ROS: Robot Operating System

What is ROS?

History

2007

Stanford STAIR Program

PR2 robot

Morgan Quigley

Box Turtle

2010 ROS 1.0

2012 ROSCon

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)

2014 to 2020, 4 long term support (LTS) distributions

Willow Garage

2008

Willow Garage

2011

TurtleBot

2013 OSRF

ROS

Open Source Robotics Foundation

2017 ROS 2 Ardent

2

We will use ROS2 Jazzy in Ubuntu 24.

Distro	Release date	Logo	EOL date
Kilted Kaiju	May 23, 2025		December 2026
Jazzy Jalisco	May 23, 2024		May 2029
Iron Irwini	May 23, 2023		December 4, 2024
Humble Hawksbill	May 23, 2022		May 2027
Galactic Geochelone	May 23, 2021		December 9, 2022

ROS1 VS ROS2

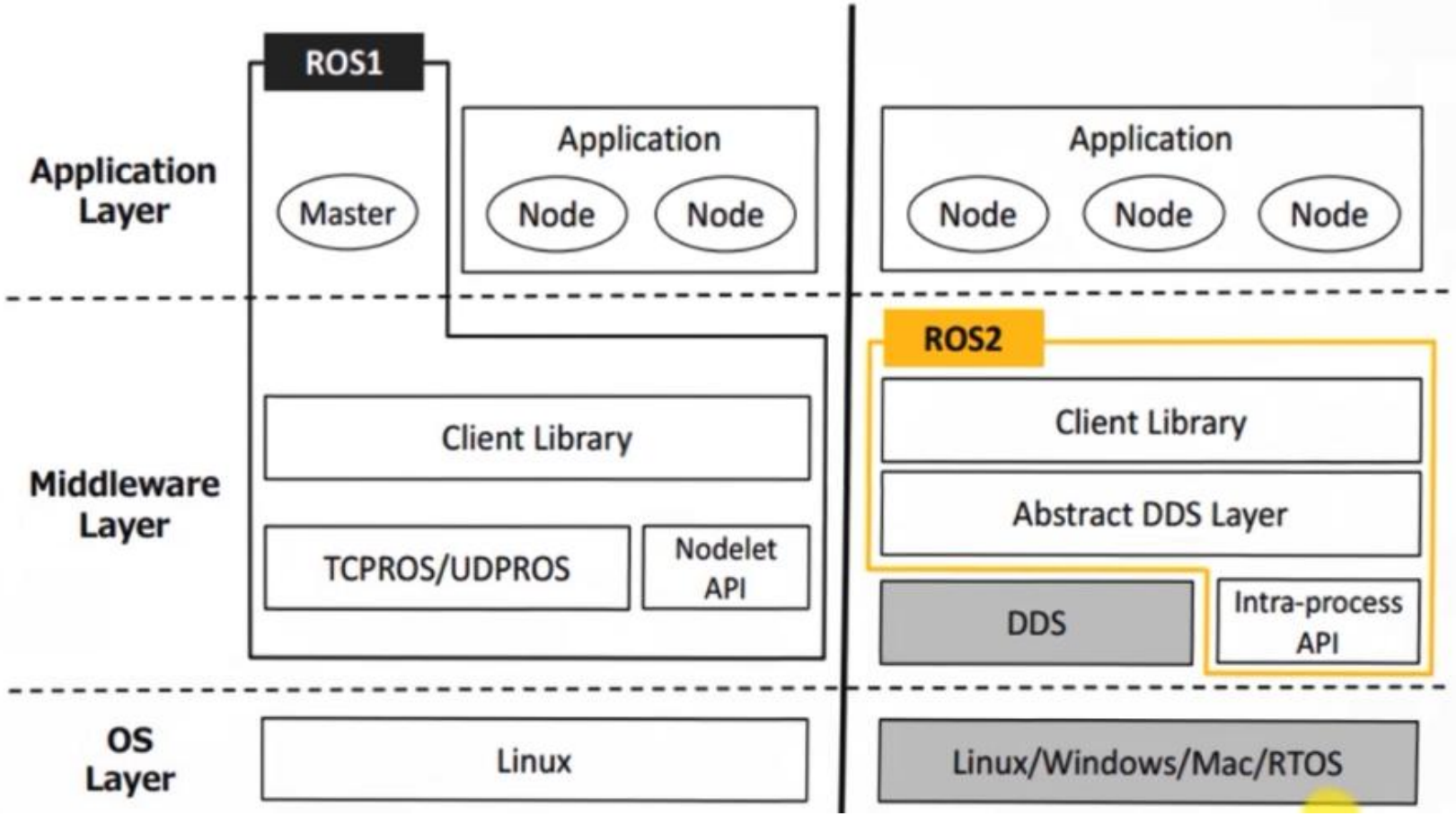
- Why we need ROS2?
  - PR2 (ROS1) is designed for independently working on daily tasks, it is mounted with workstation-level computing power, no communication, ideal environments in the lab.
- What does ROS2 have?
  - Multi-robot standard communication mechanism
  - Cross platform, Windows, Linux, and so on.
  - Data transition security
  - Commercialization

ROS2 is not just an extension of ROS 1

- Framework
  - ROS 1 needs a master node to connect with all nodes (processes), ROS2 is built upon a novel communication mechanism, without a master node and can be run in decentralized.
- API
  - ROS1 API is built upon the code implemented in 2009. ROS 2 re-design the APIs, but the user interfaces are similar.
- Compiler:
  - ROS 1 uses rosbuild or catkin to build the packages and ROS 2 uses ament or colcon

ROS2 is not just an extension of ROS 1

- OS
  - ROS 1: Linux
  - ROS2: Linux, Windows, MAC, RTOS
- Communication:
  - ROS 1: TCP/UPD
  - ROS2: DDS
- Node:
  - ROS 1: publish/subscribe
  - ROS2: discovery
- Process:
  - ROS 1: Nodelet
  - ROS2: Intra-process

