Module:3

ROS Programming (part_3)

Credit hours:5 Hours

Dr. Abhishek Rudra Pal
Assistant Professor (Senior Grade-1)
SMEC
Vellore Institute of Technology - Chennai Campus
Email: abhishek.rudrapal@vit.ac.in
Mobile No: +91-9043534478;+91-9051728314(Whatsapp)

Action

- ROS services, which are useful for synchronous request/response interactions—that is, for those cases where asynchronous ROS topics don't seem like the best fit.
- However, services aren't always the best fit, either, in particular when the request that's being made is more than a simple instruction of the form "get (or set) the value of X."
- While services are handy for simple get/set interactions like querying status and man aging configuration, they don't work well when you need to initiate a long-running task.
- For example, imagine commanding a robot to drive to some distant location; call it goto_position.
- The robot will require significant time (seconds, minutes, per-haps longer) to do so, with the exact amount of time impossible to know in advance, since obstacles may arise that result in a longer path.
- Imagine what a service interface to goto_position might look like to the caller: you send a request containing the goal location, then you wait for an indeterminate amount of time to receive the response that tells you what happened.
- While waiting, your calling program is forced to block, you have no information about the robot's progress toward the goal, and you can't cancel or change the goal. To address these shortcomings, ROS provides actions

Action

- ROS actions are the best way to implement interfaces to time-extended, goal-oriented behaviors like goto_position.
- While services are synchronous, actions are asynchronous.
- Similar to the request and response of a service, an action uses a goal to initiate a behavior and sends a result when the behavior is complete.
- But the action further uses feedback to provide updates on the behavior's progress toward the goal and also allows for goals to be canceled.
- Actions are themselves implemented using topics.
- An action is essentially a higher-level protocol that specifies how a set of topics (goal, result, feedback, etc.) should be used in combination.
- Using an action interface to goto_position, you send a goal, then move on to other tasks while the robot is driving.
- Along the way, you receive periodic progress updates (distance traveled, estimated time to goal, etc.), culminating in a result message (did the robot make it to the goal or was it forced to give up?).
- And if something more important comes up, you can at any time cancel the goal and send the robot some- where else.
- Actions require only a little more effort to define and use than do services, and they provide a lot more power and flexibility. Let's see how they work

Defining an Action

- The first step in creating a new action is to define the goal, result, and feedback message formats in an action definition *f* file which by convention has the suffix **‘.action’.**
- The **‘.action**’ file format is similar to the ‘**.srv**’ format used to define services, just with an additional field.
- And, as with services, each field within an **‘.action’** file will become its own message.
- As a simple example, let’s define an action that acts like a timer).
- We want this timer to count down, signaling us when the specified time has elapsed.
- Along the way, it should tell us periodically how much time is left. When it’s done, it should tell us how much time actually elapsed.

Example:Timer.action

We're building a timer because it's a simple example of an action. In a real robot system, you would use the time support that is built into ROS client libraries, such as rospy.sleep().

```
# This is an action definition file, which has three parts: the goal, the
# result, and the feedback.
#
# Part 1: the goal, to be sent by the client
#
# The amount of time we want to wait
duration time_to_wait
---
# Part 2: the result, to be sent by the server upon completion
#
# How much time we waited
duration time_elapsed
# How many updates we provided along the way
uint32 updates_sent
---
# Part 3: the feedback, to be sent periodically by the server during
# execution.
#
# The amount of time that has elapsed from the start
duration time_elapsed
# The amount of time remaining until we're done
duration time_remaining
```

Changes in file

- Just like with service-definition files, we use three dashes (---) as the separator between the parts of the definition.
- While service definitions have two parts (request and response), action definitions have three parts (goal, result, and feedback).
- The action file Timer.action should be placed in a directory called action within a ROS package.
- As with our previous examples, this file is already present in the basics package.
- With the definition file in the right place, we need to run catkin_make to create the code and class definitions that we will actually use when interacting with the action, just like we did for new services.

Changes in file

To get catkin_make to generate this code, we need to add some lines to the CMakeLists.txt file.

- First, add actionlib_msgs to the () call (in addition to any other packages that are already there)

```
find_package(catkin REQUIRED COMPONENTS
 # other packages are already listed here
 actionlib_msgs
)
```

Then, use the add_action_files() call to tell catkin about the action files you want to compile:

```
add_action_files(
 DIRECTORY action
 FILES Timer.action
)
```

Make sure you list the dependencies for your actions. You also need to explicitly list actionlib_msgs as a dependency in order for actions to compile properly:

```
generate_messages(
 DEPENDENCIES
 actionlib_msgs
 std_msgs
)
```

Finally, add actionlib_msgs as a dependency for catkin:

```
catkin_package(
 CATKIN_DEPENDS
 actionlib_msgs
)
```

Changes in file

With all of this information in place, running catkin_make in the top level of our catkin workspace does quite a bit of extra work for us.

