https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

ROS is a mid-level operator that breaks down complex systems into manageable parts. Each part of the robotic system is referred to as a "node." For example, let's think of a simple car with a distance sensor mounted on top and a couple of LEDs. The nodes in this robotic system would be the wheels, the distance sensor, and the LEDs. For more precision, we could even divide nodes down to individual components of the same functional part. By this, I mean we could have separate nodes for the right-side wheel and the left-side wheel of our car.

Understanding Nodes

In the context of ROS, Laika's functionality is broken down into a network of **nodes**, each responsible for a specific task or hardware component. For example, one node might control the **front left leg**, handling the angles and servo commands needed to move it. Another node could handle **IMU data**, publishing information about orientation and acceleration. A **camera node** might stream video or publish processed data like object detections or depth maps.

These nodes don't directly talk to each other, indeed they **publish and subscribe** to topics. For instance, a **motion controller node** might subscribe to the IMU and camera topics to understand Laika's current state, then publish velocity commands to the leg control nodes. If you're running a gait pattern, you might have a **gait scheduler node** that coordinates leg movement by publishing timed position commands to each limb node.

```
rqt
```

rqt is a graphical tool in ROS that helps visualize, monitor, and debug what's happening in your robot system. It gives an interface to see which nodes are running, what topics they're publishing or subscribing to, and how data is flowing between them. You can also use it to plot real-time sensor data, view system logs, and even change parameters on the fly. (for example in the turtlesim example package from the ROS documentation, we use rqt to change the thickness of the line drawn by the virtual turtle.) It's especially useful when you're trying to understand how different parts of your robot are interacting or when something isn't working and you need to trace the problem visually.

Remapping nodes

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

Remapping a node to work for another node is sometimes useful. In this example we are remapping the **turtle_teleop_key node**, first by flagging the args, then flagging the remap to in this case "turtle1/cmd_vel…" (to :=) "turtle2/cmd_vel. Just like on a calculator the := phrasing means "store this". Running this command will open a new node with the remapped functionality of interfacing with a different named subscriber.

Running packages in ROS (We say running the executable *from* the package)

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

Seeing the Network

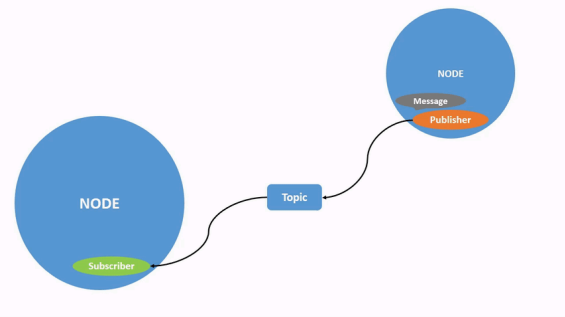
“A full robotic system consists of many nodes working in concert. In ROS 2, a single executable (a Python program for Laika,) can contain one or more nodes.”

- This “network” as we can start thinking of it as is the *ROS GRAPH*
- List running nodes using `ros2 node list`
- Find info like publishing, subscribing, etc using `ros2 node info <node_name>`

Understanding Topics

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Tops are essential buses that move information in the ROS architecture

Topics are like forks, they can push data to many or many to one. Below is a ROS graph.



To visualize a graph of how nodes are connected with each other we can run the command:

```
rqt_graph
```

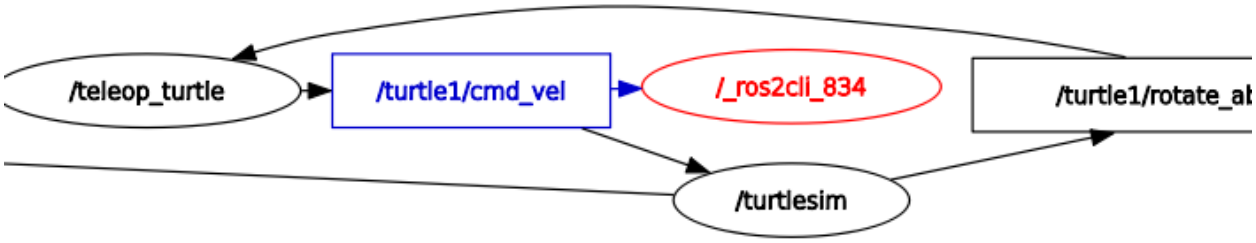
After running the command you can select to see the Nodes/Topics that are active in the top left of the rqt viewer