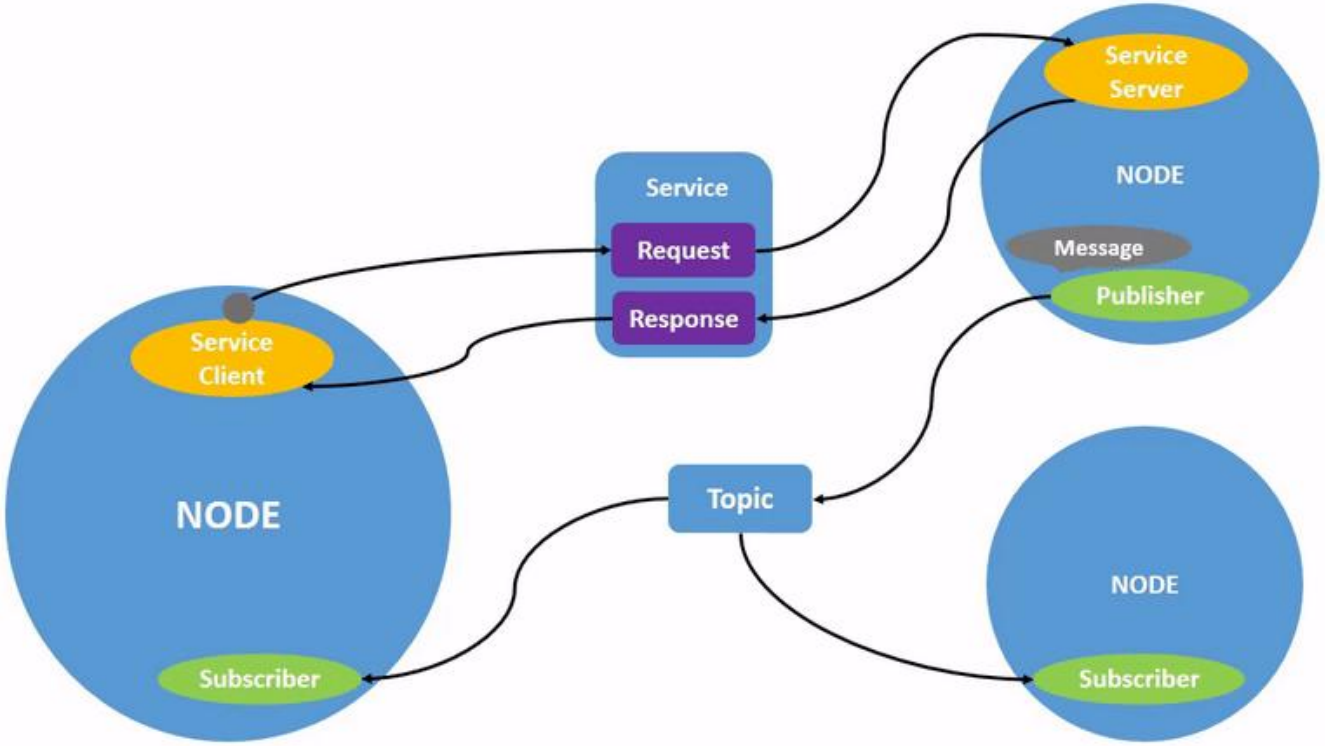


ROS: Robot Operating System

ROS2 main concepts

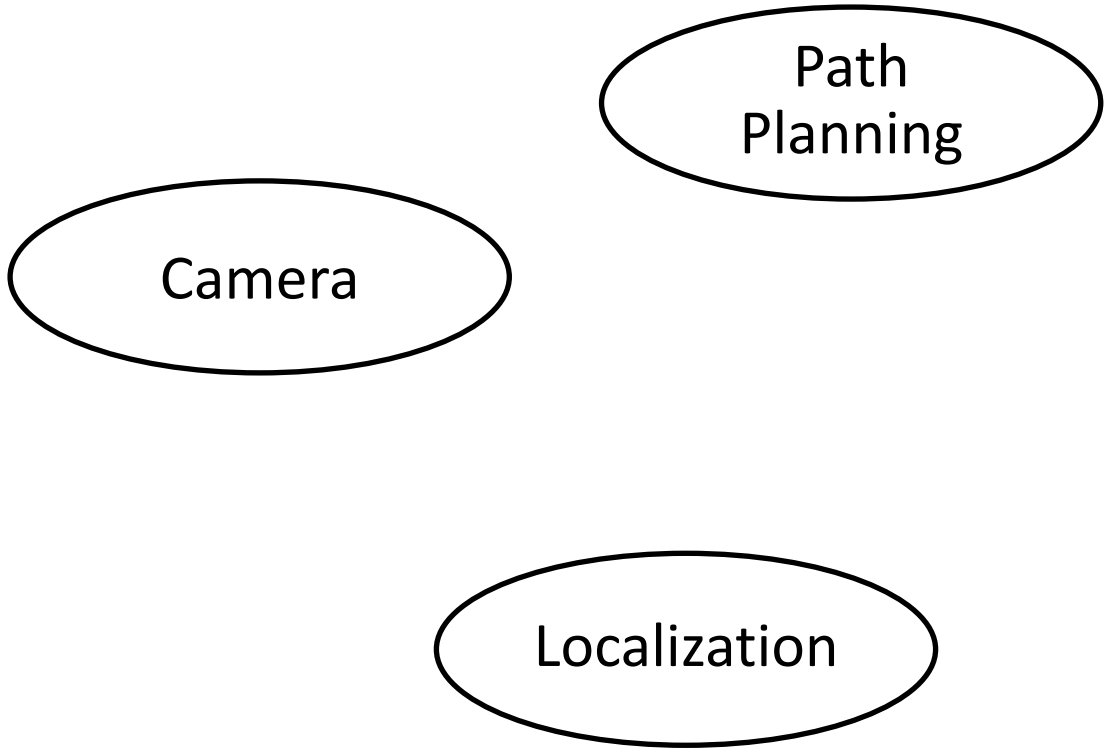
ROS2: Main Concepts

- Nodes
- Topics
- Services
- Actions
- Parameters



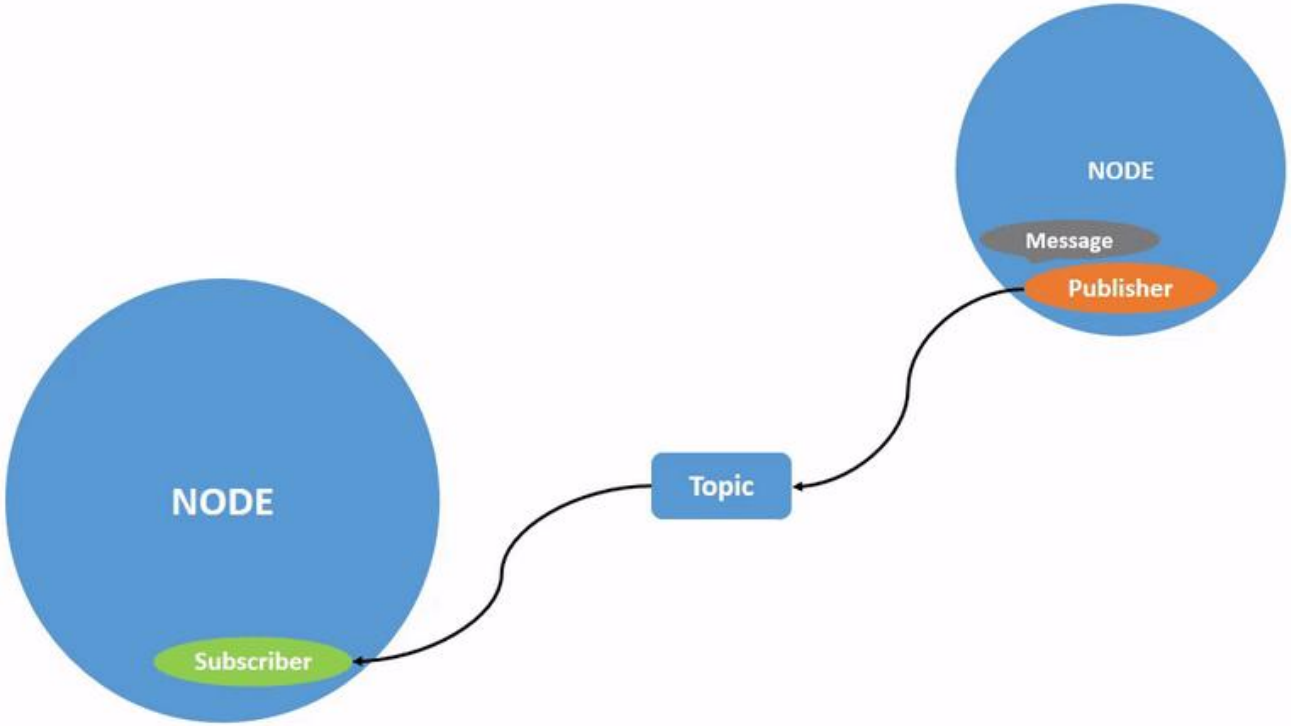
ROS: Main Concepts

- **Nodes**
  - Single-purposed executable programs that perform computation.
    - E.g., sensor driver(s), actuator driver(s), mapper, planner, UI, etc.
  - Each node has a unique name.
  - Different nodes can be written in different languages and run in different OS.



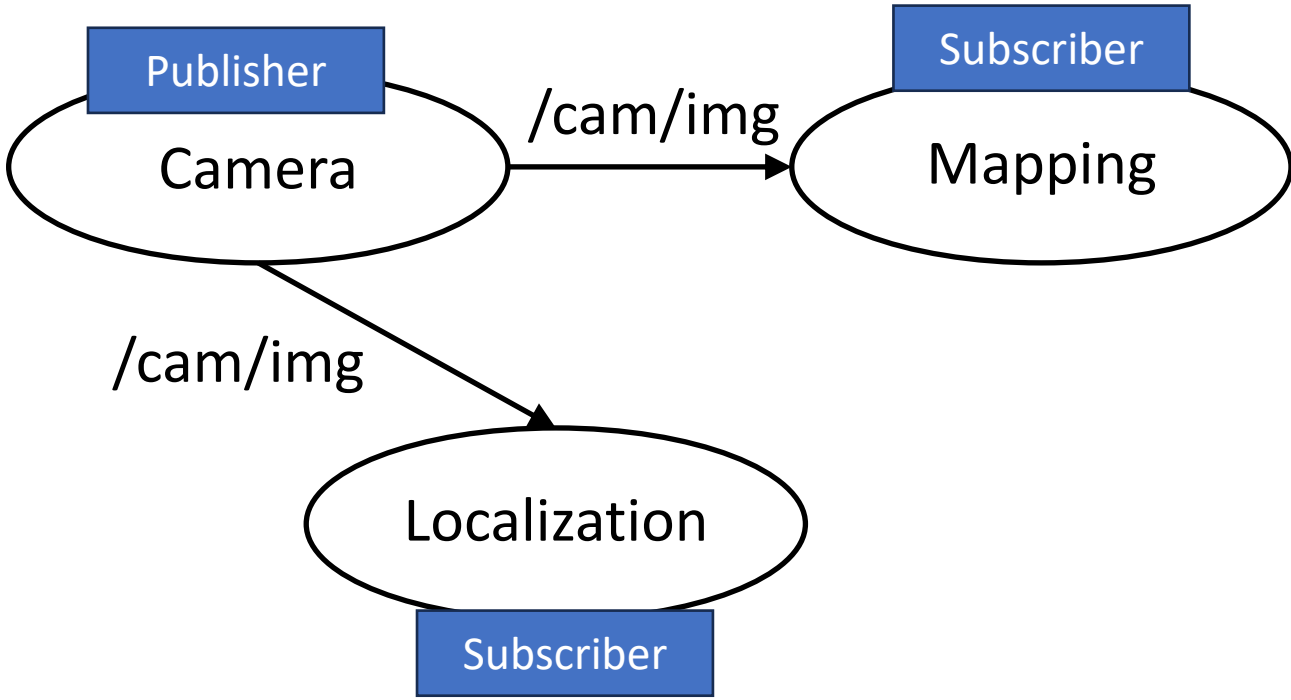
ROS: Main Concepts

- **Topics**
  - Data transfer bus between nodes (Publisher & Subscriber)
  - Always transfer data from a Publisher to a Subscriber



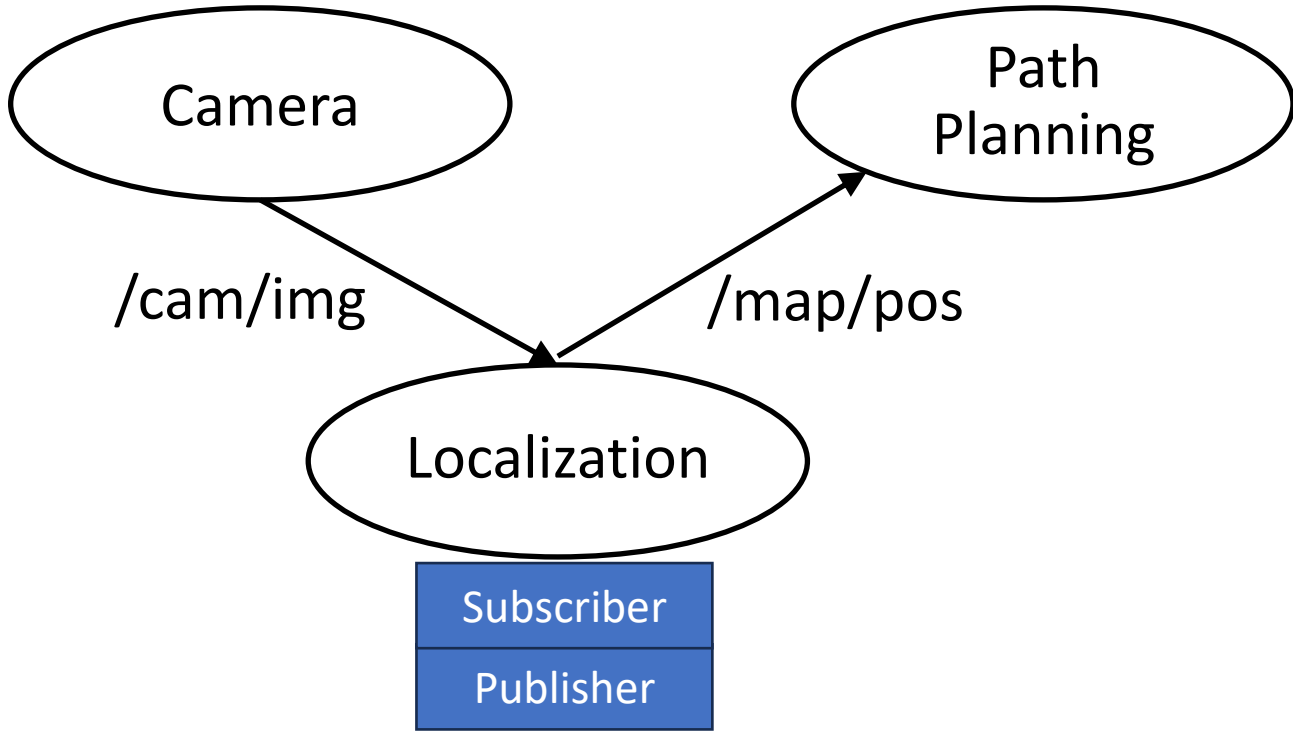
ROS: Main Concepts

- **Topics**
  - Data transfer bus between nodes (Publisher & Subscriber)
  - Always transfer data from a Publisher to a Subscriber
  - The same topic may have different subscribers and publishers.



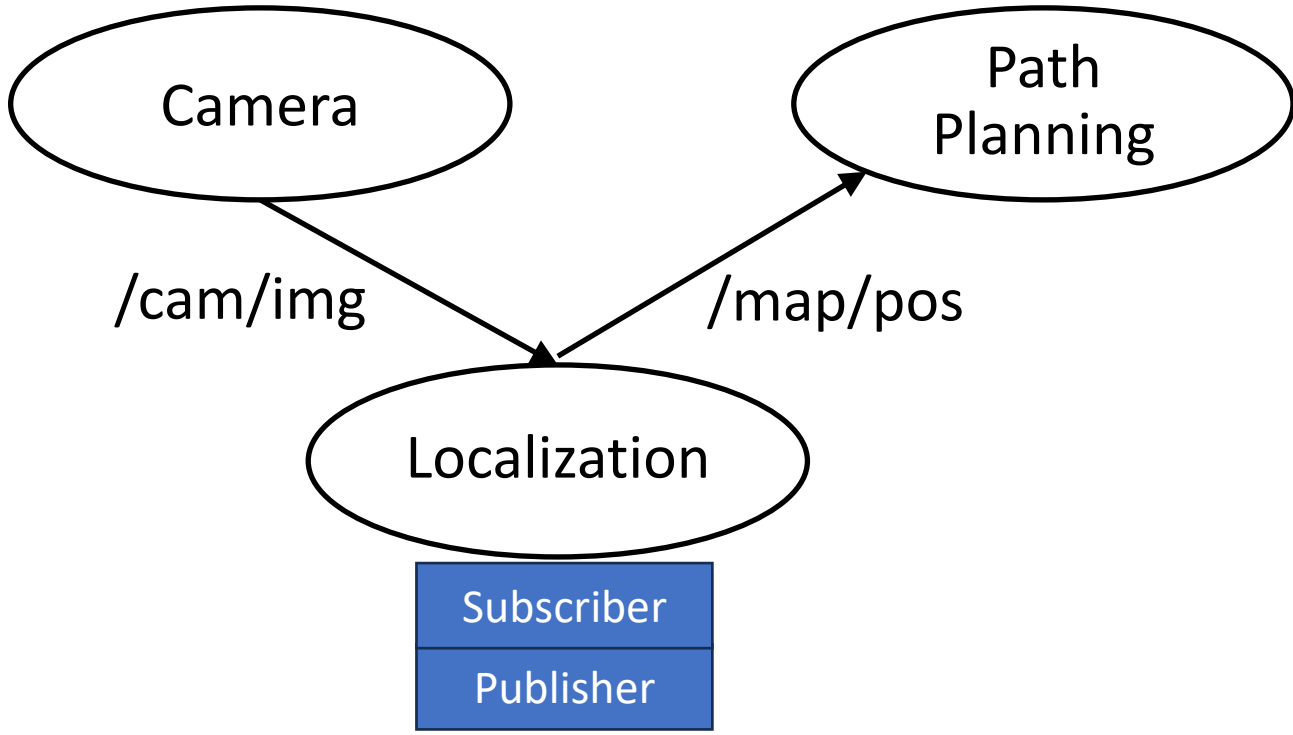
ROS: Main Concepts

- **Topics**
  - Data transfer bus between nodes (Publisher & Subscriber)
  - Always transfer data from a Publisher to a Subscriber
  - The same topic may have different subscribers and publishers.
  - A node can be a publisher and subscriber at the same time



