## Random Walk Architecture

#### Dan Blanchette

February 27, 2023

### 1 Class Function Descriptions

#### 1.1 def listener callback

This function registers a bumper reflex routine based on a collision. The function is executed as a separate thread to handle the collision detection as an interrupt routine. If the main program thread runs, this function will cancel the current goal and run its own separate routine. Despite my best efforts at this stage in the project, I have been unsuccessful in getting this routine to return control to the drive-away function without an error state being triggered.

```
# bumper stuff
      def listener_callback(self, msg):
2
          This function is called every time self.subscription gets a
          from the Robot. Here it parses the message from the Robot
      and if its
          a 'bump' message, cancel the current action.
          For this to work, make sure you have:
          ros__parameters:
9
              reflexes_enabled: false
10
          in your Application Configuration Parameters File!!!
13
          # If it wasn't doing anything, there's nothing to cancel
14
          if self._goal_uuid is None:
15
16
               return
17
          # msg.detections is an array of HazardDetection from
      HazardDetectionVectors.
          # Other types can be gotten from HazardDetection.msg
19
          for detection in msg.detections:
20
               if detection.type == 1: #If it is a bump
21
                   self.get_logger().warning('HAZARD DETECTED')
22
23
                   with lock: # Make this the only thing happening
24
                       self.get_logger().warning('CANCELING GOAL')
25
                       self._goal_uuid.cancel_goal_async()
26
                       # Loop until the goal status returns canceled
27
```

```
while self._goal_uuid.status is not GoalStatus.
28
      STATUS_CANCELED:
29
                            pass
                       print('Goal canceled.')
30
31
                   # self.get_logger().warning('OUCH! THAT HURTS!')
                   # self._rotate_ac.wait_for_server()
33
                   # # updating dialogue
34
                   # self.get_logger().warning('SERVER AVAILIBLE')
35
                   # self.get_logger().warning('NOW INSIDE SPIN ROUND
36
      PROTOCOL')
37
                   # rot_goal = RotateAngle.Goal()
38
39
                   # rot_goal.angle = (math.pi)
40
                   # self.sendRotationGoal(rot_goal)
41
```

#### 1.2 def sendDriveGoal

This function handles action client drive goals. It uses asynchronous goal handling because the action client is a resource that needs to be protected. If another thread tries to access the same action client, this can result in a race condition or deadlock. To avoid this, a lock is used to block until the results from the action are updated. If the goal is inactive, the unid is reset until this function is called again.

```
def sendDriveGoal(self,goal):
1
2
          Sends a drive goal asynchronously and 'blocks' until the
3
      goal is complete
5
          with lock:
6
              drive_handle = self._drive_ac.send_goal_async(goal)
               while not drive_handle.done():
                  pass # Wait for Action Server to accept goal
9
10
11
               # Hold ID in case we need to cancel it
               self._goal_uuid = drive_handle.result()
13
14
15
16
          while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
              pass # Wait until a Status has been assigned
17
18
          # After getting goalID, Loop while the goal is currently
19
      running
         while self._goal_uuid.status is not GoalStatus.
20
      STATUS_SUCCEEDED:
              if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
21
                   break # If the goal was canceled, stop looping
      otherwise loop until finished
23
              pass
24
```

#### 1.3 def sendRotationGoal

In the same way as the sendDriveGoal() function, sendRotationGoal() uses asynchronous goal handling because the action client is a resource that needs to be protected. If another thread tries to access the same action client, this can result in a race condition or deadlock. To avoid this, a lock is used to block until the results from the action are updated. If the goal is inactive, the unid is reset until this function is called again.

In a later function driveAway(), this function will "trade-off" with other threads once the action client goal functions give up their lock state and are no longer blocking). This is theoretical because while we have some control over the sequencing of instructions, the kernel has the final say in the order it executes the threads. The idea behind this function is that while the action client uses the rotation action list, any other call to the rotation action list will be blocked temporarily.

```
# Rotation Goal
      def sendRotationGoal(self, goal):
2
3
          Sends a rotation goal asynchronously and 'blocks' until the
       goal is complete
          with lock:
6
              rotation_handle = self._rotate_ac.send_goal_async(goal)
              while not rotation_handle.done():
                  pass
               self._goal_uuid = rotation_handle.result()
          while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
12
              pass # Wait until a Status has been assigned
13
14
          while self._goal_uuid.status is not GoalStatus.
15
      STATUS_SUCCEEDED:
              if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
16
                  break # If the goal was canceled, stop looping;
      otherwise, loop until finished
              pass
19
20
          with lock:
              # Reset the goal ID; nothing should be running
21
              self._goal_uuid = None
```

