Developing a ROS 2 Service package

Kheng Lee Koay

February 10, 2026

Contents

Chapter 1 Developing a ROS 2 Service package

ROS 2 services enable nodes to communicate using a call-and-response model versus the publisher-subscriber model of topics. This makes them ideal for tasks requiring immediate data or action before continuing. This pattern is particularly useful for querying sensor data, requesting computations, or synchronously controlling devices.

1.1 Implementing ROS 2 Services in Robotics

This tutorial will guide you through implementing a simple ROS 2 service that takes two integers and returns their sum.

Creating a ROS 2 service involves three main steps:

1. **Define the Service**
   A `.srv` file is used to define the request (inputs) and response (outputs) of a service in ROS 2. This file specifies the structure of the data exchanged between the client and server.
2. **Implement the Service Server**
   The service server responsible for handling client requests. It processes the input, performs computations (such as reading sensor data, executing algorithms, or sending commands to hardware), and returns the response.
3. **Implement the Service Client**
   The service client sends requests to the server when data retrieval, computation, or an action is required. It then waits for a response before proceeding with further execution.

1.1.1 Setting Up Your ROS 2 Workspace

Before creating the service, ensure your ROS 2 environment is properly set up.

1. **Source the ROS 2 Environment**
   If you haven't added the ROS 2 setup script to your shell startup scrip, i.e. `~/.bashrc`, manually source the ROS 2 setup script using the command below. This ensures your shell knows where to find ROS 2 commands and the default packages:

```
~$ source /opt/ros/humble/setup.bash
```

2. **Create ROS 2 Workspace**
   If you don't already have a ROS 2 workspace, create one. By convention, the workspace is typically named `ros2_ws` and include a `src` subdirectory to store packages.

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

3. **Build the Workspace**

   Use `colcon` in your workspace directory to build your workspace.

```
~/Workspaces/ros2_ws$ colcon build
```

1.1.2 Creating a ROS 2 Package

ROS 2 packages are the basic building blocks for organising and distributing ROS 2 software. Each package groups together nodes, scripts, and configuration files needed for a specific function or application. Therefore, you will begin by creating a new ROS 2 package called `my_srvcli` in your workspace directory. This package will be used to organise and store your Service and Client nodes.

1. **Navigate to your workspace's `src` directory:**

```
~$ cd ~/Workspaces/ros2_ws/src
```

2. **Create the Package:** You will use the `ros2 pkg create` command to create a package named `my_srvcli`, which will be built using `ament_python` (i.e., standard Python workflow to build Python package) and the package will depend on `rclpy` and `example_interfaces` packages. This creates a new package with dependencies required for our ROS 2 service.

```
~/Workspaces/ros2_ws/src$ ros2 pkg create --build-type ament_python --\
   license Apache-2.0 my_srvcli --dependencies rclpy example_interfaces
```

1.1.3 Using the `AddTwoInts.srv` Service File

ROS 2 services use `.srv` files to define the `request` and `response` formats for ROS 2 Services Server. For our `my_srvcli` package, we will use the `AddTwoInts.srv` definition file from the `example_interfaces` package to define the service interface for our `add_two_ints` Service Server. The `AddTwoInts.srv` service definition file has the following contents:

```
int64 a
int64 b
```

```
---
int64 sum
```

Note that everything before the `---` line represents the fields of the `request` message and everything after represents the fields of `response` message. In this example, the request field consists of two `int64` integers (`a` and `b`) and the response field consists of type is a single `int64` integer (`sum`).

1.1.4 Implement the Service Server

The service server processes incoming requests and returns responses. In our example, it will receive two integers, add them, and return the sum.

1. **Create the Server Script**

   Navigate to the `my_srvcli` directory in the package directory and create a file named `add_two_ints_server.py`

   ```
   ~/Workspaces/ros2_ws/src$ cd my_srvcli/my_srvcli
   ~/Workspaces/ros2_ws/src/my_srvcli/my_srvcli$ touch \
       add_two_ints_server.py
   ```

2. **Populate the Server Script**

   Study and implement the `add_two_ints_server.py` code below carefully as it explains and shows how to add the necessary imports, initialise a ROS 2 node, and define the service callback function. The callback function is where you define how the service processes requests.

   You can open the `add_two_ints_server.py` in any text editor to implement the service's code.