Looking at the center of the graph a flow chart is visible with the topic in the center and arrows to the topic of nodes that are publishing to it. Arrows from the topic are nodes subscribed to the topic

To see topics that are active using just command line you can call `ros2 topic list`
Or `ros2 topic list -t`

To show an active call out of what a topic is seeing you can echo it. When doing so, all data published from nodes to this topic will be displayed here as they go to their subscribing nodes

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



Looking back at the rqt_graph we can see that this echo call (*/ros2cli…)* is seeing the data published to the cmd_vel topic and is acting as a visual subscriber

- 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**.

Find the structure of a msg (what the system expects using:

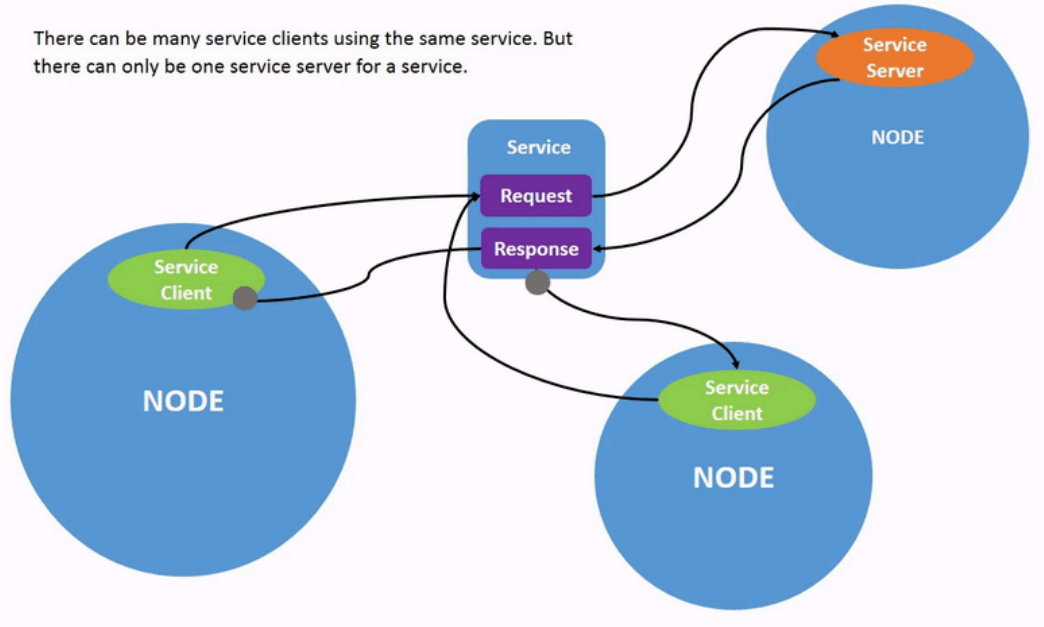
```
ros2 interface show geometry_msgs/msg/Twist
```

- Where you can replace the message with whatever your trying to inspect

Understanding Services

Services are another method of communication for nodes in the ROS graph. S**ervices are based on a call-and-response model** versus the publisher-subscriber model of topics.

- With a service a response is conditional
- Many nodes (clients) can use a service, but only one node can act as the server
-




```
ros2 service list
```

Brings up the list of available services when multiple nodes are running.
The services seen here are command line versions of what was seen when opening rqt

To call a service from the command line we need its type and to understand what arguments it takes.

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

Gives us the type.

