3.2 Topics - Publisher/Subscriber communication

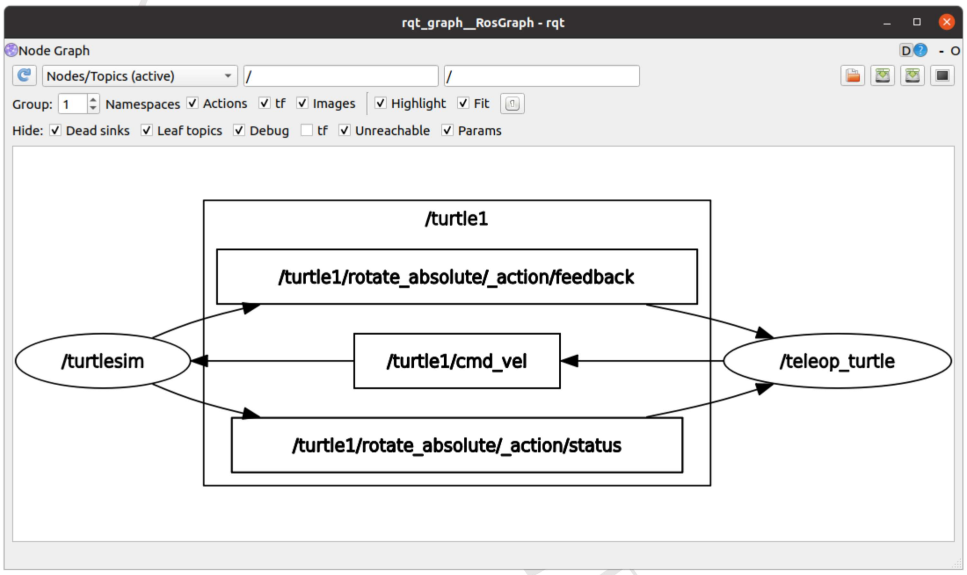
To start understanding Topics in practice we will use our two already known nodes, 'turtlesim_node' and 'teleop_key'.

```
$ ros2 run turtlesim turtlesim_node
```

```
$ ros2 run turtlesim turtle_teleop_key
```

In the third tab of our terminal, we will run the 'rqt_graph', to observe what is happening:

```
$ rqt_graph
```


rqt_graph__RosGraph - rqt
Node Graph
Nodes/Topics (active)
/
/
Group: 1
☑ Namespaces ☑ Actions ☑ tf ☑ Images ☑ Highlight ☑ Fit
Hide: ☑ Dead sinks ☑ Leaf topics ☑ Debug ☐ tf ☑ Unreachable ☑ Params
/turtle1
/turtle1/rotate_absolute/_action/feedback
/turtlesim
/turtle1/cmd_vel
/teleop_turtle
/turtle1/rotate_absolute/_action/status

Now, what we see is how these two nodes are connected. If You hover Your mouse over a topic You will also see a coloring of the elements, which can change depending on the direction of the communication. Now let's analyze what we see a bit more.

The graph tells us, that there is a '/turtlesim' and a '/teleop_turtle' node running currently. These two nodes are now connected with 3 channels, one topic (in the middle), and two actions (up and down, but we will deal with actions later). We can see that when checking only the publisher and subscriber communication via a topic, there is one case. The 'teleop_turtle' acts as a publisher, publishing on the 'turtle1/cmd_vel' topic, for which the '/turtlesim' is subscribed. This seems logical according to our experience since we are using the 'teleop_key' for commanding the turtle inside 'turtlesim'.

We are not limited to rqt when trying to analyze the currently active communication via topics. we can easily use the "list" command to reveal the currently active ones:

```
$ ros2 topic list
```

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

(We can also use a "-t" at the end to also print the type of the topics)

```
/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]
```

(If you wonder where are these in rqt, You just need to uncheck the checkboxes at the hide part of the GUI)

Sometimes it is also useful for a quick check to see the dataflow in a topic for which we can use the echo command:

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

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

(Don't panic if You see nothing until You don't command the turtle You will see no communication on the topic. Move Your turtle a little.)

If You go back to the rqt GUI and You unchecked all the hide options, You can see that now this echo command brings in a new node '/_ros2cli_xxxx', since to be able to visualize the data we need to subscribe to the topic.

