UNIVERSITY OF OXFORD

OxRAM
Oxford Robotics & Additive Manufacturing Society

OxARM

Software Lecture 5:

ROS Actions

Jacques Cloete

Contents

In this lecture, we shall cover:

- Overview for ROS Actions
- Defining a new ROS Action
- Brief introduction to Object-Oriented Programming
- Creating our first ROS Action Client/Server pair
- Brief introduction to launch files

Before We Begin

- Again, I strongly suggest bookmarking the following link: https://github.com/OxRAMSociety/RobotArm

- All example code can be found in:

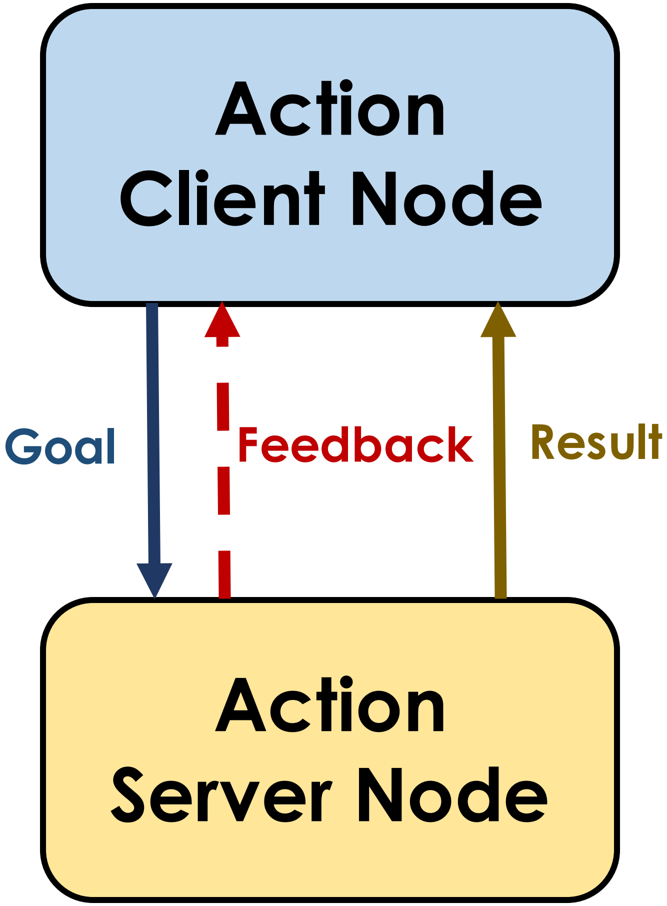
Tutorials/Software Tutorials (2022)/Example Scripts

Of course, code for this lecture will be in the Lecture 5 folder, and so on for future lectures

- I will highlight some code in these presentations, but you should refer to the Example Scripts for the entire code

ROS Actions – Overview

- Request/response system, with feedback!
- Client node sends a goal and awaits a result
- Server node listens to the goal, processes it (while providing feedback) and sends a result back to that client when (eventually) complete
- Much better suited for processes that take an extended period of time (for which feedback would be useful)



Creating an Action Client/Server Pair

- Consider some abstract action that we may want to request
- For this example:
  - We shall specify for how many seconds that action should be carried out in our goal
  - We also desire feedback telling us the number of seconds for which the action has been carried out
  - Finally, we want to receive a result that tells us whether the action was completed successfully

Defining a new ROS Action

- ROS actions must be defined in a unique 'action' file
- **Navigate to tutorial_scripts and create a folder named action**
  - This folder will contain all our custom ROS action definitions
- **Inside that folder, create a file Example.action and open it**

I didn't call it ExampleAction.action for a reason that will become apparent later...

Defining a new ROS Action

- We first define all data contained within the goal message
  - Same as for ROS Services
- We then define all data contained within the result message
- Finally, we define all data contained within the feedback message
- We separate these three groups with 3 hyphens (i.e. ---)

```
# Goal
int32 seconds_requested

---
# Result
bool success

---
# Feedback
int32 seconds_elapsed
```

Reproduce this!