   For VS Code (code), go to the File > Open Folder, then navigate to you ROS 2 workspace src directory (e.g. `~/Workspaces/ros2_ws/src`), select the `my_srvcli` package and click the Open button on the top right. Now under the EXPLORER column, click the `my_srvcli` package to show its contents. Expand the `my_srvcli` folder and click the `add_two_ints_server.py` file to open it.

   You can get a copy of the original ROS 2 Service's code (without comments) from the official ROS 2 (Humble) tutorial on Writing a simple service and client. Consult the official tutorial for additional information.

```python
# Import the necessary ROS 2 and Python packages
from example_interfaces.srv import AddTwoInts # Make sure this matches the
    ↪ name of your .srv file and package
import rclpy
```

```python
from rclpy.node import Node

# Define the service server class, inheriting from Node
class AddTwoIntsServer(Node):
    def __init__(self):
        # Initialize the node with the name 'add_two_ints_server'
        super().__init__('add_two_ints_server')
        # Create the service named 'add_two_ints' using the AddTwoInts service
            ↪  type
        # 'add_two_ints_callback' is the method that will be called when a
            ↪  request is received
        self.service = self.create_service(AddTwoInts, 'add_two_ints', self.
            ↪  add_two_ints_callback)
        self.get_logger().info('Service is running..')

    # This function processes the service request.
    def add_two_ints_callback(self, request, response):
        # The request contains two integers 'a' and 'b'.
        # The response will contain the sum of these integers.
        response.sum = request.a + request.b
        # Log the request and response
        self.get_logger().info(f'Receiving request: a={request.a}, b={request.
            ↪  b} (sum: {response.sum})')
        # Return the response object, which contains the sum
        return response

def main(args=None):
    # Initialize the ROS client library
    rclpy.init(args=args)
    # Create an instance of your service server
    server = AddTwoIntsServer()
    # Spin the node so the callback function is called.
    # This will keep your service server alive and responsive to incoming
        ↪  service requests.
    rclpy.spin(server)
    # Shutdown the ROS client library before exiting the program
    rclpy.shutdown()
```

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

Listing 1.1. add_two_ints_server.py

1.1.5 Implement the Service Client

The service client sends requests to the server and waits for a response. In this example, the request consists of two int64 integers.

1. **Crete the Client Script**
   Create a Python file named `add_two_ints_client.py` in the same directory as the server script. This script will act as the client node, sending requests to the `AddTwoInts` service.
2. **Populate the Client Script**
   The client script should import necessary modules, initialise a ROS 2 node, and send a service request. It must also handle the response message received from the service server.
   Study the implementation of the `add_two_ints_server.py` code below. It explains and shows how to create a service client class, setting up and sending a request message, waiting and listening for the service to respond, receiving and retrieving the respond message.
   Open the `add_two_ints_client.py` in any text editor and implement the client's code. You can get a copy of the original Client's code (without comments) from the official ROS 2 (Humble) tutorial on Writing a simple service and client. Consult the official tutorial for additional information

```
# Import the necessary ROS 2 and Python packages
import sys
from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node

# Define the service client class, inheriting from Node
class AddTwoIntsClient(Node):
    def __init__(self):
        # Initialise the node witht he name 'add_two_ints_client'
        super().__init__('add_two_ints_client')
        # Create the client node with the same type (AddTwoInts) and name (
            ↪ add_two_ints} as service node
```

```
        self.client = self.create_client(AddTwoInts, 'add_two_ints')
        # Check and wait for the sevice with the same type and name to become
            ↪  available
        while not self.client.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Service not available, waiting again...')

    def send_request(self, a, b):
        # Create a new request with the integers to be added
        req = AddTwoInts.Request()
        req.a = a
        req.b = b
        # Async call to the service
        self.future = self.client.call_async(req)

def main():
    # Initialise the ROS client library
    rclpy.init()
    if len(sys.argv) == 3:
        try:
            # get the passed-in command-line arguments
            a = int(sys.argv[1])
            b = int(sys.argv[2])
        except ValueError:
            print('Usage: add_two_ints_client.py <int> <int>')
            sys.exit(1)

        # Create an instance of the service client
        client = AddTwoIntsClient()
        # Send a request with the provided integers
        client.send_request(a,b)

        #Wait fo the response from the service server
        while rclpy.ok():
            rclpy.spin_once(client)
            if client.future.done():
                try:
                    response = client.future.result()
                except Exception as e:
```

```
                client.get_logger().info('Service call failed %r' % (e,))
            else:
                client.get_logger().info(f'Resulf of add_two_ints: {a} + {b}
                    ↪   = {response.sum}')
                break
    else:
        print('Usage: add_two_ints_client.py <int> <int>')
        sys.exit(1)

    # Shutdown the ROS client library
    rclpy.shutdown()

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

Listing 1.2. add_two_ints_client.py

1.1.6 Updating setup.py

To make ROS 2 recognise the scripts, modify the setup.py file in your package directory (`~/Workspaces/ros2_ws/src/my_srvcli/setup.py`). Find the `entry_point` section and add the '`add_two_ints_server=my_srvcli.add_two_ints_server:main`' and '`add_two_ints_client=my_srvcli.add_two_ints_client:main`' entries as shown below:

```
entry_points={
    'console_scripts': [
        'add_two_ints_server=my_srvcli.add_two_ints_server:main',
        'add_two_ints_client=my_srvcli.add_two_ints_client:main',
    ],
},
```

Listing 1.3. setup.py

1.1.7 Building and Running Your Service

1. **Check for and install missing dependencies**
   Run rosdep in the root of your ROS 2 workspace (`~/Workspaces/ros2_ws/`) to check for missing dependencies before building your package.

```
~/Workspaces/ros2_ws$ rosdep install -i --from-path src --rosdistro \
   humble -y
```

2. **Build the Package**

   In the root of your ROS 2 workspace run colcon build.

```
~/Workspaces/ros2_ws$ colcon build --packages-select my_srvcli
```

3. **Source the Workspace**

   After building, source your workspace to have access your newly built package.

```
~/Workspaces/ros2_ws$ source install/setup.bash
```

4. **Run the Service and Client**

   Open two terminals and source your ROS 2 Environment and Workspace in each before proceeding. In one terminal, launch your service server.

```
~$ ros2 run my_srvcli add_two_ints_server
```

   In another terminal, run your service client with two integers as arguments. Observe how the server receives the request, processes it, and returns the sum to the client

```
~$ ros2 run my_srvcli add_two_ints_client 5 3
```

   There are multiple ways to test your service. One common method is to call it from the terminal using the following command format:

```
   ros2 service call <service_name> <service_type> "{<arguments>}"
```

   Below is an example of making a service call with arguments `a: 1, b: 2`.

```
~$ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts \
   "{a: 1, b: 2}"
```

   Can you think of another way? Hint: rqt!

1.2 Conclusion

You have successfully implemented a ROS 2 service that takes two integers, computes their sum, and returns the result. This fundamental example demonstrates the request-response pattern in ROS 2, which is crucial for handling computations, querying sensor data, or controlling devices synchronously.

1.3 Homework

1.3.1 Homework I: Custom ROS 2 Service Interface

This tutorial demonstrates how to define a custom service `AddThreeInts.srv` in a separate package i.e. `my_custom_package` that can then be used by other packages like how we utilise the the `AddTwoInts.srv` definition file from the `example_interfaces` package to build our `my_srvcli` package.

1.3.1.1 Creating a Custom Service Interface package

1. Navigate to the `src` folder within your workspace and proceed to create a new package named `my_custom_interfaces`. Keep in mind that when generating custom messages (`.msg`) and services messages (`.srv`), we will need to use the `ament_cmake` build type. However, there are no constraints on other packages to utilise these custom interfaces even if those packages utilise a different build type such as `amen_python`.

```
~$ cd ~/Workspaces/ros2_ws/src
~/Workspaces/ros2_ws/src$ ros2 pkg create --build-type ament_cmake --\
   license Apache-2.0 my_custom_interfaces
```

2. Create a `srv` directory in our package directory to store our service interface file (`AddThreeInts.srv`)

```
~/Workspaces/ros2_ws/src$ cd my_custom_interfaces
~/Workspaces/ros2_ws/src/my_custom_interfaces$ mkdir srv
~/Workspaces/ros2_ws/src/my_custom_interfaces$ cd srv
~/Workspaces/ros2_ws/src/my_custom_interfaces/srv$ touch \
   AddThreeInts.srv
```

3. Edit the `AddThreeInts.srv` file and add the following request and response structure:

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

Listing 1.4. Content of AddThreeInts.srv

4. Add the following lines to the package's `CMakeLists.txt` to enable ROS Interface Definition Language (`ROSIDL`) code generation. This will generate language-specific code (e.g., C++ and Python) from the `.srv` service definitions, making the service interfaces available for use in both languages.