ROS: Main Concepts

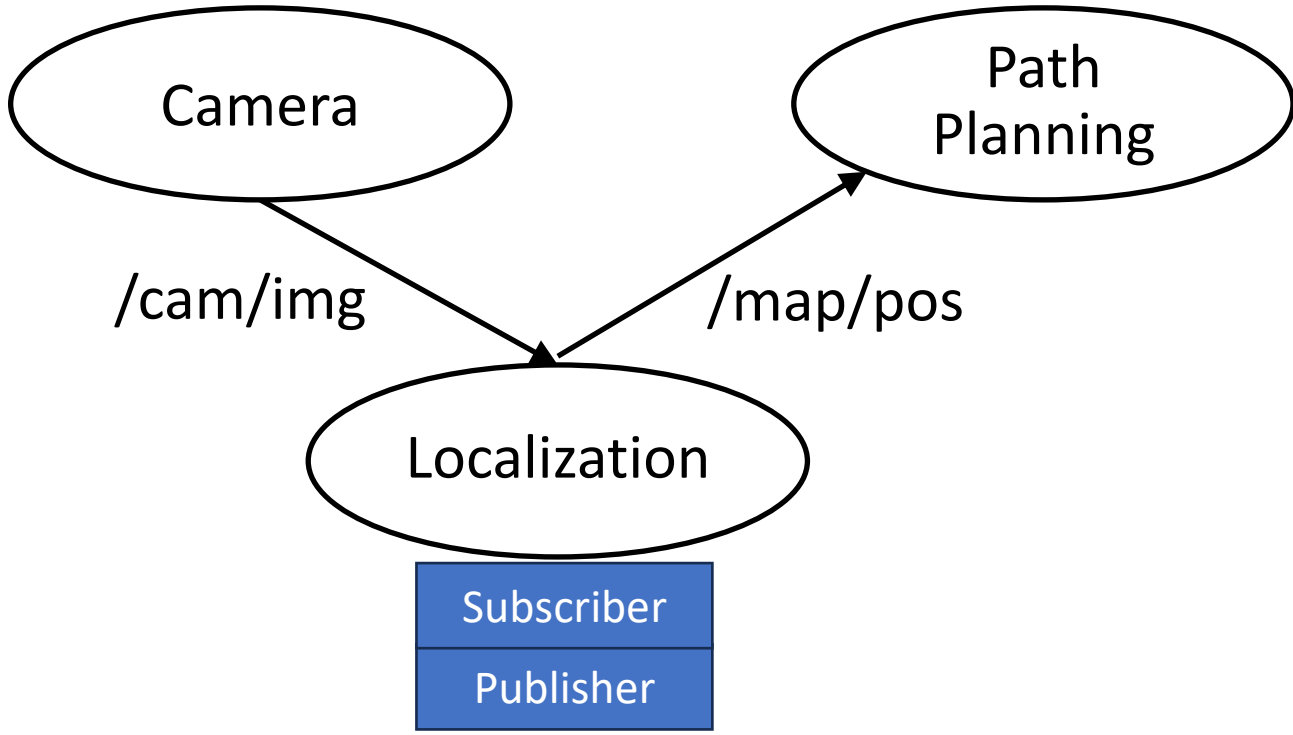
- **Topics**
  - Data transfer bus between nodes (Publisher & Subscriber)
  - Always transfer data from a Publisher to a Subscriber
  - The same topic may have different subscribers and publishers.
  - A node can be a publisher and subscriber at the same time



Can a node have multiple publishers and subscribers?

ROS: Main Concepts

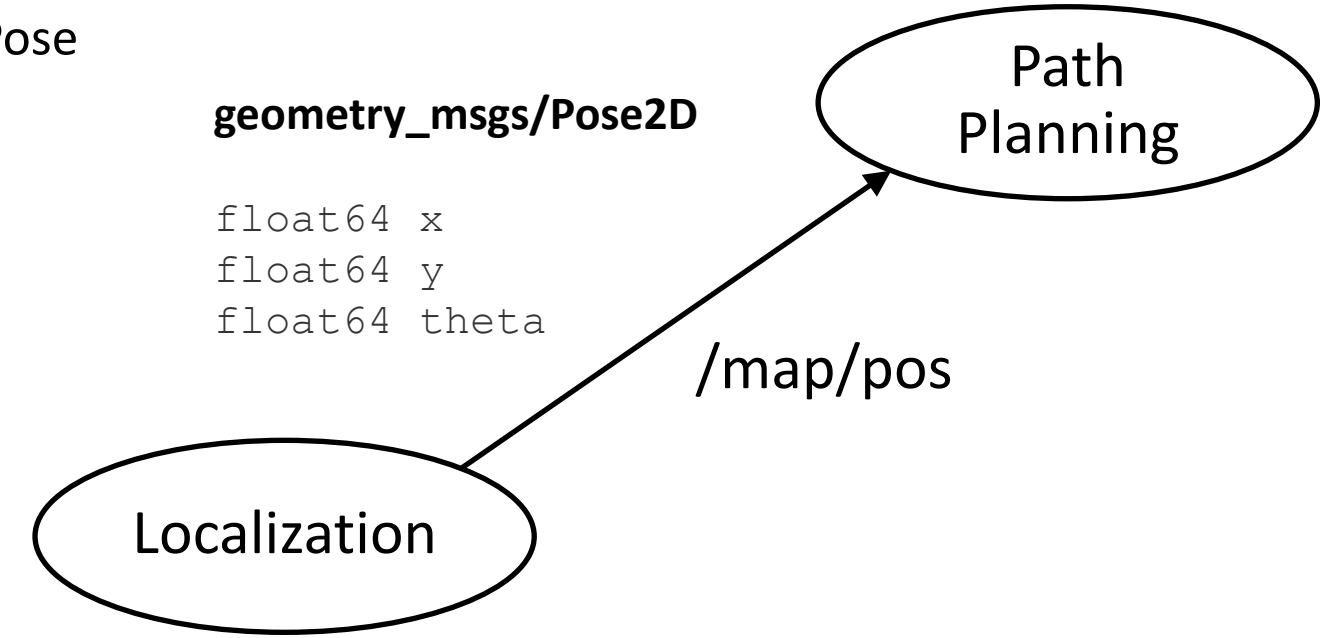
- **Topics**
  - Data transfer bus between nodes (Publisher & Subscriber)
  - Always transfer data from a Publisher to a Subscriber
  - The same topic may have different subscribers and publishers.
  - A node can be a publisher and subscriber at the same time



If there are nodes that are not neither publisher nor subscriber?

ROS: Main Concepts

- **Messages**
  - Strictly-typed data structures for node-node communication, e.g., Standard and user-defined in .msg file.
  - Language agnostic data structures, so that Python nodes can talk to C++ nodes.
  - Examples:
    - Pose2D, Pose

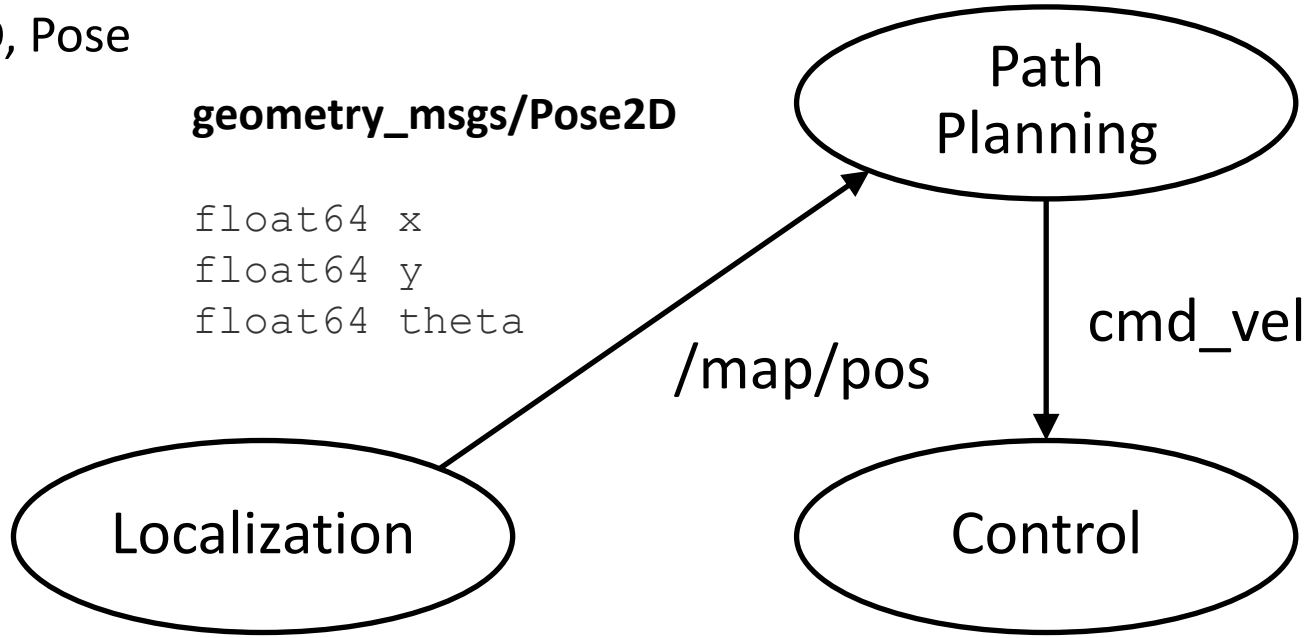


ROS: Main Concepts

- **Messages**
  - Strictly-typed data structures for node-node communication, e.g., Standard and user-defined in .msg file.
  - Language agnostic data structures, so that Python nodes can talk to C++ nodes.
  - Examples:
    - Pose2D, Pose
    - Twist

geometry_msgs/Pose2D

```
float64 x
float64 y
float64 theta
```

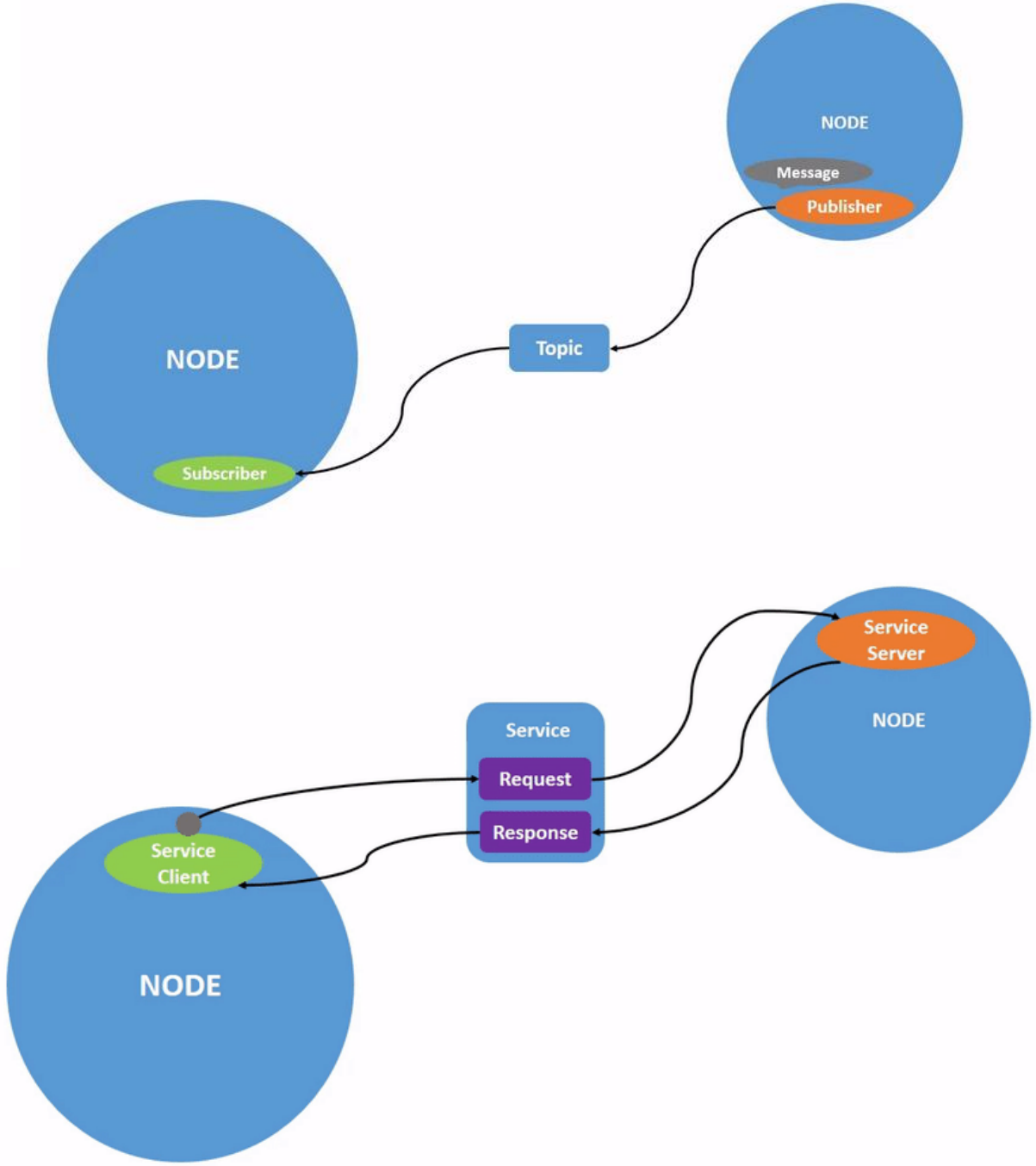


geometry_msgs/Twist

```
Vector3 linear
    float64 x
    float64 y
    float64 z
Vector3 angular
    float64 x
    float64 y
    float64 z
```

