1 Introduction to ROS2

ROS stands for Robotic Operating System. ROS2 is the second major iteration after ROS1. For convenience, let's refer ROS2 as ROS. ROS is a middleware layer for robotics applications. Provides the libraries necessary for handling communication and building robotic applications.

1.1 Code organization and packaging

Get your workspace organized with the packages. In an application with robotic arms, one package can handle gripper-related functions (or nodes), and another package handles your complex control logic-related nodes. The following code will create a package for your ROS2 Python code.

```
$ cd ~/<your_workspace>/src
$ ros2 pkg create --build-type ament_python --license Apache-2.0 <pkg_name>
```

You should see the following files created inside the src folder.

- *package.xml* file containing meta information about the package
- *resource/* < *package_name* > marker file for the package
- *setup.cfg* is required when a package has executable, so ros2 run can find them
- *setup.py* containing instructions for how to install the package
- < *package_name* > - a directory with the same name as your package, used by ROS 2 tools to find your package, contains *__init__.py*

Move to the root of your workspace to build the ROS package you just created.

```
$ cd ~/<your_workspace>
$ colcon build
```

Next, use the following command to see if ROS2 has added your newly created package to its list of packages.

```
$ ros2 pkg list
```

If not, you have to use the source command to make available to package you build to the current terminal session of ros2.

```
$ source ~/<your_workspace>/install/setup.bash
```

Follow this link to learn more!

1.2 Creating the first ROS node

A node is a core component of ROS. The logic of your program goes inside a node. A single node should be modular and provide a single purpose. For example, the wheel control implementation can be a single node, and the camera control can be another node. An ROS-based robotic application would have multiple nodes running in parallel and communicating with each other.

The following Python code will create an empty ROS node.

```
import rclpy
from rclpy.node import Node

class HelloClassNode(Node):
    def __init__(self):
        super().__init__('hello_class_node')
        self.get_logger().info('Hello Class')
```

```python
def main(args=None):
    rclpy.init(args=args)
    node = HelloClassNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Self-learning assignment: examine what *rclpy.spin(node)* does.

However, this node is not yet visible to the package builder. Make it visible by changing the setup.py file.

```python
from setuptools import find_packages, setup

package_name = 'hello_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='root',
    maintainer_email='root@todo.todo',
    description='TODO: Package description',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'hello_node_at_Setup = hello_pkg.hello_node:main'
        ],
    },
)
```

Next, add the dependency on *rclpy* library to *package.xml* file.

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
    schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>hello_pkg</name>
  <version>0.0.0</version>
  <description>Introduces basics of ROS2</description>
  <maintainer email="chandima@ccsu.edu">root</maintainer>
  <license>Apache-2.0</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>ament_xmllint</test_depend>
  <test_depend>python3-pytest</test_depend>
```

```
  <exec_depend>rclpy</exec_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Execute your node by running the command:

```
$ ros2 run <package_name> <node_name>
```

In a new terminal, make sure that your node is alive.