```
find_package(rosidl_default_generators REQUIRED)

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

Listing 1.5. Enabling automatic code generation for ROS interfaces using `ROSIDL` in `CMakeLists.txt`

5. Add the following options

   `rosidl_default_generators`: for generating language-specific code

   `rosidl_default_runtime`: for using the interface during runtime

   `rosidl_interface_packages`: to associate our package the interface packages

   to the `package.xml` file as shown below:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<depend>action_msgs</depend>
```

Listing 1.6. Declaring the `ROSIDL` dependency in `package.xml`

6. Check/install missing dependencies then build the package with the following commands:

```
~/Workspaces/ros2_ws$ rosdep install -i --from-path src --rosdistro \
   humble -y
~/Workspaces/ros2_ws$ colcon build --packages-select \
   my_custom_interfaces
```

7. Source the ROS 2 workspace and make sure our new service has been associated with the `ros2 interface` packages:

```
~/Workspaces/ros2_ws$ source install/setup.bash
~/Workspaces/ros2_ws$ ros2 interface show \
   my_custom_interfaces/srv/AddThreeInts
```

1.3.1.2 Create service nodes with custom interface

To use the new service interface `AddThreeInts`, you need to import it from the `my_custom_interfaces` package in your Python script:

```
from my_custom_interfaces.srv import AddThreeInts # import AddThreeInts.srv
    ↪ from my_custom_interface package
```

Listing 1.7. `add_three_ints_server.py`

Next, add `my_custom_interfaces` as an executable dependency in your package's `package.xml` file by inserting the following line:

```
<depend>my_custom_interfaces</depend>
```

Listing 1.8. Adding `my_custom_interfaces` to the dependency list in `package.xml` to enable usage of its custom message/service definitions.

Afterward, run the following command to install any missing dependencies:

```
~/Workspaces/ros2_ws$ rosdep install -i --from-path src --rosdistro \
   humble -y
```

Once dependencies are resolved, build your package:

```
~/Workspaces/ros2_ws$ colcon build --packages-select <yourpackagename>
```

For more details on creating custom ROS 2 interfaces (.msg and .srv), refer to the following resources:

- ✿ Creating custom msg and srv files
- ✿ Implementing custom interfaces

1.3.2 Homework II: Virtual Robot Manipulator

In this assignment, you will develop a virtual control system for a robot manipulator using ROS. The focus is on service-based client-server interaction, where a client sends commands and a server simulates the manipulator's response.

You will implement the following components:

A Service server node that acts as the virtual robot arm should:

- ✿ Receive and process commands sent by the client (e.g., target positions for the manipulator).
- ✿ Simulate the outcome of the requested manipulator operation (e.g., calculating motion or reaching a position).

- ✿ Send back a response that includes the result or status of the simulated operation (e.g., success/failure and a message).

A Service client node that acts as the controller to:

- ✿ Sends commands (requests) to the service server.
- ✿ Waits for and receives the server's response.
- ✿ Handles the response appropriately (e.g., prints success/failure or takes further action).

Your implementation should demonstrate how the client and the server communicate using ROS services, and how the server simulates a manipulator's behavior based on the client's requests.

The interaction should follow this sequence: Client requests an action → Server processes it → Simulates manipulator behavior → Sends back a result.

Activity 1.1

Create a Custom Service Interface for the Robot Manipulator

Create a manipulator interface package with a custom ROS 2 service definition file, `ManipulatorControl.srv`. The service request should include an operation type (e.g., `move_to`, `grasp`, `release`) and any required target parameters (e.g., position coordinates). The response should include a status field indicating whether the operation succeeded or failed, along with any relevant result data.

Activity 1.2

Create a Manipulator Service package

Develop a Service Server Node : Implement a server node in Python that offers the `ManipulatorControl` service. This node should simulate the manipulator's behavior by processing incoming requests and determining outcomes based on predefined scenarios. For example, if a `move_to` command is received, the server should simulate the manipulator attempting to move the gripper to the specified position and return a success or failure status. For `grasp` or `release` operations, the server should update the gripper's state accordingly.

Develop a Service Client Node : Implement a Python service client node that communicates with a manipulator service server by sending control instructions based on user input. The client should parse the desired operation and its arguments from the command-line input (i.e. `argv`). It must send the formatted request to the server, handle the server's response appropriately, and

clearly display the result of the operation to the user.

Activity 1.3

Testing: Ensure that the server and client nodes can communicate correctly within the ROS 2 ecosystem. Test the system by executing a series of operations, such as moving the manipulator to different positions, grasping objects, and releasing them. Confirm that the client receives and displays the correct responses for each action.

1.4 Appendix: Service Interface with Existing Message Type

This appendix demonstrates how to incorporate an existing message types (interface definitions) into a custom ROS 2 service definition.

1.4.1 Integrating Point Messages Type

ROS 2 provides a variety of standard message types that can be reused in your own custom interfaces. One such example is `Point.msg` from the `geometry_msgs` package, which includes three fields: `x`, `y`, and `z`, commonly used to represent positions or coordinates in 3D space.

In this section, we will demonstrate how to use `Point.msg` as part of a service request in your simple manipulator control service (developed for HW2). Reusing standard messages like `Point` allows you to build more expressive and interoperable service definitions while reducing redundancy and ensuring consistency with other ROS 2 components.

Below is an example of how to define a `position` service message in a `.srv` file.

First, we declare a request field named `position`, using the `Point` message type from the `geometry_msgs` package.