Defining a new ROS Action

- For us to be able to use our new action, we must first enable functionality for ROS actions in our package

- **Navigate to tutorial_scripts and open the CMakeLists.txt file**

- **Go to line ~10 and uncomment/edit the find_package section as follows:**

```
10  find_package(catkin REQUIRED COMPONENTS
11    rospy
12    std_msgs
13    message_generation
14    actionlib_msgs
15  )
```

Defining a new ROS Action

- **Go to line ~60 and uncomment/edit the add_action_files and generate_messages sections as follows:**

```
63 ## Generate actions in the 'action' folder
64 add_action_files(
65   FILES
66   Example.action
67 )
68
69 ## Generate added messages and services with any dependencies listed here
70 generate_messages(
71   DEPENDENCIES
72   std_msgs
73   actionlib_msgs
74 )
```

Defining a new ROS Action

- **Go to line ~100 and uncomment/edit the catkin_package section as follows:**

```
105   catkin_package(
106   #  INCLUDE_DIRS include
107      LIBRARIES tutorial_scripts
108      CATKIN_DEPENDS rospy std_msgs actionlib_msgs
109   #  DEPENDS system_lib
110      DEPENDS message_runtime
111   )
```

Defining a new ROS Action

- **Now open the package.xml file, go to line ~58 and add/edit the following (note: we have already added the first two lines):**

```xml
59    <build_depend>message_generation</build_depend>
60    <exec_depend>message_runtime</exec_depend>
61
62    <build_depend>actionlib_msgs</build_depend>
63    <build_export_depend>actionlib_msgs</build_export_depend>
64    <exec_depend>actionlib_msgs</exec_depend>
```

- **Finally, save all files and rebuild the workspace**

Creating an Action Server Node

- **In tutorial_scripts' scripts folder, create the file Example_Action_Server.py**
- From now on, these scripts will be too large to display in their entirety on these slides
- Therefore, find at them in the GitHub repository while following along to these lectures
- I will continue to highlight and explain key points

An Aside – Object-Oriented Programming

- You will see that our action server is implemented as an object
- Objects are complex data structures with their own internal data and methods
- Classes are the blueprints for these objects, and define these data and methods
- This is the basis for Object-Oriented Programming (OOP), a very popular and intuitive programming paradigm

Creating an Action Server Node

- The initialiser function automatically runs once when the object is first created
  - In this case, we use it to declare a new action server and start it up

```
# This function is run when the constructor for the class is called, and is used to initialise a new object of that class
def __init__(self):

    # Declare an action server, called 'example_as', of type ExampleAction, using function execute_cb to handle requests
    self.a_server = actionlib.SimpleActionServer("example_as", ExampleAction, execute_cb=self.execute_cb, auto_start=False)
    # Start the action server
    self.a_server.start()
```

Creating an Action Server Node

- We also define a callback function to handle goals from action clients

```python
# This function is used to handle goals requested by an action client
def execute_cb(self, goal):

    feedback = ExampleFeedback()    # Instantiate feedback variable (of type ExampleFeedback)
    result = ExampleResult()    # Instantiate result variable (of type ExampleResult)
    result.success = True
    rate = rospy.Rate(1)

    rospy.loginfo("Action Request Received (%s) - Performing Action...",goal)

    for i in range(0, goal.seconds_requested):
        # Check whether the client has cancelled the request
        if self.a_server.is_preempt_requested():
            self.a_server.set_preempted()
            result.success = False
            break

        feedback.seconds_elapsed = i
        self.a_server.publish_feedback(feedback)    # Return feedback to the client
        rate.sleep()
        i += 1

    if result.success:  # Action successful
        feedback.seconds_elapsed += 1
        rospy.loginfo("SUCCESS - Action Complete after %s seconds \n ---",feedback.seconds_elapsed)
        self.a_server.set_succeeded(result) # Return the result (success) to the client
    else:   # Action failed (e.g. cancelled by client)
        rospy.logerr("FAILURE - Action Aborted!")
        self.a_server.set_aborted(result)   # Return the result (failure) to the client
```

Creating an Action Server Node

- Finally, we need a main function that executes when the program is executed
  - In this case, it sets up the program as a ROS node, instantiates our action server as an object, and keeps the node running

```
if __name__ == "__main__":
    rospy.init_node("example_action_server_node")
    # Call the constructor for our class, instantiating it as object 's'
    s = ExampleActionServer()
    rospy.spin()
```

Creating an Action Client Node

- **In tutorial_scripts' scripts folder, create the file Example_Action_Client.py**

- The first function we define is for handling feedback received from the action server

- This will execute even while the client waits for a result; in this way, the client can dynamically respond to feedback from the server

```
# Feedback from the action server is handled by this function
def feedback_cb(msg):
    rospy.loginfo("Feedback Received: %s", msg)
```