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

Gives us the structure of arguments for this type

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

When calling a service, its type, and the correct structure of arguments the subscribing

or client nodes will respond as such. For the running example of turtlesim,

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

Clears the turtlesim board. As shown, we call the /clear service, state its type, and in this example clear requires no arguments so this was left blank

Remember! To find available services you can call ros2 service list, then from there find a services type, list its arguments and then call the whole from the command line (or save some sanity and use rqt if doing it by hand!)

- When you get your list of arguments you must structure them with "{ type: arg}"

```
joshkraus@AX8Max:~$ ros2 interface show turtlesi
m/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be c
reated and returned if this is empty
```

-
- So to call the spawn service I would use the following:

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta:
0.2, name: ''}"
```

Note here how the arguments are formatted. Even the name is left blank! The exact structure of this service call can be derived using the commands above, again: *list, type, interface, call*

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

Understanding Parameters

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.

To see the parameters of open nodes you can run the command

```
ros2 param list
```

Once you see a list of params you can see the values of params using:

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

In turtle sim, I see that the param for background_g is up (along with red and blue) so to see the green value

```
joshkraus@AX8Max:~$ ros2 param get /turtlesim background_g
Integer value is: 86
```

To change a parameter value right away you use the command:

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

Doing this change will only affect the node param *during these sessions* and **does not keep the change permanently. To make the change non-volatile:**

To first see all the set params for a node use `ros2 param dump <node name>`

```
joshkraus@AX8Max:~$ ros2 param dump /turtlesim > turtlesim.yaml
joshkraus@AX8Max:~$ ls
turtlesim.yaml
joshkraus@AX8Max:~$ ros2 param load /turtlesim turtlesim.yaml
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
```

Here is an example of saving my set params to a file in my working directory of ros, then loading it to the turtlesim. To do this on launch of a node, use the following:

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

To continue with the running example this would be the command line run for ts:

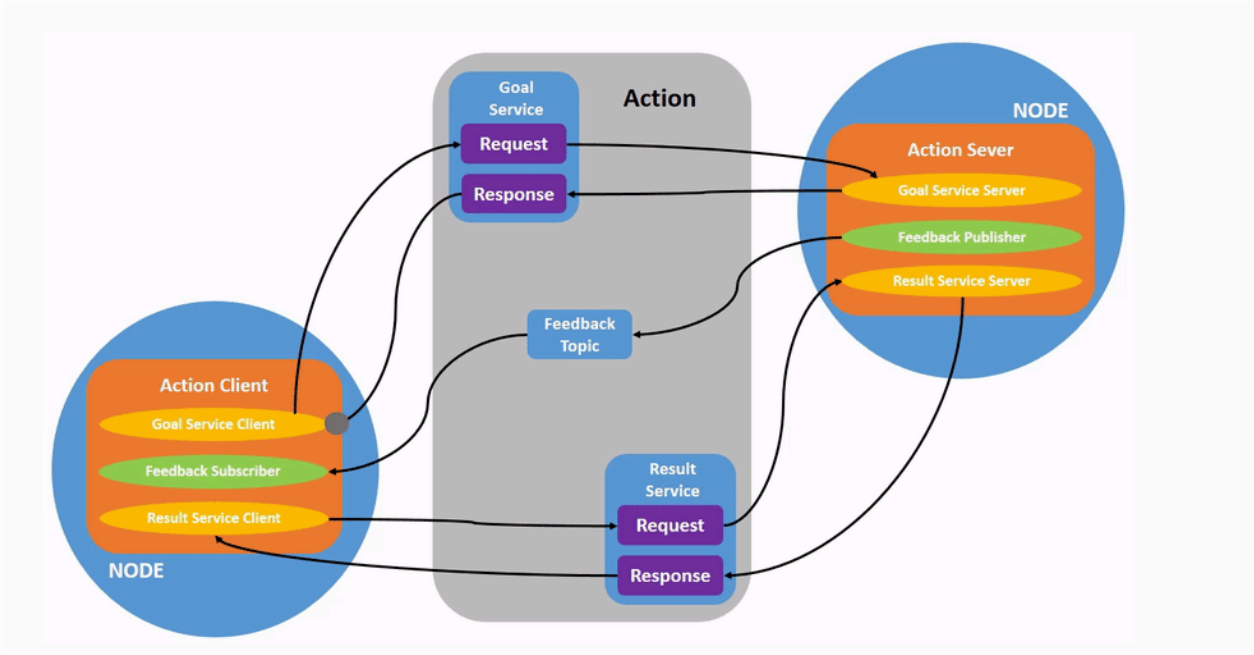
```
ros2 run turtlesim turtlesim_node --ros-args --params-file
turtlesim.yaml
```

“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.”

[This space is left intentionally blank]

Understanding Actions

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 a framework of Services and topics, and can be canceled. The topic provides feedback on the action and the service manages the requests and responses of the nodes.

Actions can be canceled on the server side or by the client. Cancelation by the client is “stopping the goal” or “canceling the goal”. Cancelation on the server side is “aborting the goal”

Using the running example of turtlesim, node and telop_key, the node is the client and the telop_key is the server. Using the GBVC.. keys that form a box around f are setting “goals” from the server side to send to the client, which provides feedback in the form of messages on the goal topic!

- *Dont assume* that when 2 goals are given in a server (one given after the other but before the first has a chance to complete) that the server will choose to abort the old goal and execute the new one, this is just how this server was set up)

You can display a package nodes action servers using the following command:

```
ros2 node info /<package name>
```

```
ypes
    /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/SetPara
metersAtomically
  Service Clients:

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

joshkraus@AX8Max:~$
```

Using turtlesim as the example (/turtlesim), the previous information in the image appears

Note: the rotate_absolute action is under a server for the turtlesim node, which means that the node responds to and provides feedback for the /turtle1/rotate_absolute action.

To list just the actions available in a ROS2 system and grab command line format:

```
ros2 action list -t
```

- *You can take the action this returns and run ros2 action info <action> to see its clients and servers*

As before you can call from the command line. Once the action is fully grabbed with `ros2 action list -t` run what was in the [brackets] `ros2 interface show <[ ]>`

```
joshkraus@AX8Max:~$ ros2 action list -t
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
joshkraus@AX8Max:~$ ros2 interface show turtlesim/action/RotateAbsol
ute
# 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
```

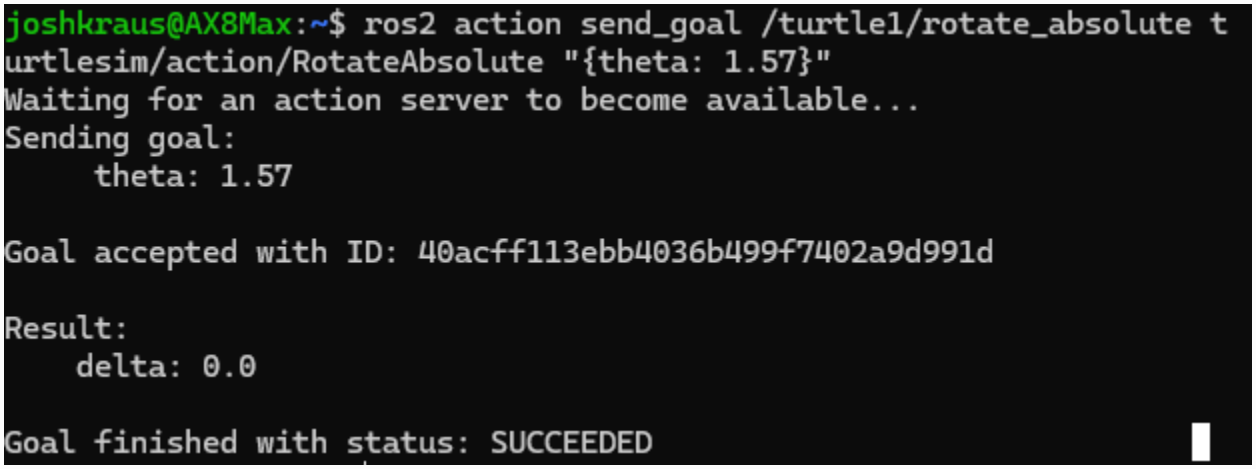
Now that we have the structure we can send goals to actions via:

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

For example:

```
ros2 action send_goal /turtle1/rotate_absolute
turtlesim/action/RotateAbsolute "{theta: 1.57}"
```

Which puts the little turtle pointing straight up (this makes sense since the top of the unit circle is at pi/2 or 3.1415/2 which = 1.57 rad!



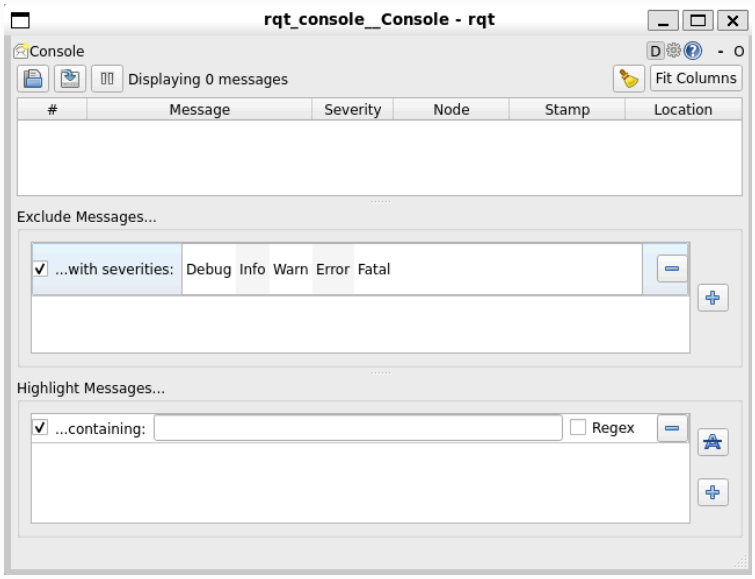
- Delta shows the change from the original position, here I ran the command two timers and thus the second had a delta of 0 since we were moving to exactly where we were.

- All goals get a unique ID, and return with a goal status

“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.”

Using rqt_console to view logs

Rqt_console is the rqt graphical tool that you can use to view logs collected over time. Much like the section before on rqt, you interact with it not through commands but the pop up interface (hence GUI)



- 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.

Rqt console can be helpful in determining where something went wrong, or the events that led up to it. There are a number of good reasons to examine the log messages.

Launching a system of nodes together

By now, every node has been launched in its own shell. This of course becomes cumbersome in complex robotics systems. That's why we use “launch files,” and the *ros2 launch* command.

Using the running example we can launch 2 turtlesims at once using the following:

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

Where multisim.launch.py is our starter script, in python: (more on how to write this later) (note that turtlesim is the demo package where the file is)

```
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'),
    ])
```

Data Logging

Sometimes we want to record our data to check experiments or share our topics with others. This can be done with a built in ros feature. Recorded data can also be played back to enhance reproducibility. Use:

```
ros2 bag
```

Best practice is to make a dedicated folder to store data. This can be done in command line as expected:

```
mkdir bag_files
cd bag_files
```

You want to be inside the folder where you plan to record your logged data.

```
joshkraus@AX8Max:~$ mkdir bag_files
cd bag_files
joshkraus@AX8Max:~/bag_files$
```

The ros2 bag only records data that is being *published* from topics. Note again that to see the topics available for us to listen to we can use **ros2 topic list**

The ros2 bag can record a single or multiple topics:

- Single topic

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

- Multi topic

```
ros2 bag record <topic_name> <topic_name2> <topic_name3>
```

You may pause recording with space and stop with ctrl + c (remember to record in the folder you want to store data in!) (use record -a to record all topics)

To grab the info on a bag and see some stats on it:

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

```
ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose
```

NOTE: from within a folder you can have many bags, name them with -o “name”
Here the bag was named subset.

You can play back bags which will communicate right to the same nodes as when recorded:

```
ros2 bag play <”name”>
```

End of Beginner: CLI Tools tutorials!