Lecture 09.02 Serving and calling a ROS action

Creating an *action server node* is our first consideration.

action server node

09.02.1 Creating an action server node

Here are some key aspects of a `rospy` action server, listed below as instructions for creating such a node.

1. Import the Python package `actionlib`:

```
import actionlib
```

2. Import the action's generated message types:

```
from <pkg>.msg import <action_type>Action, \
  <action_type>Goal, <action_type>Result, <action_type>Feedback
```

3. Define an action function to serve:

```
def do_action(goal):
```

   a) Check for errors in client request and abort with result if necessary:

```
result = <action_type>Result() # create result object
result.<field_name> = <field_value> # set result(s)
server.set_aborted(result,"An abort message")
```

   b) While `goal` isn't yet met, do the following in a timed loop.

      i. Check for client request to preemptively abort goal and abort with result if necessary:

```
if server.is_preempt_requested():
  result = <action_type>Result() # create result obj
  server.set_preempted(result,"Preempted abort msg")
  return
```

      ii. Set and publish feedback:

```
feedback = <action_type>Feedback() # create fdbck obj
feedback.<field_name> = <field_value> # set feedback
server.publish_feedback(feedback) # publish feedback
```

   c) When `goal` is met, publish the result:

```
result = <action_type>Result() # create result object
result.<field_name> = <field_value> # set result(s)
server.set_succeeded(result,"A success message")
```

4. Register an action:

```
server = actionlib.SimpleActionServer(
  'server_name',          # server name string
  <action_type>Action,    # action Action message type
```

```
    do_action,              # action function
    False # autostart server? always set to False
)
```

5. Start the action and wait for requests:

```
server.start()
rospy.spin()
```

09.02.2 An example action server node

Let's implement our new action `Timer`, created in Lecture 09.01. We need an action server node to do so. Create (`touch`) a Python node file `my_actions/src/fancy_action_server.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_services/src/fancy_action_server.py` file shown in Figure 09.1 (be sure to change `rico_actions` to `my_actions`).

```python
#! /usr/bin/env python
import rospy
import time       # for regular Python timing
import actionlib  # for actions!
from rico_actions.msg import \
    TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_timer(goal): # action function
    start_time = time.time()
    update_count = 0
    if goal.time_to_wait.to_sec() > 60.0: # check req duration
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(
            time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Aborted: too long to wait")
        return # too long of a requested wait
    while (time.time()-start_time) < goal.time_to_wait.to_sec():
        # waiting to meet goal duration
        if server.is_preempt_requested(): # check preemption
            result = TimerResult()
            result.time_elapsed = rospy.Duration.from_sec(
                time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted")
            return
        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(
            time.time() - start_time)
        feedback.time_remaining = \
            goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1
        time.sleep(1.0)
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(
        time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, "Timer completed successfully")

rospy.init_node('timer_action_server') # initialize node
server = actionlib.SimpleActionServer(
    'timer', TimerAction, do_timer, False
)
server.start()
rospy.spin()
```

Figure 09.1: `rico_actions/src/fancy_action_server.py` listing.

09.02.3 Creating an action client node

action client node

The key elements to creating an *action client node* are:

1. Import the Python package `actionlib`:
   ```
   import actionlib
   ```
2. Import the action's generated message types:
   ```
   from <pkg>.msg import <action_type>Action, \
     <action_type>Goal, \
     <action_type>Result, \
     <action_type>Feedback
   ```
3. Define a feedback callback function:
   ```
   def feedback_cb(feedback):
   ```
   Do whatever you like with the feedback in here.
4. Register an action client and wait for server connection:
   ```
   client = actionlib.SimpleActionClient(
     'server_name',      # action server name
     <action_type>Action # action Action message
   )
   client.wait_for_server()
   ```
5. Define and send `goal` to server:
   ```
   goal = <action_type>Goal()
   goal.<field_name> = <field_value> # set goal field(s)
   client.send_goal(goal, feedback_cb=feedback_cb) # send to
   ```
6. Wait for results, then do what you like with them:
   ```
   client.wait_for_result()
   ```

09.02.4 An example action client node

Let's create a client for our new action `Timer`. We need a client node to do so. Create (`touch`) a Python node file `my_actions/src/fancy_action_client.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_actions/src/fancy_action_client.py` file shown in Figure 09.2 (be sure to change `rico_actions` to `my_actions`).