```
geometry_msgs/Point position
```

Listing 1.9. An example on creating a `position` request field.

To use the `Point.msg` message type in our package, we need to declare a dependency on the `geometry_msgs` package. This is done by adding the following lines to our`package.xml` file.

```
<build_depend>geometry_msgs</build_depend>
<exec_depend>geometry_msgs</exec_depend>
```

Listing 1.10. Declaring `geometry_msgs` as a dependency in `package.xml`.

Since the package requires geometry_msgs at both build-time and execution-time, we can simplify the dependency declaration by using a single <depend> tag, which covers both. This is especially recommended in ROS 2, where <depend> replaces the need for separate <build_depend> and <exec_depend> tags.

```
<depend>geometry_msgs</depend>
```

Listing 1.11. Replacing separate build-time and execution-time dependency declarations with a single <depend> tag for the geometry_msgs package in package.xml.

To ensures that CMake can locate the geometry_msgs package during the build process, you must add the following entry to your package's CMakeLists.txt. This line instructs CMake to find the geometry_msgs package and configure the build environment accordingly:

```
find_package(geometry_msgs REQUIRED)
```

Listing 1.12. Locate and setup geometry_msgs package

Additionally, in the same CMakeLists.txt file, you need to declare that the ManipulatorControl.srv service definition depends on the geometry_msgs package, as it uses the Point message type from this package. This is done using the DEPENDENCIES keyword, followed by the package name geometry_msgs, as shown below:

```
rosidl_generate_interfaces(\${PROJECT_NAME}
    "srv/ManipulatorControl.srv"
    DEPENDENCIES geometry_msgs
)
```

Listing 1.13. Declare the geometry_msgs dependency

This step ensures that the necessary code for handling messages from geometry_msgs is included when generating the service interface, enabling your service to use message types like Point properly.

1.4.2 Creating Waypoints with Point Message Type

In ROS, you can use Point.msg to define a waypoints message , which is essentially a list (or an array) of Point messages. This allows you to represent multiple positions (each with x, y, and z coordinates) as a collection in your custom message. This is useful for storing or sending a list of waypoints for a robot to follow. Add the following line to the request section of your .srv file to define a waypoints field that holds a list of Point messages:

```
geometry_msgs/Point[] waypoints
```

Listing 1.14. An example on creating a waypoints request field.

This allows the service request to include multiple 3D coordinates (each with `x`, `y`, and `z` coordinate) for the manipulator to follow and allows your service to handle multiple waypoints efficiently. You do not need to add any new entries to the package's `package.xml` and `CMakeLists.txt`, as the necessary configurations for the `geometry_msg` package were already set up when we included `Point.msg` earlier to define the `position` field. To construct and manage a list of `waypoints` for your service client, you need to import the `Point` message from the `geometry_msgs` package into your Python script:

```
from geometry_msgs.msg import Point
```

Listing 1.15. An example of importing `Point` message type into Pyton script.

Then create an empty list to store your waypoints:

```
waypoints = []
```

Listing 1.16. An example on creating an empty list.

After that you can construct each waypoint as a Point message type and then append it to the waypoints list as shown in the example below:

```
waypoint = Point()
waypoint.x = 1.0
waypoint.y = 1.0
waypoint.z = 1.0
waypoints.append(waypoint)
```

Listing 1.17. An example on creating a list of waypoints.

Once you have completed waypoints, assign the list to the service request:

```
req.waypoints = waypoints
```

Listing 1.18. Assigned waypoints to service request message.

This approach allows you to dynamically create and populate a list of `waypoints` using the `Point` message to be included in your service request. See the link below for more information on this. Use an existing interface definition.

Bibliography