#### 1.4 def driveAway

this function runs the following operations: undocking the robot, driving away from the dock 1.0m, looping the random walk for ten iterations, calculating and rotating to a random angle(by radian value), driving .75m meters in the selected direction, then docking the robot. The self.-undock-ac.-send-goal(undock goal), sendDriveGoal() and sendRotationGoal() and self.-dock-ac.send-goal(dock-goal) send the goal status and receives a callback once the routine is either successful or fails. In sequence, the actions run asynchronously, with each action server routine temporarily blocking until it becomes available again.

```
def drive_away(self):
2
3
       # list of rotations by radian values for the rotateAngle action
       list
          radians = [1/6, 1/4, 1/3, 1/2, 2/3, 3/4, 5/6, 1, 7/6, 5/4,
      4/3, 3/2, 5/3, 7/4, 11/6, 2,
       -1/6, -1/4, -1/3, -1/2, -2/3, -3/4, -5/6, -1, -7/6, -5/4, -4/3, -3/2, -5/3, -7/4, -11/6, -2]
5
6
      Undocks the robot and drives out a meter asynchronously.
      # Freshly started, undock
9
           self.get_logger().warning('WAITING FOR SERVER')
      # wait until the robot server is found and ready to receive a
      new goal
           self._undock_ac.wait_for_server()
           self.get_logger().warning('SERVER AVAILABLE')
13
           self.get_logger().warning('UNDOCKING')
14
      # create a new Undock goal object to send to the server
           undock_goal = Undock.Goal()
17
18
           self._undock_ac.send_goal(undock_goal)
19
           self.get_logger().warning('UNDOCKED')
20
21
      # wait for DriveDistance action server (blocking)
22
           self._drive_ac.wait_for_server()
23
           self.get_logger().warning('DRIVING!')
24
25
      # create a goal object and specify the distance to drive
26
           drive_goal = DriveDistance.Goal()
27
           drive_goal.distance = 1.0
28
29
      # send goal to async function
30
31
           self.sendDriveGoal(drive_goal)
           # repeat the random walk routine for 10 interations
32
33
           for i in range(0, 10):
               # send rotation goal
               self.get_logger().warning('WAITING FOR SERVER')
35
36
               # block until the rotate action server is ready
37
               self._rotate_ac.wait_for_server()
               # update dialogue
38
               self.get_logger().warning('SERVER AVAILABLE')
39
               self.get_logger().warning('ROTATING')
40
41
```

```
create a new rotation goal object and define the
42
43
               rotation (in radians) to spin.
44
               rot_goal = RotateAngle.Goal()
45
46
               using the random library, choose a fractional value
47
               from the list and multiply by pi (to get radians)
48
49
50
               rot_goal.angle = (random.choice(radians) * math.pi)
51
               \# send the goal to the async rotation function
               self.sendRotationGoal(rot_goal)
52
53
               self.get_logger().warning('WAITING FOR SERVER')
54
55
               # prepare to drive forward
               self._drive_ac.wait_for_server()
56
               self.get_logger().warning('SERVER AVAILIBLE')
57
               self.get_logger().warning('DRIVING!')
58
59
60
               drive_goal2 = DriveDistance.Goal()
               \mbox{\tt\#} drive .75 meters in the direction of the previously
61
       selected angle
               drive\_goal2.distance = 0.75
62
63
64
               self.sendDriveGoal(drive_goal2)
               self.get_logger().warning(f'ITERATION:{i+1}')
65
66
           self.get_logger().warning('WAITING FOR SERVER')
67
           self._dock_ac.wait_for_server()
68
           self.get_logger().warning('SERVER AVAILIBLE')
69
           self.get_logger().warning('DOCKING!')
70
71
           dock_goal = Dock.Goal()
           self._dock_ac.send_goal(dock_goal)
72
```

