5.2 Our first Service

We will proceed with our first service server and service client as we previously did with the publisher and subscriber. We will create a package and add two Python files to it and while we fill those up, we will attempt to understand its logic. Let's make our package in the 'src' folder first:

```
$ ros2 pkg create --build-type ament_python py_srvcli --dependencies
   rclpy example_interfaces
```

Now If You check the "package.xml", You will see something different from the previous package creations. Since we added the keyword '–dependencies', 'rclpy' and 'example_interface' is already added to the dependency list, where the latter one contains our definition of the service, which we will use. The definition looks like the following:

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

Now, we need to update the 'package.xml' and the 'setup.py' with the important details of the package (maintainer, description, etc.).

Now let's create the python file in our package (inside the package's "srvcli" folder), and name it 'serviceserver.py'. The main structure will be similar to the previous ones:

```
import rclpy
from rclpy.node import Node

from example_interfaces.srv import AddTwoInts


class MinimalService(Node):

    def __init__(self):
        pass

    def add_two_ints_callback(self, request, response):
        pass


def main(args=None):
    pass


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

We import the 'Node' class, and also the service definition. Now er initialize our own custom node:

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

Here, as usual, we take the 'Node' class as a base and then create the service. Our service will use the "AddTwoInts" as the service type, then it will be offered on the "first-service" service, and finally, we name our service callback function "add_two_ints_callback". Now let's define the 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
```

Here we first calculate the response by adding together the request's "a" and "b" fields, which will be the response's "sum" field. We also log our request and response to the terminal.

Finally, we fill the "main" function, with the same logic, that we already discussed:

```
def main(args=None):
    rclpy.init(args=args)
    minimal_service = MinimalService()
    rclpy.spin(minimal_service)
    rclpy.shutdown()
```

That's it (You can find the complete version in: A.2.1). Now let's move to the client part and name it "serviceclient.py"

```
import sys
import rclpy
from rclpy.node import Node

from example_interfaces.srv import AddTwoInts


class MinimalClientAsync(Node):

    def __init__(self):
        pass

    def send_request(self, a, b):
        pass

def main(args=None):
    pass


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

Now the difference is in the importing to the "sys", which allows us to sort out the command line input arguments for the request via the 'sys.argv' command. Let's initialize our node:

```
def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddTwoInts, 'firstservice')
```

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

After taking the 'Node' class as a base, we create the client with the "AddTwoInts" type and the "firstservice" as the correspondent service name. Now we also take care of a really important thing, what if the service is with the same type and the service name is not available? We should get at least some feedback about it. Exactly that's what the while loop does, checks the availability of such a service server in every second. Finally, if the service is available we make an instance of the request called "req".

Now we can define how to send our request. We sort out the fields of the request first, then we call the service (asynchronously) and wait until the response is created and sent.

```
def send_request(self, a, b):
        self.req.a = a
        self.req.b = b
        self.future = self.cli.call_async(self.req)
        rclpy.spin_until_future_complete(self, self.future)
        return self.future.result()
```

In the "main", we make an instance of our node, and use the "send_request". After we get the answer we log it out to the terminal with some formatting.

```
def main(args=None):
    rclpy.init(args=args)

    minimal_client = MinimalClientAsync()
    response = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    minimal_client.get_logger().info(
        'Result of add_two_ints: for %d + %d = %d' %
        (int(sys.argv[1]), int(sys.argv[2]), response.sum))

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

We completed the client as well (for the completed version see: A.2.2). Now we need these two as entry points to the 'setup.py' file:

```
entry_points={
    'console_scripts': [
        'service = srvcli.serviceserver:main',
        'client = srvcli.serviceclient:main',
    ],
},
```

It is time to build and try out our package (inside "ros2_ws"):

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

```
$ colcon build --packages-select srvcli
```

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

And try in two tabs:

```
$ ros2 run srvcli service
```

```
ros2 run srvcli client 2 3
```

Now, what we will get from the server as a log:

```
[INFO] [minimal_service]: Incoming request
a: 2 b: 3
```

and from the client:

```
[INFO] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

This concludes our section about services, and we move on to a more advanced method called actions, so for now clear Your tabs.

5.3 Our first Action

To examine actions we will create a new package called "actionsrvcli" (inside 'src'):

