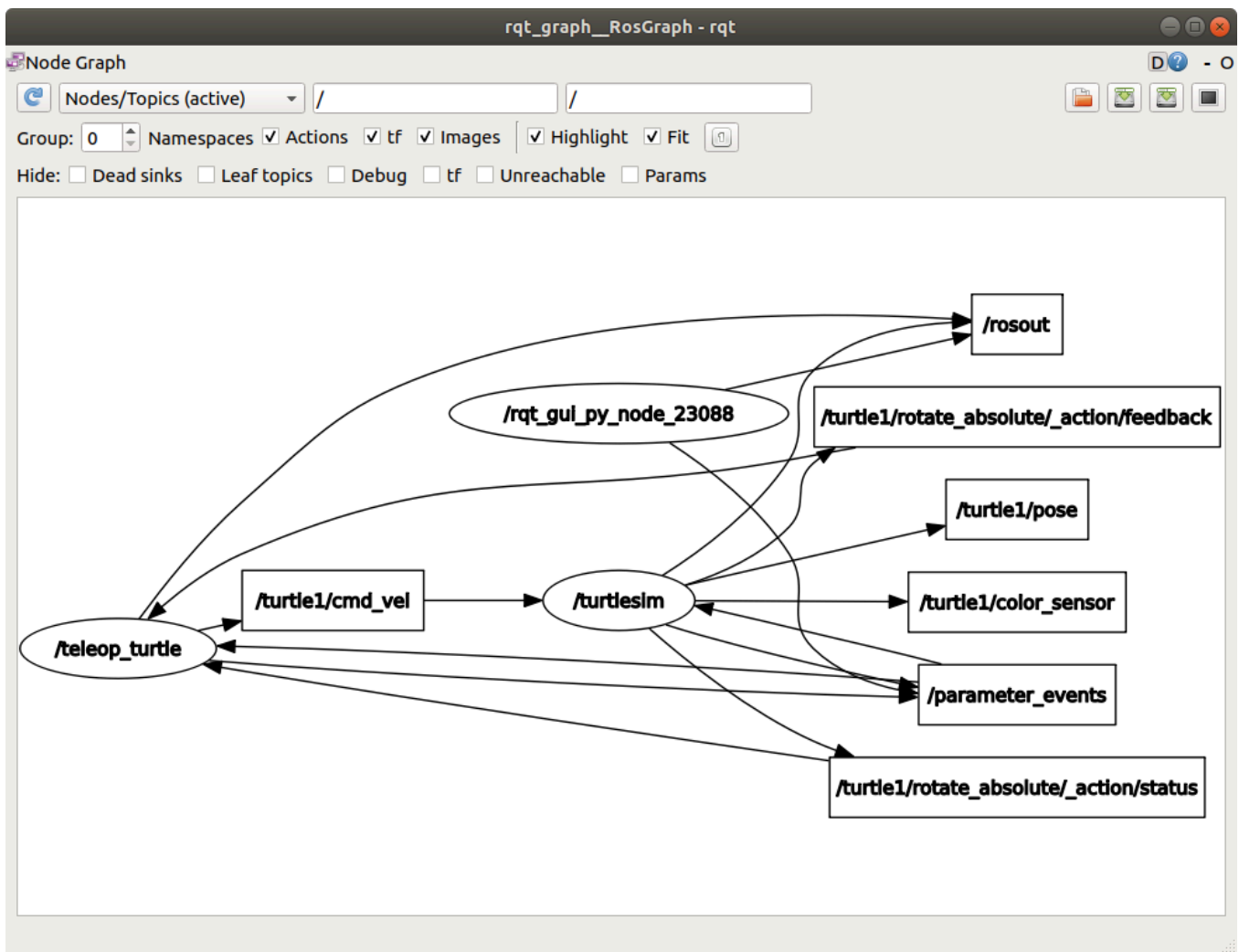
```
$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

`ros2 topic list -t` will return the same list of topics, this time with the topic type appended in brackets:

```
$ ros2 topic list -t
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

These attributes, particularly the type, are how nodes know they're talking about the same information as it moves over topics.

If you're wondering where all these topics are in rqt_graph, you can uncheck all the boxes under **Hide**:



For now, though, leave those options checked to avoid confusion.

4 ros2 topic echo

To see the data being published on a topic, use:

```
$ ros2 topic echo <topic_name>
```

Since we know that `/teleop_turtle` publishes data to `/turtlesim` over the `/turtle1/cmd_vel` topic, let's use `echo` to introspect that topic:

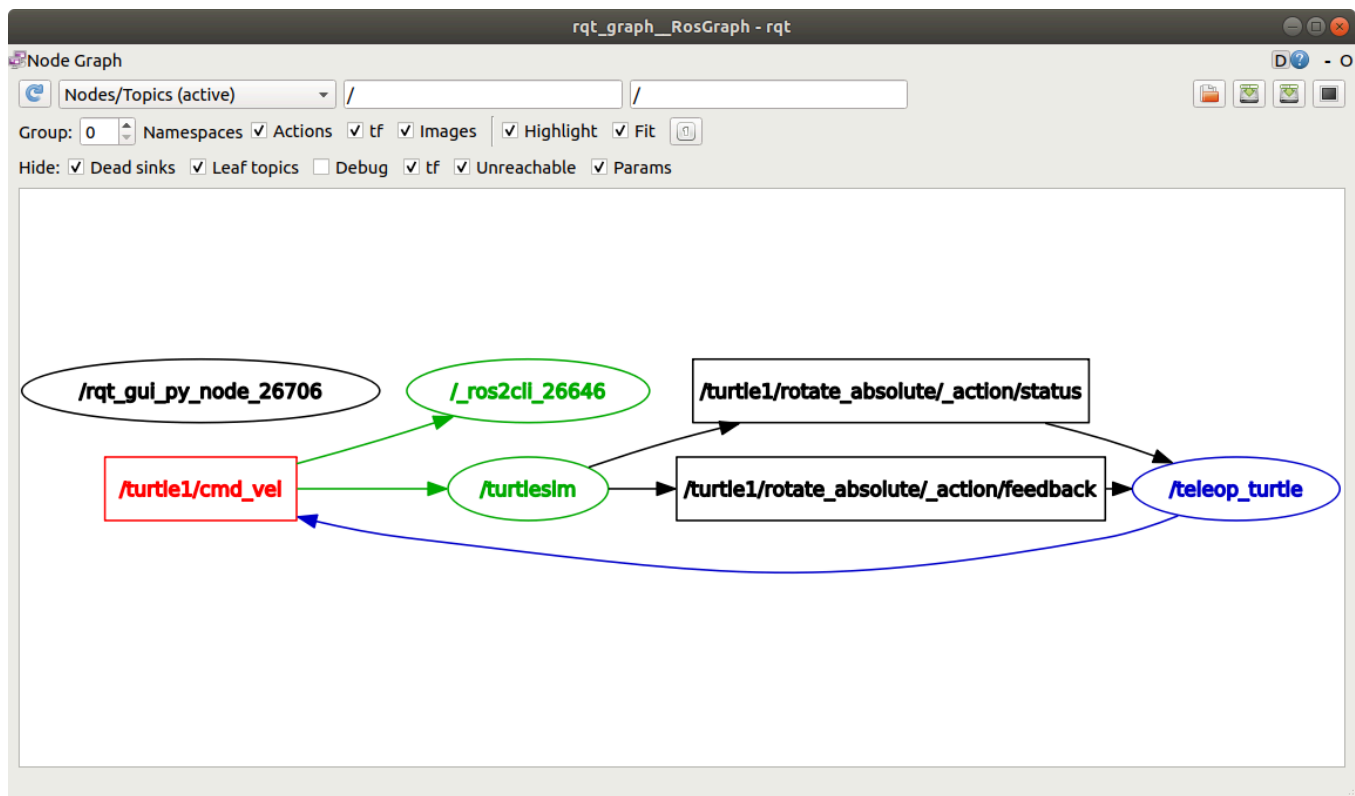
```
$ ros2 topic echo /turtle1/cmd_vel
```

At first, this command won't return any data. That's because it's waiting for `/teleop_turtle` to publish something.

Return to the terminal where `turtle_teleop_key` is running and use the arrows to move the turtle around. Watch the terminal where your `echo` is running at the same time, and you'll see position data being published for every movement you make:

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Now return to `rqt_graph` and uncheck the **Debug** box.



`/_ros2cli_26646` is the node created by the `echo` command we just ran (the number might be different). Now you can see that the publisher is publishing data over the `cmd_vel` topic, and two subscribers are subscribed to it.

5 ros2 topic info

Topics don't have to only be one-to-one communication; they can be one-to-many, many-to-one, or many-to-many.

Another way to look at this is running:

```
$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

6 ros2 interface show

Nodes send data over topics using messages. Publishers and subscribers must send and receive the same type of message to communicate.

The topic types we saw earlier after running `ros2 topic list -t` let us know what message type is used on each topic. Recall that the `cmd_vel` topic has the type:

```
geometry_msgs/msg/Twist
```

This means that in the package `geometry_msgs` there is a `msg` called `Twist`.

Now we can run `ros2 interface show <msg_type>` on this type to learn its details. Specifically, what structure of data the message expects.

```
$ ros2 interface show geometry_msgs/msg/Twist
```

Which will return:

```
# This expresses velocity in free space broken into its linear and angular parts.
  Vector3  linear
    float64 x
    float64 y
    float64 z
  Vector3  angular
    float64 x
    float64 y
    float64 z
```

This tells you that the `/turtlesim` node is expecting a message with two vectors, `linear` and `angular`, of three elements each. If you recall the data we saw `/teleop_turtle` passing to `/turtlesim` with the `echo` command, it's in the same structure:

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

7 ros2 topic pub

Now that you have the message structure, you can publish data to a topic directly from the command line using:

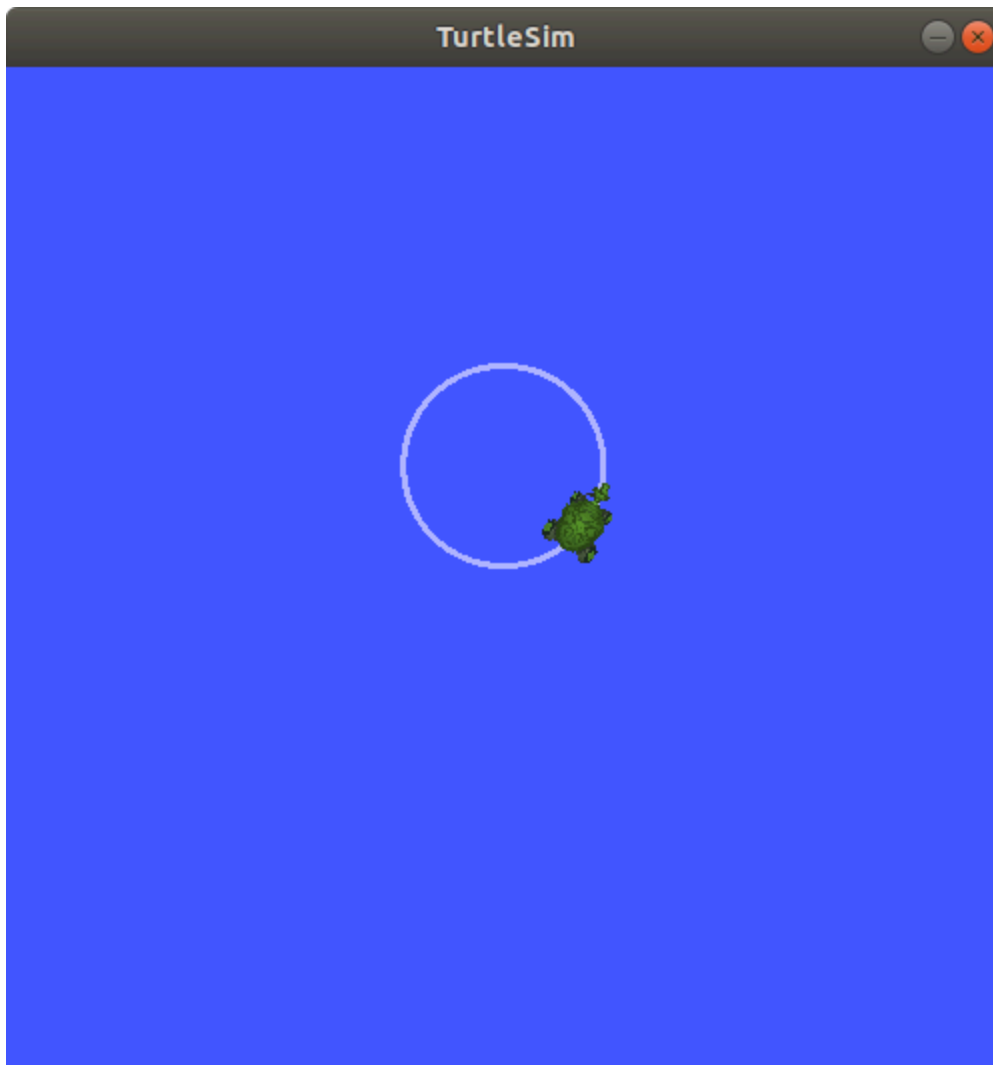
```
$ ros2 topic pub <topic_name> <msg_type> '<args>'
```

The `'<args>'` argument is the actual data you'll pass to the topic, in the structure you just discovered in the previous section.

The turtle (and commonly the real robots which it is meant to emulate) require a steady stream of commands to operate continuously. So, to get the turtle moving, and keep it moving, you can use the following command. It's important to note that this argument needs to be input in YAML syntax. Input the full command like so:

```
$ ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

With no command-line options, `ros2 topic pub` publishes the command in a steady stream at 1 Hz.



