ActionServer User Manual

Compiled by: David Kooi

July 2018

1 Introduction

The **ActionServer** located in:

`src/movex/nodes/services/action_server.py`

The **ActionServer** contains methods to do timed and setpoint actuation on the Housecat and Movex. Bucket movements may be timed or setpoint driven. Track motors may only be timed.

Contents

List of Figures

2 Imports and Organization

Various structure keep the action server logic generic. **Line 9** imports the movement messages. Each message acts as a 'goal'. One goal may have multiple actions. This will be detailed in Section ....

Lines 20 and 21 are an attempt to organize the different command names. A better way to define parameters over multiple modules is to use the ROS parameter server:

`http://wiki.ros.org/Parameter%20Server`

This allows parameters to be used in both C++ and Python nodes.

```
 1 #!/usr/bin/env python
 2
 3 import rospy
 4 import actionlib
 5 import threading
 6 import time
 7
 8 # ROS Actions
 9 from movex.msg import movementAction, movementGoal,\
10                       movementFeedback, movementResult
11
12 from std_msgs.msg import Float64
13
14
15 # Sambuca Command messages
16 from movex.srv import Grab, Lift, Move, Turn
17
18 from rospy import ServiceException
19 from sambuca_util import enum
20 from nodes.services.http_service import SambucaCmdNames
21 from definitions.sambuca_settings import SambucaActionServerName
22
23
24 # Define some organizational structures
25 grab_request = rospy.ServiceProxy(SambucaCmdNames.GRAB, Grab)
26 lift_request = rospy.ServiceProxy(SambucaCmdNames.LIFT, Lift)
27 move_request = rospy.ServiceProxy(SambucaCmdNames.MOVE, Move)
28 turn_request = rospy.ServiceProxy(SambucaCmdNames.TURN, Turn)
29
30 SambucaPoseTopics = {SambucaCmdNames.GRAB:"pose/angle_2",
31                      SambucaCmdNames.LIFT:"pose/angle_1"}
32
33
34 # Dictionary of service requests
35 SambucaCmdFunc = {SambucaCmdNames.MOVE:move_request,
36                   SambucaCmdNames.LIFT:lift_request,
37                   SambucaCmdNames.GRAB:grab_request,
38                   SambucaCmdNames.TURN:turn_request}
```

Figure 1: Action Server Imports and Function Organization

Lines 25-28 define service proxies to the HTTP service located in:

`ROS/catkin_ws/src/movex/nodes/services/http_service.py`

Lines 35-39 define a dictionary of service proxies accessible by name.

3 Initialization

Lines 116 - 119 enable concurrent movement operations. One goal may start a **grab, turn, move** and **lift** at the same time. The goal is "finished" when all commands are not running. I.e, each of the running flags are false.

Line 122 creates the action server and sets an execute callback method. This method is run whenever a **MovementAction** is received.

From now on, a **MovementAction** will be refered to as a **Goal**.

```
110 class SambucaActionServer(object):
111
112     def __init__(self, name):
113
114         self.actionName = name
115
116         self.grabRunning = False
117         self.turnRunning = False
118         self.moveRunning = False
119         self.liftRunning = False
120
121
122         self.server = actionlib.SimpleActionServer(self.actionName,\
123                 movementAction, execute_cb=self.executeCallback,\
124                 auto_start=False)
125
126         self.server.start()
```

Figure 2: Action Server Initialization

4 Goals (movement.action)

```
 1 # Goal Definition
 2 string requester
 3
 4 ActionMsg moveRequest
 5 ActionMsg turnRequest
 6 ActionMsg grabRequest
 7 ActionMsg liftRequest
 8
 9 ---
10 # Result Definition
11 int8 result
12 ---
13 # Feedback
14 uint16 timeElapsed
```

Figure 3: Action Server Goal

The definition of a **Goal** is shown in Figure 3. This definition is located in:

`../ROS/catkin_ws/src/movex/action/movement.action`

There are four **ActionMsgs** one may configure. **ActionMsgs** are shown in Figure 4. They are defined in:

`../ROS/catkin_ws/src/movex/msg/ActionMsg.msg`

```
1 bool   isActive
2 float32 duration
3 int16 magnitude
```

Figure 4: ActionMsg

4.1 Testing Goals

It is useful to test a sequence of movements. Figure 5 shows a test file that does this. The file is located in the test folder:

`../ROS/catkin_ws/src/movex/test/test_actionlib.py`

The module is launched by running:

`roslaunch movex test_action_client.launch`

Other launch files may be seen in:

`../ROS/catkin_ws/src/movex/launch`

```
1  #!/usr/bin/env python
2
3  from time import sleep
4  import rospy
5  import actionlib
6
7  from nodes.services.http_service import SambucaCmdNames
8  from definitions.sambuca_settings import SambucaActionServerName
9
10 from movex.msg import movementGoal, movementAction
11
12 if __name__ == "__main__":
13
14     rospy.init_node("test_movement_client")
15     client = actionlib.SimpleActionClient(SambucaActionServerName,\
16                                           movementAction)
17     client.wait_for_server()
18
19     print("EXECUTING NOW")
20     sleep(1)
21
22     # Action 1
23     goal1 = movementGoal()
24     goal1.requester = "Action Tester"
25
26     goal1.moveRequest.isActive  = True
27     goal1.moveRequest.duration  = 0.5
28     goal1.moveRequest.magnitude = 200
29
30     goal1.liftRequest.isActive  = True
31     goal1.liftRequest.magnitude = 45
32
33     goal1.grabRequest.isActive  = True
34     goal1.grabRequest.duration  = 3
35     goal1.grabRequest.magnitude = 1
36
37     # Action 2
38     goal2 = movementGoal()
39     goal2.requester = "Action Tester"
40     goal2.moveRequest.isActive = True
41     goal2.moveRequest.duration= 4
42     goal2.moveRequest.magnitude = -200
43
44
45     # Send goals
46     goalSequence = [goal1, goal2]
47
48     for goal in goalSequence:
49         client.send_goal(goal)
50         client.wait_for_result(rospy.Duration.from_sec(20))
```

Figure 5: Testing Goals on the ActionServer

ActionMsgs default their active flag to **False**. So if you are not using a request, not including it will not run the request. E.g, Goal 1 will not run a **turn** request and Goal 2 will only run a **move** request.

grab and **lift** requests can operate with setpoints and with timers. To use either request with a set point, only assign the magnitude. To use a timer, assign a magnitude (-1 for down, 1 for up) and a duration(seconds). Timed operations can be less than 1, but must be greater than 0. E.g, 0.5 seconds, 0.2 seconds, ect...

move and **turn** requests require a magnitude and duration. The higher the magnitude, the more power you give to the motors. This magnitude may be negative for reverse operation. The duration is seconds.

Good practice is to create a list of goals and send the sequentially as in **lines 46 - 50**.

5 Executing Goals

Figure 6 shows the **ActionServer** execute callback function. This function receives all goals. **Lines 242 - 264** handle each request. **self.handleRequest** does not block, but kicks off threads and timers. In effect, each request is run in parallel.

Note if a request is active, the corresponding run flag is set to true. E.g, when a **grabRequest** is active, the **grabRunning** flag is set **True**.

Also note, each requests passes a **stop** method. This method will be used by a timer or **PoseObserver** thread to stop the request. Stop functions are detailed in Section 8.

```
235     def executeCallback(self, goal):
236         print("EXECUTE")
237         print(goal)
238         rospy.loginfo("ActionServer: Goal received from :\
239                         {}".format(goal.requester))
240
241
242         # Handle grab request
243         if(goal.grabRequest.isActive):
244             request = goal.grabRequest
245             self.handleRequest(SambucaCmdNames.GRAB, request, self.stopGrab)
246
247             self.grabRunning = True
248
249         # Handle turn request
250         if(goal.turnRequest.isActive):
251             request = goal.turnRequest
252             self.handleRequest(SambucaCmdNames.TURN, request, self.stopTurn)
253
254             self.turnRunning = True
255
256         # Handle move request
257         if(goal.moveRequest.isActive):
258             request = goal.moveRequest
259             self.handleRequest(SambucaCmdNames.MOVE, request, self.stopMove)
260
261             self.moveRunning = True
262
263
264         # Handle lift request
265         if(goal.liftRequest.isActive):
266             request = goal.liftRequest
267             self.handleRequest(SambucaCmdNames.LIFT, request, self.stopLift)
268
269             self.liftRunning = True
```

Figure 6: ActionServer Execute Callback(1)

Figure 7 shows the **ActionServer** waiting for all requests to stop. **Line 274** creates a list of run flags. The **goal** will not be finished until all requests are finished, i.e all run flags are **False**.

When all requests are finished **allStopped** evaluates to **True** via single line python magic on 285.

Finally, we **line 297** tells the client this goal has succeded.

```
272         # Spin while all actions finish
273         rospy.loginfo("Waiting for actions to finish")
274
275         # Assume none are stopped
276         allStopped = False
277         while(not allStopped):
278             runningList = [self.grabRunning, self.turnRunning,
279                            self.liftRunning, self.moveRunning]
280
281             # Spin until all are not running
282
283             # When isRunning is False for all in runningList
284             # all(not isRunning...) will return True
285             allStopped = all([not isRunning for isRunning in runningList])
286
287
288
289
290         # Make sure all flags are false for next request
291         self.grabRunning = False
292         self.turnRunning = False
293         self.moveRunning = False
294         self.liftRunning = False
295
296
297         self.server.set_succeeded()
```

Figure 7: ActionServer Execute Callback(2)

6 Handling Goals

6.1 Handling Move and Turn Requests

handleRequest consumes requests and starts either timers for timed movements and **PoseObserver** threads for setpoint movments.