```
$ ros2 node list
```

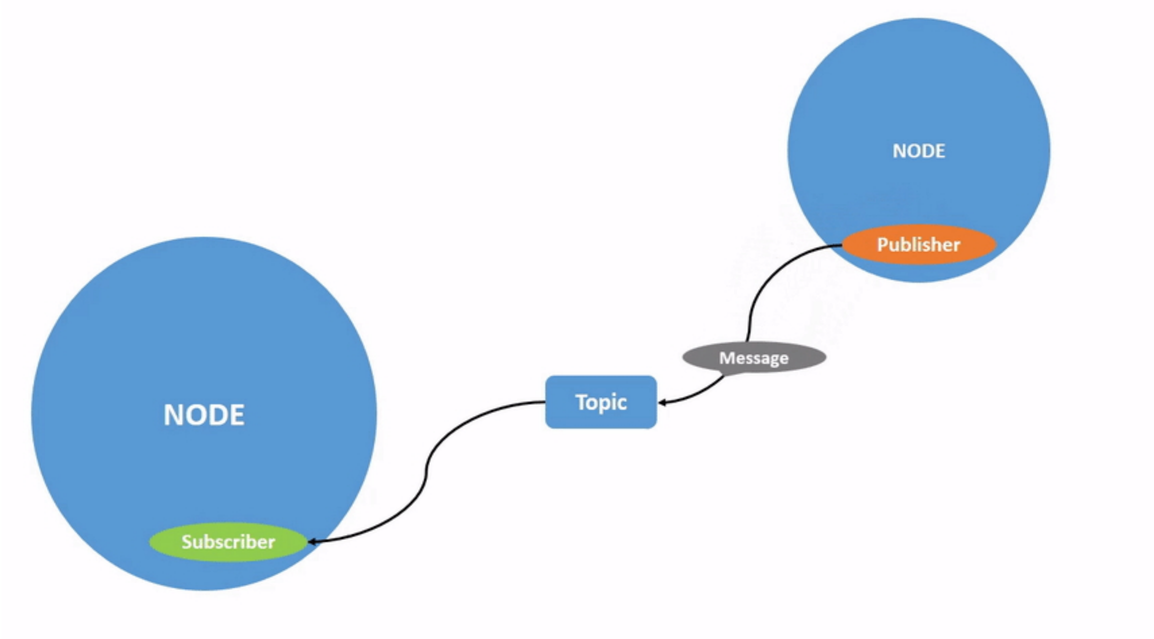
Follow this link to learn more!

2 ROS2 Communications

ROS nodes communicate through topics, services, actions, or parameters.

2.1 Publishers and Subscribers though Topics

A publisher residing inside a node continuously publishes a strongly formatted message to a topic at a certain rate. A subscriber who listens to the same topic can receive the published message continuously.



The following code will create a simple publisher node that publishes a String type message to a topic called 'tokyo_time'.

```
import rclpy
from rclpy.executors import ExternalShutdownException
from rclpy.node import Node
from datetime import datetime
from zoneinfo import ZoneInfo
from std_msgs.msg import String

class PublisherNode(Node):

    def __init__(self):
        super().__init__('time_publisher')
        self.publisher_ = self.create_publisher(String, 'tokyo_time', 10)
        timer_period = 1.0  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        msg = String()
        tokyo_time = datetime.now(ZoneInfo("Asia/Tokyo")).strftime("%Y-%m-%d
            %H:%M:%S")
        msg.data = tokyo_time
        self.publisher_.publish(msg)
        self.get_logger().info('Time in Tokyo: "%s"' % msg.data)
```

```python


def main(args=None):
    try:
        with rclpy.init(args=args):
            minimal_publisher = PublisherNode()

            rclpy.spin(minimal_publisher)
    except (KeyboardInterrupt, ExternalShutdownException):
        pass


if __name__ == '__main__':
    main()
```

Make sure to add the dependency *std_msg* to the *package.xml* file and link your newly created publisher file to the *setup.py*. Then build and run your node to check the output. In a new terminal, run the following command to see if your topic has already been created.

```
$ ros2 topic list
```

The following command investigates the contents of your topic.