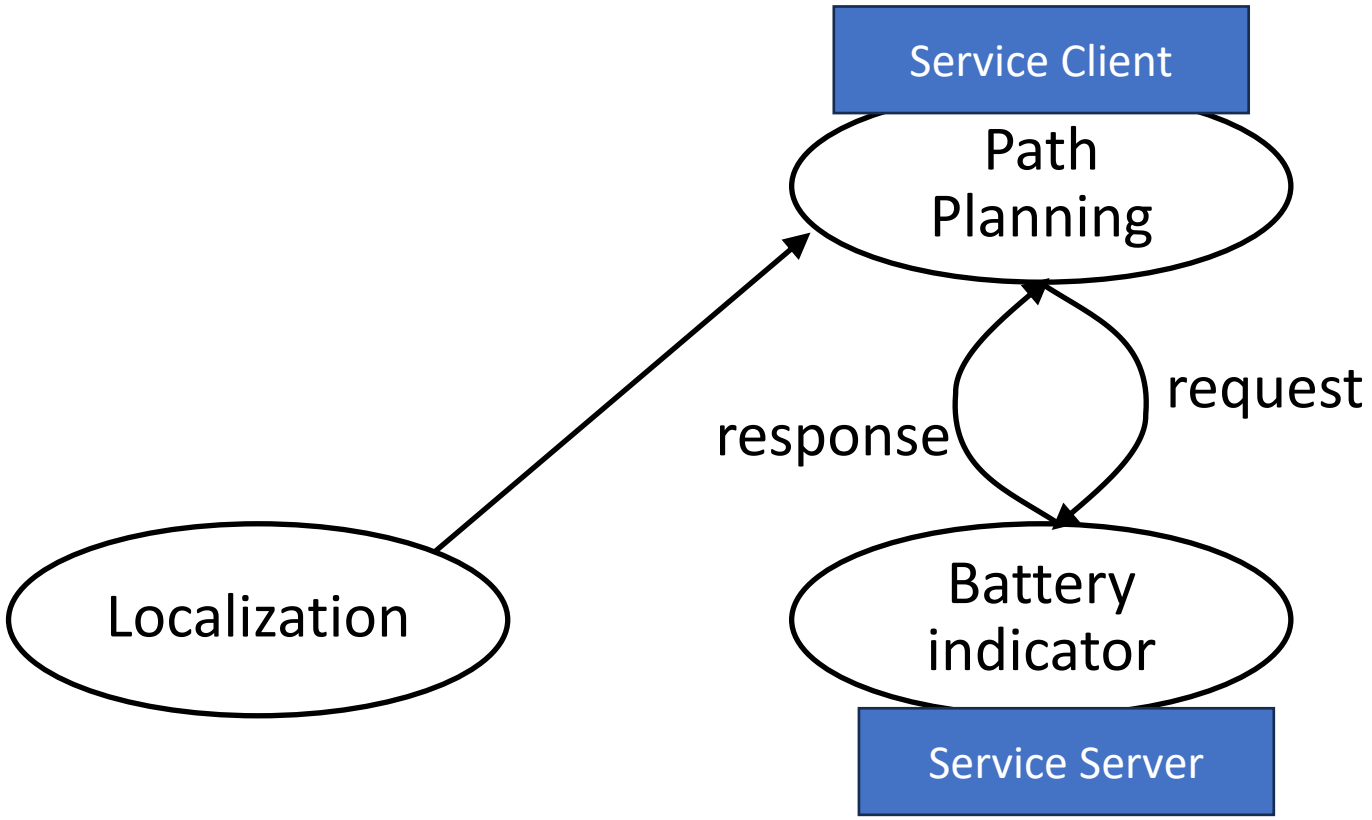
ROS: Main Concepts

- **Services**
  - Inter-node transactions by request and response.
  - Service roles:
    - Trigger functionality/behavior.
    - Perform remote or asynchronized computation/storage.



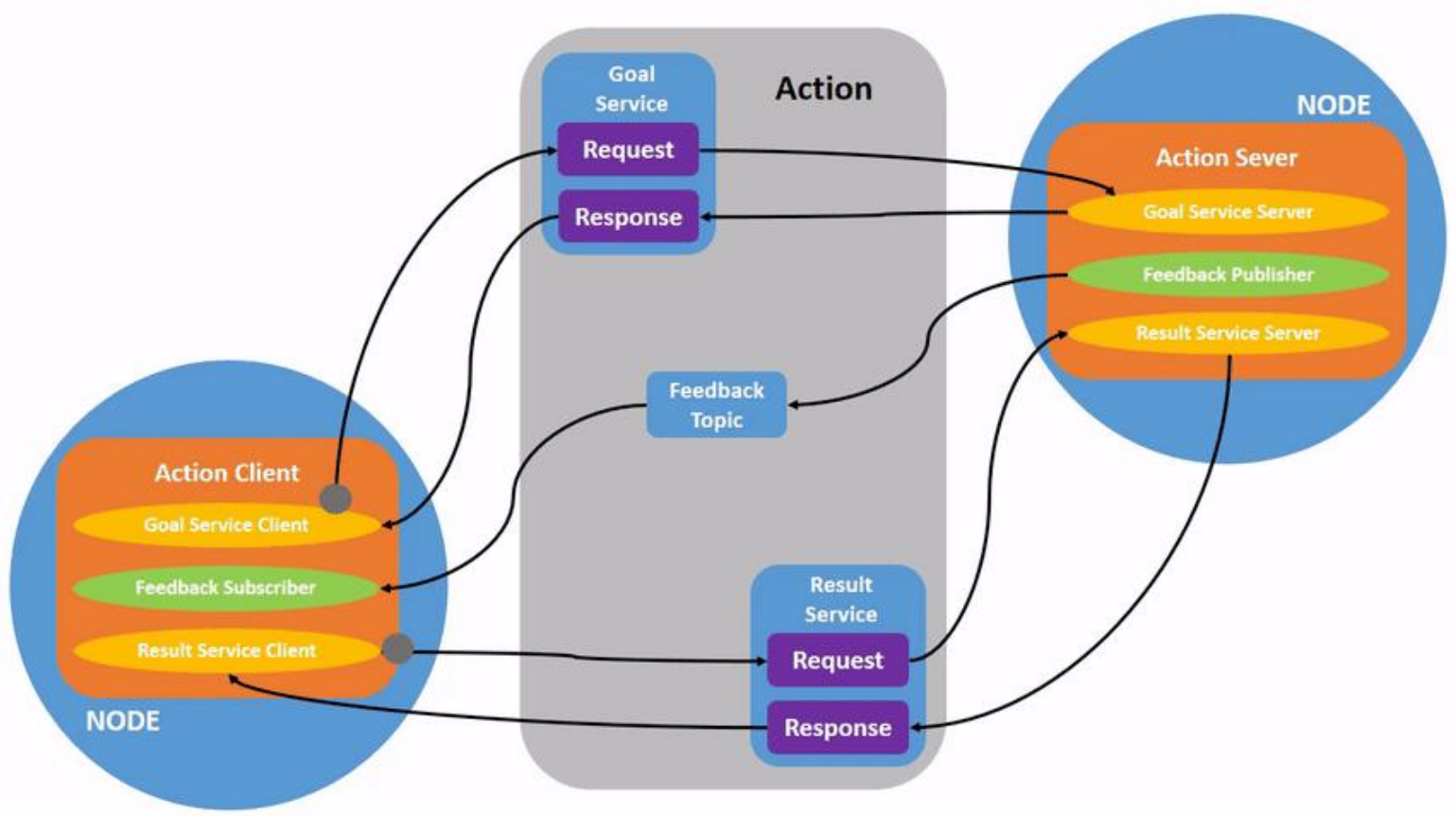
ROS: Main Concepts

- **Services**
  - Inter-node transactions by request and response.
  - Service roles:
    - Trigger functionality/behavior.
    - Perform remote or asynchronized computation/storage.



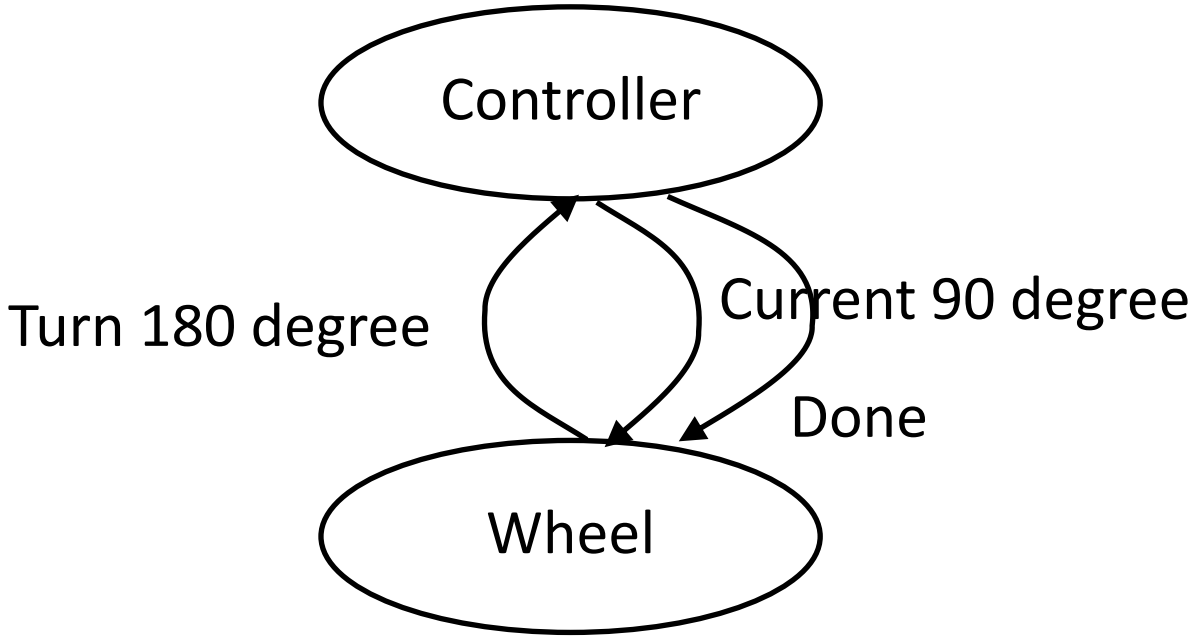
ROS: Main Concepts

- **Action**
  - 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.



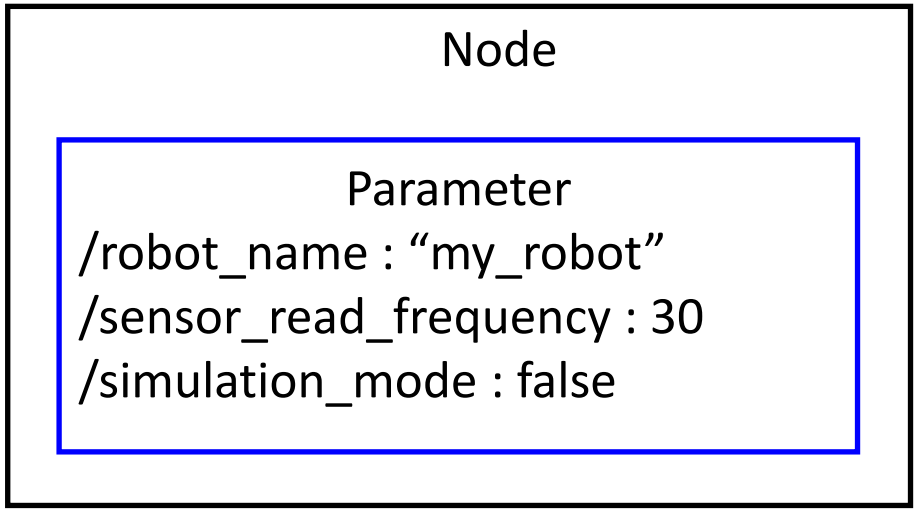
ROS: Main Concepts

- **Action**
  - 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.



ROS: Main Concepts

- **Parameter**
  - 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.



