Mobile Robotics Crash Course #1
Introduction to ROS 2

Cameron Matthew, Angus Stewart

February 2022

1 Introduction

The purpose of this worksheet is to introduce the basics of ROS needed for the upcoming modules in this Crash Course. To do this, we first explore how to implement ROS publishers & subscribers and ROS clients & services using Python. We then introduce turtlesim as a simple but fun environment to play with ROS concepts. We finish with some exercises to reinforce the ideas presented in this document.

Much of this document is heavily based on the ROS 2 Galactic tutorials - a useful resource that you should keep close at hand throughout this module and the rest of the Crash Course.

2 Setup

Before we begin, we must create a ROS workspace. A workspace is a directory which contains all of your ROS "stuff". More specifically, it is a place where we put our ROS packages (a container for your ROS nodes). Everything you do with ROS will be from this workspace directory, so it's very important!

You can choose to create your new directory in any file location of your choosing. For the purposes of this tutorial we will create it in the home directory. If you choose to create this in a different place, you will have to adjust the relevant commands accordingly.

```
$ mkdir -p ~/dev_ws/src
$ cd ~/dev_ws/src
```

It is considered "best practice" to put your packages in the `src` directory. The code above creates a new directory called `dev_ws` containing a `src` directory, and navigates into the `src` directory.

3 Topics

As mentioned in the presentation at the beginning of the workshop, topics are fundamental to ROS. Topics allow for nodes to communicate using a continuous stream of data. Publishers send data onto these data streams, and subscribers listen to the data.

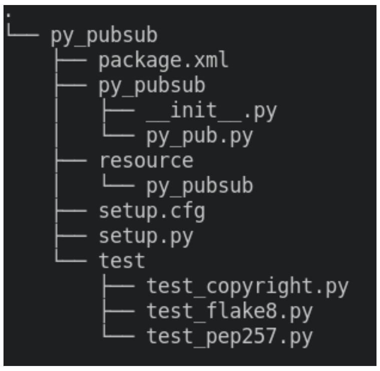
3.1 Publisher

Before we can create our publisher node which "publishes" information to a topic, we need to create a ROS package in our ROS workspace to contain this ROS node. Navigate to the `src` directory of your ROS workspace (`~/dev_ws/src` if you followed the instructions above) and run the following command to create a package called `py_pubsub`:

```
$ ros2 pkg create --build-type ament_python py_pubsub
```

You should now see a `py_pubsub` directory in your `src` directory. Navigate into `py_pubsub/py_pubsub` and create a new file called `py_pub.py`.

After all of this, your directory structure should look something like:



```
.
└── py_pubsub
    ├── package.xml
    ├── py_pubsub
    │   ├── __init__.py
    │   └── py_pub.py
    ├── resource
    │   └── py_pubsub
    ├── setup.cfg
    ├── setup.py
    └── test
        ├── test_copyright.py
        ├── test_flake8.py
        └── test_pep257.py
```

Now open the py_pub.py file you just created and paste the following code. If you find the code difficult to copy you can copy it from this ROS 2 Galactic tutorial. Take a minute to try and understand what the code is doing (you should do this with all copied code in this tutorial).

```python
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, 'topic', 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)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


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

So what does this all mean?

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String
```

Here we import the ROS client library for python, the node object, and the message type that we are using for this publisher. Messages types are highly customisable, and can be formatted to suit specific applications. For example, a message type containing displacement, velocity and acceleration vectors. All in one message! You can read more about these here.

Another useful feature of ROS 2 messages is that the API is consistent for all message types. For example, a message named FooMessage with attributes bar, velocity and angle can be initialised and populated in your node file using:

```
foo_msg = FooMessage()
foo_msg.bar = 2.73
foo_msg.velocity = 3000
foo_msg.angle = 3.14
```

Anyway...

```
class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
```

Now, we create and initialise our MinimalPublisher Node object. First we call the constructor for the parent Node object, and then we create our publisher object. Note that "self" refers to the Node class, which is why we have the `create_publisher` method available to us. Next we set up a timer which calls the `timer_callback` method every half a second.

```
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
```

The `timer_callback` method creates our string message, and then publishes that method using the publisher we created earlier.

Now we (more or less) understand the code we copied, let's continue to implement our publisher...