At times you may want to publish data to your topic only once (rather than continuously). To publish your command just once add the `--once` option.

```
$ ros2 topic pub --once -w 2 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

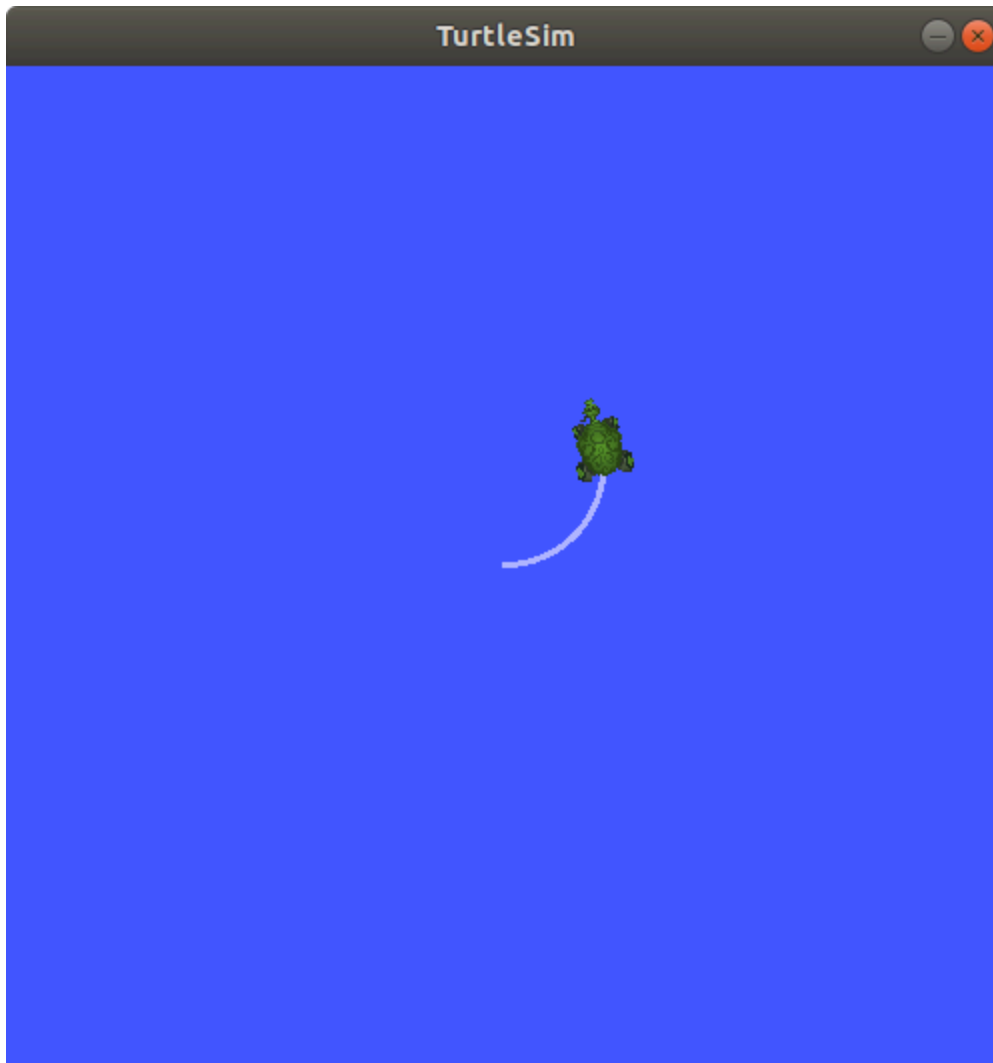
`--once` is an optional argument meaning “publish one message then exit”.

`-w 2` is an optional argument meaning “wait for two matching subscriptions”. This is needed because we have both turtlesim and the topic echo subscribed.

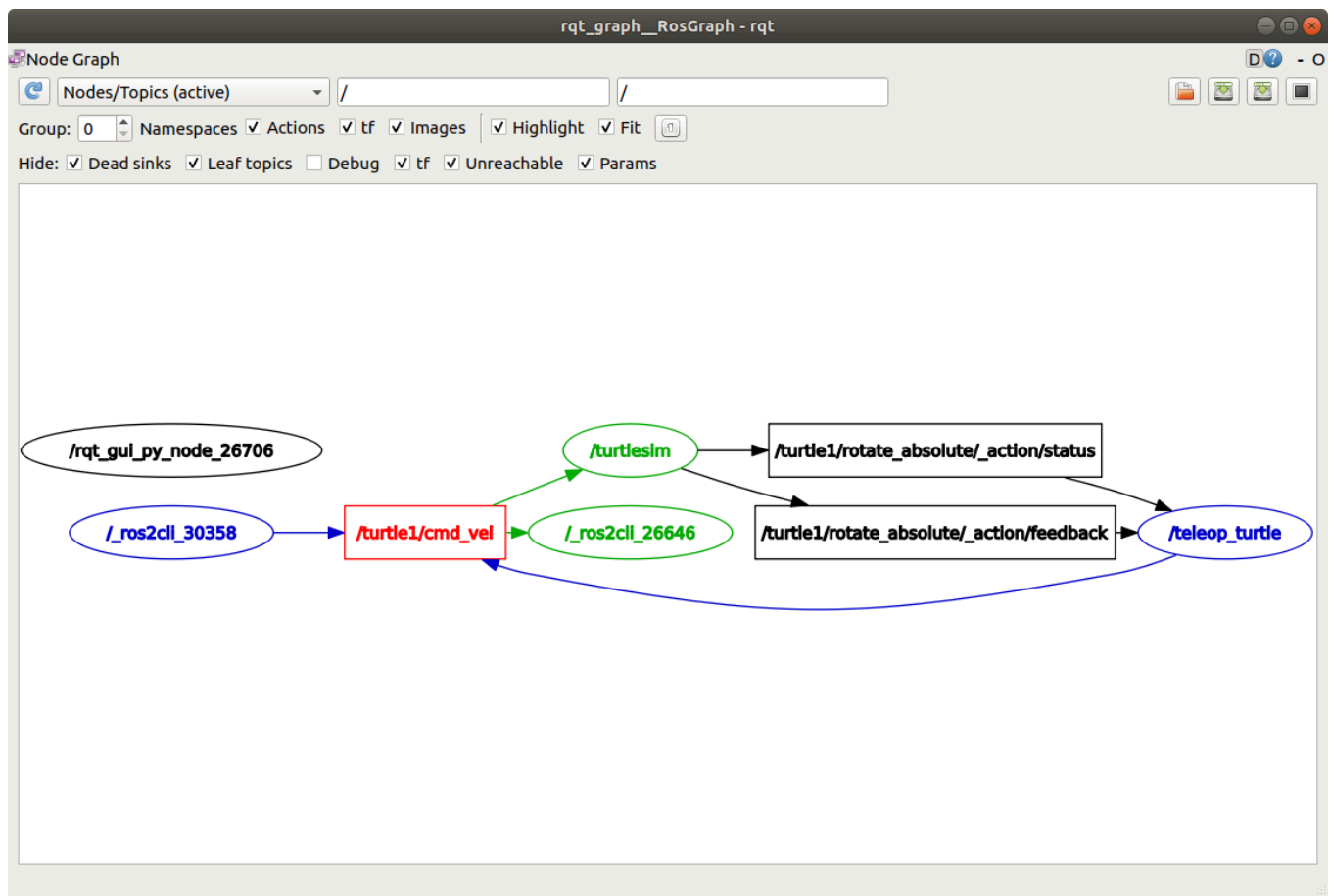
You will see the following output in the terminal:

```
Waiting for at least 2 matching subscription(s)...
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0.0, z=0.0),
angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.8))
```

And you will see your turtle move like so:

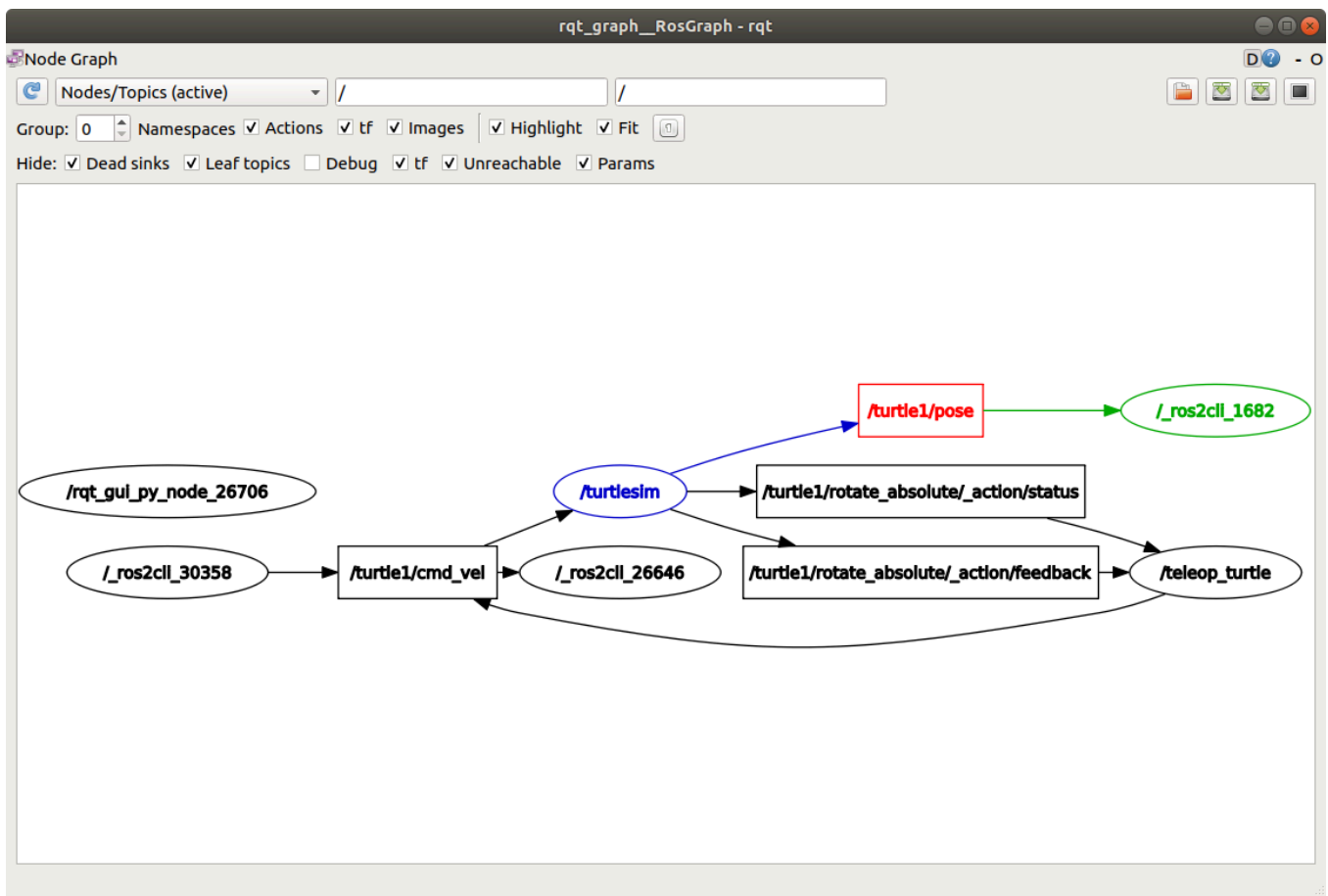


You can refresh `rqt_graph` to see what's happening graphically. You will see that the `ros2 topic pub ...` node (`/_ros2cli_30358`) is publishing over the `/turtle1/cmd_vel` topic, which is being received by both the `ros2 topic echo ...` node (`/_ros2cli_26646`) and the `/turtlesim` node now.



Finally, you can run `echo` on the `pose` topic and recheck `rqt_graph`:

```
$ ros2 topic echo /turtle1/pose
```



You can see that the `/turtlesim` node is also publishing to the `pose` topic, which the new `echo` node has subscribed to.

When publishing messages with timestamps, `pub` has two methods to automatically fill them out with the current time. For messages with a `std_msgs/msg/Header`, the header field can be set to `auto` to fill out the `stamp` field.