ROS: Robot Operating System

ROS2 common commands

Run a package

- **ROS Environment**
  - ROS uses the shell environment.
  - After you install ROS 2, you will have to setup.*sh files in '/opt/ros/<distro>/', and you could source them like so:

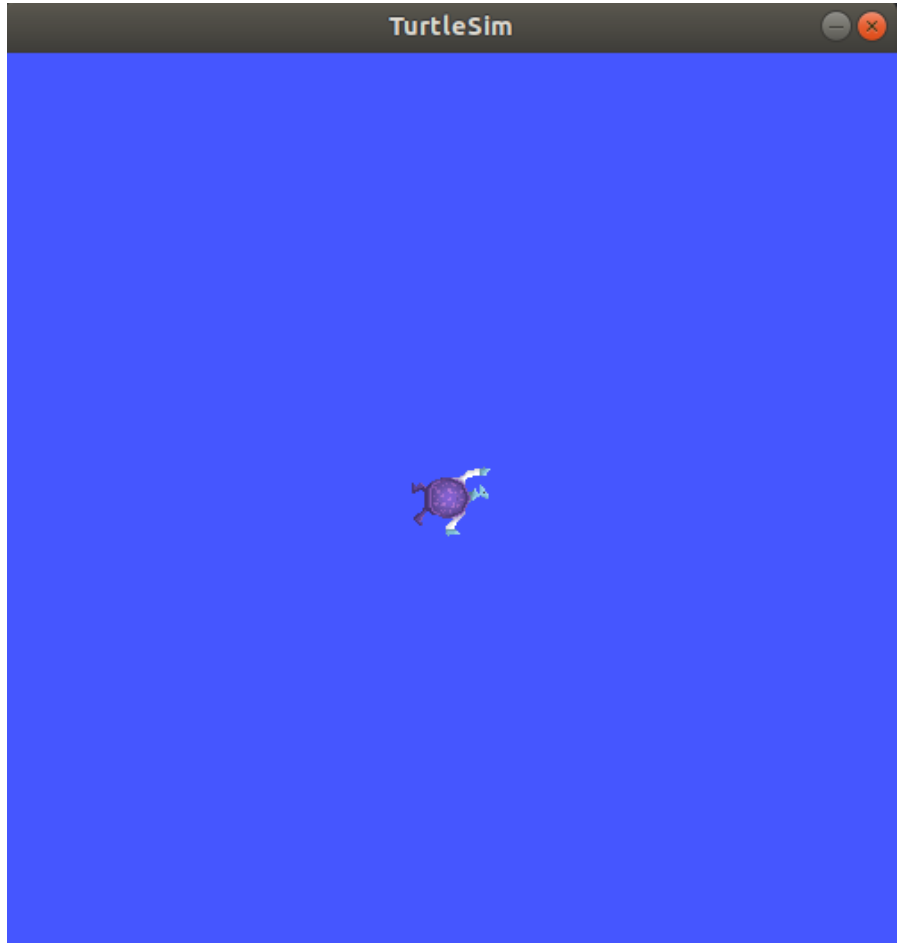
```
$ source /opt/ros/jazzy/setup.bash
```

  - You will need to run this command on every new shell you open to have access to the ROS commands, unless you add this line to your bash startup file (~/.bashrc).
  - This makes developing against different versions of ROS or against different sets of packages easier.

Run a package

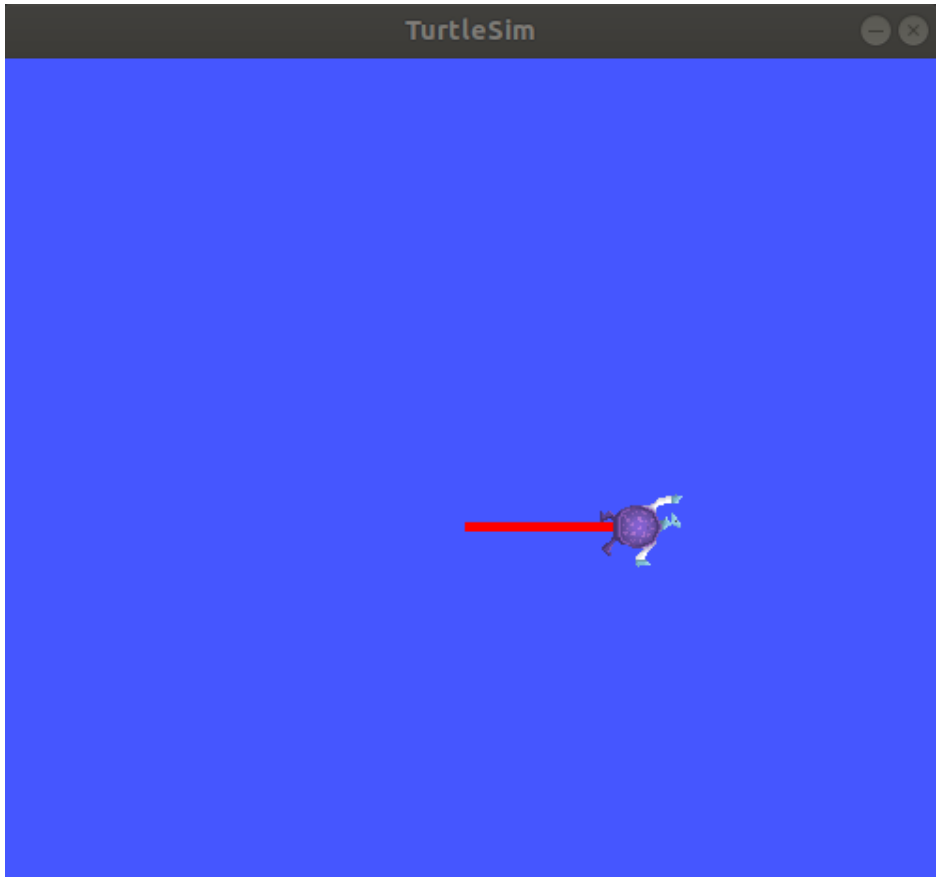
- To run a package in ROS2
  - `ros2 run <package_name> <executable_name>`

- To run turtlesim, open a new terminal, and enter the following command:
  - `ros2 run turtlesim turtlesim_node`



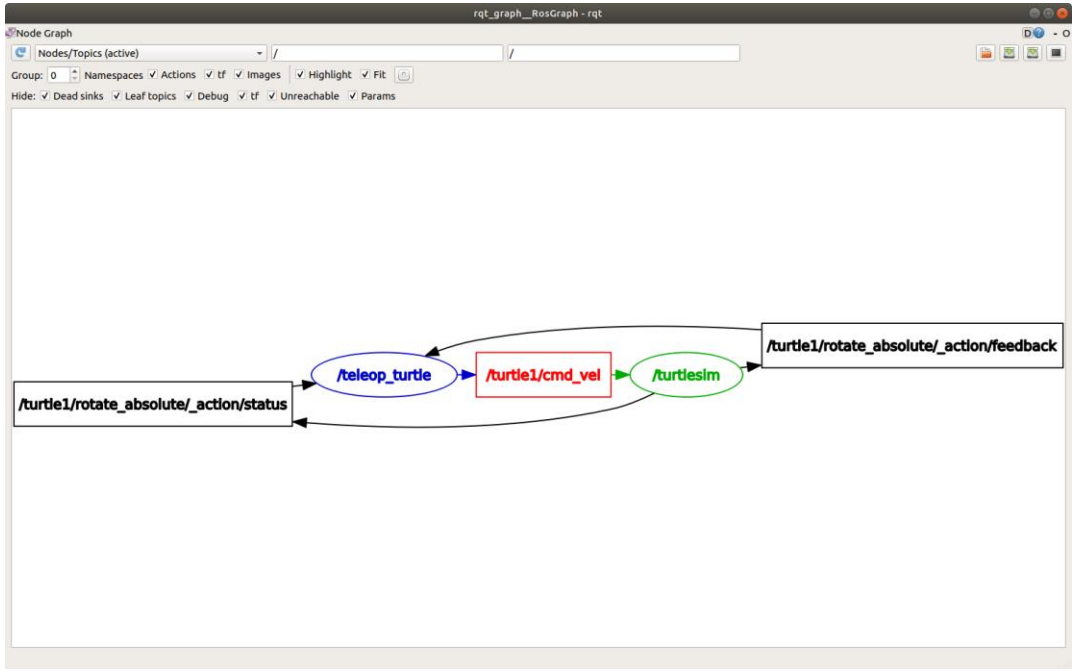
Run a package

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`



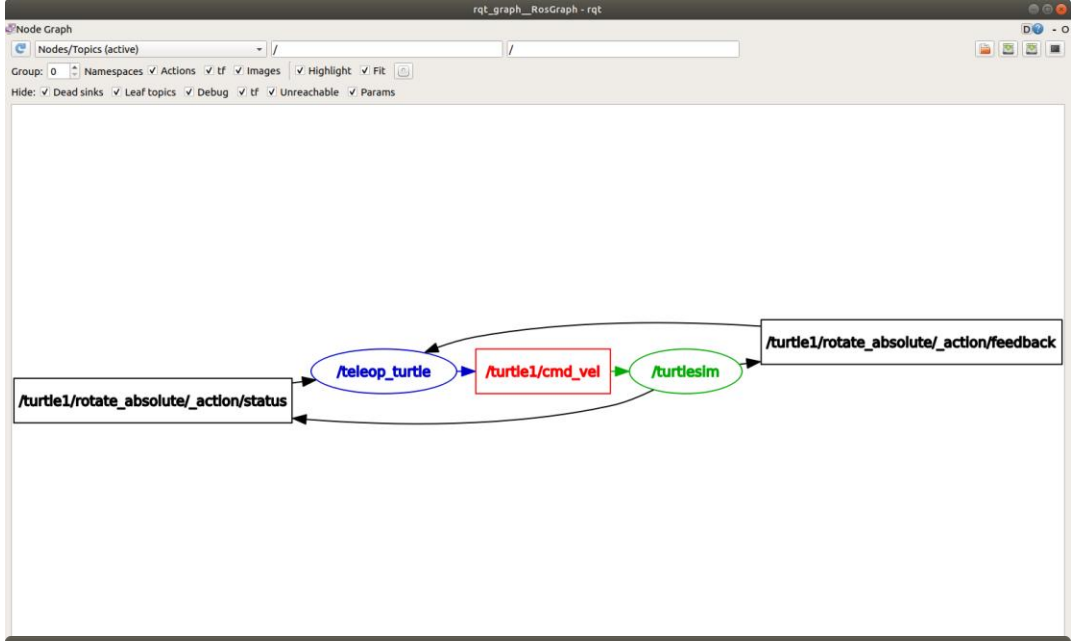
Run a package

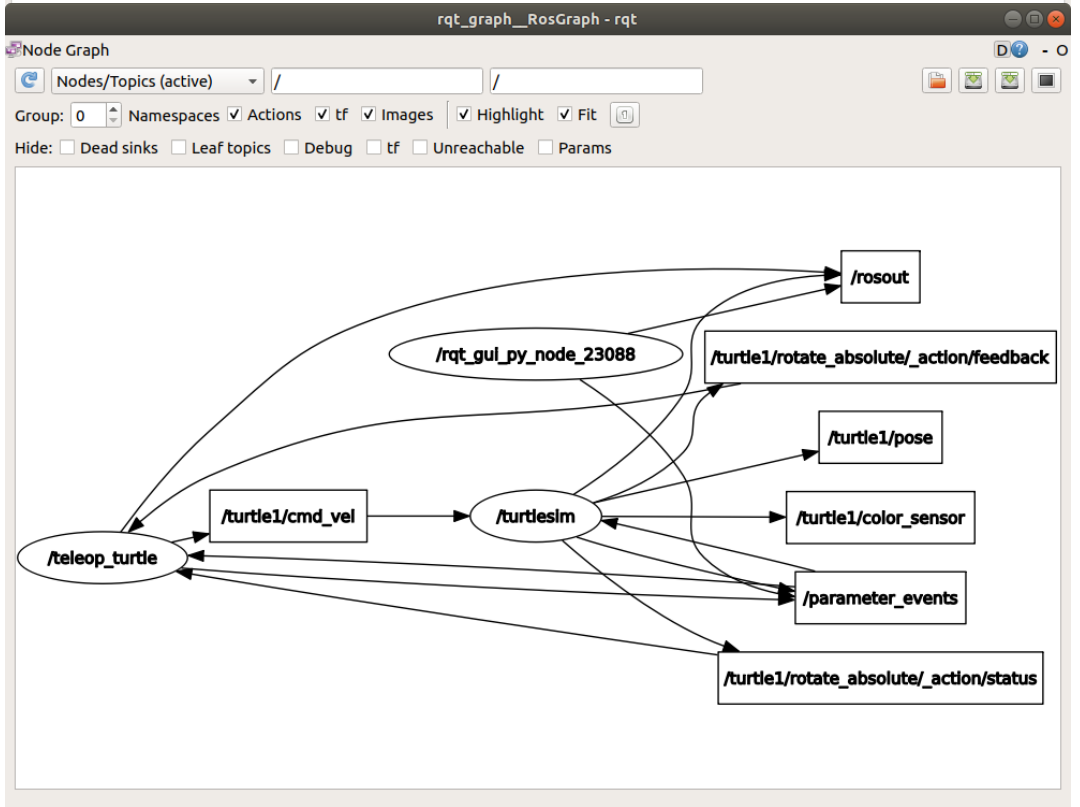
- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 run rqt_graph rqt_graph`