We need to update the `package.xml` in the ROS package we created to tell ROS what other ROS packages we need for our package to work properly. Add the following to the file:

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

Next, we need to add some extra lines to our `setup.py` file enable ROS to execute our publisher. Add the following lines within the `setup()` command:

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

These are called entry points.

Now we know what the publisher does, it is now time to test it. Before we can do this we need to do 3 things:

1. Install dependencies
2. Build
3. Source

These operations are fundamental to using ROS and you will be doing this a lot over this Crash Course.

3.1.1 Install dependencies

Navigate to the dev_ws directory and install dependencies using rosdep. Strictly, we don't have to install dependencies in this case, since they were installed by defualt with ROS 2, but it's usually a good idea to run it at least once when building a package for the first time, so can do this step anyway.

```
$ rosdep update
$ rosdep install --from-paths src --rosdistro galactic -y
```

3.1.2 Build packages

Navigate to the `dev_ws` directory and build the packages with:

```
$ colcon build
```

This will generate `build`, `install` and `log` directories within `dev_ws`. Note that it is considered "good practice" to build in a **different** terminal to the terminal you use to run your application.

3.1.3 Source overlay

ROS 2 contains layers called the "underlay" and the "overlay". The underlay is the set of packages that make up the core ROS 2 workspace - the default packages installed in ROS 2. The overlay is a workspace which does not interfere with packages in the underlay. In our case, our publisher package is in the overlay. For a layer to "take effect" we must source it. Since we source the underlay on terminal startup (as we added it to the .bashrc file in the installation instructions document), we only need to source the overlay.

To do this, open a new terminal and navigate to the `dev_ws` directory and run:

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

Once this is done, we are ready to run our publisher.

3.1.4 Run the publisher

```
$ ros2 run py_pubsub talker
```

Where the `py_pubsub` is the name you chose for your publisher package, and `talker` is the name of the entry point described in your package `setup.py`.

You should see that the publisher starts logging in your terminal, and it should also start publishing to `/topic`. You can check this by running the following command in another terminal:

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

Some additional topic commands you can play around with are:

```
$ ros2 topic info /topic
$ ros2 topic hz /topic
$ ros2 topic bw /topic
```

Some additional commands to inspect your node:

```
$ ros2 node info /talker
```

For more, use the `-h` flag for all `ros2` commandline commands and explore!

3.2 Subscriber

We will now create a subscriber node to listen to the topic that we have previously published to. To do this we will create a new ROS 2 node inside our package.

Create a new file in the `py_pubsub/py_pubsub` directory called `py_sub.py`. Open this file and paste the following code. You can take this from the ROS 2 galactic documentation if you need.

```python
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,
            'topic',
            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)
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


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

How does this code work then?

The format of the subscriber node is very similar to the publisher node, and indeed the service and client nodes (spoilers), so from now on, we will only explain the distinctive parts of the code.

For the subscriber we do **not** need to create a timer. This is because our subscription object, automatically calls the supplied method whenever it receives a message from the topic it is subscribed to. In our case this is the `listener_callback` method. For our minimal subscriber this callback just logs the message recieved.

Continuing with completing our subscriber node, we need to add another entry point to allow us to execute our command. Add lines to your packages setup.py file such that the `entry_points` now look like this.

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

Now all that's left is to build, source then run! In your terminal run:

```
$ cd ~/dev_ws
# Make sure all dependencies are installed
$ rosdep install -i --from-path src --rosdistro galactic -y
$ colcon build
```

Open a new terminal and run the following.

```
$ cd ~/dev_ws
$ . install/setup.bash
$ ros2 run py_pubsub talker
```

Open a second terminal and run the following.

```
$ cd ~/dev_ws
$ . install/setup.bash
$ ros2 run py_pubsub listener
```

You should see the publisher node logging what it is publishing to `/topic` in its terminal, and the subscriber node logging what it is receiving from `/topic` in its terminal. You have now successfully set up communication between two nodes!

4 Services

As described in the talk, services are the second way nodes can communicate with each other. They are different to topics in that information isn't transferred continuously. A client sends a request to the service. The service executes some code, then sends a response back to the client.

4.1 Service and Client

To create our service and client system, we will create yet another new package. Packages should be purpose specific, so we will keep the publisher and subscriber package separate.

Create a new package in `dev_ws/src`