```
$ ros2 topic echo /tokyo_time
```

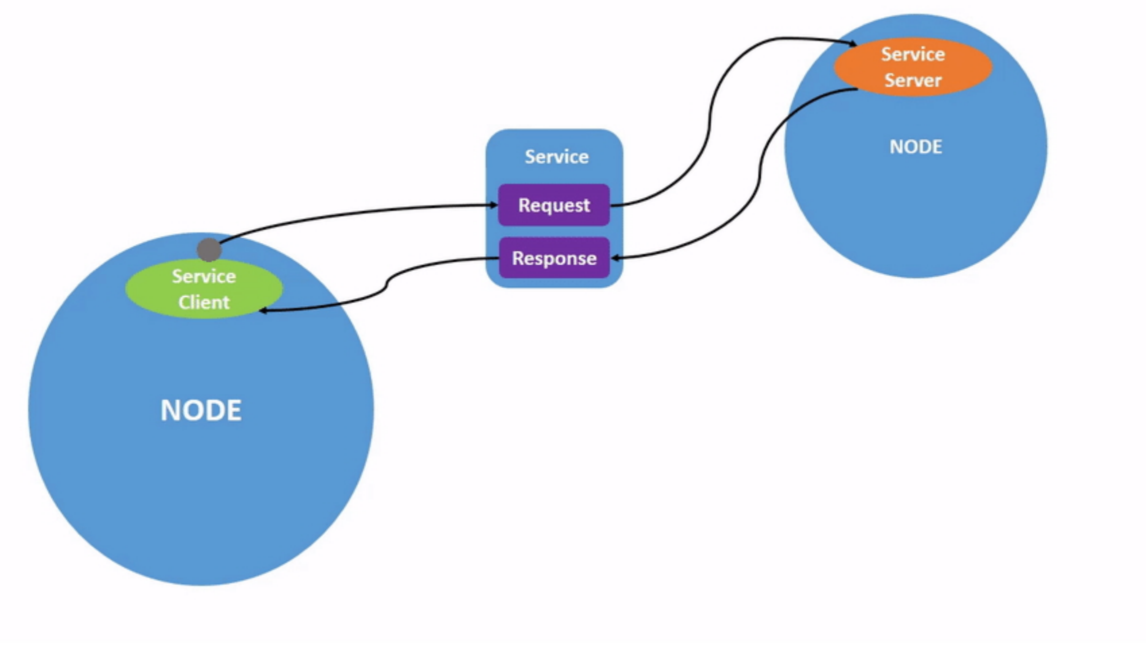
The following subscriber node listens to the topic *tokyo_time* and prints the content of the message.

```python
import rclpy
from rclpy.executors import ExternalShutdownException
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('time_subscriber')
        self.subscription = self.create_subscription(
            String,
            'tokyo_time',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard, "%s"' % msg.data)


def main(args=None):
    try:
        with rclpy.init(args=args):
            minimal_subscriber = MinimalSubscriber()

            rclpy.spin(minimal_subscriber)
    except (KeyboardInterrupt, ExternalShutdownException):
        pass


```

```
if __name__ == '__main__':
    main()
```

2.2 Clients and Servers though Services

A client-server relationship where the client requests a certain update / value from the server, and the server returns a response message after fulfilling the request.



Self-learning assignment: How do topics differ from services?

We used an already available message type called "String" in the subscriber-publisher example. However, most of the time you would have to create a custom message for your client-server application as the service calls become complex.

2.2.1 Create a custom service message

Create a new package to host the custom service message file. You can create the same package to include your custom messages for topics. It is customary to have a dedicated package for your custom message interfaces in a typical ROS workspace. This package has to be created using $ament_c make$.

```
$ cd ~/<your_workspace>/src
$ ros2 pkg create --build-type ament_cmake --license Apache-2.0
    custom_interfaces
```

Create a directory inside *src* called *srv*.

```
$ cd ~/<your_workspace>/src/custom_interfaces
$ mkdir srv
```

Make your strongly typed custom service file inside the *srv* folder. Give it a meaningful name and an extension *.srv*.

```
string zone
---
string time
```

Next, extend your CMakeLists.txt file and the package.xml file inside the custom interfaces package by adding the following lines.

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/TimeRetrevalMsg.srv"
)
```

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

Build your workspace.

Follow this link to learn more!

