4.2 Colcon and our workspace

If we want to build up functionality from the ground, we need to make a package. Inside that package, we have our nodes running and other files as well. Now, to integrate this package into our system we need to build it. For this, we will use the tools of colcon. To do so, first, we need to install it:

```
$ sudo apt install python3-colcon-common-extensions
```

Now we create our first workspace, we can name it anything we like, but for simplicity, we will name it ”ros2_ws”, let’s make it with its source folder (”src”) and also move the working directory inside the workspace:

```
$ mkdir -p ~/ros2_ws/src
$ cd ~/ros2_ws
```

In the next steps, we will download an already-made repository, to skip the building part for now and focus only on the building of the package. Clone the examples repository to the src folder:

```
$ git clone https://github.com/ros2/examples src/examples -b foxy
```

It is always a good practice, to check for the dependencies of the package we want to install. We can do so by navigating to the workspace’s main directory (currently: ”/ros2_ws”):

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

If we get the answer:

```
#All required rosdeps installed successfully
```

We are all good and proceed further. Now we can build our whole workspace with the following command:

```
$ colcon build --symlink-install
```

This command now builds our whole workspace, but sometimes it is not efficient to build the whole workspace. For the method of how to build only certain packages see the Cooking tips section. After building we should test our packages:

```
$ colcon test
```

After testing we need to source our ”setup.bash” file again since it has changed from the last time we sourced it:

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

We can also add this line to our ’bashrc’, so we will get this always sourced as well:

```
$ echo "source install/local_setup.bash" >> ~/.bashrc
```

(For a hint on how to make an easier process regarding these two extra command see the Cooking tips section.)

And finally, we can try out the premade nodes in two separate terminals:

```
$ ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
```

```
$ ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

This will start a publisher that will simply count the time and a subscriber node. For more practice, try doing the same with the 'ros_turtorials' repository. Inside 'ros2_ws/src' use the following command:

```
$ git clone https://github.com/ros/ros_tutorials.git -b foxy-devel
```

Check for the dependencies, build the package, test, and source Your workspace.

4.3 Package

Now we can go further and make our first own package. For that we only need to move to the 'src' directory of our workspace, and use the following command:

```
$ ros2 pkg create --build-type ament_python --node-name my_node
   my_package
```

We use the argument '–build-type' with the option 'ament_python' and argument '–node-name' with option 'my_node', to tell the system, that we want a python-based package and we want a node inside that package named "my_node", also we name our package to 'my_package'. After the system finishes executing this command, we need to build our package, for which we will follow our previously learned steps. from 'ros2_ws' dictionary, we build the packages (just now we can skip the dependency checking part since we didn't added any special dependency to the package):

```
$ colcon build --packages-select my_package
```

and source:

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

If we done everything all right, then we can try out our new package with its node.

```
$ ros2 run my_package my_node
```

The node should respond with the following message on the screen:

```
Hi from my_package.
```

This is the basic package, that the command created for us. Now let's examine the content of the package, which can be seen in 'ros2_ws/src/my_package'. We will see thee directories (my_package, resource, test) and three files (package.xml, setup.cfg, setup.py). For now we need to observe two of them: package.xml and setup.py.

package.xml contains all the basic information about the package, like name, version, license, and most importantly the dependencies. If we open it we will see the following:

```
<?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>
```

You can add the details to the "TODO" parts. (Don't forget to save the changes)

Now, the second important will be the setup.py, this contains similar information about the package, so if You changed something in the XML file, You should do so here as well. (Don't forget to save the changes) Here we can also see which nodes and how can be used later in the 'entry_points' field.

```python
from setuptools import setup

package_name = 'my_py_pkg'

setup(
 name=package_name,
 version='0.0.0',
 packages=[package_name],
 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'
     ],
   },
)
```

4.4 First publisher and subscriber

Let's move forward and make something a bit more interesting. We will remake the previously downloaded publisher-subscriber system. Here, our goal will be to make a package, that will contain two nodes, one will make a count and send its result to a topic, while the other will subscribe to this topic and print its received data. Let's prepare the package and name it pubsub, after navigating to the 'src' folder :

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

Now it is time to open our favorite Python compatible IDE and open the package in that. Inside our package, we have a same-named folder, which will contain our nodes. Let's create a file for the publisher, called "publisher.py"

We will fill this file up with Python script while attempting to understand each part simultaneously. You can copy and paste the code snippets according to the instructions. At the end, You will be able to check Your code, if it is not working in the appendices: A.1.1.

First, we import 'rclpy' and also from 'rclpy' its 'Node' class, we will build our code based in this class.

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

After this, since we know that we will need some messages to send, and all message has a type and definition, we need to import one. Now it will be enough for us to import from the standard message types the String:

```python
from std_msgs.msg import String
```

After this, we will have three parts, a class definition, a 'main' function, and the switch between the operational modes. We will name our class "MinimalPublisher" with two method definitions, that we will fill later as well as the main function:

```python

class MinimalPublisher(Node):
    def __init__(self):
        pass


    def timer_callback(self):
        pass


def main(args=None):
    pass


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

Now, that we have all the main structural elements let's fill up the missing parts, starting with the "__init__" method:

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

In this code snippet above, we first call the 'Node' class's constructor and give our node's name "minimal_publisher'. Then we define our publisher with the 'create_publisher' using "String" type for the message sent, "firsttopic" for the name of the topic to the messages, and give "10" as the queue size. After this, we set a timer period in seconds, and also create the timer with its period time and callback function called "timer_callback"

Next, the callback function:

```python
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
```

Here, we will create our message to send. We make a message definition that will be a "String" typed one. After that to the message's data field, we compose the exact message. After that, we will send it to the topics by our publisher's "publish" method. It is also good to have some logging, so we add that as well. Finally, we integrate our variable.

Lastly, we will make the main function's definition which will be simple:

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

We initialize the 'rclpy' library then we create the node, and "spin" it so it will be usable. Finally, we are assuring that the node will be destroyed correctly if we stop its execution.

Let's move to the subscriber node, create a python file called "listener.py" and let's fill it up. We will do so similarly (explain only the new parts) and at the end, if something is not working, You can check it also in the appendices: A.1.2

We will start from a very similar structured script, only the name of our node class will differ:

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

from std_msgs.msg import String

class MinimalSubscriber(Node):

    def __init__(self):
        pass

    def listener_callback(self, msg):
        pass


def main(args=None):
    pass


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

The "__init__" method has a very similar logic to the publisher, except we now define a subscriber, for which we give the message type, topic name, callback function, and queue size:

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

After that, we define the callback, so the node will know what to do with the message received. Now we will only print out as a log to our terminal:

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

The "main" part is the same as previously with the publisher, adjusted to the new variable names:

```
def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()
```

Now, we have the functional part of our two nodes, so we can integrate them into the package. For this, we will need to update the "package.xml" and the "setup.py" files.

For the "package.xml" (completed see at appendices: A.1.3), we will need to update the main details (maintainer, license, etc.) and most importantly add the dependencies. Since we are using both "rlcpy and "std_msgs" during execution, we add them to the list of the execution-dependencies as follows:

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

Now the next thing to update before building the package, is the "setup.py" (completed version here: A.1.4). Here we need to add the two nodes as entry points, so we will be able to easily access them via 'ros2 run', let's refer to the publisher node as "talker" and to the subscriber node as "listener":

```
entry_points={
        'console_scripts': [
            'talker = pubsub.publisher:main',
            'listener = pubsub.subscriber:main',
        ],
    },
```

Here the convention is the following:

```
'<ALIAS FOR ROS2 RUN> = <PACKAGE NAME>.<FILE NAME>:main'
```

Now we can build the package with the usual way, dependency check, building, testing and sourcing:

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

```
$ colcon build --packages-select pubsub
```

```
$ colcon test
```

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

And now we can try out our first package in two terminal tabs:

```
$ ros2 run pubsub talker
```

```
$ ros2 run pubsub listener
```

This concludes our practice regarding the packages and the start of the communication protocols used. In the next practice, we will create our second and third packages to try out the communication protocols via services and actions.

4.5 Cooking tips

4.5.1 Quick finding of the built packages

We can use the ’colcon_cd’ to quickly jump our current working directory to the package. For this, however, we need first to update our ”.bashrc” file by the following commands:

```
$ echo "source /usr/share/colcon_cd/function/colcon_cd.sh" >> ~/.bashrc
$ echo "export _colcon_cd_root=/opt/ros/foxy/" >> ~/.bashrc
```

4.5.2 Tab completion

We can also use tab completion (double tap ”tab” button, for quickly complete package or node names, or to check all the same starting packages/nodes.) For this functionality, we need to add to the ”.bashrc” the following line with this command:

```
$ echo "source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.
    bash" >> ~/.bashrc
```

4.5.3 Building only certain packages

To build only selected packages we can use ’–packages-select’ while building in the following form:

```
$ colcon build --packages-select <PACKAGE_NAME>
```

4.5.4 Merge testing and sourcing into the build

We can make our lives a little easier if we define an alias for checking the dependencies, inside our ”.bashrc” file

```
alias ck_dep='cd ~/ros2_ws && rosdep install -i --from-path src --
    rosdistro foxy -y'
```

While, for the complete building process we can use a custom function inside the ".bashrc" file. Something like the following:

```
col_mk() { ck_dep && colcon build --packages-select "$1" && source
    install/setup.bash; }
```