## 2 Block Diagram

# Random Walk Dan Blanchette | February 27, 2023

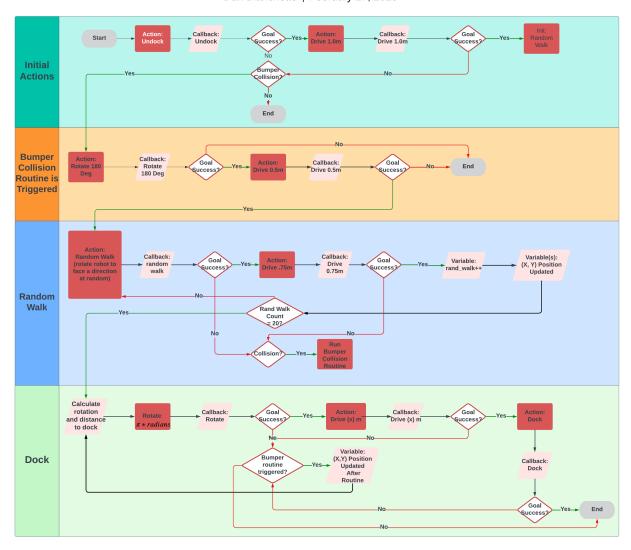


Figure 1: Random Walk Block Diagram

#### 3 Source Code

```
# Project: ROS2 Random Walk irobot Create3
2 # Date Started: 2/22/2023
3 # Date Modified: 2/27/2023
_{4} # Author Dan Blanchette
5 # Credit: Jordan Reed for catching my indent errors on a couple of
      my functions
7 import rclpy
8 from rclpy.node import Node
9 from rclpy.action.client import ActionClient
10 from rclpy.qos import qos_profile_sensor_data
from rclpy.callback_groups import MutuallyExclusiveCallbackGroup
12 from geometry_msgs.msg import PoseStamped
13 from action_msgs.msg._goal_status import GoalStatus
15 import irobot_create_msgs
16 from irobot_create_msgs.action import DriveDistance, Undock, Dock,
      RotateAngle
17 from irobot_create_msgs.msg import HazardDetectionVector
19 from pynput.keyboard import KeyCode
20 from key_commander import KeyCommander
21 from threading import Lock
22 from rclpy.executors import MultiThreadedExecutor
24 import random
25 import math
26
27
28 # To help with Multithreading
29 lock = Lock()
31 class Chocobo(Node):
32
33
      Class to coordinate actions and subscriptions
34
35
      def __init__(self, namespace):
36
           super().__init__('chocobo')
37
38
          # 2 Seperate Callback Groups for handling the bumper
39
      Subscription and Action Clients
           cb_Subscripion = MutuallyExclusiveCallbackGroup()
40
41
           #cb_Action = cb_Subscripion
          cb_Action =MutuallyExclusiveCallbackGroup()
42
43
          # Subscription to Hazards, the callback function attached
44
      only looks for bumper hits
           self.subscription = self.create_subscription(
45
               HazardDetectionVector, f'/{namespace}/hazard_detection'
46
       , self.listener_callback, qos_profile_sensor_data,
      callback_group=cb_Subscripion)
47
           # Action clients for movements
48
           self._undock_ac = ActionClient(self, Undock, f'/{namespace
49
```

```
}/undock', callback_group=cb_Action)
           # added dock action
           self._dock_ac = ActionClient(self, Dock, f'/{namespace}/
      dock', callback_group=cb_Action)
           {\tt self.\_drive\_ac} \; = \; {\tt ActionClient(self, DriveDistance, f'/\{} \\
      namespace}/drive_distance',callback_group=cb_Action)
           # added rotate angle topics
           self._rotate_ac = ActionClient(self, RotateAngle, f'/{
54
      namespace}/rotate_angle',callback_group=cb_Action )
           # Variables
56
57
           self._goal_uuid = None
58
59
      # bumper stuff
60
       def listener_callback(self, msg):
61
62
          This function is called every time self.subscription gets a
63
       message
           from the Robot. Here it parses the message from the Robot
64
      and if its
          a 'bump' message, cancel the current action.
65
66
67
          For this to work, make sure you have:
           ros__parameters:
68
               reflexes_enabled: false
69
           in your Application Configuration Parameters File!!!
70
71
72
           # If it wasn't doing anything, there's nothing to cancel
73
74
           if self._goal_uuid is None:
75
               return
76
          \# msg.detections is an array of HazardDetection from
77
      HazardDetectionVectors.
           # Other types can be gotten from HazardDetection.msg
           for detection in msg.detections:
79
80
               if detection.type == 1: #If it is a bump
                   self.get_logger().warning('HAZARD DETECTED')
81
82
                   with lock: # Make this the only thing happening
83
                        self.get_logger().warning('CANCELING GOAL')
84
85
                       self._goal_uuid.cancel_goal_async()
                       # Loop until the goal status returns canceled
86
                       while self._goal_uuid.status is not GoalStatus.
87
      STATUS_CANCELED:
88
                       print('Goal canceled.')
89
90
                   # self.get_logger().warning('OUCH! THAT HURTS!')
91
                   # self._rotate_ac.wait_for_server()
92
                   # # updating dialogue
93
94
                   # self.get_logger().warning('SERVER AVAILIBLE')
                   # self.get_logger().warning('NOW INSIDE SPIN ROUND
95
      PROTOCOL')
96
                   # rot_goal = RotateAngle.Goal()
97
```

```
# rot_goal.angle = (math.pi)
98
99
                   # self.sendRotationGoal(rot_goal)
101
         -----Async send goal calls
103
       def sendDriveGoal(self,goal):
104
106
           Sends a drive goal asynchronously and 'blocks' until the
       goal is complete
108
109
           with lock:
               drive_handle = self._drive_ac.send_goal_async(goal)
               while not drive_handle.done():
111
                   pass # Wait for Action Server to accept goal
114
               # Hold ID in case we need to cancel it
               self._goal_uuid = drive_handle.result()
118
119
           while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