2.2.2 Create the server node

The following code will create a simple server that returns the time of a given zone.

```
from custom_interfaces.srv import TimeRetrevalMsg

from datetime import datetime
from zoneinfo import ZoneInfo

import rclpy
from rclpy.executors import ExternalShutdownException
from rclpy.node import Node


class TimeServer(Node):

    def __init__(self):
        super().__init__('time_server')
        self.srv = self.create_service(TimeRetrevalMsg, 'global_time_server'
            , self.get_time_from_zone)

    def get_time_from_zone(self, request, response):
        response.time = datetime.now(ZoneInfo(request.zone)).strftime("%Y-%m
            -%d %H:%M:%S")
        self.get_logger().info('Incoming request for zone: %s' % (request.
            zone))

        return response


def main():
    try:
        with rclpy.init():
            time_server = TimeServer()

            rclpy.spin(time_server)
    except (KeyboardInterrupt, ExternalShutdownException):
        pass


if __name__ == '__main__':
    main()
```

Note that our server node depends on the *custom_interfaces* library. Therefore, we need to add the following line to the *package.xml* file.

```
    <exec_depend>custom_interfaces</exec_depend>
```

2.2.3 Create the client node

The following code will create a client that requests the time from the server for a specific zone.

```python
from custom_interfaces.srv import TimeRetrevalMsg

import rclpy
from rclpy.executors import ExternalShutdownException
from rclpy.node import Node


class TimeClientAsync(Node):

    def __init__(self):
        super().__init__('time_client_async')
        self.cli = self.create_client(TimeRetrevalMsg, 'global_time_server')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...'
                )
        self.req = TimeRetrevalMsg.Request()

    def send_request(self, zone):
        self.req.zone = zone
        return self.cli.call_async(self.req)


def main(args=None):
    try:
        with rclpy.init(args=args):
            time_client = TimeClientAsync()
            future = time_client.send_request("Asia/Tokyo")
            rclpy.spin_until_future_complete(time_client, future)
            response = future.result()
            time_client.get_logger().info(
                'Current time of the region: %s is %s' %
                (time_client.req.zone, response.time))
    except (KeyboardInterrupt, ExternalShutdownException):
        pass


if __name__ == '__main__':
    main()
```

Run both the server and the client nodes in two terminals and check the output.

3 Launch Files

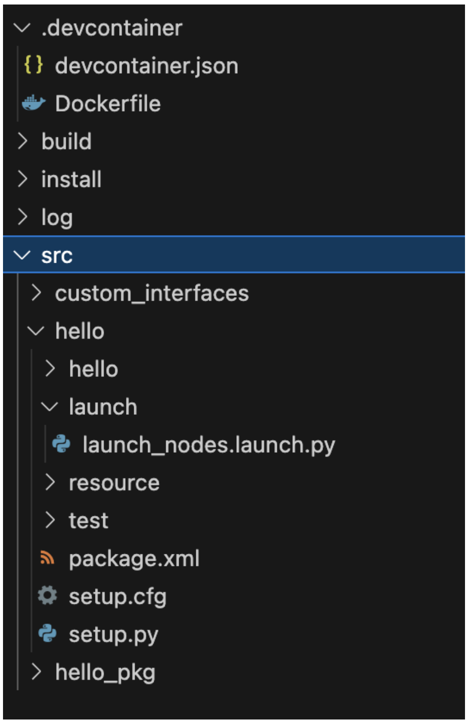
When your ROS2 powered robotic program gets complex, most of the time you would end up running a number of ros2 nodes. Running them individually in separate terminals becomes cumbersome. Therefore, ROS provides a 'launch' mechanism to run multiple nodes simultaneously with their respective parameters.

It is generally agreed upon that you would create these launch files for running nodes belonging to a package. (However, there is nothing preventing you from writing a launch file to start nodes belonging to multiple packages.)

Create a sub-folder called 'launch' inside your package. Create a Python file inside the launch folder with the following contents. It is a good practice to name the launch files with the '.launch.py' extension to tag them as launch files.

```
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='hello',
            executable='publisher_node',
            output="screen"
        ),
        Node(
            package='hello',
            executable='subscriber_node',
            output="screen"
        )
    ])
```

Here is what my workspace looks like:



Next, amend the data_files section of the setup.py file to inform the colcon build about the newly added launch folder and its files.