```
$ ros2 topic pub /pose geometry_msgs/msg/PoseStamped '{header: "auto", pose: {position: {x: 1.0, y: 2.0, z: 3.0}}}'
```

If the message does not use a full header, but just has a field with type `builtin_interfaces/msg/Time`, that can be set to the value `now`.

```
$ ros2 topic pub /reference sensor_msgs/msg/TimeReference '{header: "auto", time_ref: "now", source: "dummy"}'
```

8 ros2 topic hz

You can also view the rate at which data is published using:

```
$ ros2 topic hz /turtle1/pose
average rate: 59.354
min: 0.005s max: 0.027s std dev: 0.00284s window: 58
```

It will return data on the rate at which the `/turtlesim` node is publishing data to the `pose` topic.

Recall that you set the rate of `turtle1/cmd_vel` to publish at a steady 1 Hz using `ros2 topic pub --rate 1`. If you run the above command with `turtle1/cmd_vel` instead of `turtle1/pose`, you will see an average reflecting that rate.

! Note

The rate reflects the receiving rate on the subscription created by the `ros2 topic hz` command, which might be affected by platform resources and QoS configuration, and may not exactly match the publisher rate.

9 ros2 topic bw

The bandwidth used by a topic can be viewed using:

```
$ ros2 topic bw /turtle1/pose
Subscribed to [/turtle1/pose]
1.51 KB/s from 62 messages
  Message size mean: 0.02 KB min: 0.02 KB max: 0.02 KB
```

It returns the bandwidth utilization and number of messages being published to the `/turtle1/pose` topic.

! Note

The bandwidth reflects the receiving rate on the subscription created by the `ros2 topic bw` command, which might be affected by platform resources and QoS configuration, and may not exactly match the publisher's bandwidth.

10 ros2 topic find

To list a list of available topics of a given type use:

```
$ ros2 topic find <topic_type>
```

Recall that the `cmd_vel` topic has the type:

```
geometry_msgs/msg/Twist
```

Using the `find` command outputs topics available when given the message type:

```
$ ros2 topic find geometry_msgs/msg/Twist  
/turtle1/cmd_vel
```

11 Clean up

At this point you'll have a lot of nodes running. Don't forget to stop them by entering `Ctrl+C` in each terminal.

Summary

Nodes publish information over topics, which allows any number of other nodes to subscribe to and access that information. In this tutorial you examined the connections between several nodes over topics using `rqt_graph` and command line tools. You should now have a good idea of how data moves around a ROS 2 system.

Next steps

Next you'll learn about another communication type in the ROS graph with the tutorial [Understanding services](#).

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Understanding services

Goal: Learn about services in ROS 2 using command line tools.

Tutorial level: Beginner

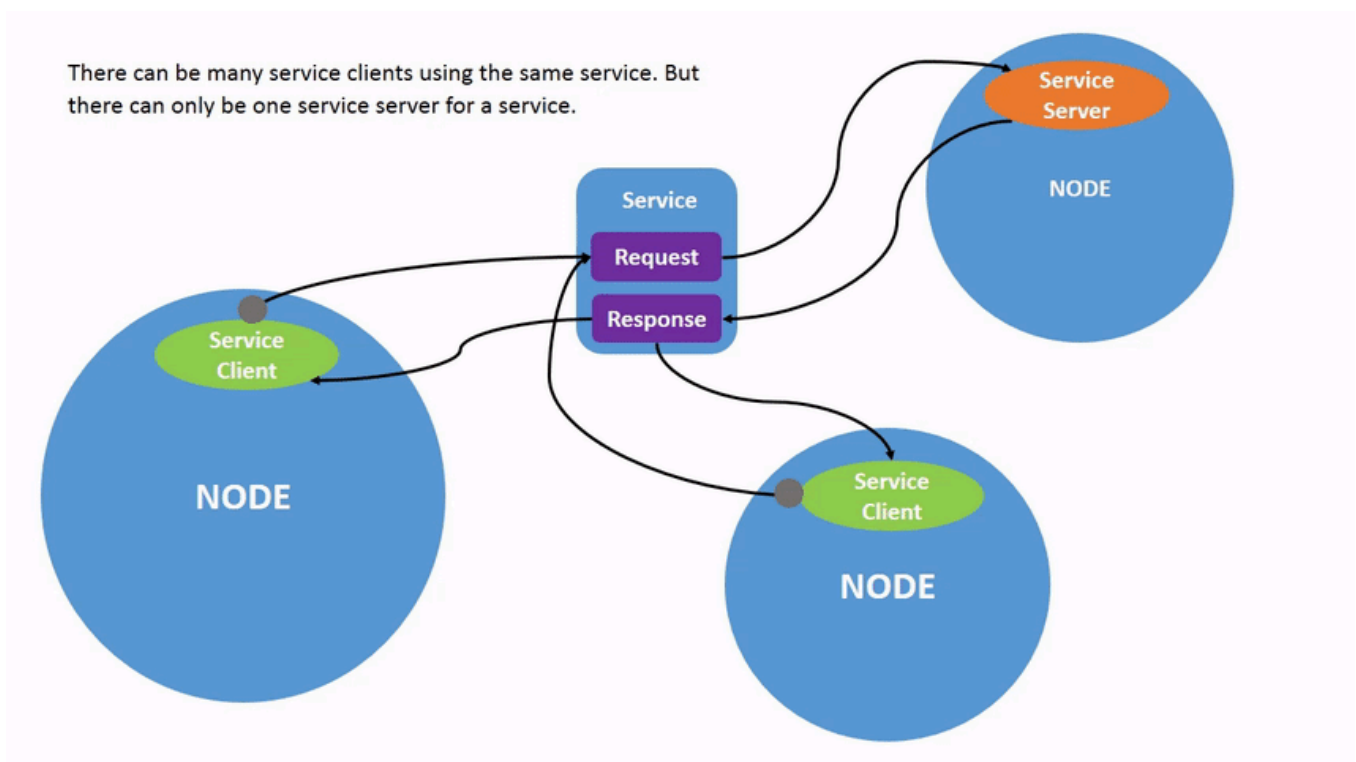
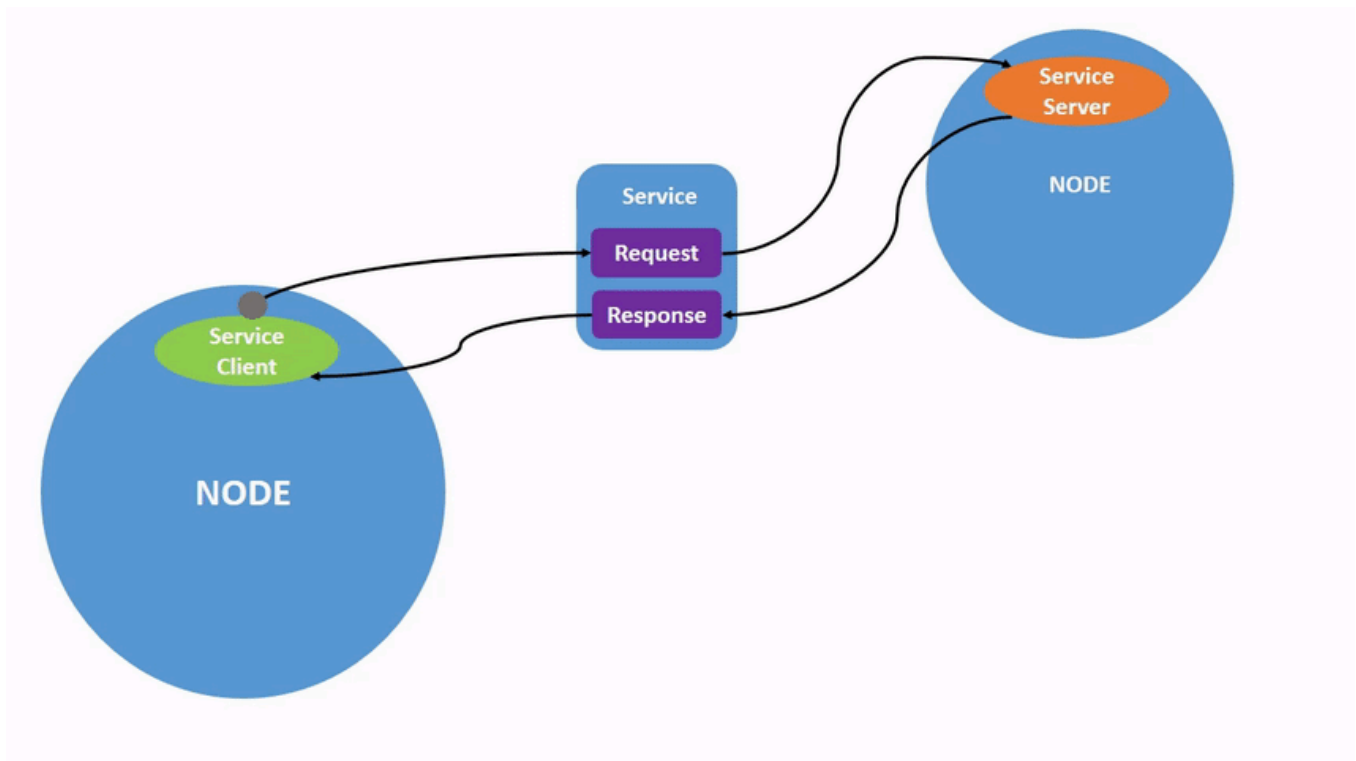
Time: 10 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Setup](#)
 - [2 ros2 service list](#)
 - [3 ros2 service type](#)
 - [4 ros2 service find](#)
 - [5 ros2 interface show](#)
 - [6 ros2 service call](#)
- [Summary](#)
- [Next steps](#)
- [Related content](#)

Background

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model versus the publisher-subscriber model of topics. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.



Prerequisites

Some concepts mentioned in this tutorial, like **Nodes** and **Topics**, were covered in previous tutorials in the series.

You will need the **turtlesim** package.

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start up the two turtlesim nodes, `/turtlesim` and `/teleop_turtle`.

Open a new terminal and run:

```
$ ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```
$ ros2 run turtlesim turtle_teleop_key
```

2 ros2 service list

Running the `ros2 service list` command in a new terminal will return a list of all the services currently active in the system:

```
$ ros2 service list
/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
```

You will see that both nodes have the same six services with `parameters` in their names. Nearly every node in ROS 2 has these infrastructure services that parameters are built off of. There will be more about parameters in the next tutorial. In this tutorial, the parameter services will be omitted from the discussion.

For now, let's focus on the turtlesim-specific services, `/clear`, `/kill`, `/reset`, `/spawn`, `/turtle1/set_pen`, `/turtle1/teleport_absolute`, and `/turtle1/teleport_relative`. You may recall interacting with some of these services using rqt in the [Use turtlesim, ros2, and rqt](#) tutorial.

3 ros2 service type

Services have types that describe how the request and response data of a service is structured. Service types are defined similarly to topic types, except service types have two parts: one message for the request and another for the response.

To find out the type of a service, use the command:

```
$ ros2 service type <service_name>
```

Let's take a look at turtlesim's `/clear` service. In a new terminal, enter the command:

```
$ ros2 service type /clear
std_srvs/srv/Empty
```

The `Empty` type means the service call sends no data when making a request and receives no data when receiving a response.

3.1 ros2 service list -t

To see the types of all the active services at the same time, you can append the `--show-types` option, abbreviated as `-t`, to the `list` command:

```
$ ros2 service list -t
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
...
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
...
```


4 ros2 service find

If you want to find all the services of a specific type, you can use the command:

```
$ ros2 service find <type_name>
```

For example, you can find all the `Empty` typed services like this:

```
$ ros2 service find std_srvs/srv/Empty  
/clear  
/reset
```

5 ros2 interface show

You can call services from the command line, but first you need to know the structure of the input arguments.