Our Timer.action file is processed to produce several message-definition files: TimerAction.msg, TimerAction- Feedback.msg, TimerActionGoal.msg, TimerActionResult.msg, TimerFeedback.msg, TimerGoal.msg, and TimerResult.msg.

These messages are used to implement the action client/server protocol, which, as mentioned previously, is built on top of ROS topics.

The generated message definitions are in turn processed by the message generator to produce corresponding class definitions.

Most of the time, you'll use only a few of those classes, as you'll see in the following examples

Implementing a Basic Action Server

- Now that we have a definition of the goal, result, and feedback for the timer action, we're ready to write the code that implements it.
- Like topics and services, actions are a callback-based mechanism, with your code being invoked as a result of receiving messages from another node.
- The easiest way to build an action server is to use the SimpleActionServer class from the actionlib package.
- We'll start by defining only the callback that will be invoked when a new goal is sent by an action client. In that callback, we'll do the work of the timer, and then return a result when we're done.

Implementing a Basic Action Server

simple_action_server.py

```
#! /usr/bin/env python
import rospy
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult
def do_timer(goal):
 start_time = time.time()
 time.sleep(goal.time_to_wait.to_sec())
 result = TimerResult()
 result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
 result.updates_sent = 0
 server.set_succeeded(result)
rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

Explanation

- Next, we define do_timer(), the function that will be invoked when we receive a new goal. In this function, we handle the new goal in-place and set a result before returning. The type of the goal argument that is passed to do_timer() is TimerGoal, which corresponds to the goal part of Timer.action. We save the current time, using the standard Python time.time() function, then sleep for the time requested in the goal, converting the time_to_wait field from a ROS duration to seconds.

```
def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
```

The next step is to build up the result message, which will be of type TimerResult; this corresponds to the result part of Timer.action. We fill in the time_elapsed field by subtracting our saved start time from the current time, and converting the result to a ROS duration. We set updates_sent to zero, because we didn't send any updates along the way (we'll add that part shortly):

```
 result = TimerResult()
 result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
 result.updates_sent = 0
```

Explanation

- Our final step in the callback is to tell the SimpleActionServer that we successfully achieved the goal by calling set_succeeded() and passing it the result. For this sim- ple server, we always succeed; we'll address failure cases later in this chapter

server.set_succeeded(result)

- Back in the global scope, we initialize and name our node as usual, then create a Sim pleActionServer. The first constructor argument for SimpleActionServer is the server's name, which will determine the namespace into which its constituent topics will be advertised; we'll use timer. The second argument is the type of the action that the server will be handling, which in our case is TimerAction. The third argument is the goal callback, which is the function do_timer() that we defined earlier. Finally, we pass False to disable autostarting the server. Having created the action server, we explicitly start() it, then go into the usual ROS spin() loop to wait for goals to arrive:

rospy.init_node('timer_action_server')

server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)

server.start()

rospy.spin()

Checking That Everything Works as Expected

- **user@hostname$ rosrun basics simple_action_server.py**
- Let’s check that the expected topics are present:
- **user@hostname$ rostopic list**
- /rosout
- /rosout_agg
- /timer/cancel
- /timer/feedback
- /timer/goal
- /timer/result
- /timer/status
- That looks good: we can see the five topics in the timer namespace that are used
- under the hood to manage the action. Let’s take a closer look at the /timer/goal
- topic, using rostopic:
- **user@hostname$ rostopic info /timer/goal**
- Type: basics/TimerActionGoal
- Publishers: None
- Subscribers: * /timer_action_server (http://localhost:63174/)

Checking That Everything Works as Expected

- user@hostname$ **rosmsg show TimerActionGoal**
- [basics/TimerActionGoal]:
- std_msgs/Header header
- uint32 seq
- time stamp
- string frame_id
- actionlib_msgs/GoalID goal_id
- time stamp
- string id
- basics/TimerGoal goal
- duration time_to_wait

Checking That Everything Works as Expected

- Interesting; we can see our goal definition in there, as the goal.time_to_wait field, but there are also some extra fields that we didn't specify. Those extra fields are used by the action server and client code to keep track of what's happening. Fortunately, that bookkeeping information is automatically stripped away before our server code sees a goal message. While a TimerActionGoal message is sent over the wire, what we see in our goal execution is a bare TimerGoal message, which is just what we defined in our .action file:
- **user@hostname$ rosmsg show TimerGoal**

  [basics/TimerGoal]:

  duration time_to_wait
- In general, if you're using the libraries in the actionlib package, you should not need to access the autogenerated messages with Action in their type name. The bare Goal, Result, and Feedback messages should suffice. If you like, you can publish and subscribe directly to an action server's topics using the autogenerated Action message types. This is a nice feature of ROS actions: they are just a higher-level protocol built on top of ROS messages. But for most applica- tions (including everything that we'll cover in this book), the actionlib libraries will do the job, handling the underlying messages for you behind the scenes

Using an Action (client)

simple_action_client.py