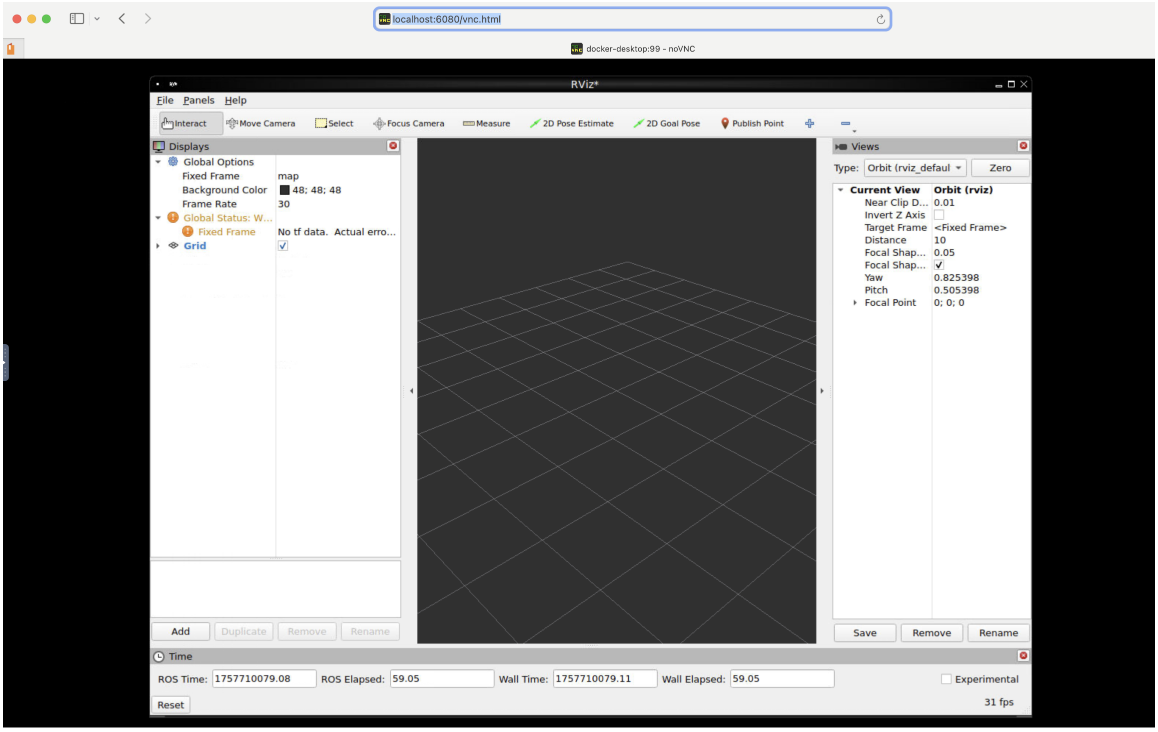
```
from setuptools import find_packages, setup
import os
from glob import glob

package_name = 'hello'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob('launch/*.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='root',
    maintainer_email='root@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'hello_class_node = hello.hello_class:main',
            'publisher_node = hello.publisher_node:main',
            'subscriber_node = hello.subscriber_node:main',
            'server_node = hello.server_node:main',
            'client_node = hello.client_node:main'
        ],
    },
)
```

Next, build your workspace with

```
colcon build
```

and launch the nodes with the command:

```
ros2 launch your_package_name your_launch_file_name.launch.py
```



4 RViz

Use RViz in ROS ecosystem to visualize your robot model in 3D.

4.1 Set up inside a DevContainer

Running RViz does not require prior setup if you have ROS2 installed on your machine. However, when executing RViz from a VSCode Dev Container or a Dokcer / Podman container, we need to execute the following commands to make sure the RViz has a screen to output its GUI.