```
$ ros2 pkg create --build-type ament_python actionsrvcli --dependencies
    rclpy action_tutorials_interfaces
```

Quickly we can check the Fibonacci action's structure:

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

And make the server file called "actionserver.py":

```
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node

from action_tutorials_interfaces.action import Fibonacci


class FibonacciActionServer(Node):

    def __init__(self):
        pass

    def execute_callback(self, goal_handle):
        pass


def main(args=None):
    pass


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

We do our import similarly to the previous methods. Only now we have to add the "ActionServer" as well. Now let's fill the initializer:

```
def __init__(self):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(
            self,
            Fibonacci,
            'firstaction',
            self.execute_callback)
```

Very similar to what we have done with the service server, we take the 'Node' class as a base, and create an action server, with the type of "Fibonacci", action named "firstaction" and a callback function named "execute_callback". Now let's fill in this callback function.

Since an action takes usually a longer period to execute, we send a logger message. Then, as we want to get the values of the elements of the given order in the Fibonacci sequence.

Now the sequence starts with 0 and 1 let's take those into a list variable called "sequence. After that, we can calculate and add the next element to the list until we reach the required order.

To ensure the functionality in the case an empty request is sent we use "succeed()" and also if the request is fulfilled. Finally, we return the result. Now this so far is similar to the service that we did, and also is missing a very important component, the Feedback part of the action.

```
def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')

        sequence = [0, 1]
        for i in range(1, goal_handle.request.order):
            sequence.append(sequence[i] + sequence[i-1])

        goal_handle.succeed()
        result = Fibonacci.Result()
        return result
```

We need to add the feedback part as well. For this, before the calculations, we make an instance of the Feedback of the Fibonacci's type, and add it to the feedback's only field "partial_sequence". Now we change the calculation part a bit, so it will calculate the partial sequence elements, and only after finishing the for loop will we send the list as our result.

```
def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')

        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]

        for i in range(1, goal_handle.request.order):
            feedback_msg.partial_sequence.append(
                feedback_msg.partial_sequence[i] + feedback_msg.
   partial_sequence[i-1])
```

```
            self.get_logger().info('Feedback: {0}'.format(feedback_msg.
    partial_sequence))
            goal_handle.publish_feedback(feedback_msg)
            time.sleep(1)

        goal_handle.succeed()

        result = Fibonacci.Result()
        result.sequence = feedback_msg.partial_sequence
        return result
```

Now the "main" part is simple as usual:

```
def main(args=None):
    rclpy.init(args=args)
    fibonacci_action_server = FibonacciActionServer()
    rclpy.spin(fibonacci_action_server)
```

Let's move on to the client part, for which we create the "actionclient.py" file.

```
import sys
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node

from action_tutorials_interfaces.action import Fibonacci


class FibonacciActionClient(Node):

    def __init__(self):
        pass

    def send_goal(self, order):
        pass

    def goal_response_callback(self, future):
        pass

    def get_result_callback(self, future):
        pass

    def feedback_callback(self, feedback_msg):
        pass

def main(args=None):
    pass


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

Now we can see, as actions are the most complex of the communication protocols, the code is the most complex as well. We will have an initializer, a goal sending, and three callback functions (one for goal response, result, and feedback each). Now let's fill in the initializer first:

```
def __init__(self):
    super().__init__('fibonacci_action_client')
    self._action_client = ActionClient(self, Fibonacci, 'firstaction')
```

Nothing fancy here, we take the 'Node' class as a basis and create an action client with the "Fibonacci2 type and the "firstaction" action name. Now, the interesting things start only after this.

First, we need to be able to send a goal. Now we make an instance of the Goal, and we add the value of it, followed by the waiting process for the action server to be available. Now we can send a goal, we will do this asynchronously, and also register a callback function called "goal_response_callback" for the future that we will get back.

```
def send_goal(self, order):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order

        self._action_client.wait_for_server()

        self._send_goal_future = self._action_client.send_goal_async(
    goal_msg, feedback_callback=self.feedback_callback)

        self._send_goal_future.add_done_callback(self.
    goal_response_callback)
```

Now this future is completed when the action server either accepts or rejects the goal request. This is exactly what is happening here, we simply log out the answer, and if we get a reject, then we return from the execution.