```
#! /usr/bin/env python
import rospy
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult
rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

Explanation

- The first constructor argument is the name of the action server, which the client will use to determine the topics that it will use when communicating with the server.
- This name must match the one that we used in creating the server, which is timer.
- The second argument is the type of the action, which must also match the server: TimerAction.
- Having created the client, we tell it to wait for the action server to come up, which it does by checking for the five advertised topics that we saw earlier when testing the server.
- Similar to rospy.wait_for_service(), which we used to wait for a service to be ready, SimpleActionClient.wait_for_server() will block until the server is ready:

```
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
```

Explanation

Now we create a goal of type TimerGoal and fill in the amount of time we want the timer to wait, which is five seconds. Then we send the goal, which causes the transmission of the goal message to the server:

```
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
```

Next, we wait for a result from the server. If things are working properly, we expect to block here for about five seconds. After the result comes in, we use get_result() to retrieve it from within the client object and print out the time_elapsed field that was reported by the server:

```
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

Checking That Everything Works as Expected

- Now that we have implemented the action client, we can get to work. Make sure that your roscore and action server are still running, then run the action client:

  user@hostname$ rosrun basics simple_action_client.py

  Time elapsed: 5.001044
- Between the invocation of the client and the printing of the result data, you should
- see a delay of approximately five seconds, as requested. The time elapsed should be
- slightly more than five seconds, because a call to time.sleep() will usually take a lit-
- tle longer than requested.

Implementing a More Sophisticated Action Server

- So far, actions look a lot like services, just with more configuration and setup. Now it’s time to exercise the asynchronous aspects of actions that set them apart from services. We’ll start on the server side, making some changes that demonstrate how to abort a goal, how to handle a goal preemption request, and how to provide feedback while pursuing a goal.

Implementing a More Sophisticated Action Server

fancy_action_server.py

```
#! /usr/bin/env python
import rospy
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
def do_timer(goal):
 start_time = time.time()
 update_count = 0
 if goal.time_to_wait.to_sec() > 60.0:
 result = TimerResult()
 result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
 result.updates_sent = update_count
 server.set_aborted(result, "Timer aborted due to too-long wait")
 return
```

Implementing a More Sophisticated Action Server

```
while (time.time() - start_time) < goal.time_to_wait.to_sec():
 if server.is_preempt_requested():
 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')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

Explanation

Because we will be providing feedback, we add TimerFeedback to the list of message types that we import:

from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

Stepping inside our do_timer() callback, we add a variable that will keep track of how many times we publish feedback:

update_count = 0

Next, we add some error checking. We don't want this timer to be used for long waits, so we check whether the requested time_to_wait is greater than 60 seconds, and if so, we explicitly abort the goal by calling set_aborted(). This call sends a message to the client notifying it that the goal has been aborted. Like with set_succeeded(), we include a result; doing this is optional, but a good idea if possible. We also include a status string to help the client understand what happened; in this case, we aborted because the requested wait was too long

Explanation

we’re done with this goal:

```
if goal.time_to_wait.to_sec() > 60.0:
result = TimerResult()
result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
result.updates_sent = update_count
server.set_aborted(result, "Timer aborted due to too-long wait")
return
```

Now that we’re past the error check, instead of just sleeping for the requested time in one shot, we’re going to loop, sleeping in increments. This allows us to do things while we’re working toward the goal, such as checking for preemption and providing feedback:

```
while (time.time() - start_time) < goal.time_to_wait.to_sec():
```

Explanation

we’re done with this goal:

In the loop, we first check for preemption by asking the server is_preempt_reques ted(). This function will return True if the client has requested that we stop pursuing the goal (this could also happen if a second client sends us a new goal). If so, similar to the abort case, we fill in a result and provide a status string, this time calling **set_preempted():**

```
    if server.is_preempt_requested():
    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
```

Next we send feedback, using the type TimerFeedback, which corresponds to the feedback part of Timer.action. We fill in the time_elapsed and time_remaining fields, then call publish_feedback() to send it to the client. We also increment update_count to reflect the fact that we sent another update:

Explanation

Next we send feedback, using the type TimerFeedback, which corresponds to the feedback part of Timer.action. We fill in the time_elapsed and time_remaining fields, then call publish_feedback() to send it to the client. We also increment update_count to reflect the fact that we sent another update:

```
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
```

Then we sleep a little and loop. Sleeping for a fixed amount of time here is not the right way to implement a timer, as we could easily end up sleeping longer than requested, but it makes for a simpler example:

```
time.sleep(1.0)
```