```
$ ros2 interface show <type_name>
```

Try this on the `/clear` service's type, `Empty`:

```
$ ros2 interface show std_srvs/srv/Empty  
---
```

The `---` separates the request structure (above) from the response structure (below). But, as you learned earlier, the `Empty` type doesn't send or receive any data. So, naturally, its structure is blank.

Let's introspect a service with a type that sends and receives data, like `/spawn`. From the results of `ros2 service list -t`, we know `/spawn`'s type is `turtlesim/srv/Spawn`.

To see the request and response arguments of the `/spawn` service, run the command:

```
$ ros2 interface show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional. A unique name will be created and returned if this is empty
---
string name
```

The information above the `---` line tells us the arguments needed to call `/spawn`. `x`, `y` and `theta` determine the 2D pose of the spawned turtle, and `name` is clearly optional.

The information below the line isn't something you need to know in this case, but it can help you understand the data type of the response you get from the call.

6 ros2 service call

Now that you know what a service type is, how to find a service's type, and how to find the structure of that type's arguments, you can call a service using:

```
$ ros2 service call <service_name> <service_type> <arguments>
```

The `<arguments>` part is optional. For example, you know that `Empty` typed services don't have any arguments:

```
$ ros2 service call /clear std_srvs/srv/Empty
```

This command will clear the turtlesim window of any lines your turtle has drawn.



Now let's spawn a new turtle by calling `/spawn` and setting arguments. Input `<arguments>` in a service call from the command-line need to be in YAML syntax.

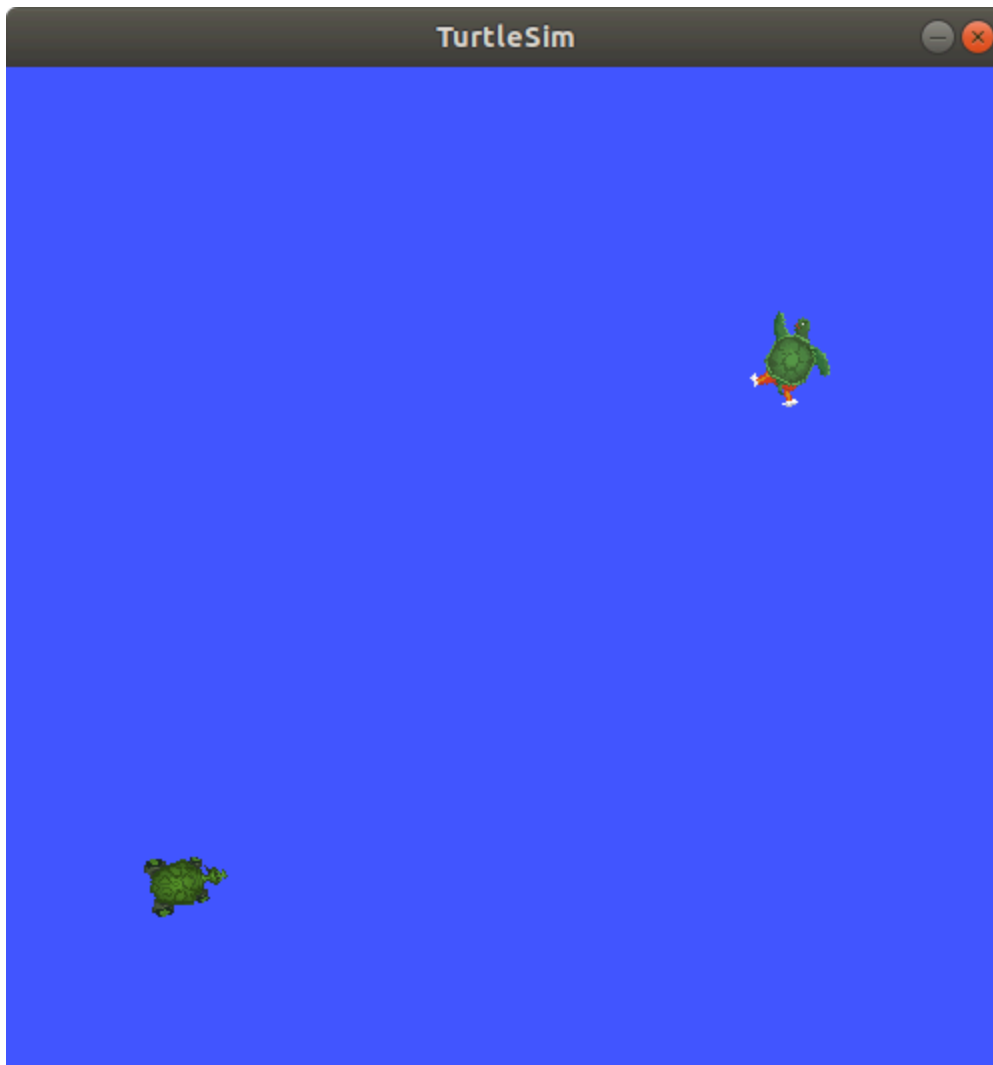
Enter the command:

```
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.2, name='')

response:
turtlesim.srv.Spawn_Response(name='turtle2')
```

You will get this method-style view of what's happening, and then the service response.

Your turtlesim window will update with the newly spawned turtle right away:



Summary

Nodes can communicate using services in ROS 2. Unlike a topic - a one way communication pattern where a node publishes information that can be consumed by one or more subscribers - a service is a request/response pattern where a client makes a request to a node providing the service and the service processes the request and generates a response.

You generally don't want to use a service for continuous calls; topics or even actions would be better suited.

In this tutorial you used command line tools to identify, introspect, and call services.

Next steps

In the next tutorial, [Understanding parameters](#), you will learn about configuring node settings.

Related content

Check out [this tutorial](#); it's an excellent realistic application of ROS services using a Robotis robot arm.

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Understanding parameters

Goal: Learn how to get, set, save and reload parameters in ROS 2.

Tutorial level: Beginner

Time: 5 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Setup](#)
 - [2 ros2 param list](#)
 - [3 ros2 param get](#)
 - [4 ros2 param set](#)
 - [5 ros2 param dump](#)
 - [6 ros2 param load](#)
 - [7 Load parameter file on node startup](#)
- [Summary](#)
- [Next steps](#)

Background

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists. In ROS 2, each node maintains its own parameters. For more background on parameters, please see [the concept document](#).

Prerequisites

This tutorial uses the [turtlesim](#) package.

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start up the two turtlesim nodes, `/turtlesim` and `/teleop_turtle`.

Open a new terminal and run:

```
$ ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```
$ ros2 run turtlesim turtle_teleop_key
```

2 ros2 param list

To see the parameters belonging to your nodes, open a new terminal and enter the command:

```
$ ros2 param list
/teleop_turtle:
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  use_sim_time
```

Every node has the parameter `use_sim_time`; it's not unique to turtlesim.

Based on their names, it looks like `/turtlesim`'s parameters determine the background color of the turtlesim window using RGB color values.

To determine a parameter's type, you can use `ros2 param get`.

3 ros2 param get

To display the type and current value of a parameter, use the command:

```
$ ros2 param get <node_name> <parameter_name>
```

Let's find out the current value of `/turtlesim`'s parameter `background_g`:

```
$ ros2 param get /turtlesim background_g  
Integer value is: 86
```

Now you know `background_g` holds an integer value.

If you run the same command on `background_r` and `background_b`, you will get the values `69` and `255`, respectively.

4 ros2 param set

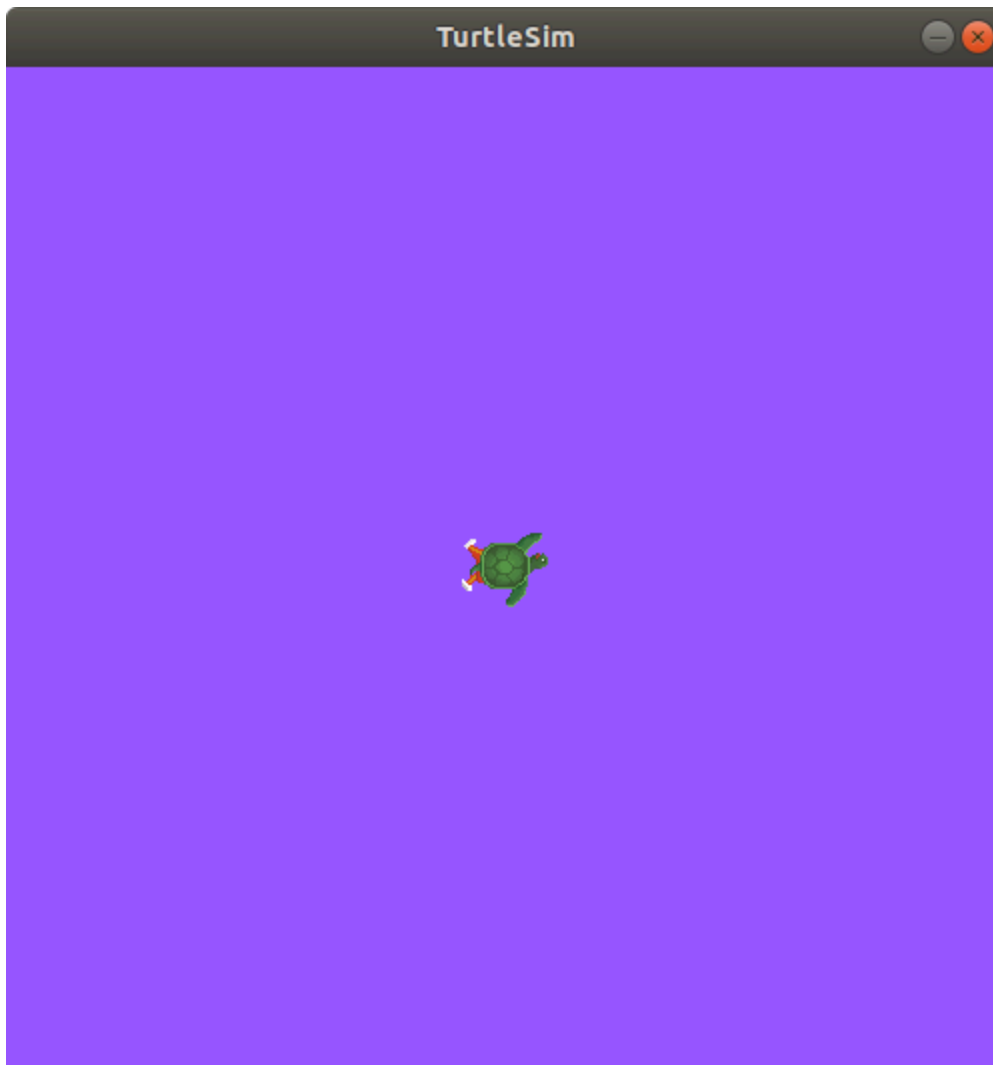
To change a parameter's value at runtime, use the command:

```
$ ros2 param set <node_name> <parameter_name> <value>
```

Let's change `/turtlesim`'s background color:

```
$ ros2 param set /turtlesim background_r 150  
Set parameter successful
```

The background of your turtlesim window should change colors:



Setting parameters with the `set` command will only change them in your current session, not permanently. However, you can save your settings and reload them the next time you start a node.

5 ros2 param dump

You can view all of a node's current parameter values by using the command:

```
$ ros2 param dump <node_name>
```

The command prints to the standard output (stdout) by default but you can also redirect the parameter values into a file to save them for later. To save your current configuration of `/turtlesim`'s parameters into the file `turtlesim.yaml`, enter the command:

```
$ ros2 param dump /turtlesim > turtlesim.yaml
```

You will find a new file in the current working directory your shell is running in. If you open this file, you'll see the following content:

```
/turtlesim:
ros__parameters:
  background_b: 255
  background_g: 86
  background_r: 150
  qos_overrides:
    /parameter_events:
      publisher:
        depth: 1000
        durability: volatile
        history: keep_last
        reliability: reliable
  use_sim_time: false
```

Dumping parameters comes in handy if you want to reload the node with the same parameters in the future.

6 ros2 param load

You can load parameters from a file to a currently running node using the command:

```
$ ros2 param load <node_name> <parameter_file>
```

To load the `turtlesim.yaml` file generated with `ros2 param dump` into `/turtlesim` node's parameters, enter the command:

```
$ ros2 param load /turtlesim turtlesim.yaml
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter qos_overrides./parameter_events.publisher.depth failed: parameter
'qos_overrides./parameter_events.publisher.depth' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.durability failed: parameter
'qos_overrides./parameter_events.publisher.durability' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.history failed: parameter
'qos_overrides./parameter_events.publisher.history' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.reliability failed: parameter
'qos_overrides./parameter_events.publisher.reliability' cannot be set because it is read-only
Set parameter use_sim_time successful
```

Read-only parameters can only be modified at startup and not afterwards, that is why there are some warnings for the “qos_overrides” parameters.

7 Load parameter file on node startup

To start the same node using your saved parameter values, use:

```
$ ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

This is the same command you always use to start turtlesim, with the added flags `--ros-args` and `--params-file`, followed by the file you want to load.

Stop your running turtlesim node, and try reloading it with your saved parameters, using:

```
$ ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

The turtlesim window should appear as usual, but with the purple background you set earlier.

! Note

When a parameter file is used at node startup, all parameters, including the read-only ones, will be updated.

Summary

Nodes have parameters to define their default configuration values. You can `get` and `set` parameter values from the command line. You can also save the parameter settings to a file to reload them in a future session.

Next steps

Jumping back to ROS 2 communication methods, in the next tutorial you'll learn about [actions](#).

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Understanding actions

Goal: Introspect actions in ROS 2.

Tutorial level: Beginner

Time: 15 minutes

Contents

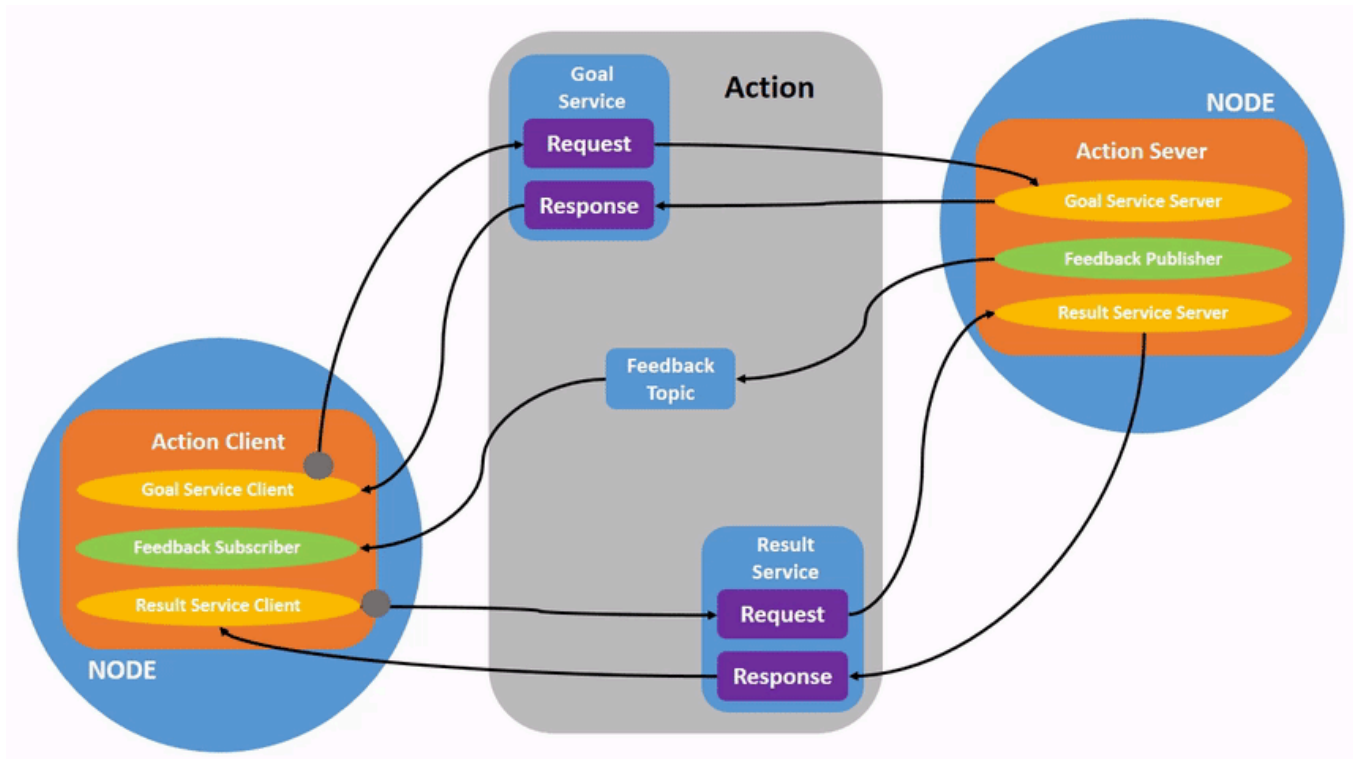
- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Setup](#)
 - [2 Use actions](#)
 - [3 ros2 node info](#)
 - [4 ros2 action list](#)
 - [5 ros2 action info](#)
 - [6 ros2 interface show](#)
 - [7 ros2 action send_goal](#)
- [Summary](#)
- [Next steps](#)
- [Related content](#)

Background

Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the [topics tutorial](#)). An “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result.



Prerequisites

This tutorial builds off concepts, like [nodes](#) and [topics](#), covered in previous tutorials.

This tutorial uses the [turtlesim](#) package.

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start up the two turtlesim nodes, `/turtlesim` and `/teleop_turtle`.

Open a new terminal and run:

```
$ ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```
$ ros2 run turtlesim turtle_teleop_key
```

2 Use actions

When you launch the `/teleop_turtle` node, you will see the following message in your terminal:

```
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.
```

Let's focus on the second line, which corresponds to an action. (The first instruction corresponds to the "cmd_vel" topic, discussed previously in the [topics tutorial](#).)

Notice that the letter keys `G|B|V|C|D|E|R|T` form a "box" around the `F` key on a US QWERTY keyboard (if you are not using a QWERTY keyboard, see [this link](#) to follow along). Each key's position around `F` corresponds to that orientation in turtlesim. For example, the `E` will rotate the turtle's orientation to the upper left corner.

Pay attention to the terminal where the `/turtlesim` node is running. Each time you press one of these keys, you are sending a goal to an action server that is part of the `/turtlesim` node. The goal is to rotate the turtle to face a particular direction. A message relaying the result of the goal should display once the turtle completes its rotation:

```
[INFO] [turtlesim]: Rotation goal completed successfully
```

The `F` key will cancel a goal mid-execution.

Try pressing the `C` key, and then pressing the `F` key before the turtle can complete its rotation. In the terminal where the `/turtlesim` node is running, you will see the message:

```
[INFO] [turtlesim]: Rotation goal canceled
```

Not only can the client-side (your input in the teleop) stop a goal, but the server-side (the `/turtlesim` node) can as well. When the server-side chooses to stop processing a goal, it is said to "abort" the goal.

Try hitting the `D` key, then the `G` key before the first rotation can complete. In the terminal where the `/turtlesim` node is running, you will see the message:

```
[WARN] [turtlesim]: Rotation goal received before a previous goal finished. Aborting previous goal
```

This action server chose to abort the first goal because it got a new one. It could have chosen something else, like reject the new goal or execute the second goal after the first one finished. Don't assume every action server will choose to abort the current goal when it gets a new one.

3 ros2 node info

To see the list of actions a node provides, `/turtlesim` in this case, open a new terminal and run the command:

```
$ ros2 node info /turtlesim
/turtlesim
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/color_sensor: turtlesim/msg/Color
  /turtle1/pose: turtlesim/msg/Pose
Service Servers:
  /clear: std_srvs/srv/Empty
  /kill: turtlesim/srv/Kill
  /reset: std_srvs/srv/Empty
  /spawn: turtlesim/srv/Spawn
  /turtle1/set_pen: turtlesim/srv/SetPen
  /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
  /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
  /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
  /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
  /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
  /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
Action Clients:
```

The command returns a list of `/turtlesim`'s subscribers, publishers, services, action servers and action clients.

Notice that the `/turtle1/rotate_absolute` action for `/turtlesim` is under `Action Servers`. This means `/turtlesim` responds to and provides feedback for the `/turtle1/rotate_absolute` action.

The `/teleop_turtle` node has the name `/turtle1/rotate_absolute` under `Action Clients` meaning that it sends goals for that action name. To see that, run:

```
$ ros2 node info /teleop_turtle
/teleop_turtle
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
Service Servers:
  /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
  /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
  /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
  /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:

Action Clients:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

4 ros2 action list

To identify all the actions in the ROS graph, run the command:

```
$ ros2 action list
/turtle1/rotate_absolute
```

This is the only action in the ROS graph right now. It controls the turtle's rotation, as you saw earlier. You also already know that there is one action client (part of `/teleop_turtle`) and one action server (part of `/turtlesim`) for this action from using the `ros2 node info <node_name>` command.

4.1 ros2 action list -t

Actions have types, similar to topics and services. To find `/turtle1/rotate_absolute`'s type, run the command:


```
$ ros2 action list -t
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

In brackets to the right of each action name (in this case only `/turtle1/rotate_absolute`) is the action type, `turtlesim/action/RotateAbsolute`. You will need this when you want to execute an action from the command line or from code.

5 ros2 action info

You can further introspect the `/turtle1/rotate_absolute` action with the command:

```
$ ros2 action info /turtle1/rotate_absolute
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

This tells us what we learned earlier from running `ros2 node info` on each node: The `/teleop_turtle` node has an action client and the `/turtlesim` node has an action server for the `/turtle1/rotate_absolute` action.

6 ros2 interface show

One more piece of information you will need before sending or executing an action goal yourself is the structure of the action type.

Recall that you identified `/turtle1/rotate_absolute`'s type when running the command `ros2 action list -t`. Enter the following command with the action type in your terminal:

```
$ ros2 interface show turtlesim/action/RotateAbsolute
```

Which will return:

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

The section of this message above the first `---` is the structure (data type and name) of the goal request. The next section is the structure of the result. The last section is the structure of the feedback.

7 ros2 action send_goal

Now let's send an action goal from the command line with the following syntax:

```
$ ros2 action send_goal <action_name> <action_type> <values>
```

`<values>` need to be in YAML format.

Keep an eye on the turtlesim window, and enter the following command into your terminal:

```
$ ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"
Waiting for an action server to become available...
Sending goal:
  theta: 1.57

Goal accepted with ID: f8db8f44410849eaa93d3feb747dd444

Result:
  delta: -1.568000316619873

Goal finished with status: SUCCEEDED
```

You should see the turtle rotating.