Make sure your Dockerfile installs the following programs.

```
xvfb fluxbox x11vnc novnc net-tools
```

Next, launch virtual framebuffer X server on display :99 via Xvfb.

```
Xvfb :99 -ac -screen 0 1920x1080x24 &
```

Exports environment variable to point applications (like RViz) to virtual display :99.

```
export DISPLAY=:99
```

Fluxbox running in background manages windows in the virtual display for better GUI handling.

```
fluxbox &
```

The following command connects VNC server to display :99.

```
x11vnc -display :99 -nopw -forever -shared -rfbport 5900 &
```

Next command proxies VNC to web interface on port 6080 for browser access.

```
/usr/share/novnc/utils/novnc_proxy --vnc localhost:5900 --listen 6080 &
```

Run `http://localhost:6080/vnc.html` in your browser to access the vnc server.

4.2 Launch Rviz

In your terminal, run *rviz*2 or

```
ros2 run rviz2 rviz2
```

to launch the RViz program.

4.3 Display objects in RViz

This section is tricky and needs to keep track of many items. You would need the following items. First, create a new package called 'rviz_models' in your workspace. Use $--build-typeament_cmake$ for your package.

- A .xacro file of your robot model. This should go under a new folder inside your package. A new folder is for better code organization. The example code below creates a simple box and attaches it to the simulation 'world'. Name it box.xacro

```
<?xml version="1.0"?>
<robot name="box_robot">
  <link name="world"/>
  <link name="base_link">
    <visual>
      <geometry>
        <box size="1 1 1"/>
      </geometry>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <material name="blue">
        <color rgba="0 0 1 1"/>
      </material>
    </visual>
  </link>
  <joint name="world_to_base" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </joint>
</robot>
```

- A custom configuration file for RViz to properly setup its environment for your liking. Again, create a new folder called 'rviz' inside your package, and create the following config.rviz file.

```
%YAML 1.1
---
Panels:
  - Class: rviz_common/Displays
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /MotionPlanning1
  - Class: rviz_common/Help
    Name: Help
  - Class: rviz_common/Views
    Name: Views
Visualization Manager:
  Class: ""
  Displays:
    - Class: rviz_default_plugins/Grid
      Alpha: 0.5
```

```
          Cell Size: 1
          Color: 160; 160; 164
          Enabled: true
          Line Style: Lines
          Name: Grid
          Normal Cell Count: 0
          Offset: 0; 0; 0
          Plane: XY
          Plane Cell Count: 10
          Value: true
        - Class: rviz_default_plugins/RobotModel
          Alpha: 1.0
          Description Source: Topic
          Description Topic: /robot_description
          Enabled: true
          Name: RobotModel
          TF Prefix: ""
          Value: true
          Visual Enabled: true
        - Class: rviz_default_plugins/TF
          Enabled: true
          Name: TF
          Show Axes: true
          Show Names: true
          Value: true
      Enabled: true
      Global Options:
        Background Color: 48; 48; 48
        Fixed Frame: world
        Frame Rate: 30
      Name: root
      Tools:
        - Class: rviz_default_plugins/Interact
          Hide Inactive Objects: true
        - Class: rviz_default_plugins/MoveCamera
        - Class: rviz_default_plugins/Select
      Value: true
      Views:
        Current:
          Class: rviz_default_plugins/Orbit
          Distance: 2.0
          Focal Point:
            X: -0.1
            Y: 0.25
            Z: 0.30
          Name: Current View
          Pitch: 0.5
          Target Frame: world
          Yaw: -0.623
        Saved: ~
    Window Geometry:
      Height: 975
      Width: 1200
```

- Next, create a launch folder and a launch.py file inside your package to launch two nodes. One node will be the rviz node that gets your config.rviz as a parameter. The other node is the 'robot_state_publisher' that displays your robot. It takes the model of the robot (in our case box.xacro) as a parameter. Name the file box.launch.py