Creating an Action Client Node

- We then define the function used to call the action server and return the result:

```python
# This function is run to call the action server
def call_server():

    # Declare an action client for action 'example_as' of type ExampleAction
    client = actionlib.SimpleActionClient('example_as', ExampleAction)

    # Wait for the corresponding action server to become available
    client.wait_for_server()

    # Instantiate a goal of type ExampleGoal
    goal = ExampleGoal()
    goal.seconds_requested = int(input("For how many seconds should I perform the action? (1<=t<=9): "))
    if goal.seconds_requested < 1:
        goal.seconds_requested = 1
    elif goal.seconds_requested > 9:
        goal.seconds_requested = 9

    # Send the goal to the action server, using the function feedback_cb to handle feedback
    client.send_goal(goal, feedback_cb=feedback_cb)

    # Wait for the result
    client.wait_for_result()

    # Retrieve the result once it is available
    result = client.get_result()

    return result
```

Creating an Action Client Node

- Finally, we need a main function that executes when the program is executed
  - In this case, it sets up the program as a ROS node, and then calls the action server

```
if __name__ == '__main__':

    try:
        rospy.init_node('example_action_client_node')
        result = call_server()
        rospy.loginfo("Result Received: %s", result)
    except rospy.ROSInterruptException as e:    # An exception will be raised if the request fails
        rospy.logerr('Something went wrong: %s', e)
```

Testing Our Code

- **Remember to give permissions to our code**
- **Now, in three separate Terminals, run the following:**
  1. **roscore**
  2. **rosrun tutorial_scripts Example_Action_Server.py**
  3. **rosrun tutorial_scripts Example_Action_Client.py**
- **In this example, our client requires a user input; make sure to give it one when requested!**

Testing Our Program

- You should see something like this:

```
For how many seconds should I perform the action? (1<=t<=9): 5
[INFO] [1664618815.533085]: Feedback Received: seconds_elapsed: 0
[INFO] [1664618816.530901]: Feedback Received: seconds_elapsed: 1
[INFO] [1664618817.530846]: Feedback Received: seconds_elapsed: 2
[INFO] [1664618818.530541]: Feedback Received: seconds_elapsed: 3
[INFO] [1664618819.531781]: Feedback Received: seconds_elapsed: 4
[INFO] [1664618820.539802]: Result Received: success: True
```

Client

```
[INFO] [1664618815.529325]: Action Request Received (seconds_requested: 5) - Per
forming Action...
[INFO] [1664618820.530307]: SUCCESS - Action Complete after 5 seconds
 ---
```

Server

- The client node sends a goal to the server to perform the action for a given number of seconds

- The service node does this while providing feedback each second, and eventually returns success when done

Introducing Launch Files

- Launch files allow you to execute multiple programs at once
  - Also allow you to specify launch arguments and parameters for your programs
- You can also include launch files within other launch files
  - Allows you to build up more complex launch systems from simpler ones!

Creating a Launch File

- **Navigate to tutorial_scripts and create a folder named launch**
    - This folder will contain all our custom launch files
- **Inside that folder, create a file Example_Action_Server.launch and open it**
- **Enter the following (note - not in Example Scripts!):**

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

  <node name="ExampleActionServerNode" pkg="tutorial_scripts" type="Example_Action_Server.py" respawn="false" output="screen"/>

</launch>
```

Creating a Launch File

- This launch file sets up our Example Action Server as a node
- We must specify:
  - The name for the node in the ROS system
  - The package wherein the program can be found
  - The program file (in this case the Python script)

The last two arguments are also important so make sure to always specify them as well

```xml
<?xml version="1.0" encoding="UTF-8"?>
<launch>

  <node name="ExampleActionServerNode" pkg="tutorial_scripts" type="Example_Action_Server.py" respawn="false" output="screen"/>

</launch>
```

Testing Our Code (Again)

- **Now, in two separate Terminals, run the following:**

  1. **roslaunch tutorial_scripts Example_Action_Server.launch**
  2. **rosrun tutorial_scripts Example_Action_Client.py**

- Note that setting up a node using a launch file automatically starts ROSCORE in the background, therefore we don't have to start it ourselves

Closing Thoughts – Comparing Methods

Publisher/Subscriber:

- Many-to-many system
- One-way communication
- Basis for all ROS systems, including Services & Actions

ROS Services:

- Request/response system
- Two-way communication
- Use for requesting fast processes

ROS Actions:

- Request/response system...
- …with feedback
- Two-way communication
- Use for requesting extended processes

Summary

We covered:

- Overview for ROS Actions
- Defining a new ROS Action
- Brief introduction to Object-Oriented Programming
- Creating our first ROS Action Client/Server pair
- Brief introduction to launch files

Homework: Revise ROS Actions (p.4), go over the code and make sure you can follow what's going on

Next time, we will learn how to create autonomous behaviour for robots using State Machines!

Thank You!

Any Questions? Contact jacques.cloete@trinity.ox.ac.uk