Writing an action server and client (Python)

Goal: Implement an action server and client in Python.

Tutorial level: Intermediate

Time: 15 minutes

Contents

Background

Actions are a form of asynchronous communication in ROS 2. *Action clients* send goal requests to *action servers*. *Action servers* send goal feedback and results to *action clients*.

Prerequisites

You will need the `action_tutorials_interfaces` package and the `Fibonacci.action` interface defined in the previous tutorial, Creating an action.

Tasks

1 Writing an action server

Let's focus on writing an action server that computes the Fibonacci sequence using the action we created in the Creating an action tutorial.

Until now, you've created packages and used `ros2 run` to run your nodes. To keep things simple in this tutorial, however, we'll scope the action server to a single file. If you'd like to see what a complete package for the actions tutorials looks like, check out action_tutorials.

Open a new file in your home directory, let's call it `fibonacci_action_server.py`, and add the following code:

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

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        result = Fibonacci.Result()
        return result


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

    fibonacci_action_server = FibonacciActionServer()

    rclpy.spin(fibonacci_action_server)


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

Line 8 defines a class `FibonacciActionServer` that is a subclass of `Node`. The class is initialized by calling the `Node` constructor, naming our node `fibonacci_action_server`:

```
        super().__init__('fibonacci_action_server')
```

In the constructor we also instantiate a new action server:

```
        self._action_server = ActionServer(
            self,
            Fibonacci,
            'fibonacci',
            self.execute_callback)
```

An action server requires four arguments:

1. A ROS 2 node to add the action client to: `self`.
2. The type of the action: `Fibonacci` (imported in line 5).
3. The action name: `'fibonacci'`.
4. A callback function for executing accepted goals: `self.execute_callback`. This callback **must** return a result message for the action type.

We also define an `execute_callback` method in our class:

```
    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        result = Fibonacci.Result()
        return result
```

This is the method that will be called to execute a goal once it is accepted.

Let's try running our action server:

Linux | macOS | Windows

```
python3 fibonacci_action_server.py
```

In another terminal, we can use the command line interface to send a goal:

```
ros2 action send_goal fibonacci action_tutorials_interfaces/action/Fibonacci "{order: 5}"
```

In the terminal that is running the action server, you should see a logged message “Executing goal...” followed by a warning that the goal state was not set. By default, if the goal handle state is not set in the execute callback it assumes the *aborted* state.

We can use the method succeed() on the goal handle to indicate that the goal was successful:

```
    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        goal_handle.succeed()
        result = Fibonacci.Result()
        return result
```

Now if you restart the action server and send another goal, you should see the goal finished with the status `SUCCEEDED`.

Now let’s make our goal execution actually compute and return the requested Fibonacci sequence:

```
    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()
        result.sequence = sequence
        return result
```

After computing the sequence, we assign it to the result message field before returning.

Again, restart the action server and send another goal. You should see the goal finish with the proper result sequence.

1.2 Publishing feedback

One of the nice things about actions is the ability to provide feedback to an action client during goal execution. We can make our action server publish feedback for action clients by calling the goal handle’s publish_feedback() method.

We'll replace the `sequence` variable, and use a feedback message to store the sequence instead. After every update of the feedback message in the for-loop, we publish the feedback message and sleep for dramatic effect:

```
import time

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):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(
            self,
            Fibonacci,
            'fibonacci',
            self.execute_callback)

    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


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

    fibonacci_action_server = FibonacciActionServer()

    rclpy.spin(fibonacci_action_server)


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

After restarting the action server, we can confirm that feedback is now published by using the command line tool with the `--feedback` option:

```
ros2 action send_goal --feedback fibonacci action_tutorials_interfaces/action/Fibonacci "{order:
5}"
```

2 Writing an action client

We'll also scope the action client to a single file. Open a new file, let's call it `fibonacci_action_client.py`, and add the following boilerplate code:

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

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

        self._action_client.wait_for_server()

        return self._action_client.send_goal_async(goal_msg)


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

    action_client = FibonacciActionClient()

    future = action_client.send_goal(10)

    rclpy.spin_until_future_complete(action_client, future)


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

We've defined a class `FibonacciActionClient` that is a subclass of `Node`. The class is initialized by calling the `Node` constructor, naming our node `fibonacci_action_client`:

```
        super().__init__('fibonacci_action_client')
```

Also in the class constructor, we create an action client using the custom action definition from the previous tutorial on Creating an action:

```
        self._action_client = ActionClient(self, Fibonacci, 'fibonacci')
```

We create an `ActionClient` by passing it three arguments:

1. A ROS 2 node to add the action client to: `self`
2. The type of the action: `Fibonacci`
3. The action name: `'fibonacci'`

Our action client will be able to communicate with action servers of the same action name and type.

We also define a method `send_goal` in the `FibonacciActionClient` class:

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

        self._action_client.wait_for_server()

        return self._action_client.send_goal_async(goal_msg)
```

This method waits for the action server to be available, then sends a goal to the server. It returns a future that we can later wait on.

After the class definition, we define a function `main()` that initializes ROS 2 and creates an instance of our `FibonacciActionClient` node. It then sends a goal and waits until that goal has been completed.

Finally, we call `main()` in the entry point of our Python program.

Let's test our action client by first running the action server built earlier:

Linux | macOS | Windows

```
python3 fibonacci_action_server.py
```

In another terminal, run the action client:

Linux | macOS | Windows

```
python3 fibonacci_action_client.py
```

You should see messages printed by the action server as it successfully executes the goal:

```
[INFO] [fibonacci_action_server]: Executing goal...
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1])
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2])
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3, 5])
# etc.
```

The action client should start up, and then quickly finish. At this point, we have a functioning action client, but we don't see any results or get any feedback.

2.1 Getting a result

So we can send a goal, but how do we know when it is completed? We can get the result information with a couple steps. First, we need to get a goal handle for the goal we sent. Then, we can use the goal handle to request the result.

Here's the complete code for this example:

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

    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)

        self._send_goal_future.add_done_callback(self.goal_response_callback)

    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)

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


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

    action_client = FibonacciActionClient()

    action_client.send_goal(10)

    rclpy.spin(action_client)


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

The ActionClient.send_goal_async() method returns a future to a goal handle. First we register a callback for when the future is complete:

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

Note that the future is completed when an action server accepts or rejects the goal request. Let's look at the `goal_response_callback` in more detail. We can check to see if the goal was rejected and return early since we know there will be no result:

```
    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 :)')
```

Now that we've got a goal handle, we can use it to request the result with the method get_result_async(). Similar to sending the goal, we will get a future that will complete when the result is ready. Let's register a callback just like we did for the goal response:

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

In the callback, we log the result sequence and shutdown ROS 2 for a clean exit:

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

With an action server running in a separate terminal, go ahead and try running our Fibonacci action client!

Linux | macOS | Windows

```
python3 fibonacci_action_client.py
```

You should see logged messages for the goal being accepted and the final result.

2.2 Getting feedback

Our action client can send goals. Nice! But it would be great if we could get some feedback about the goals we send from the action server.

Here's the complete code for this example:

```python
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):
        super().__init__('fibonacci_action_client')
        self._action_client = ActionClient(self, Fibonacci, 'fibonacci')

    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)

    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)

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

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


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

    action_client = FibonacciActionClient()

    action_client.send_goal(10)

    rclpy.spin(action_client)


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

Here's the callback function for feedback messages:

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

In the callback we get the feedback portion of the message and print the `partial_sequence` field to the screen.

We need to register the callback with the action client. This is achieved by additionally passing the callback to the action client when we send a goal:

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

We're all set. If we run our action client, you should see feedback being printed to the screen.

Summary

In this tutorial, you put together a Python action server and action client line by line, and configured them to exchange goals, feedback, and results.

Related content

- There are several ways you could write an action server and client in Python; check out the `minimal_action_server` and `minimal_action_client` packages in the ros2/examples repo.
- For more detailed information about ROS actions, please refer to the design article.