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.substitutions import PathJoinSubstitution, Command, FindExecutable
from launch_ros.substitutions import FindPackageShare
from launch_ros.parameter_descriptions import ParameterValue


def generate_launch_description():
    # Assuming ROS 2 style for Python launch

    robot_description_content = Command(
        [
            PathJoinSubstitution([FindExecutable(name="xacro")]),
            " ",
            PathJoinSubstitution([FindPackageShare('rviz_models'), "urdf", 'box.xacro'])
        ]
    )

    robot_description = ParameterValue(robot_description_content, value_type=str)

    return LaunchDescription([
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'robot_description': robot_description}]
        ),
        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            output='screen',
            arguments=['-d', PathJoinSubstitution([FindPackageShare('rviz_models'), 'rviz', 'config.rviz'])]
        )
    ])
```

- Let's modify the package.xml file to install dependencies for running our box model.

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>rviz_models</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="root@todo.todo">root</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

```

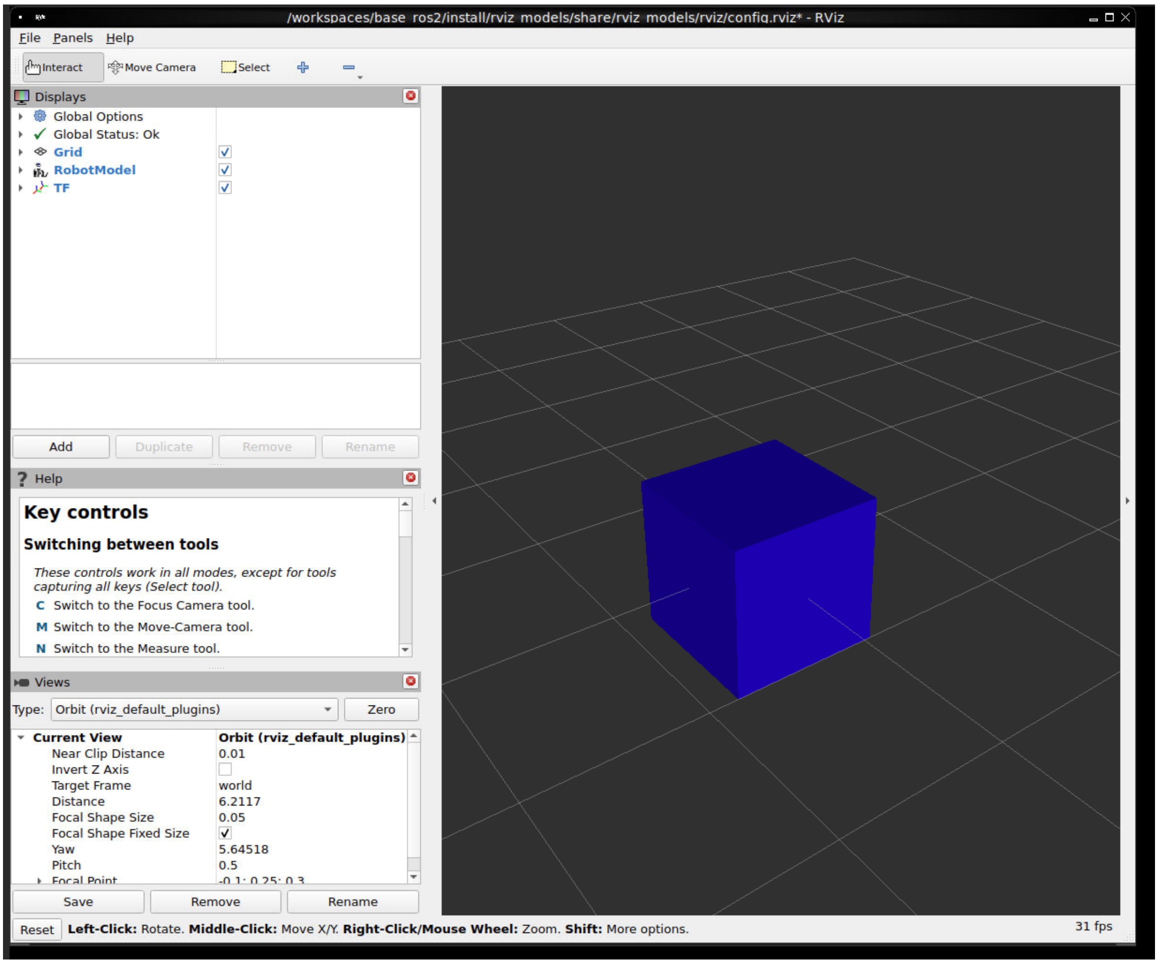
```
  <depend>ros2launch</depend>
