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

1 Introduction 1

2 Imports and Organization 3

3 Initialization 4

4 Goals (movement.action) 5
   4.1 Testing Goals .......................................................... 6

5 Executing Goals 7

6 Handling Goals 8
   6.1 Handling Move and Turn Requests .................................. 8
   6.2 Handling Bucket Requests ........................................... 8
      6.2.1 Timed Operation ................................................... 8
      6.2.2 Setpoint Operation .............................................. 8

7 PoseObserver Feedback Thread 9

8 Stop Functions 10
   8.1 Timed Stops ............................................................ 10
   8.2 Setpoint Stops .......................................................... 10

List of Figures

1 Action Server Imports and Function Organization ............................. 3
2 Action Server Initialization ..................................................... 4
3 Action Server Goal ............................................................. 5
4 ActionMsg ................................................................. 5
5 Testing Goals on the ActionServer ........................................... 6
6 ActionServer Execute Callback(1) ............................................ 7
7 ActionServer Execute Callback(2) ............................................ 7
8 ActionServer Request Handling .............................................. 8
9 PoseObserver ............................................................... 9
10 ActionServer Timed Stop Methods ......................................... 10
11 ActionServer Timed Stop Methods ......................................... 10
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.

```
#!/usr/bin/env python

import rospy
import actionlib
import threading
import time

# ROS Actions
from movex.msg import movementAction, movementGoal,
    movementFeedback, movementResult

from std_msgs.msg import Float64

# SambaCommand messages
from movex.srv import Grab, Lift, Move, Turn

from rospy import ServiceException
from sambautil import import enum
from nodes.services.http_service import SambaCmdNames
from definitions.samba_settings import SambaActionServerName

# Define some organizational structures
grab_request = rospy.ServiceProxy(SambaCmdNames.GRAB, Grab)
lift_request = rospy.ServiceProxy(SambaCmdNames.LIFT, Lift)
movement_request = rospy.ServiceProxy(SambaCmdNames.MOVE, Move)
turn_request = rospy.ServiceProxy(SambaCmdNames.TURN, Turn)

SambaPoseTopics = {SambaCmdNames.GRAB: "pose/angle_2",
    SambaCmdNames.LIFT: "pose/angle_1"}

# Dictionary of service requests
SambaCmdFunc = {SambaCmdNames.MOVE: movement_request,
    SambaCmdNames.LIFT: lift_request,
    SambaCmdNames.GRAB: grab_request,
    SambaCmdNames.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 referred to as a Goal.

![Figure 2: Action Server Initialization](image)
4 Goals (movement.action)

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
```

---

Figure 3: Action Server Goal

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:

```python
../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:

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

![Python code snippet](image)

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, etc...

**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.

```python

```def executeCallback(self, goal):
    print("EXECUTE")
    print(goal)
    rospy.loginfo("ActionServer: Goal received from : 
    (\).format(goal.requester))

    # Handle grab request
    if goal.grabRequest.isActive():
        request = goal.grabRequest
        self.handleRequest(SambucaCommandNames.GRAB, request, self.stopGrab)
        self.grabRunning = True

    # Handle turn request
    if goal.turnRequest.isActive():
        request = goal.turnRequest
        self.handleRequest(SambucaCommandNames.TURN, request, self.stopTurn)
        self.turnRunning = True

    # Handle move request
    if goal.moveRequest.isActive():
        request = goal.moveRequest
        self.handleRequest(SambucaCommandNames.MOVE, request, self.stopMove)
        self.moveRunning = True

    # Handle lift request
    if goal.liftRequest.isActive():
        request = goal.liftRequest
        self.handleRequest(SambucaCommandNames.LIFT, request, self.stopLift)
        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 succeeded.

```python

``````# Spin while all actions finish
rospy.loginfo("Waiting for actions to finish")
allStopped = False
while(not allStopped):
    runningList = [self.grabRunning, self.turnRunning,
                  self.liftRunning, self.moveRunning]
    # Spin until all are not running
    # When isRunning is False for all in runningList
    # all(not isRunning... will return True
    allStopped = all((not isRunning for isRunning in runningList))

    # Make sure all flags are false for next request
    self.grabRunning = False
    self.turnRunning = False
    self.moveRunning = False
    self.liftRunning = False

    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 movements.

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.

```
def handleRequest(self, cmdName, request, stopMethod):
    rospy.loginfo("Received request: {0}.\ncommand: {1}. request magnitude: {2}. request duration: {3}.")
    if(cmdName == SanbucaCmdNames.MOVE or cmdName == SanbucaCmdNames.TURN):
        # Get reference to the service proxy
        # And initiate the action
        serviceFunction = SanbucaCmdFunc[cmdName]
        serviceFunction(request.magnitude)
    # Run a timer callback to stop the action if a track movement
    duration = rospy.Duration(request.duration)
    rospy.Timer(period=duration, callback=stopMethod, oneshot=True)
    else:
        # Run a timed arm or bucket command
        if(request.duration > 0.0):
            # Get reference to the service proxy
            # And initiate the action
            serviceFunction = SanbucaCmdFunc[cmdName]
            serviceFunction(request.magnitude)
        # Run a timer callback to stop the action if a track movement
        duration = rospy.Duration(request.duration)
        rospy.Timer(period=duration, callback=stopMethod, oneshot=True)
        else:
            # Create and start a pose observer thread
            poseObserver = PoseObserver(self, SanbucaPoseTopics[cmdName],
                                         request.magnitude)
            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.

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 there 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 unsubscribes 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 ServiceException. A ServiceException occurs when the stop command fails. When a stop command fails, the ActionServer retries the stop command after a small duration.

```
# Timer Callbacks
def stopGrab(self, event):
    try:
        grab_request(0)
    except ServiceException:
        ros prt.err("GRAB STOP FAILED")
    # There was an error in the HTTP Service
    self.grabRunning = False # We are not done yet
    return

def stopTurn(self, event):
    try:
        turn_request(0)
    except ServiceException:
        ros prt.err("TURN STOP FAILED")
    # There was an error in the HTTP Service
    self.turnRunning = False
    return

def stopMove(self, event):
    try:
        move_request(0)
    except ServiceException:
        ros prt.err("MOVE STOP FAILED")
    # There was an error in the HTTP Service
    self.moveRunning = False
    return

def stopLift(self, event):
    try:
        lift_request(0)
    except ServiceException:
        ros prt.err("LIFT STOP FAILED")
    # There was an error in the HTTP Service
    self.liftRunning = False
    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.

```
def notifyStop(self, cmdName):
    if cmdName == SambucaCmdNames.LIFT:
        self.stopLift(None)
    elif cmdName == SambucaCmdNames.Grab:
        self.stopGrab(None)
    else:
        ros prt.err("Unsupported PoseObserver command {}.format(cmdName))
```

Figure 11: ActionServer Timed Stop Methods