Run a package

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 run rqt_graph rqt_graph`





Topic

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 run rqt_graph rqt_graph`
- `ros2 topic list`

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

Topic

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 run rqt_graph rqt_graph`
- `ros2 topic list`
- `ros2 topic info /turtle1/cmd_vel`

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

Topic

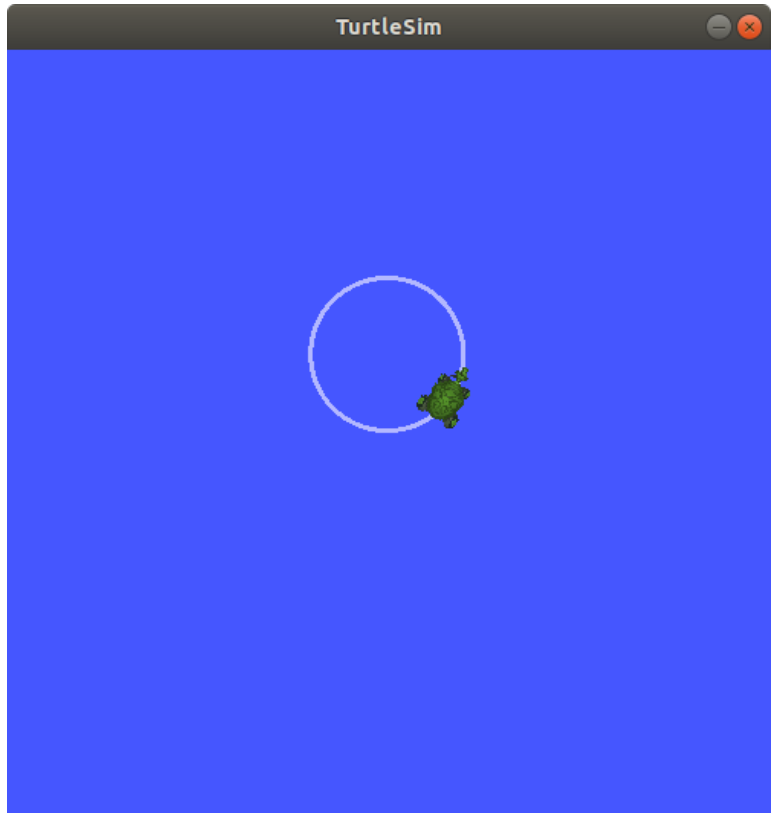
- source /opt/ros/jazzy/setup.bash
- ros2 run turtlesim turtlesim_node
- ros2 run turtlesim turtle_teleop_key
- ros2 run rqt_graph rqt_graph
- ros2 topic list
- ros2 topic info /turtle1/cmd_vel
- ros2 interface show geometry_msgs/msg/Twist

```
# This expresses velocity
    Vector3  linear
            float64 x
            float64 y
            float64 z
    Vector3  angular
            float64 x
            float64 y
            float64 z
```

Topic

- source /opt/ros/jazzy/setup.bash
- ros2 run turtlesim turtlesim_node
- ros2 run turtlesim turtle_teleop_key
- ros2 run rqt_graph rqt_graph
- ros2 topic list
- ros2 topic info /turtle1/cmd_vel
- ros2 interface show geometry_msgs/msg/Twist
- 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}}"

```
# This expresses velocity i
    Vector3  linear
            float64 x
            float64 y
            float64 z
    Vector3  angular
            float64 x
            float64 y
            float64 z
```



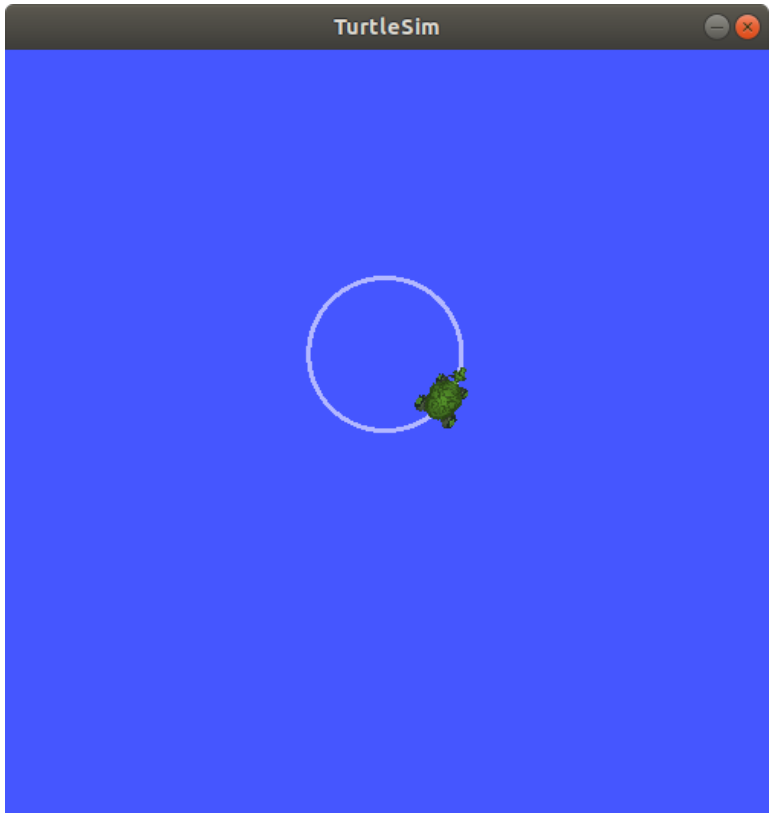
Topic

- ros2 topic <cmd>
- Remember to use Tab, it helps to complete the cmd line.

cmd	
$ list	List active topics
$ interface show	Show what structure of data the message expects
$ pub	Publish data to a topic from the cmd line
$ echo	Visualize the data being published on a topic
$ hz	Visualize the data publishing rate
$ find	List all available topic of a given message type

Service

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 run rqt_graph rqt_graph`



Service

- source /opt/ros/jazzy/setup.bash
- ros2 run turtlesim turtlesim_node
- ros2 run turtlesim turtle_teleop_key
- ros2 run rqt_graph rqt_graph
- ros2 service list

```
$ 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
```

Service

- source /opt/ros/jazzy/setup.bash
- ros2 run turtlesim turtlesim_node
- ros2 run turtlesim turtle_teleop_key
- ros2 run rqt_graph rqt_graph
- ros2 service list
- ros2 service list -t

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

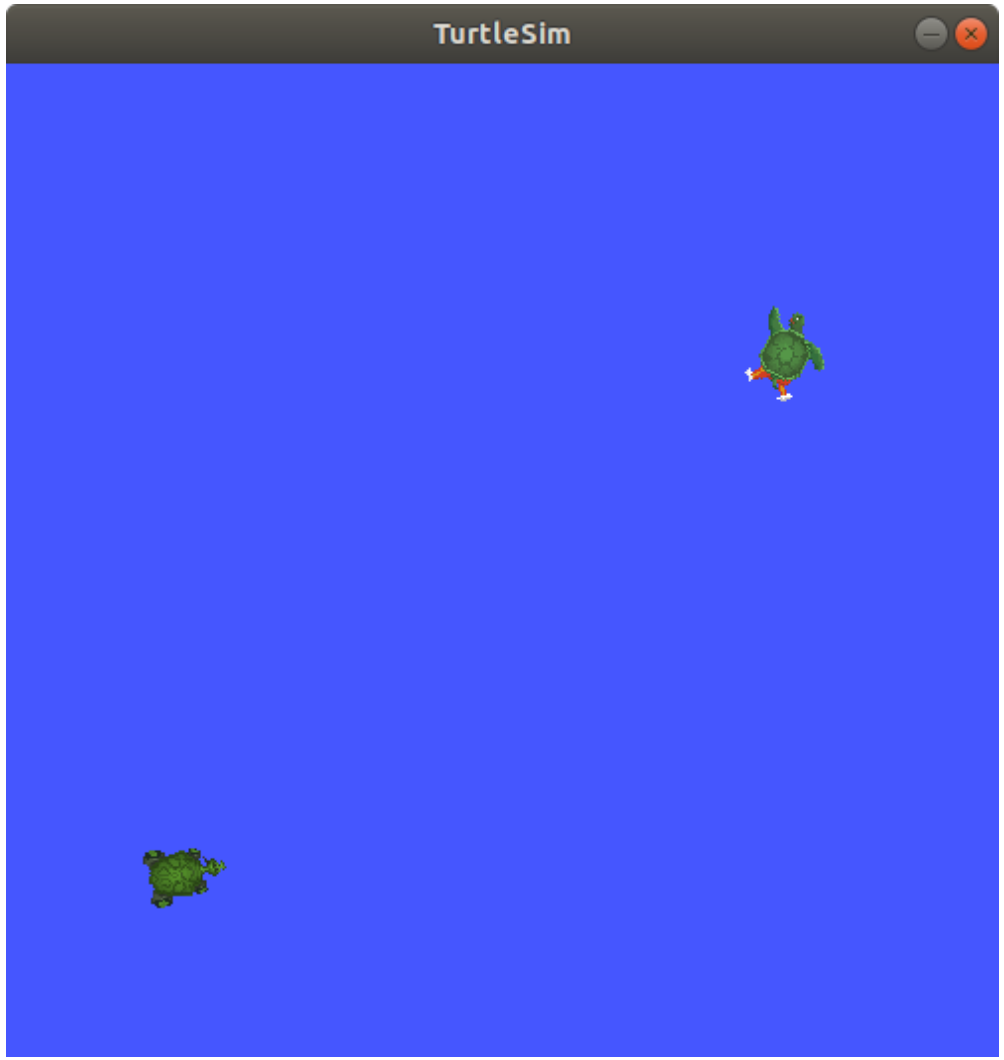
Service

- source /opt/ros/jazzy/setup.bash
- ros2 run turtlesim turtlesim_node
- ros2 run turtlesim turtle_teleop_key
- ros2 run rqt_graph rqt_graph
- ros2 service list
- ros2 service list -t
- ros2 interface show turtlesim_msgs/srv/Spawn

```
$ ros2 interface show turtlesim_msgs/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be c
---
string name
```

Service

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 run rqt_graph rqt_graph`
- `ros2 service list`
- `ros2 service list -t`
- `ros2 interface show turtlesim_msgs/srv/Spawn`
- `ros2 service call /spawn turtlesim_msgs/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: 'turtle2'}"`



Service

- ros2 service <>

Command	
$ list	List active services
$ list -t	Show service type
$ info	Print information about a service
$ find	Find all services of a specific type
$ call	Call a service
$ echo	See the data communication

Action

- `source /opt/ros/jazzy/setup.bash`
- `ros2 run turtlesim turtlesim_node`
- `ros2 run turtlesim turtle_teleop_key`
- `ros2 action list -t`