  <depend>robot_state_publisher</depend>
  <depend>rviz2</depend>
  <depend>xacro</depend>

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

- Before building your workspace, lets connect all the new files we made via the CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(rviz_models)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang"
   )
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

install(DIRECTORY rviz/
  DESTINATION share/${PROJECT_NAME}/rviz
)

install(DIRECTORY urdf/
  DESTINATION share/${PROJECT_NAME}/urdf
)

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

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all
     source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

- build your new package / workspace
- Instead of running *rviz2*, run the launch the file we created.

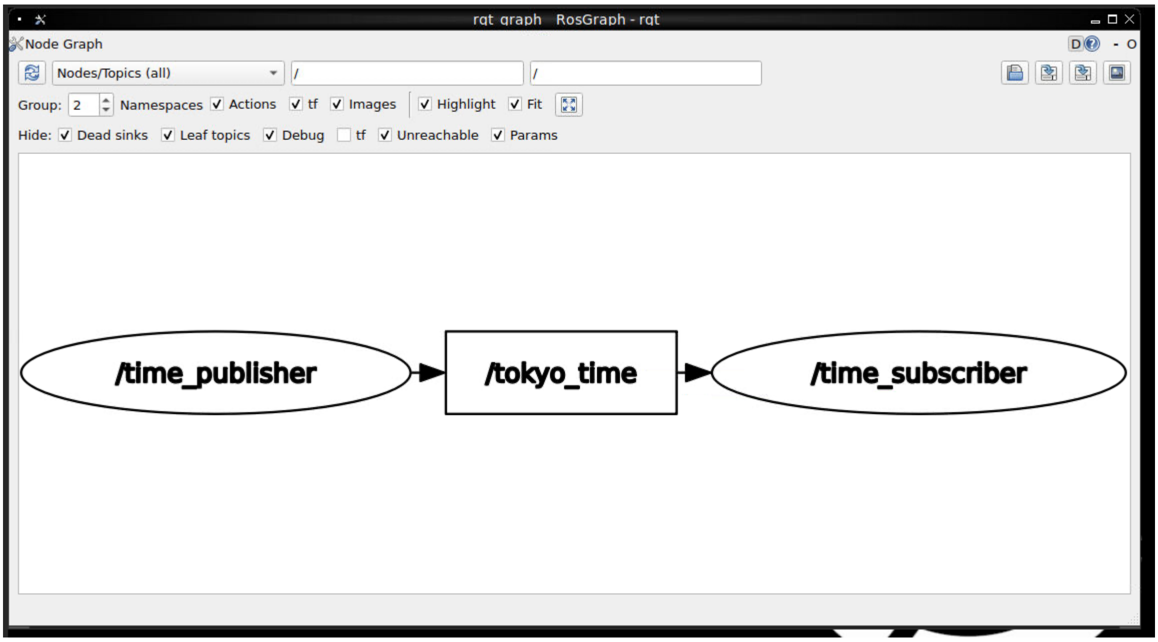
```
ros2 launch rviz_models box.launch.py
```



5 Analyze your workspace

When RUNNING multiple ROS nodes and topics and services, you may lose track of which node communicates with which node. Hence, ROS provides you a program to visualize your whole ROS2 environment.

```
ros2 run rqt_graph rqt_graph
```


rqt_graph__RosGraph - rqt
Node Graph
Nodes/Topics (all)
/
/
Group: 2 Namespaces Actions tf Images Highlight Fit
Hide: Dead sinks Leaf topics Debug tf Unreachable Params
/time_publisher
/tokyo_time
/time_subscriber