09.02.5 Launching and verifying the server and client nodes

launch file

Let's make a *launch file* for our action server and client nodes. Navigate to your `my_actions` directory and create (`touch`) a file `fancy_action.launch`. Now edit it to include the contents shown in Figure 09.3.

```python
#! /usr/bin/env python
import rospy
import time          # for regular Python timing
import actionlib     # for actions!
from rico_actions.msg import \
    TimerAction, TimerGoal, TimerResult, TimerFeedback

def the_feedback_cb(feedback): # feedback callback function
    print('[Feedback] Time elapsed: %f' %
      (feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f' %
      (feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client') # initialize node
client = actionlib.SimpleActionClient( # register client
    'timer',         # action server name
    TimerAction    # action Action message
)
client.wait_for_server()     # wait for action server
goal = TimerGoal()            # create goal object
goal.time_to_wait = rospy.Duration.from_sec(5.0) # set field
# Uncomment this line to test server-side abort:
# goal.time_to_wait = rospy.Duration.from_sec(500.0)

client.send_goal(goal, feedback_cb=the_feedback_cb ) # send goal
# Uncomment these lines to test goal preemption:
# time.sleep(3.0)
# client.cancel_goal()

client.wait_for_result() # wait for action server to finish
# print results:
print('[Result] State: %d' % (client.get_state()))
print('[Result] Status: %s' % (client.get_goal_status_text()))
if client.get_result():
    print('[Result] Time elapsed: %f' %
        (client.get_result().time_elapsed.to_sec()))
    print('[Result] Updates sent: %d' %
        (client.get_result().updates_sent))
```

Figure 09.2: `rico_actions/src/fancy_action_client.py` listing.

```
<launch>
  <node name="server" pkg="my_actions"
  ↪  type="fancy_action_server.py" output="screen"/>
  <node name="client" pkg="my_actions"
  ↪  type="fancy_action_client.py" output="screen"/>
</launch>
```

Figure 09.3: code listing for launch file `fancy_action.launch`.

Navigate to your workspace root and build the workspace.

```
catkin_make
```

Source your workspace: `source devel/setup.bash`. Now launch the ROS graph with the following.

```
roslaunch my_actions fancy_action.launch
```

```
... logging to /home/socrates/.ros/log/....log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
started roslaunch server http://socrates:44495/
SUMMARY
========
PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.5
NODES
  /
    client (rico_actions/fancy_action_client.py)
    server (rico_actions/fancy_action_server.py)
auto-starting new master
process[master]: started with pid [9827]
ROS_MASTER_URI=http://localhost:11311
setting /run_id to 9f2db0a0-8c0f-11ea-a1ae-0800272f9db4
process[rosout-1]: started with pid [9838]
started core service [/rosout]
process[server-2]: started with pid [9845]
process[client-3]: started with pid [9846]
[Feedback] Time elapsed: 0.000013
[Feedback] Time remaining: 4.999987
[Feedback] Time elapsed: 1.002952
[Feedback] Time remaining: 3.997048
```

```
[Feedback] Time elapsed: 2.008340
[Feedback] Time remaining: 2.991660
[Feedback] Time elapsed: 3.017035
[Feedback] Time remaining: 1.982965
[Feedback] Time elapsed: 4.022336
[Feedback] Time remaining: 0.977664
[Result] State: 3
[Result] Status: Timer completed successfully
[Result] Time elapsed: 5.028862
[Result] Updates sent: 5
[client-3] process has finished cleanly
log file: /home/socrates/.ros/log/....log
```

We see the feedback printing as expected, along with the final results. The resulting goal status, accessed by `client.get_goal_status_text()`, is `'Timer completed successfully'`.

09.02.6 More `actionlib` documentation

The core of what we have used to construct our action server and client is the `actionlib` ROS package:

`wiki.ros.org/actionlib`.

The `rospy` (Python) library API is here for the `SimpleActionServer` class:

`ricopic.one/redirect/SimpleActionServer`.

And here for the `SimpleActionClient` class:

`ricopic.one/redirect/SimpleActionClient`.