               pass # Wait until a Status has been assigned
120
121
           # After getting goalID, Loop while the goal is currently
       running
           while self._goal_uuid.status is not GoalStatus.
123
       STATUS_SUCCEEDED:
               if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
                   break # If the goal was canceled, stop looping
125
       otherwise, loop until finished
126
               pass
127
           with lock:
128
129
               # Reset the goal ID; nothing should be running
               self._goal_uuid = None
130
131
       # Rotation Goal
132
       def sendRotationGoal(self, goal):
133
134
           with lock:
               rotation_handle = self._rotate_ac.send_goal_async(goal)
136
137
               while not rotation_handle.done():
                   pass
138
               self._goal_uuid = rotation_handle.result()
139
140
           while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
141
               pass # Wait until a Status has been assigned
142
143
           while self._goal_uuid.status is not GoalStatus.
144
       STATUS_SUCCEEDED:
               if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
145
                  break # If the goal was canceled, stop looping;
146
```

```
otherwise, loop until finished
                pass
148
            with lock:
149
                # Reset the goal ID; nothing should be running
                self._goal_uuid = None
151
152
153 #
154
       def drive_away(self):
156
157
            Undocks the robot and drives out a meter asynchronously.
158
159
160
           # list of rotations by radian values for the rotateAngle
       action list
           radians = [1/6, 1/4, 1/3, 1/2, 2/3, 3/4, 5/6, 1, 7/6, 5/4,
       4/3, 3/2, 5/3, 7/4, 11/6, 2,
       -1/6, -1/4, -1/3, -1/2, -2/3, -3/4, -5/6, -1, -7/6, -5/4, -4/3, -3/2, -5/3, -7/4, -11/6, -2]
164
       # Freshly started, undock
            self.get_logger().warning('WAITING FOR SERVER')
       # wait until the robot server is found and ready to receive a
166
       new goal
167
            self._undock_ac.wait_for_server()
            self.get_logger().warning('SERVER AVAILABLE')
168
            self.get_logger().warning('UNDOCKING')
169
       # create new Undock goal object to send to server
171
            undock_goal = Undock.Goal()
172
173
174
            self._undock_ac.send_goal(undock_goal)
            self.get_logger().warning('UNDOCKED')
176
177
       # wait for DriveDistance action server (blocking)
            self._drive_ac.wait_for_server()
178
179
            self.get_logger().warning('DRIVING!')
180
       # create a goal object and specify the distance to drive
181
182
            drive_goal = DriveDistance.Goal()
            drive\_goal.distance = 1.0
183
184
185
       # send goal to async function
            self.sendDriveGoal(drive_goal)
186
187
            for i in range(0, 10):
                # send rotation goal
188
                self.get_logger().warning('WAITING FOR SERVER')
189
190
                self._rotate_ac.wait_for_server()
191
                # updating dialogue
                self.get_logger().warning('SERVER AVAILIBLE')
                self.get_logger().warning('ROTATING')
194
195
                rot_goal = RotateAngle.Goal()
196
```

```
rot_goal.angle = (random.choice(radians) * math.pi)
197
198
                self.sendRotationGoal(rot_goal)
199
200
                self.get_logger().warning('WAITING FOR SERVER')
201
                self._drive_ac.wait_for_server()
202
203
                self.get_logger().warning('SERVER AVAILIBLE')
                self.get_logger().warning('DRIVING!')
204
205
                drive_goal2 = DriveDistance.Goal()
206
                drive_goal2.distance = 0.05
207
208
                self.sendDriveGoal(drive_goal2)
209
210
                self.get_logger().warning(f'ITERATION:{i+1}')
211
            self.get_logger().warning('WAITING FOR SERVER')
212
213
            self._dock_ac.wait_for_server()
            self.get_logger().warning('SERVER AVAILIBLE')
214
215
            self.get_logger().warning('DOCKING!')
            dock_goal = Dock.Goal()
216
217
            self._dock_ac.send_goal(dock_goal)
218
219
220
   if __name__ == '__main__':
221
222
       rclpy.init()
       namespace = 'create3_05B9'
224
       c = Chocobo(namespace)
225
226
227
       # 1 thread for the Subscription, another for the Action Clients
       exec = MultiThreadedExecutor(2)
228
       exec.add_node(c)
229
230
       keycom = KeyCommander([
231
232
            (KeyCode(char='r'), c.drive_away),
233
234
       print("r: Start drive_away")
235
236
       try:
           exec.spin() # execute slash callbacks until shutdown or
237
       destroy is called
238
       except KeyboardInterrupt:
           print('KeyboardInterrupt, shutting down.')
239
            print("Shutting down executor")
240
241
            exec.shutdown()
           print("Destroying Node")
242
243
           c.destroy_node()
           print("Shutting down RCLPY")
244
           rclpy.try_shutdown()
```