Another useful technique to observe a functioning system is to use the topics info, that gives us the type of the messages used in the topic and also the number of publishers and subscribers:

```
$ ros2 topic info /turtle1/cmd_vel
```

```
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

We can see, that the '/turtle1/cmd_vel' has a type of 'geometry_msgs/msg/Twist' from which we can conclude, that there is a package called 'geometry_msgs' which has a message definition called 'Twist'. Let's see what this 'Twist' message definition holds:

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

```
# This expresses velocity in free space broken into its linear and
   angular parts.

    Vector3  linear
    Vector3  angular
```

This tells us, that this message consists of two vectors called "linear" and "angular", both three-dimensional, that will tell the turtle where to move. This is so far according to our experiences with the echo command, that we used earlier.

We can easily send messages (data) as well to any topic similarly to how we echo them to the terminal. The only restriction is that we need to organize our message in YAML syntax. Now let's try two things, let's send one message to the turtle and then try sending a stream, while observing the differences

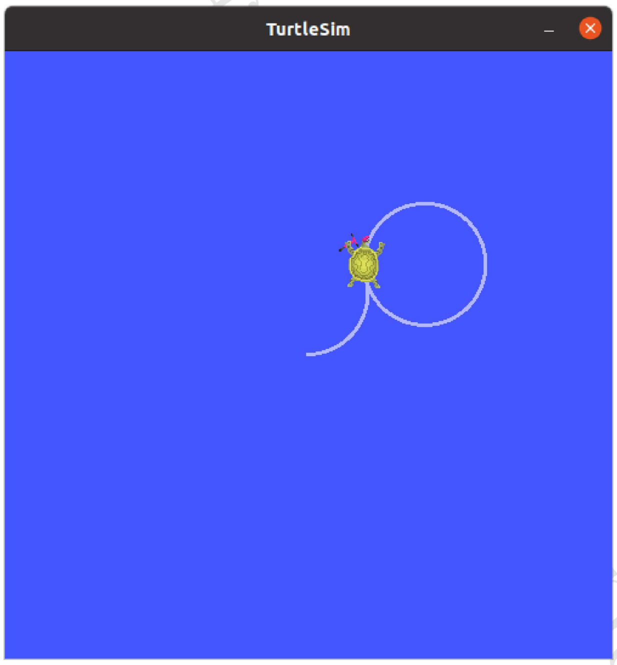
```
$ ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

The "–once" will induce a message sending once.

```
$ ros2 topic pub --rate 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: -1.8}}"
```

With the "–rate" we are telling our system to repeat sending this same message until we stop this process by pressing ctrl+C. The 1 after the "–rate" argument tells the frequency in Hertz.

Now what is the difference? The turtle as usually all robots needs constant commanding orders to work properly. We sent the command once the turtle executed the order and stopped. In the case of message stream sending we can see a constantly cycling turtle. (We will also see a similar change in the rqt graph that we had in the case of echoing.)



Lastly, in some cases, we want to see that what is the communication frequency on a certain topic. For this, we can use the 'hz' command:

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

```
average rate: 59.354
  min: 0.005s max: 0.027s std dev: 0.00284s window: 58
```

You can see that the average is pretty close to 60 which is the number of messages that will be sent in one minute. This is pretty close to our 1Hz, but not exactly one.

3.3 Services - Server/Client communication

Before starting to experiment with this next concept called Services, let's have a blank page. Stop all nodes and clear the tabs as well so nothing will cause any unexpected behavior. We will use again our guinea pigs the 'turtlesim' and 'teleop_key'. If the two node is started, we can start our observations with the currently available services:

```
$ 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
```

We can see that currently many more services can be used, from which there are 6 pairs for these two nodes, that are concerning the parameters of the nodes. (Similarly to messages over topics, if we want just a bit more information about the services we can use the "-t" here as well to print the types of the services as well.)

As it was with the messages, services have type as well. We can check a type of the service definition by the "type" command. Let's check the '/spawn':

```
$ ros2 service type /reset
```

```
std_srvs/srv/Empty
```

This answer lets us know that this service is not giving back any data nor when receiving a response.