```
$ cd ~/dev_ws/src
$ ros2 pkg create --build-type ament_python py_srvcli --dependencies rclpy
    example_interfaces
```

Next we will write the service node. Create a new file in your package called `py_srv.py`. And copy the following code into the file. If you are struggling to copy from this document then try copying from this document

```python
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node


class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints',
            self.add_two_ints_callback)

    def add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Incoming request\na: %d b: %d' % (request.a,
            request.b))

        return response


def main():
    rclpy.init()

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()


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

Code explanation:

```
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node
```

As previously, we import the ROS 2 python client library and the Node object, we now also import the service type we will be using. Just like message types, service types are also highly customisable. If you're interested, see this document.

The particular message used here is defined as follows:

```
int64 a
int64 b
---
int64 sum
```

Above the triple dash is the message definition for the our request, below is the definition for the response. In this case, our request contains two integers, and the response contains one integer (which should hopefully be the sum of a and b).

```
def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints',
            self.add_two_ints_callback)
```

In the service constructor we create our service object using our imported message type. This has an associated callback, which in our case is labelled `add_two_ints_callback`.

```
add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Incoming request\na: %d b: %d' % (request.a,
            request.b))

        return response
```

As you can see, the callback for a given service must have a particular format, in that it must contain inputs for a request and a response, and it must return the response. In this callback, we create our response message, and then log the request.

Another useful feature of service message types is that the method of creating the request and response messages is consistent across message types.

For example, if I defined the following message called FooBar...

```
bool foo
float64 bar
---
bool success
```

Then I could create a request by writing in my node file:

```
request = FooBar.Request()
request.foo = True
request.bar = 3.14
```

And equivalently a response in my service callback...

```
def callback(self, request, response):
    response.success = True
    return response
```

We should now create the client. Create a new file called `py_cli.py` and copy the following code. If you are struggling to copy from this document then try copying from the ROS 2 galactic tutorials.

```
import sys

from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node


class MinimalClientAsync(Node):

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

    def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        self.future = self.cli.call_async(self.req)


def main():
    rclpy.init()

    minimal_client = MinimalClientAsync()
    minimal_client.send_request()

    while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
            except Exception as e:
                minimal_client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                minimal_client.get_logger().info(
                    'Result of add_two_ints: for %d + %d = %d' %
                    (minimal_client.req.a, minimal_client.req.b, response.sum))
            break

    minimal_client.destroy_node()
    rclpy.shutdown()
```

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

Lets run through how the client node works...

```
def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddTwoInts, 'add_two_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddTwoInts.Request()
```

In the client constructor we initialise the client with our message type and the service name. We then wait until either the service is available, or the timeout of 1 second elapses, logging every time we wait. Finally we create our `request` message, which we will send to the service.

```
def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        self.future = self.cli.call_async(self.req)
```

The `send_request` method creates our request method using the arguments provided to our client, and then asynchronously calls our service node. This essentially means that we have made a request to call the service, which will be completed at a time that is "convenient" for ROS. This helps prevent a crash if too many nodes call a service simultaneously. This returns a `future` object which is completed when the call request is fulfilled.

```
while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
            except Exception as e:
                minimal_client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                minimal_client.get_logger().info(
                    'Result of add_two_ints: for %d + %d = %d' %
                    (minimal_client.req.a, minimal_client.req.b, response.sum))
            break
```

Finally, this section in the `main` method handles the `future` object. This maintains our node while ROS is waiting for the future to complete. Once the request is fulfilled we either log the result or raise an exception.

Explanation over, now we best continue with our package...

Add entry points for both the service and the client in setup.py.

```
entry_points={
    'console_scripts': [
        'service = py_srvcli.py_srv:main',
        'client = py_srvcli.py_cli:main',
    ],
},
```

Finally, build, source and run.

```
# Navigate to workspace directory
$ cd ~/dev_ws
# Make sure all dependencies are installed
$ rosdep install -i --from-path src --rosdistro galactic -y
# Only build service and client package
$ colcon build --packages-select py_srvcli
```

Open a new terminal and run:

```
$ cd ~/dev_ws
$ . install/setup.bash
$ ros2 run py_srvcli service
```

Open an additional terminal and run:

```
$ cd ~/dev_ws
$ . install/setup.bash
$ ros2 run py_srvcli client 2 3
```

You should see the result of your calculation printed in the service terminal.