All goals have a unique ID, shown in the return message. You can also see the result, a field with the name `delta`, which is the displacement to the starting position.

To see the feedback of this goal, add `--feedback` to the `ros2 action send_goal` command:

```
$ ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1.57}"
--feedback
Sending goal:
  theta: -1.57

Goal accepted with ID: e6092c831f994afda92f0086f220da27

Feedback:
  remaining: -3.1268222332000732

Feedback:
  remaining: -3.1108222007751465

...

Result:
  delta: 3.1200008392333984

Goal finished with status: SUCCEEDED
```

You will continue to receive feedback, the remaining radians, until the goal is complete.

Summary

Actions are like services that allow you to execute long running tasks, provide regular feedback, and are cancelable.

A robot system would likely use actions for navigation. An action goal could tell a robot to travel to a position. While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it's reached its destination.

Turtlesim has an action server that action clients can send goals to for rotating turtles. In this tutorial, you introspected that action, `/turtle1/rotate_absolute`, to get a better idea of what actions are and how they work.

Next steps

Now you've covered all of the core ROS 2 concepts. The last few tutorials in this set will introduce you to some tools and techniques that will make using ROS 2 easier, starting with [Using rqt_console to view logs](#).

Related content

You can read more about the design decisions behind actions in ROS 2 [here](#).

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Using `rqt_console` to view logs

Goal: Get to know `rqt_console`, a tool for introspecting log messages.

Tutorial level: Beginner

Time: 5 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Setup](#)
 - [2 Messages on rqt_console](#)
 - [3 Logger levels](#)
- [Summary](#)
- [Next steps](#)

Background

`rqt_console` is a GUI tool used to introspect log messages in ROS 2. Typically, log messages show up in your terminal. With `rqt_console`, you can collect those messages over time, view them closely and in a more organized manner, filter them, save them and even reload the saved files to introspect at a different time.

Nodes use logs to output messages concerning events and status in a variety of ways. Their content is usually informational, for the sake of the user.

Prerequisites

You will need `rqt_console` and `turtlesim` installed.

As always, don't forget to source ROS 2 in every new terminal you open.

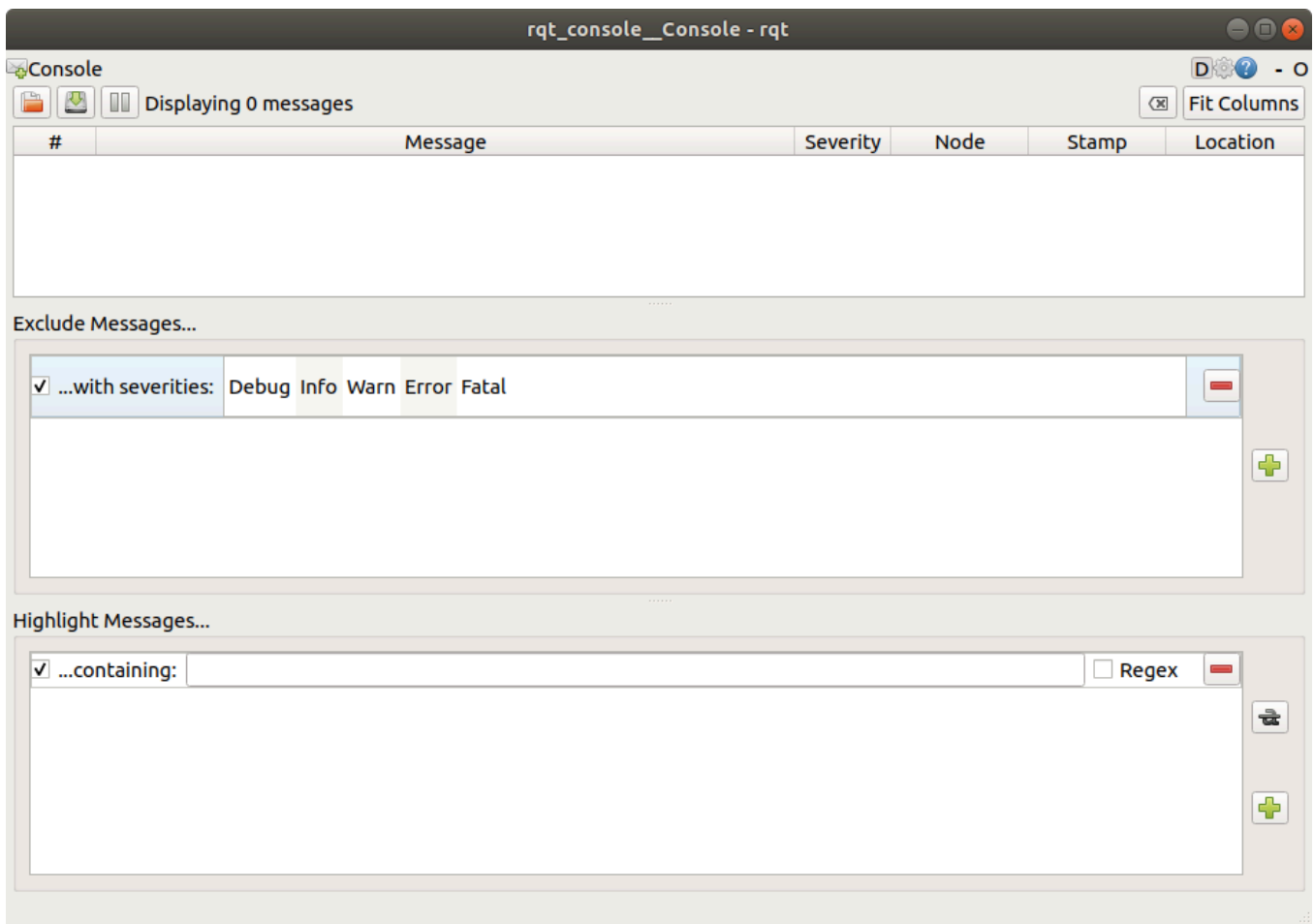
Tasks

1 Setup

Start `rqt_console` in a new terminal with the following command:

```
$ ros2 run rqt_console rqt_console
```

The `rqt_console` window will open:



The first section of the console is where log messages from your system will display.

In the middle you have the option to filter messages by excluding severity levels. You can also add more exclusion filters using the plus-sign button to the right.

The bottom section is for highlighting messages that include a string you input. You can add more filters to this section as well.

Now start `turtlesim` in a new terminal with the following command:

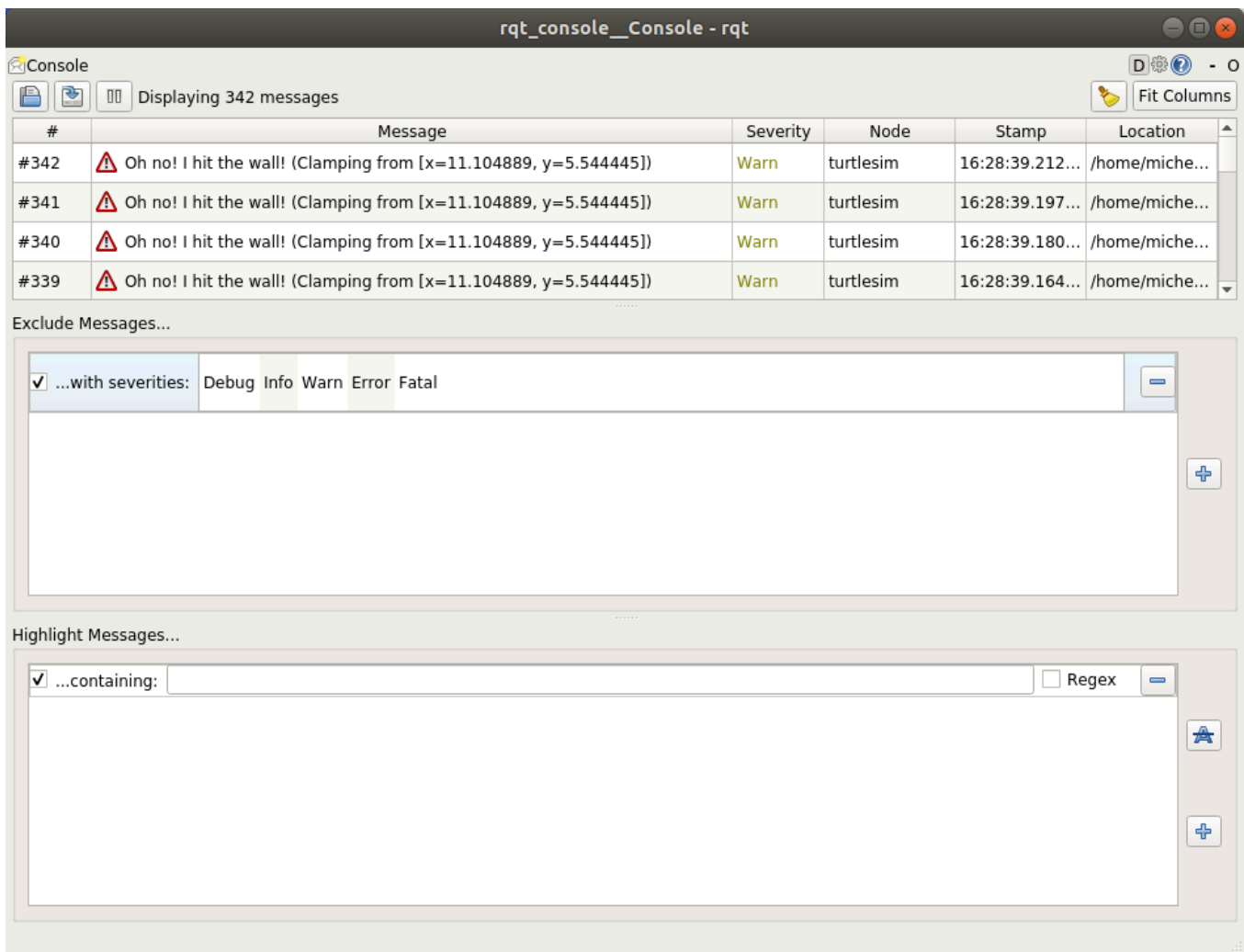
```
$ ros2 run turtlesim turtlesim_node
```

2 Messages on rqt_console

To produce log messages for `rqt_console` to display, let's have the turtle run into the wall. In a new terminal, enter the `ros2 topic pub` command (discussed in detail in the [topics tutorial](#)) below:

```
$ ros2 topic pub -r 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0},  
angular: {x: 0.0,y: 0.0,z: 0.0}}"
```

Since the above command is publishing the topic at a steady rate, the turtle is continuously running into the wall. In `rqt_console` you will see the same message with the `Warn` severity level displayed over and over, like so:



The screenshot shows the `rqt_console` window titled `rqt_console__Console - rqt`. The window displays a table of messages with the following columns: #, Message, Severity, Node, Stamp, and Location. The messages are warnings from the `turtlesim` node, indicating that the turtle has hit the wall and is clamping its velocity. The messages are numbered #339, #340, #341, and #342. The severity level is `Warn` for all messages. The location is `/home/miche...` for all messages.

#	Message	Severity	Node	Stamp	Location
#342	Oh no! I hit the wall! (Clamping from [x=11.104889, y=5.544445])	Warn	turtlesim	16:28:39.212...	/home/miche...
#341	Oh no! I hit the wall! (Clamping from [x=11.104889, y=5.544445])	Warn	turtlesim	16:28:39.197...	/home/miche...
#340	Oh no! I hit the wall! (Clamping from [x=11.104889, y=5.544445])	Warn	turtlesim	16:28:39.180...	/home/miche...
#339	Oh no! I hit the wall! (Clamping from [x=11.104889, y=5.544445])	Warn	turtlesim	16:28:39.164...	/home/miche...

Below the table, there are two sections: `Exclude Messages...` and `Highlight Messages...`. The `Exclude Messages...` section has a checkbox `...with severities:` which is checked, and a list of severity levels: `Debug`, `Info`, `Warn`, `Error`, and `Fatal`. The `Highlight Messages...` section has a checkbox `...containing:` which is checked, and a text input field for a search pattern. There is also a `Regex` checkbox.