Lines 192-204 operate **move** and **turn** requests. The function is retrieved from the dictionary of service proxies on **line 196** and called on the next line.. This dictionary of name-mapped service proxies is defined in Figure 1 .

A timer is started for the duration with the callback function set to the stopMethod passed in. **oneshot** is **True** because we are only using this timer for one request.

```
187     def handleRequest(self, cmdName, request, stopMethod):
188         rospy.loginfo("Received request: {}".format(cmdName))
189         rospy.loginfo("Duration: {}".format(request.duration))
190         rospy.loginfo("Magnitude: {}".format(request.magnitude))
191 
192         if(cmdName == SambucaCmdNames.MOVE or cmdName == SambucaCmdNames.TURN):
193 
194             # Get reference to the service proxy
195             # And initiate the action
196             serviceFunction = SambucaCmdFunc[cmdName]
197             serviceFunction(request.magnitude)
198 
199 
200             # Run a timer callback to stop the action if a track movement
201             duration = rospy.Duration(request.duration)
202             rospy.Timer(period=duration, callback=stopMethod, oneshot=True)
203 
204 
205         else:
206             # Run a timed arm or bucket command
207             if(request.duration > 0.0):
208 
209                 # Get reference to the service proxy
210                 # And initiate the action
211                 serviceFunction = SambucaCmdFunc[cmdName]
212                 serviceFunction(request.magnitude)
213 
214 
215                 # Run a timer callback to stop the action if a track movement
216                 duration = rospy.Duration(request.duration)
217                 rospy.Timer(period=duration, callback=stopMethod, oneshot=True)
218 
219             # Run a setpoint arm or bucket command
220             else:
221                 # Create and start a pose observer thread
222                 poseObserver = PoseObserver(self, cmdName, SambucaPoseTopics[cmdName],\
223                                             request.magnitude)
224                 poseObserver.start()
```

Figure 8: ActionServer Request Handling

6.2 Handling Bucket Requests

6.2.1 Timed Operation

Lines 205-224 handle **grab** and **lift** requests. Time requests are handled if the duration is greater than 0. I.e, the duration is defined. **Line 207** checks the duration and runs a timed operation if appropriate. The operation is the same as the above: The service function is retrieved from the service proxy dictionary and called on the next line. A timer is then created with a callback pointing to the stop method.

6.2.2 Setpoint Operation

A **PoseObserver** thread is created to run a setpoint operation. The **PoseObserver** is defined in the same file and is detailed in Section 7.

Line 222 creates a **PoseObserver** thread with the actionsever(self), command name, pose topic to subscribe to, and the desired setpoint(magnitude). The pose topic refers to the IMU data stream published by **pose.py**. With the **PoseObserver** thread instantiated, the thread is started on **line 224**.

Note again, **self** is passed to the **PoseObserver** thread. This gives **PoseObserver** thread reference to the **ActionServer**. Once the thread is finished it will notify the **ActionServer** via the public **notifyStop** method.

7 PoseObserver Feedback Thread

The **PoseObserver** thread 'listens' to IMU data streamed from **pose.py**. Once the IMU data reaches a desired setpoint the thread notifies the **ActionServer** the bucket request has completed.

Note that the **PoseObserver** is a subclass of **threading.Thread**. For more information see:

`https://docs.python.org/2/library/threading.html`

Line 56 sets the hysteresis the received angle must be within for the **PoseObserver** to stop.

Line 63 creates a **runFlag**. The function will run until the **runFlag** is cleared. For more information see:

`https://docs.python.org/2/library/threading.html#event-objects`

Line 67 starts a subscriber to listen to the pose topic.

```
41 class PoseObserver(threading.Thread):
42     """
43     A thread to watch an angle.
44     """
45     def __init__(self, actionServer, cmdName, poseTopic, refAngle):
46         """
47         @param actionServer: Reference to action server
48         @param cmdName: GRAB or LIFT
49         @param poseTopic: Name of the pose. E.g angle_0, angle_1, angle_2
50         @param refAngle: Reference angle
51         """
52         super(PoseObserver, self).__init__()
53 
54         rospy.loginfo("Creating PoseObserver: {}".format(poseTopic))
55 
56         self.threshold = 3
57 
58         self.cmdName = cmdName
59         self.poseTopic = poseTopic
60         self.refAngle = refAngle
61         self.actionServer = actionServer
62 
63         self.runFlag  = threading.Event()
64         self.runFlag.set()
65 
66         # Start listening to the pose
67         self.sub = rospy.Subscriber(poseTopic, Float64, self.handlePose,\
68                 queue_size=1)
```

```
70     def handlePose(self, pose):
71         angle = pose.data
72         rospy.loginfo("{} {}".format(self.poseTopic, angle))
73 
74 
75         # Run the motors toward the reference angle
76         if(self.cmdName == SambucaCmdNames.GRAB):
77             if(self.refAngle - angle > 0):
78                 SambucaCmdFunc[self.cmdName](1)
79             else:
80                 SambucaCmdFunc[self.cmdName](-1)
81 
82         if(self.cmdName == SambucaCmdNames.LIFT):
83             if(self.refAngle - angle > 0):
84                 SambucaCmdFunc[self.cmdName](1)
85             else:
86                 SambucaCmdFunc[self.cmdName](-1)
87 
88         # If the reference angle and the current angle are close
89         # enough, stop the arm movement
90         if(abs(angle - self.refAngle) < self.threshold):
91 
92             self.sub.unregister()
93             self.actionServer.notifyStop(self.cmdName)
94             self.runFlag.clear()
95             rospy.loginfo("Reference angle {} reached".format(self.refAngle))
96             # time.sleep(1)
97 
98 
99 
100 
101     def run(self):
102         """
103         Run thread until runFlag is cleared
104         """
105         while(self.runFlag.isSet()):
106             pass
107         print("DONE WITH THREAD")
108 
109
```

Figure 9: PoseObserver

handlePose on line 70 is run each time it receives a new IMU data. **Lines 76-86** drive the motors in the correct direction. This could be condensed into one conditional since tehre is no difference between **grab** and **arm** requests.

Line 90 checks if the received angle is within the threshold of the reference angle. If it is, the **PoseObserver** unsibsribes from the IMU data topic, notifies the **ActionServer** to stop, and clears the **runFlag**.

Line 101 defines the run function of the thread. The thread blocks until the **runFlag** is cleared.

8 Stop Functions

8.1 Timed Stops

Lines 129 - 182 in the **ActionServer** define the timer stop callbacks. Each do the same thing: They try to stop the request. (Setting the request to 0). Note, in each method there is handling for a **ServiceExceptions**. A **ServiceException** occurs when the stop command fails. When a stop command fails, the **ActionServer** retries the stop command after a small duration.

```
128     # Timer Callbacks
129     def stopGrab(self, event):
130         rospy.loginfo("Grab stop received")
131         try:
132             grab_request(0)
133         except ServiceException:
134             rospy.logerr("GRAB STOP FAILED")
135             # There was an error in the HTTP Service
136             # So call ourselves again
137             rospy.Timer(period=rospy.Duration(0.01), callback=self.stopGrab, oneshot=True)
138             self.grabRunning = True # We are not done yet
139             return
140 
141         self.grabRunning = False # Done
142 
143 
144     def stopTurn(self, event):
145         rospy.loginfo("Turn stop received")
146         try:
147             turn_request(0)
148         except ServiceException:
149             rospy.logerr("TURN STOP FAILED")
150             # There was an error in the HTTP Service
151             # So call ourselvesn again
152             rospy.Timer(period=rospy.Duration(0.01), callback=self.stopTurn, oneshot=True)
153             self.turnRunning = True # We are not done yet
154             return
155 
156         self.turnRunning = False
157 
158     def stopMove(self, event):
159         rospy.loginfo("Move stop received")
160         try:
161             move_request(0)
162         except ServiceException:
163             rospy.logerr("MOVE STOP FAILED")
164             # There was an error in the HTTP Service
165             # So call ourselves again
166             rospy.Timer(period=rospy.Duration(0.01), callback=self.stopMove, oneshot=True)
167             self.moveRunning = True # We are not done yet
168             return
169 
170         self.moveRunning = False
171 
172     def stopLift(self, event):
173         rospy.loginfo("Lift stop received")
174         try:
175             lift_request(0)
176         except ServiceException:
177             rospy.logerr("LIFT STOP FAILED")
178             # There was an error in the HTTP Service
179             # So call ourselves again
180             rospy.Timer(period=rospy.Duration(0.01), callback=self.stopLift, oneshot=True)
181             self.liftRunning = True # We are not done yet
182             return
```

Figure 10: ActionServer Timed Stop Methods

8.2 Setpoint Stops

PoseObservers call the **notifyStop** when a setpoint is reached. The appropriate stop method is called. **None** is the argument because the stop methods are called by timers which pass an event. We can pass **None** to replace the timer event.

```
227     def notifyStop(self, cmdName):
228         if(cmdName == SambucaCmdNames.LIFT):
229             self.stopLift(None)
230         elif(cmdName == SambucaCmdNames.GRAB):
231             self.stopGrab(None)
232         else:
233             rospy.logerr("Unsupported PoseObserver command {}".format(cmdName))
234 
```

Figure 11: ActionServer Timed Stop Methods