5 Introducing turtlesim

In this section we go through the steps necessary to install turtlesim, a fun environment where you can play around with the basic ROS concepts we've explored so far. You will also need this environment for the exercises described later.

5.1 Install turtlesim

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

You can verify that this was installed correctly by running

```
$ ros2 pkg executables turtlesim
```

This should return something like:

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

5.2 Start turtlesim

```
$ ros2 run turtlesim turtlesim_node
```

You should see a randomly selected turtle (named turtle1) appear in the middle of a blue window.

5.3 Use turtlesim

With the turtlesim node still running, open a new terminal and launch the turtle teleop key:

```
$ ros2 run turtlesim turtle_teleop_key
```

Instructions on how to use this node to control turtle1 will be shown in the terminal used to launch the teleop key. You can view the nodes, topics and services associated with the turtlesim node and the teleop key by running the apropriate command from the following list:

```
# List running nodes
$ ros2 node list
# List active topics
$ ros2 topic list
# List active services
$ ros2 service list
# Get publishers, subscribers, services and clients associated with a node
$ ros2 node info <node name>
# Get the number of publishers and subscribers associated with a topic
$ ros2 topic info <topic name>
# Echo data passed along a topic
$ ros2 topic echo <topic name>
```

5.4 Close turtlesim

In ROS 2 the typical method of terminating a process is to press `Ctrl+C` in the terminal in which you launched the process. So to close turtlesim, simply press `Ctrl+C` in each terminal you have opened (Note that the teleop key has a custom method of termination - you should press `q` in the terminal used to launch the teleop key).

6 Exercises

In the following exercises, we challenge you to modify what you've just done to achieve the described outcomes. These will help reinforce the concepts that you've been introduced to in this document.

To complete these exercises, you will need to create nodes which perform more than one function (e.g. publishes to topic A and subscribes to topic B simultaneously). This is generally how we write nodes in practical applications. Nodes are almost never specific to being a "Publisher node" or a "Subscriber node" alone.

6.1 Adapt the publisher to publish command messages for a turtle in turtlesim

Change the message type and name of the publisher you created earlier to match the topic name (`/turtle1/cmd_vel`) and type of the turtlesim command subscription (type should be `geometry_msgs/Twist`). To use this type of message, you must import it in your python node file. For the Twist message, add the following:

```
from geometry_msgs.msg import Twist
```

You should add `geometry_msgs` to the list of dependencies in the `package.xml` file. As an extension, edit the publisher such that it instructs the turtle to follow a predetermined route.

The end goal is to be able to run the `turtlesim_node` in one terminal, and your custom node in another, which should result in the turtle following the commands supplied by your custom node.

6.2 Change the pen during runtime

Create a client in the node you edited in 6.1. Using the `/turtle1/set_pen` service, change the pen characteristics of the turtle during runtime. You could choose to have the pen change colour every second for example. If you chose to use any external packages to achieve this, remember to include them in your package.xml dependencies.

To access the `turtlesim/SetPen` service type, add the following line to the top of your python node file:

```
from turtlesim.srv import SetPen
```

6.3 Turtle Chase

The aim of this exercise is to create a second publisher node, which creates a new turtle and instructs it to follow the original turtle.

Copy your node from the previous questions to a new file (with a different name) in the `py_pubsub` package, and create a separate entry point for this node. At this point we assume your node from 6.1 instructs `turtle1` to move, following a path of your choosing. You should now have two separate nodes with identical contents. Next, adapt one of these nodes to spawn a new turtle called `turtle2` when initialised. You can do this by creating a client in your node which calls the `/spawn` service in the constructor.

After this, adapt the publisher to publish to `turtle2`'s command topic (`/turtle2/cmd_vel`).

Now, create subscriptions to `/turtle1/pose` and `/turtle2/pose`. This will give you access to the position of each turtle.

Finally, using this node configuration, devise an algorithm which will instruct `turtle2` to chase `turtle1`.

To access the relevant message types, add the following to your imports:

```
from turtlesim.srv import Spawn
from turtlesim.msg import Pose

from geometry_msgs.msg import Twist
```

7 Completion!

Congratulations! You've completed the introduction to ROS 2 worksheet! Up next is the Perception workshop.