Press `Ctrl+C` in the terminal where you ran the `ros2 topic pub` command to stop your turtle from running into the wall.

3 Logger levels

ROS 2's logger levels are ordered by severity:

1. Fatal
2. Error
3. Warn
4. Info
5. Debug

There is no exact standard for what each level indicates, but it's safe to assume that:

- **Fatal** messages indicate the system is going to terminate to try to protect itself from detriment.
- **Error** messages indicate significant issues that won't necessarily damage the system, but are preventing it from functioning properly.
- **Warn** messages indicate unexpected activity or non-ideal results that might represent a deeper issue, but don't harm functionality outright.
- **Info** messages indicate event and status updates that serve as a visual verification that the system is running as expected.
- **Debug** messages detail the entire step-by-step process of the system execution.

The default level is **Info**. You will only see messages of the default severity level and more-severe levels.

Normally, only **Debug** messages are hidden because they're the only level less severe than **Info**. For example, if you set the default level to **Warn**, you would only see messages of severity **Warn**, **Error**, and **Fatal**.

3.1 Set the default logger level

You can set the default logger level when you first run the `/turtlesim` node using remapping. Enter the following command in your terminal:

```
$ ros2 run turtlesim turtlesim_node --ros-args --log-level WARN
```

Now you won't see the initial **Info** level messages that came up in the console last time you started `turtlesim`. That's because **Info** messages are lower priority than the new default severity, **Warn**.

Summary

`rqt_console` can be very helpful if you need to closely examine the log messages from your system. You might want to examine log messages for any number of reasons, usually to find out where something went wrong and the series of events leading up to that.

Next steps

The next tutorial will teach you about starting multiple nodes at once with [ROS 2 Launch](#).

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Launching nodes

Goal: Use a command line tool to launch multiple nodes at once.

Tutorial Level: Beginner

Time: 5 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [Running a Launch File](#)
 - [\(Optional\) Control the Turtlesim Nodes](#)
- [Summary](#)
- [Next steps](#)

Background

In most of the introductory tutorials, you have been opening new terminals for every new node you run. As you create more complex systems with more and more nodes running simultaneously, opening terminals and reentering configuration details becomes tedious.

Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.

Running a single launch file with the `ros2 launch` command will start up your entire system - all nodes and their configurations - at once.

Prerequisites

Before starting these tutorials, install ROS 2 by following the instructions on the [ROS 2 Installation](#) page.

The commands used in this tutorial assume you followed the binary packages installation guide for your operating system (deb packages for Linux). You can still follow along if you built from source, but the path to your setup files will likely be different. You also won't be able to use the `sudo apt install ros-<distro>-<package>` command (used frequently in the beginner level tutorials) if you install from source.

If you are using Linux and are not already familiar with the shell, [this tutorial](#) will help.

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

Running a Launch File

Open a new terminal and run:

```
$ ros2 launch turtlesim multisim.launch.py
```

This command will run the following launch file:

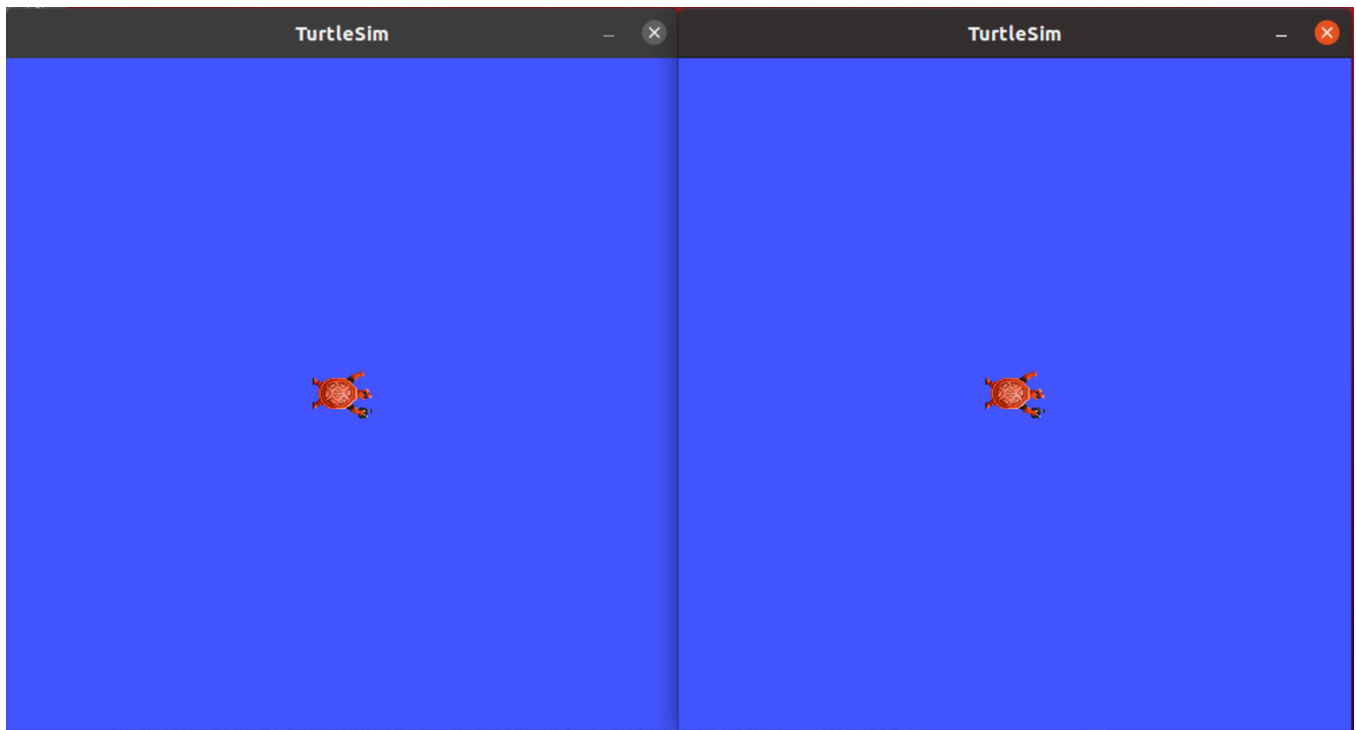
```
from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            namespace='turtlesim1', package='turtlesim',
            executable='turtlesim_node', output='screen'),
        launch_ros.actions.Node(
            namespace='turtlesim2', package='turtlesim',
            executable='turtlesim_node', output='screen'),
    ])
```

Note

The launch file above is written in Python, but you can also use XML and YAML to create launch files. You can see a comparison of these different ROS 2 launch formats in [Using XML, YAML, and Python for ROS 2 Launch Files](#).

This will run two turtlesim nodes:



For now, don't worry about the contents of this launch file. You can find more information on ROS 2 launch in the [ROS 2 launch tutorials](#).

(Optional) Control the Turtlesim Nodes

Now that these nodes are running, you can control them like any other ROS 2 nodes. For example, you can make the turtles drive in opposite directions by opening up two additional terminals and running the following commands:

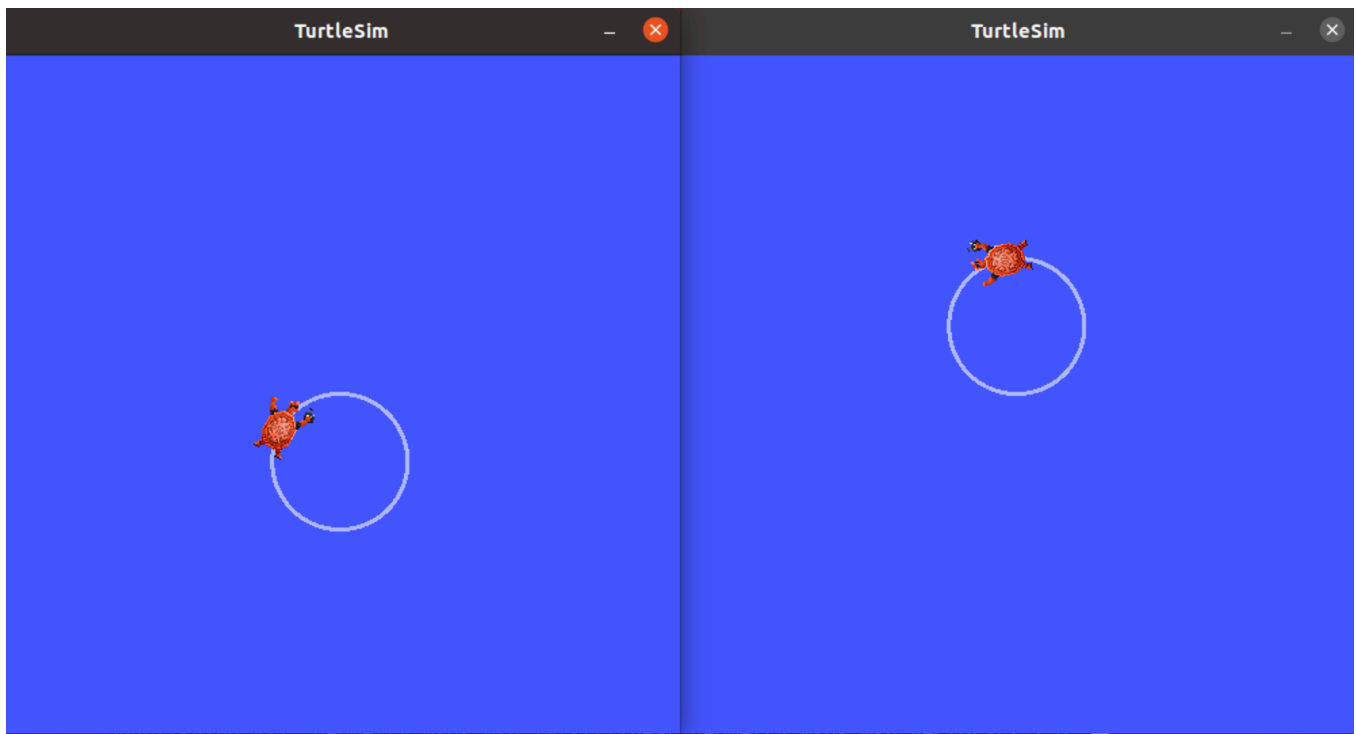
In the second terminal:

```
$ ros2 topic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

In the third terminal:

```
$ ros2 topic pub /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"
```

After running these commands, you should see something like the following:



Summary

The significance of what you've done so far is that you've run two turtlesim nodes with one command. Once you learn to write your own launch files, you'll be able to run multiple nodes - and set up their configuration - in a similar way, with the `ros2 launch` command.

For more tutorials on ROS 2 launch files, see the [main launch file tutorial page](#).

Next steps

In the next tutorial, [Recording and playing back data](#), you'll learn about another helpful tool, `ros2 bag`.

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Recording and playing back data

Goal: Record data published on a topic so you can replay and examine it any time.

Tutorial level: Beginner

Time: 10 minutes

Contents

- [Background](#)
- [Prerequisites](#)
- [Tasks](#)
 - [1 Setup](#)
 - [2 Choose a topic](#)
 - [3 ros2 bag record](#)
 - [4 ros2 bag info](#)
 - [5 ros2 bag play](#)
- [Summary](#)
- [Next steps](#)
- [Related content](#)

Background

`ros2 bag` is a command line tool for recording data published on topics in your system. It accumulates the data passed on any number of topics and saves it in a database. You can then replay the data to reproduce the results of your tests and experiments. Recording topics is also a great way to share your work and allow others to recreate it.

Prerequisites

You should have `ros2 bag` installed as a part of your regular ROS 2 setup.

If you need to install ROS 2, see the [Installation instructions](#).

This tutorial talks about concepts covered in previous tutorials, like [nodes](#) and [topics](#). It also uses the [turtlesim package](#).

As always, don't forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

You'll be recording your keyboard input in the `turtlesim` system to save and replay later on, so begin by starting up the `/turtlesim` and `/teleop_turtle` nodes.

Open a new terminal and run:

```
$ ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```
$ ros2 run turtlesim turtle_teleop_key
```

Let's also make a new directory to store our saved recordings, just as good practice:

Linux

macOS

Windows

```
$ mkdir bag_files  
$ cd bag_files
```

2 Choose a topic

`ros2 bag` can only record data from published messages in topics. To see the list of your system's topics, open a new terminal and run the command:

```
$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

In the topics tutorial, you learned that the `/turtle_teleop` node publishes commands on the `/turtle1/cmd_vel` topic to make the turtle move in turtlesim.