Explanation

Exiting the loop means that we've successfully slept for the requested duration, so it's time to notify the client that we're done. This step is very similar to the simple action server, except that we fill in the updates_sent field and add a status string:

```
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")
```

Using the More Sophisticated Action(client)

- Now we'll modify the action client to try out the new capabilities that we added to the action server: we'll process feedback, preempt a goa, and trigger an abort

fancy_action_client.py

```
#! /usr/bin/env python
import rospy
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
def feedback_cb(feedback):
 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')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
```

Using the More Sophisticated Action(client)

```
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
# Uncomment this line to test server-side abort:
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
client.send_goal(goal, feedback_cb=feedback_cb)
# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()
client.wait_for_result()
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))
```

Using the More Sophisticated Action(client)

Let's step through the changes with respect to Example 5-3. We define a callback, feedback_cb(), that will be invoked when we receive a feedback message. In this callback we just print the contents of the feedback:

```
def feedback_cb(feedback):
 print('[Feedback] Time elapsed: %f'%(feedback.time_elapsed.to_sec()))
 print('[Feedback] Time remaining: %f'%(feedback.time_remaining.to_sec()))
```

We register our feedback callback by passing it as the feedback_cb keyword argument when calling send_goal():

```
client.send_goal(goal, feedback_cb=feedback_cb)
```

Using the More Sophisticated Action(client)

After receiving the result, we print a little more information to show what happened. The get_state() function returns the state of the goal, which is an enumeration that is defined in actionlib_msgs/GoalStatus. While there are 10 possible states, in this example we'll encounter only three: PREEMPTED=2, SUCCEEDED=3, and ABORTED=4. We also print the status text that was included by the server with the result:

```
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))
```

Checking That Everything Works as Expected

Let’s try out our new server and client. As before, start up a roscore, then run the server:

user@hostname$ rosrun basics fancy_action_server.py

In another terminal, run the client:

user@hostname$ rosrun basics fancy_action_client.py

[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001626
[Feedback] Time remaining: 3.998374
[Feedback] Time elapsed: 2.003189
[Feedback] Time remaining: 2.996811
[Feedback] Time elapsed: 3.004825
[Feedback] Time remaining: 1.995175
[Feedback] Time elapsed: 4.006477
[Feedback] Time remaining: 0.993523
[Result] State: 3
[Result] Status: Timer completed successfully
[Result] Time elapsed: 5.008076
[Result] Updates sent: 5

Everything works as expected: while waiting, we receive one feedback update per second, then we receive a successful result (SUCCEEDED=3)

Checking That Everything Works as Expected

Now let's try preempting a goal. In the client, following the call to send_goal(), uncomment these two lines, which will cause the client to sleep briefly, then request that the server preempt the goal:

```
# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()
```

Run the client again:

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001651
[Feedback] Time remaining: 3.998349
[Feedback] Time elapsed: 2.003297
[Feedback] Time remaining: 2.996703
[Result] State: 2
[Result] Status: Timer preempted
[Result] Time elapsed: 3.004926
[Result] Updates sent: 3
```

That's the behavior we expect: the server pursues the goal, providing feedback, until we send the cancellation request, after which we receive the result confirming the preemption (PREEMPTED=2).

Checking That Everything Works as Expected

Now let’s trigger a server-side abort. In the client, uncomment this line to change the requested wait time from 5 seconds to 500 seconds:

```
# Uncomment this line to test server-side abort:
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
```

Run the client again:

```
user@hostname$ rosrun basics fancy_action_client.py
[Result] State: 4
[Result] Status: Timer aborted due to too-long wait[Result] Time elapsed: 0.000012
[Result] Updates sent: 0
```

As expected, the server immediately aborted the goal (ABORTED=4).

Comparison

- actions offer much more control to both the client and the server than do services. The server can provide feedback along the way while it's servicing the request: the client can can- cel a previously issued request; and, because they're built atop ROS messages, actions are asynchronous, allowing for nonblocking programming on both sides

Table 5-1. Comparison of topics, services, and actions

Type	Best used for
Topic	One-way communication, especially if there might be multiple nodes listening (e.g., streams of sensor data)
Service	Simple request/response interactions, such as asking a question about a node's current state
Action	Most request/response interactions, especially when servicing the request is not instantaneous (e.g., navigating to a goal location)

Taken together, these features of actions make them well-suited to many aspects of robot programming.

It's common in a robotics application to implement timeextended, goal-seeking behaviors, whether it's goto_position or clean_the_house.

Any time you need to be able trigger a behavior, actions are probably the right tool for the job.

In fact, any time that you're using a service, it's worth considering replacing it with an action; actions require a bit more code to use, but in return they're much more powerful and extensible than services.