We can also find services based on their type:

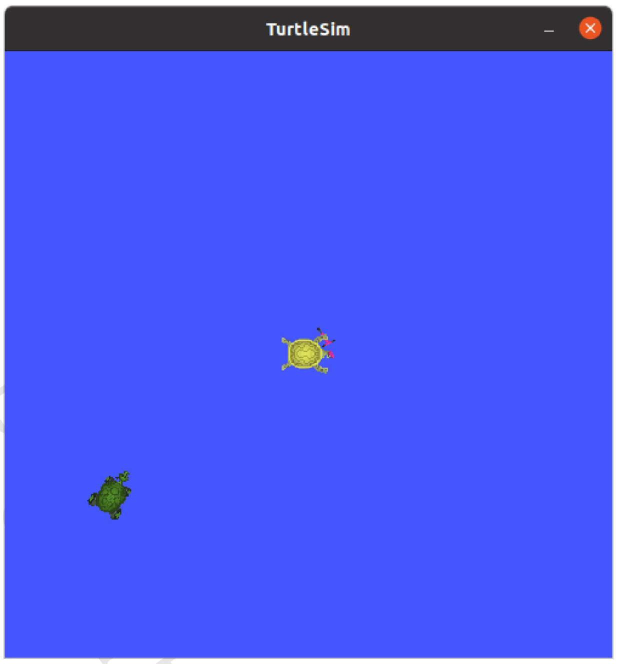
```
ros2 service find std_srvs/srv/Empty
```

```
/clear
/reset
```

This indicates, that not just '/reset', but also '/clear' has similarly the 'Empty' type. Now we can ask the system to print out the structure of this service:

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

```
---
```



Well, this hopefully was not a big surprise that we can see no structure, since we stated that the 'Empty' service has no data input nor output. For a more interesting example we can check the 'Spawn' service:

```
$ ros2 interface show turtlesim/srv/Spawn
```

Obviously, the most important tool is the service call, for which we can use the 'call' command. Let's spawn a new turtle to the board:

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 3.0, theta:
    1.0, name: Steve}"
```

Now welcome Steve at the bottom left side of the TurtleSim. After we had enough time to meet him, we are needed to bid farewell, since we need a blank page again (ctrl+C).

3.4 Actions - Action server/Action client communication

We again will use our two basic nodes for the tests, so let's run the 'turtlesim_node' and the teleop_key' again. If we use the 'info' command on the 'turtlesim', we will see all the possible communication for the node. At the bottom we can see that it has a definition for one action server:

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

If we take a closer look at the 'teleop_key' with the 'info' command, with this method we will see the counterpart (client) of this action:

```
/teleop_turtle
       .
       .
       .
  Action Servers:

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

The name suggests a functionality we can use to reorient the turtle somehow. Taking a closer look at the tab of the terminal with the running 'teleop_key' node, we can see the following:

```
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.
```

The second line seems to advise on the usage of the action. (ERTGBVCD for reorientation and F for stopping the action.) Let's try it. If we push some buttons from the list the turtle will rotate in place with absolute orientation. Also, we can try to stop an action by pressing "F". If we successfully tried out both, then we will have a similar log in the turtlesim's tab:

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

It is the same as how we get the list of active actions as we did previously, we use the 'list' command after the "action" keyword (and also the "-t" extension works as well). Now only one active action we can use:

```
$ ros2 action list -t
```

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

The usage of the 'info' command is the same as well:

```
$ ros2 action info /turtle1/rotate_absolute
```

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

We can see, that there is one action client and server.

The 'interface show' command is working similarly as well:

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

```
# 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
```

We can see that the request part has a ’theta’ parameter, the response has a variable named ’delta’, and the feedback has the ’remaining’.

Similarly to the message sending and service calling we can manually initiate an action request. This can be done by the ’send_goal’ command. Let’s ask the turtle (in radians) to turn to face the upper wall:

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

And we will see the following in the terminal tab:

```
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
```

Here we can’t see the feedback part, to make it visible we can add ’–feedback’ to the end of the 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
```

This concludes our current practice. So far You should understand the basic tools to navigate and investigate within a ROS2 project and now You learned the basics of the three communication opportunities inside ROS2: via topics, via services, and via actions. In the next session, we will learn about making our own packages and also will make the first that can utilize these communication types.