To see the data that `/turtle1/cmd_vel` is publishing, run the command:

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

Nothing will show up at first because no data is being published by the teleop. Return to the terminal where you ran the teleop and select it so it's active. Use the arrow keys to move the turtle around, and you will see data being published on the terminal running `ros2 topic echo`.

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

3 ros2 bag record

3.1 Record a single topic

To record the data published to a topic use the command syntax:

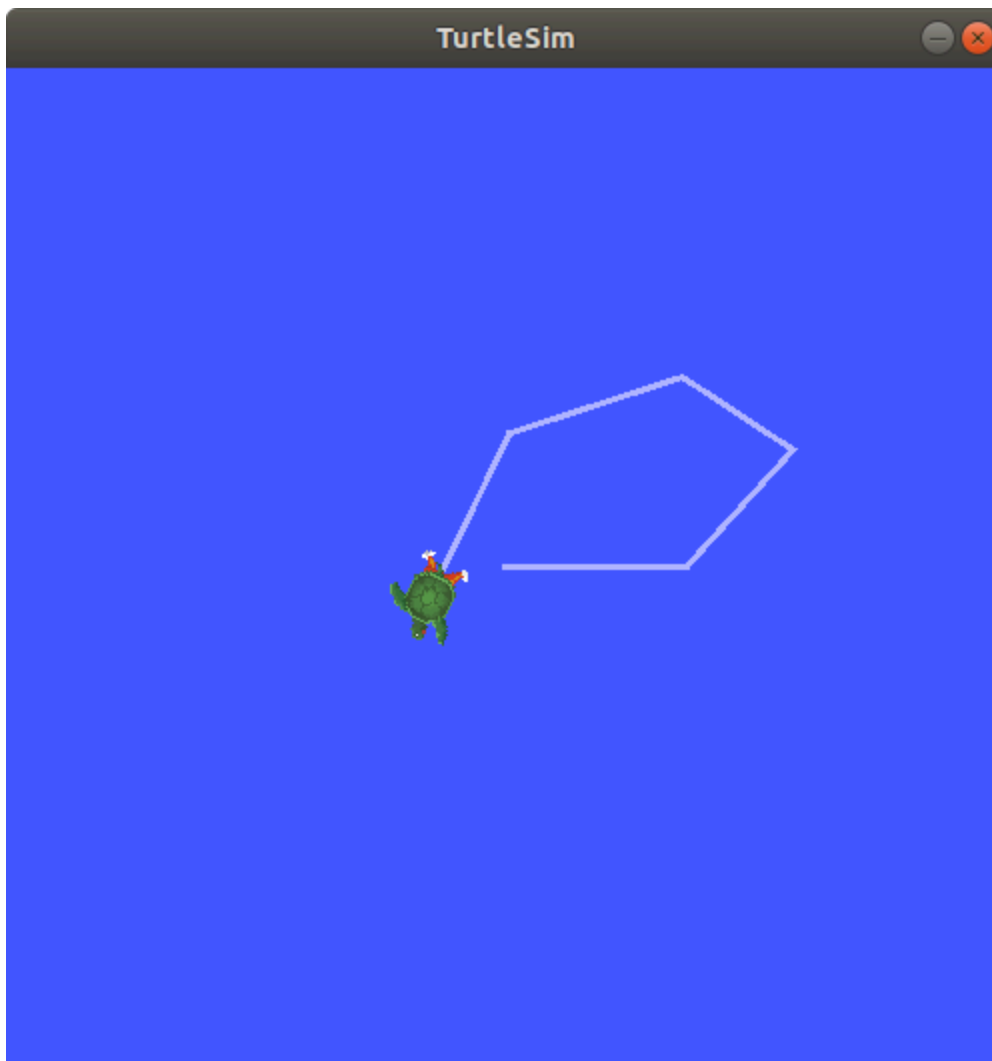
```
$ ros2 bag record <topic_name>
```

Before running this command on your chosen topic, open a new terminal and move into the `bag_files` directory you created earlier, because the rosbag file will save in the directory where you run it.

Run the command:

```
$ ros2 bag record /turtle1/cmd_vel
[INFO] [rosbag2_storage]: Opened database 'rosbag2_2019_10_11-05_18_45'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery...
```

Now `ros2 bag` is recording the data published on the `/turtle1/cmd_vel` topic. Return to the teleop terminal and move the turtle around again. The movements don't matter, but try to make a recognizable pattern to see when you replay the data later.



Press `Ctrl+C` to stop recording.

The data will be accumulated in a new bag directory with a name in the pattern of `rosbag2_year_month_day-hour_minute_second`. This directory will contain a `metadata.yaml` along with the bag file in the recorded format.

3.2 Record multiple topics

You can also record multiple topics, as well as change the name of the file `ros2 bag` saves to.

Run the following command:

```
$ ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose
[INFO] [rosbag2_storage]: Opened database 'subset'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/pose'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery...
```

The `-o` option allows you to choose a unique name for your bag file. The following string, in this case `subset`, is the file name.

To record more than one topic at a time, simply list each topic separated by a space. In this case, the command output above confirms that both topics are being recorded.

You can move the turtle around and press `Ctrl+C` when you're finished.

Note

There is another option you can add to the command, `-a`, which records all the topics on your system.

4 ros2 bag info

You can see details about your recording by running:

```
$ ros2 bag info <bag_file_name>
```

Running this command on the `subset` bag file will return a list of information on the file:

```
$ ros2 bag info subset
Files:                subset.db3
Bag size:             228.5 KiB
Storage id:           sqlite3
Duration:             48.47s
Start:               Oct 11 2019 06:09:09.12 (1570799349.12)
End:                 Oct 11 2019 06:09:57.60 (1570799397.60)
Messages:            3013
Topic information: Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/Twist | Count: 9 |
Serialization Format: cdr
                    Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count: 3004 | Serialization
Format: cdr
```

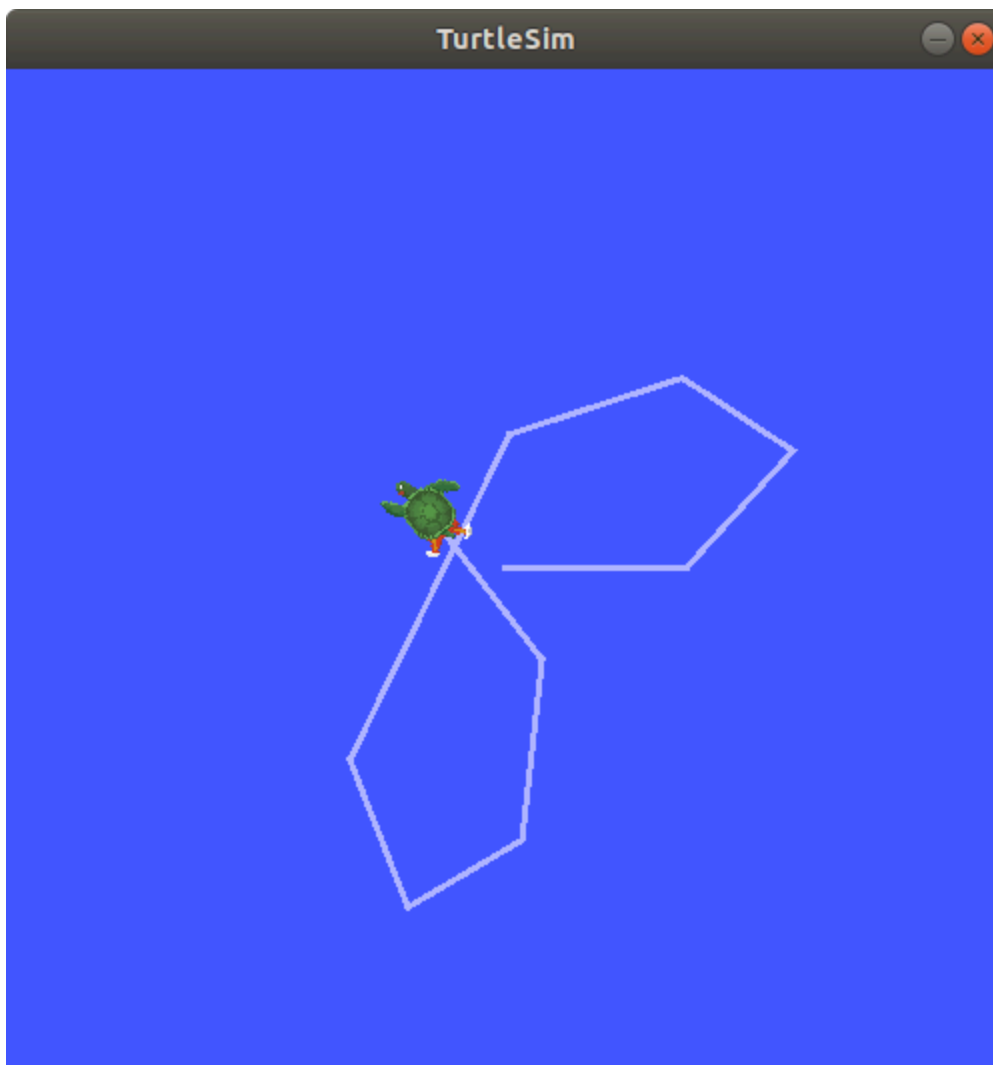
5 ros2 bag play

Before replaying the bag file, enter `Ctrl+C` in the terminal where the teleop is running. Then make sure your turtlesim window is visible so you can see the bag file in action.

Enter the command:

```
$ ros2 bag play subset
[INFO] [rosbag2_storage]: Opened database 'subset'.
```

Your turtle will follow the same path you entered while recording (though not 100% exactly; turtlesim is sensitive to small changes in the system's timing).



Because the `subset` file recorded the `/turtle1/pose` topic, the `ros2 bag play` command won't quit for as long as you had turtlesim running, even if you weren't moving.

This is because as long as the `/turtlesim` node is active, it publishes data on the `/turtle1/pose` topic at regular intervals. You may have noticed in the `ros2 bag info` example result above that the `/turtle1/cmd_vel` topic's `Count` information was only 9; that's how many times we pressed the arrow keys while recording.

Notice that `/turtle1/pose` has a `Count` value of over 3000; while we were recording, data was published on that topic 3000 times.

To get an idea of how often position data is published, you can run the command:

```
$ ros2 topic hz /turtle1/pose
```

Summary

You can record data passed on topics in your ROS 2 system using the `ros2 bag` command. Whether you're sharing your work with others or introspecting your own experiments, it's a great tool to know about.

Next steps

You've completed the "Beginner: CLI Tools" tutorials! The next step is tackling the "Beginner: Client Libraries" tutorials, starting with [Creating a workspace](#).

Related content

A more thorough explanation of `ros2 bag` can be found in the README [here](#). For more information on QoS compatibility and `ros2 bag`, see [rosbag2: Overriding QoS Policies](#).

You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at [Kilted](#).

Launch

ROS 2 Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.

1. [Creating a launch file.](#)

Learn how to create a launch file that will start up nodes and their configurations all at once.

2. [Launching and monitoring multiple nodes.](#)

Get a more advanced overview of how launch files work.

3. [Using substitutions.](#)

Use substitutions to provide more flexibility when describing reusable launch files.

4. [Using event handlers.](#)

Use event handlers to monitor the state of processes or to define a complex set of rules that can be used to dynamically modify the launch file.

5. [Managing large projects.](#)

Structure launch files for large projects so they may be reused as much as possible in different situations. See usage examples of different launch tools like parameters, YAML files, remappings, namespaces, default arguments, and RViz configs.

ⓘ Note

If you are coming from ROS 1, you can use the [ROS Launch Migration guide](#) to help you migrate your launch files to ROS 2.