```
def goal_response_callback(self, future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().info('Goal rejected :(')
            return

        self.get_logger().info('Goal accepted :)')

        self._get_result_future = goal_handle.get_result_async()
        self._get_result_future.add_done_callback(self.
    get_result_callback)
```

Now, that we have a goal handler we can request the result, which is the last two lines of this code. We will do so by again registering a callback function but now for the result, called "get_result_callback":

```
def get_result_callback(self, future):
        result = future.result().result
        self.get_logger().info('Result: {0}'.format(result.sequence))
        rclpy.shutdown()
```

In this callback, we log out the result and shut down for a clean termination.

In the "send_goal" method, when we were sending the goal, we also declared that we wanted a callback function for the feedback as well, this "feedback_callback" logs the feedback to the terminal:

```
def feedback_callback(self, feedback_msg):
```

```
        feedback = feedback_msg.feedback
        self.get_logger().info('Received feedback: {0}'.format(feedback.
    partial_sequence))
```

Finally, the main part is straightforward at this point:

```
def main(args=None):
    rclpy.init(args=args)
    action_client = FibonacciActionClient()
    action_client.send_goal(int(sys.argv[1]))
    rclpy.spin(action_client)
```

Now that the client is finished as well, we need to ad the entry points to our 'setup.py' file:

```
entry_points={
    'console_scripts': [
        'acserver = actionsrvcli.actionserver:main',
        'acclient = actionsrvcli.actionclient:main',

    ],
},
```

After building the package and running the nodes in two tabs, we can try this out. It is a good practice to use predefined messages and services, however there is occasions when You will need to define Your own. Let's see how to do so:.

5.4 Customizing the messages and services

We will create a new package to make our custom messages and services. Let's create it (in 'src'):

```
ros2 pkg create --build-type ament_cmake first_interfaces
```

As You can see how we are not making a Python-based package but a CMAKE based one (these types of packages can be only made in this style). Now important to bear in mind, that we need to make an "msg" and "srv" directory to store our definitions (this is the convention, to make the build happen). If we done it, we can go to make our definition for the message, inside "msg" dictionary make a file named "Num.msg":

```
int64 num
```

Now this can send a number from one node to another. Let's make another, where we will build on an already defined structure, let's call it "Sphere.msg":

```
geometry_msgs/Point center
float64 radius
```

We build upon the 'geometry_msgs's' 'Point' message definition and use it as 'center'. Now let's also make a service definition inside 'srv' called "AddThreeInts.srv":

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

As from the name and the structure is visible, we will use this kind of service to add three numbers "a","b", and "c", and give back their "sum".

The definitions are ready, to make them usable we need to tell the compiler how to handle them. We can do this inside the "CMakeLists.txt". Now we add the following lines (complete file: A.2.7):

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES geometry_msgs # Add packages that above messages depend
   on, in this case geometry_msgs for Sphere.msg
)
```

We need to add the following dependencies to the 'package.xml as well':

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

And we can build our package. After sourcing we can confirm the creation of messages and services with the 'interface show':

```
$ ros2 interface show first_interfaces/msg/Num
```

If we get the definitions back and not an error message, then we are good.

Now we will create a new package called "interface_tests":

```
$ ros2 pkg create --build-type ament_python interface_tests --
   dependencies rclpy first_interfaces
```

To test the 'Num' type we will create a Publisher/Subsrciber system and for the 'AddThreeInts' a Service server/client system. For this, we will make four nodes in python. Inside the package correspondent directory make four python files with the following names: "testpub.py", "testsub.py", "testsrv.py", "testcli.py". You can copy and paste them from the appendix: A.2.8, A.2.9, A.2.10, A.2.11 respectively, and the 'setup.py' A.2.12. After that we build, and voila, we can run our nodes.

This concludes our learning of communication, we have now a solid foundation, but there are many more deails to learn about for which You can refer to the tutorial section and concepts section of the docs of ROS2[1].

[1]Foxy docs