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

Action

- source /opt/ros/jazzy/setup.bash
- ros2 run turtlesim turtlesim_node
- ros2 run turtlesim turtle_teleop_key
- ros2 action list -t
- ros2 interface show turtlesim_msgs/action/RotateAbsolute
- ros2 action send_goal /turtle1/rotate_absolute turtlesim_msgs/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
```

Action

- ros2 action <>

Command	
$ list	List active actions
$ type | list -t	Show action type
$ info	More info of actions
$ send_goal	Send an action goal
$ echo	See the data communication

https://docs.ros.org/en/jazzy/Tutorials.html

Parameter

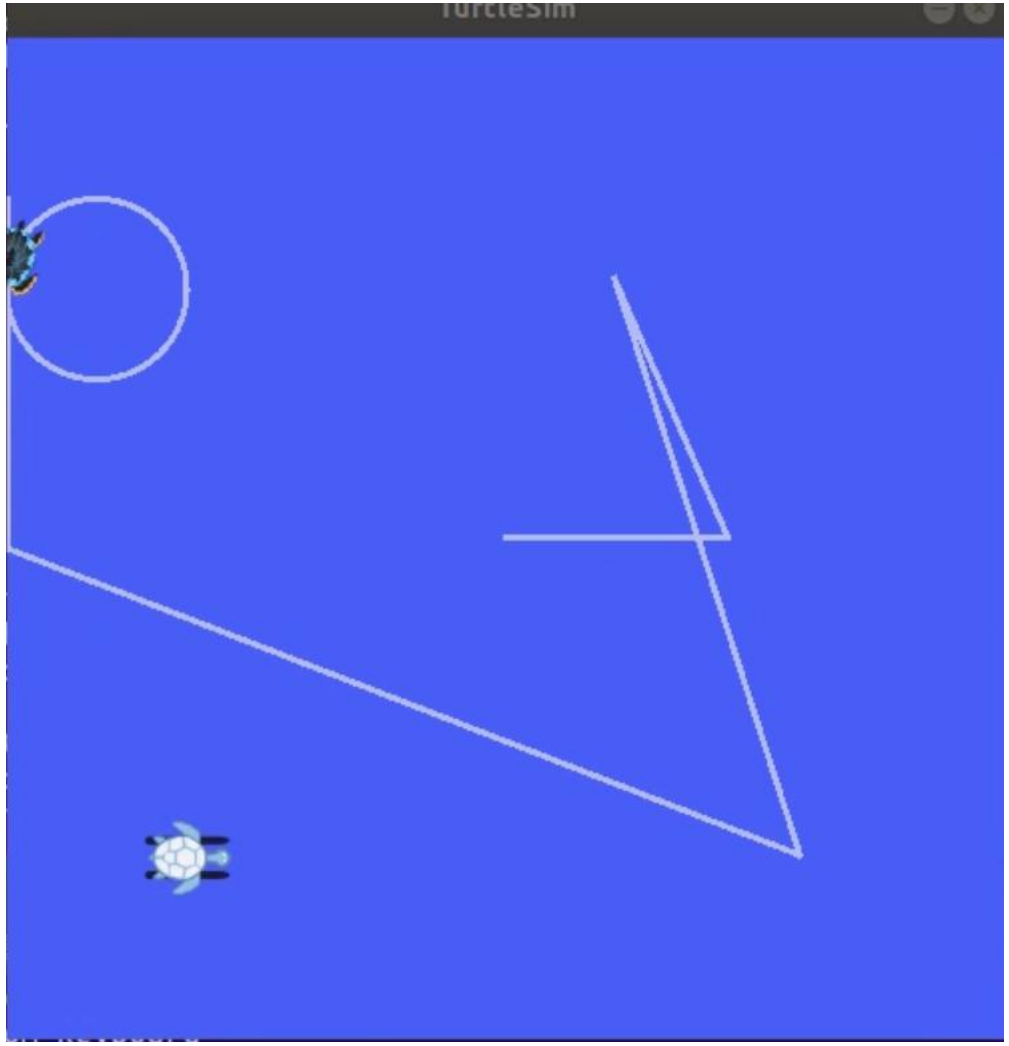
- ros2 param <>

Command	
$ list	List active parameters
$ get	Show parameter value
$ set	Set parameter value
$ dump	Get all parameters of a node and store them
$ load	Load parameters from a yaml file

Bag

- Assume that you want to develop an autonomous robot with navigation capability.
- You cannot always debug your code on the real robots. Then, you need to record the data collected by the real robot and replay it for debugging.

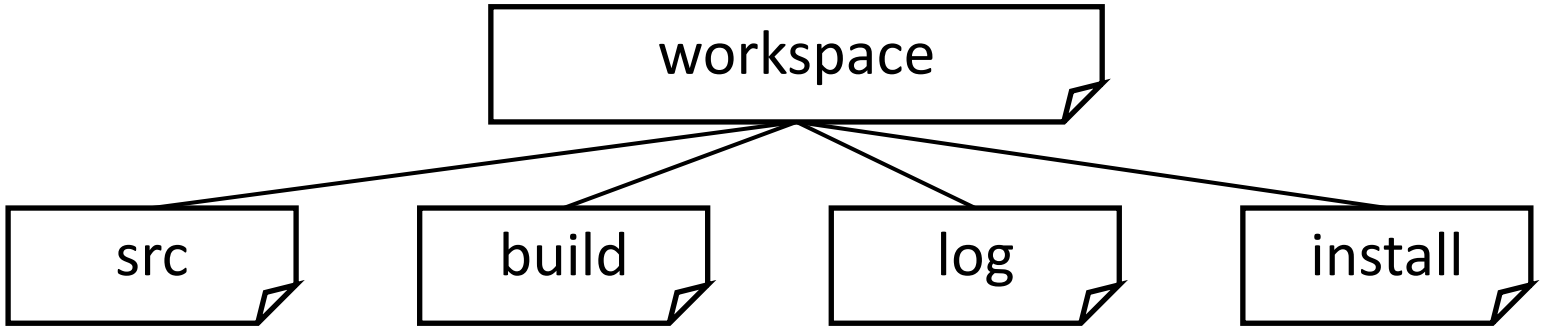
```
$ ros2 bag record -a -o cmd_record
$ Ctrl + C
$ ros2 bag play cmd_record
```



ROS: Robot Operating System

Create you own ROS package

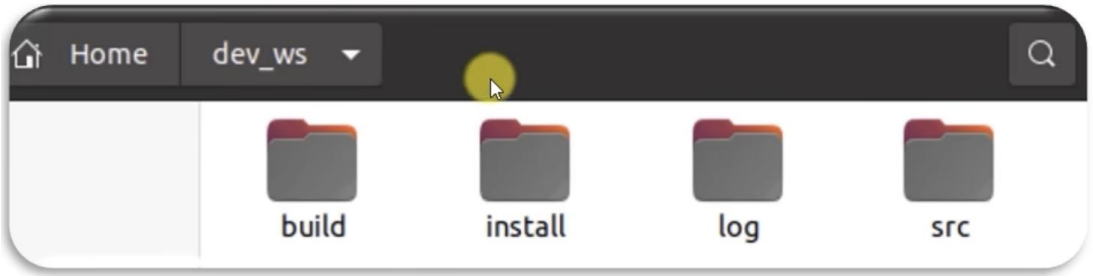
Workspace



Source space	It contains the source code. Each folder within the source space contains one or more packages.
Build Space	It stores all intermediate files during building.
Log Space	It stores all log files.
Install Space	Once targets are built, they can be installed into the install space and be executed.

Create your own workspace

```
$ source /opt/ros/jazzy/setup.bash
$ mkdir -p ~/dev_ws/src
$ …
$ cd ~/dev_ws/
$ colcon build
$ source install/local_setup.bash
```

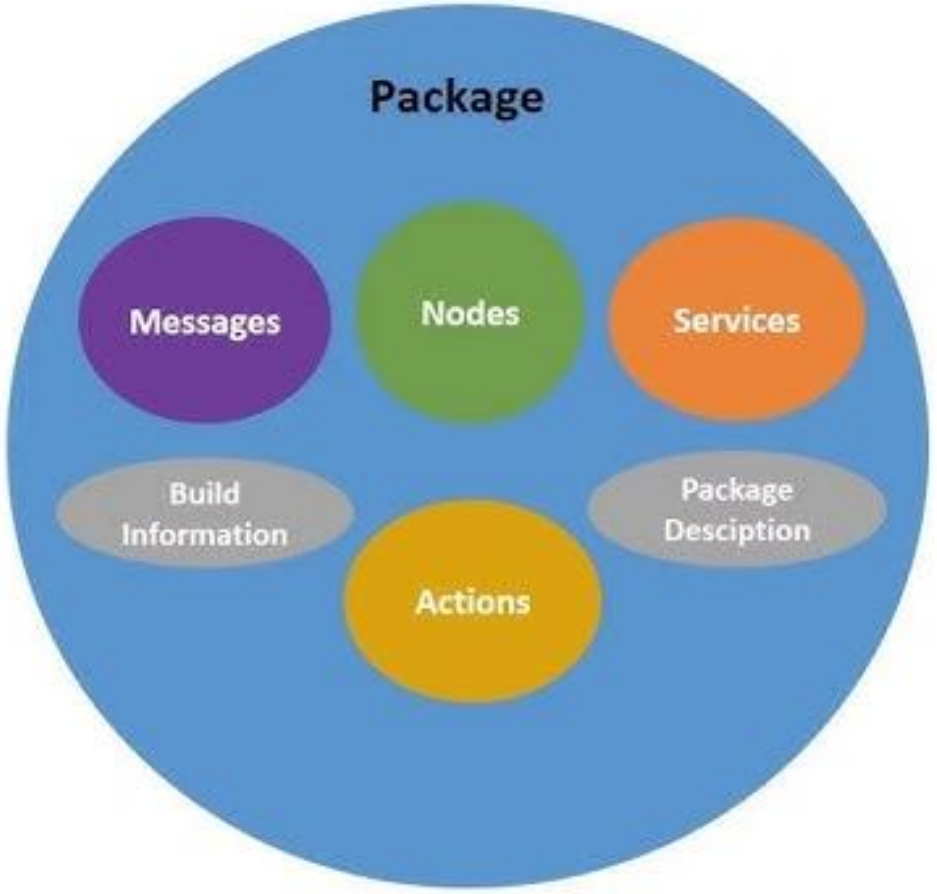


Now you can start your own project

https://docs.ros.org/en/jazzy/Tutorials.html

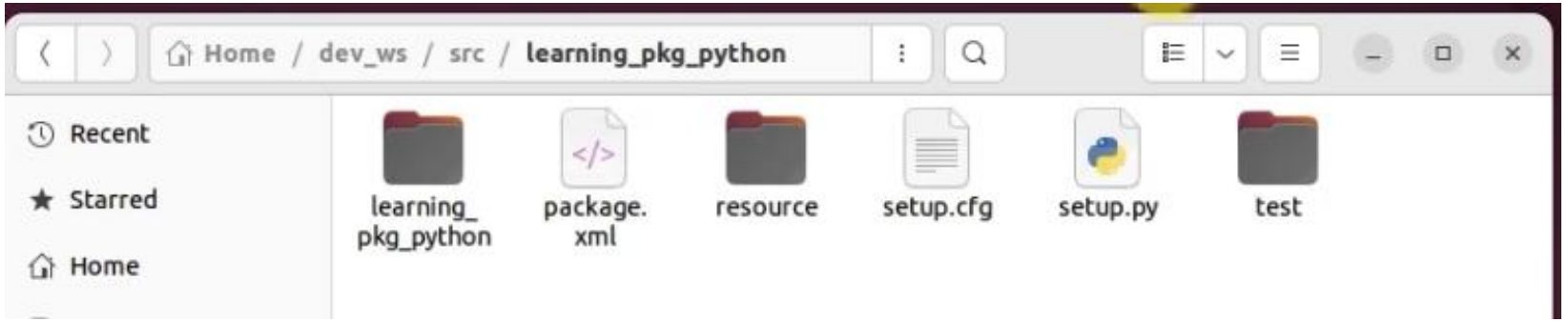
ROS Packages

- **Packages**
  - Software programs in ROS are organized in packages.
  - A package contains one or more nodes and provides a ROS interface with package information



ROS Packages

```
$ source /opt/ros/jazzy/setup.bash
$ mkdir -p ~/dev_ws/src
$ cd ~/dev_ws/src
$ ros2 pkg create --build-type ament_python <package_name>
$ ros2 pkg create --build-type ament_cmake <package_name>
cd ~/dev_ws/
$ colcon build
$ source install/local_setup.bash
$ ros2 run package_name node_name
```



Python package

ROS Packages

- A ROS package always has a package.xml file in it, containing:
  - Developer name
  - Version
  - Email
  - License
  - Dependencies
- Most of them will not influence the code running, besides dependencies.

```
<?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>my_package</name>
 <version>0.0.0</version>
 <description>TODO: Package description</description>
 <maintainer email="user@todo.todo">user</maintainer>
 <license>TODO: License declaration</license>

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

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

ROS Packages

- ROS2 also need to configure the setup.py file, to set the entry point of the code.
  - ‘my_node’: node name
  - ‘my_pk_pkg’: package name
  - ‘my_node’: the file containing main function
  - ‘main’: the main function

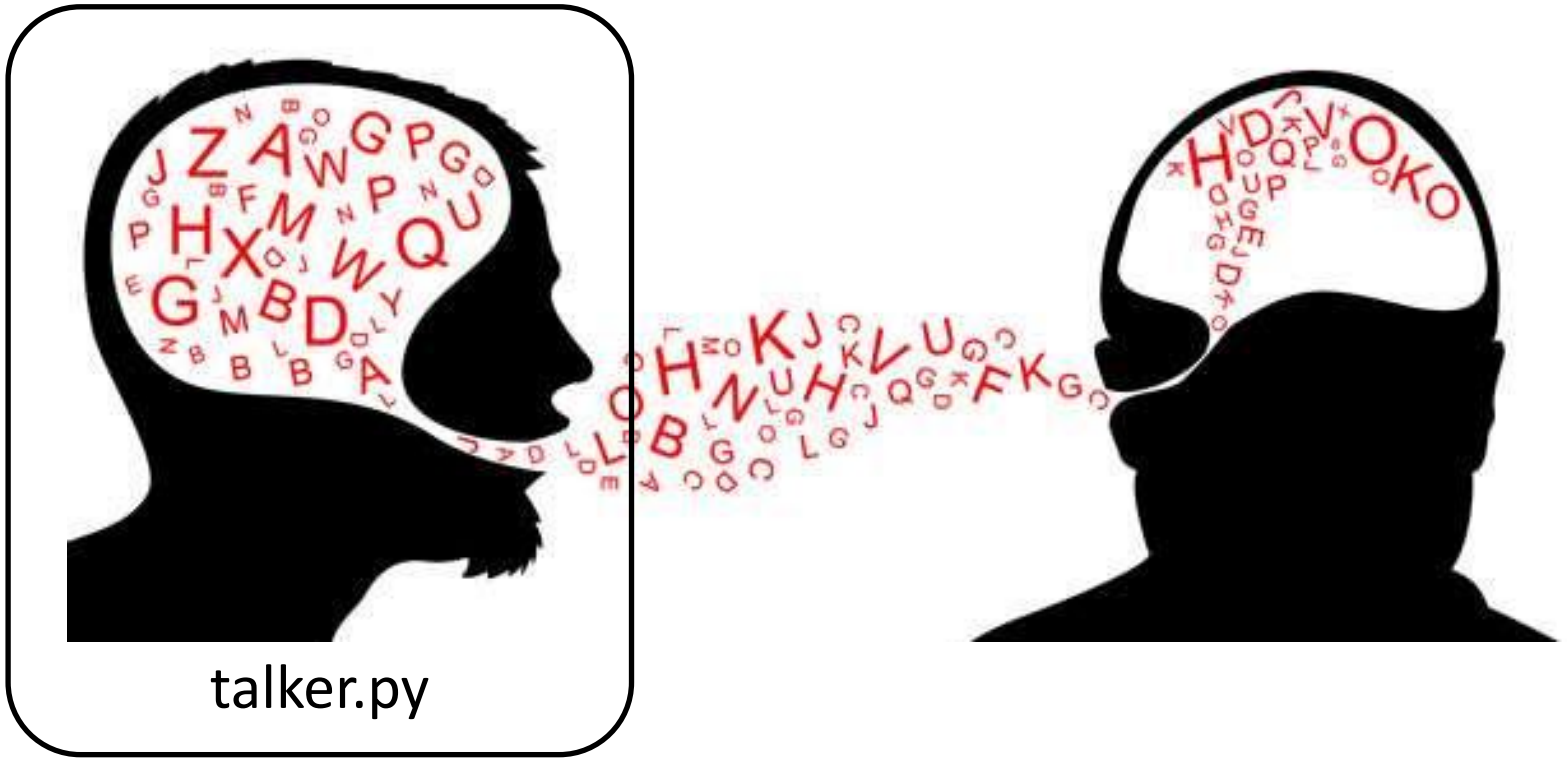
```
from setuptools import find_packages, setup

package_name = 'my_py_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='TODO',
 maintainer_email='TODO',
 description='TODO: Package description',
 license='TODO: License declaration',
 tests_require=['pytest'],
 entry_points={
     'console_scripts': [
             'my_node = my_py_pkg.my_node:main'
     ],
   },
)
```

Implementation: Talker and Listener

- We will create a package with two nodes:
  - *talker* publishes messages to topic *chatter*.
  - *listener* reads the messages from the topic and prints them out to the screen.



ROS Example: Talker Node

- Navigate to `dev_ws/src/py_pubsub/py_pubsub`
- Create a `publisher_member_function.py` file and make it executable

```
$ chmod +x talker.py
```

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Import `rclpy` and `Node` when writing a Python ROS Node.
- Import the built-in `string` message type that the node uses to structure the data.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
```

```
class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Import `rclpy` and `Node` when writing a Python ROS Node.
- Import the built-in `string` message type that the node uses to structure the data.
- These lines represent the node's dependencies. Recall that dependencies have to be added to package.xml

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
```

```
class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Create a class inherited from `Node` class.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Create a class inherited from `Node` class.
- super().__init__ calls the Node class's constructor and gives it your own name.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Create a class inherited from `Node` class.
- super().__init__ calls the Node class's constructor and gives it your own name
- Create the publisher.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Create a class inherited from `Node` class.
- super().__init__ calls the Node class's constructor and gives it your node name
- Create the publisher
- Create the timer with a callback function executed every 0.5 second

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Create a class inherited from `Node` class.
- super().__init__ calls the Node class's constructor and gives it your node name
- Create the publisher
- Create the timer with a callback function executed every 0.5 second
- The callback function will publish string data to the 'topic' every 0.5 second.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Create a class inherited from `Node` class.
- super().__init__ calls the Node class's constructor and gives it your node name
- Create the publisher
- Create the timer with a callback function executed every 0.5 second
- The callback function will publish string data to the 'chatter' topic every 0.5 second.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Initialize rclpy library.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Initialize rclpy library.
- Create the publisher node.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Initialize rclpy library.
- Create the publisher node.
- `rclpy.spin()` is a dead loop
  - waits for the callback function
  - keeps the node from exiting until the node has been shutdown.
- Ctrl-C to trigger the stop of `spin()`

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

ROS Example: Talker Node

- Initialize rclpy library.
- Create the publisher node.
- `rclpy.spin()` is a dead loop
  - waits for the callback function
  - keeps the node from exiting until the node has been shutdown.
- Ctrl-C to trigger the stop of `spin()`
- Destroy node and release resource.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5  # seconds
        self.timer =self.create_timer(timer_period,self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

Add dependencies

2.2 Add dependencies

Navigate one level back to the `ros2_ws/src/py_pubsub` directory, where the `setup.py`, `setup.cfg`, and `package.xml` files have been created for you.

Open `package.xml` with your text editor.

As mentioned in the previous tutorial, make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

```
<description>Examples of minimal publisher/subscriber using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache-2.0</license>
```

After the lines above, add the following dependencies corresponding to your node's import statements:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

This declares the package needs `rclpy` and `std_msgs` when its code is executed.

Make sure to save the file.

Add an entry point

2.3 Add an entry point

Open the `setup.py` file. Again, match the `maintainer`, `maintainer_email`, `description` and `license` fields to your `package.xml`:

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache-2.0',
```

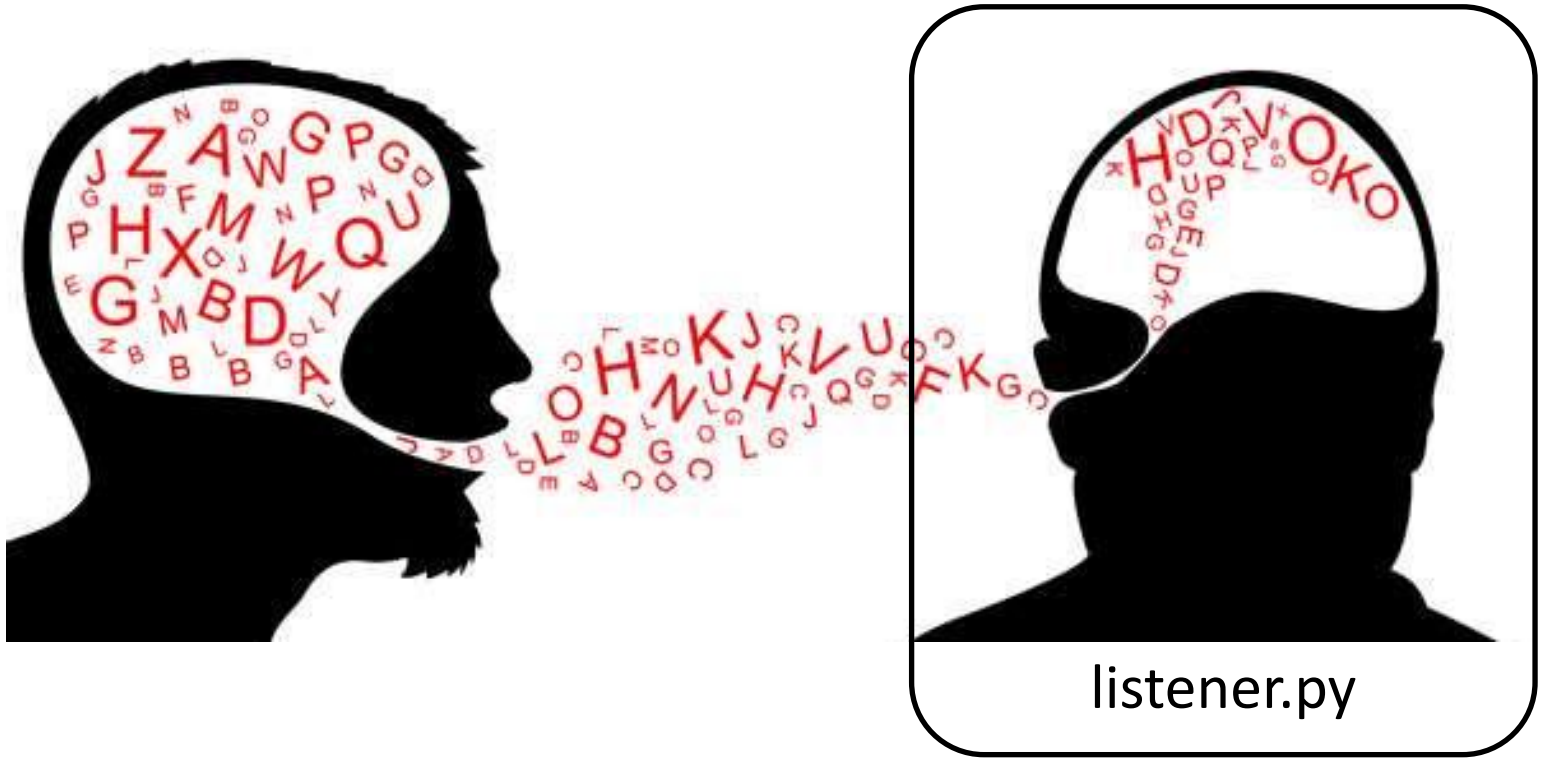
Add the following line within the `console_scripts` brackets of the `entry_points` field:

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

Don't forget to save.

ROS Example: Talker and Listener

- We will create a package with two nodes:
  - *talker* publishes messages to topic *chatter*.
  - *listener* reads the messages from the topic and prints them out to the screen.



ROS Example: Listener Node

- `subscription()` declares a subscriber to listen to a topic '`chatter`'.
- It has four parameters:
  1. Message type
  2. Topic name
  3. Callback function to handle the message
  4. Queue size

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            `chatter',
            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):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

ROS Example: Listener Node

- This line ensures the subscription object is referenced to prevent "unused variable" warnings

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            chatter',
            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):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

ROS Example: Listener Node

- The callback definition simply prints an info message to the console.
- The subscription does not need a timer, its callback get called as soon as it receives data.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            chatter',
            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):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

ROS Example: Listener Node

- The main function is identical to the publisher.
- Question, do you remember if the publisher has a callback function?

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            chatter',
            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):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

ROS Example: Listener Node

- The main function is identical to the publisher.
- Question, do you remember if the publisher has a callback function?
- The dependencies are the same, we do not need to modify package.xml

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            chatter',
            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):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

Add an entry point

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

Build your package

- Run `rosdep` in the root of your workspace to check for missing dependencies before building.
- Run `colcon build` to build your package
- Source the local setup bash to recognize the package path.

Linux macOS Windows

```
$ rosdep install -i --from-path src --rosdistro jazzy -y
```

Still in the root of your workspace, `ros2_ws`, build your new package:

Linux macOS Windows

```
$ colcon build --packages-select py_pubsub
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux macOS Windows

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

Run your package

Open a terminal

```
$ ros2 run py_pubsub talker
[info] [minimal_publisher]: publishing: "hello world: 0"
[info] [minimal_publisher]: publishing: "hello world: 1"
[info] [minimal_publisher]: publishing: "hello world: 2"
[info] [minimal_publisher]: publishing: "hello world: 3"
[info] [minimal_publisher]: publishing: "hello world: 4"
...
```

Open another terminal

```
$ ros2 run py_pubsub listener
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Ctrl+C to stop

ROS Launch

- To run the talker-listener code, we need to

```
$ ros2 run py_pubsub talker
$ ros2 run py_pubsub listener
```

- It is complex to run lots of nodes separately, especially when the project is large.

ROS Launch

- `ros2 launch` is a tool for easily launching multiple ROS nodes as well as setting parameters.
- By convention, these files have a suffix of .launch, with the syntax:

```
$ ros2 launch package_name file.launch
$ ros2 launch file.launch
```

ROS Launch

```
<launch>
  <node pkg=”py_pubsub" exec="talker" name="talker" output="screen"/>
  <node pkg="py_pubsub " exec="talker" name="talker2" output="screen"/>
  <node pkg="py_pubsub " exec="listener" name="listener" output="screen"/>
</launch>

ros2 launch first_pkg talkers_listeners.launch.xml
```

- Each <node> tag specifies the details needed to start a ROS2 node. It includes:
  - **name** → The ROS graph name of the node.
  - **pkg** → The package where the node’s executable or script is located.
  - **exec** → The executable name or script file that runs the node.
- The same script or executable can be used to create **multiple nodes** by assigning different name values.