

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.

```
1  # bumper stuff
2  def listener_callback(self, msg):
3      '''
4      This function is called every time self.subscription gets a
5      message
6      from the Robot. Here it parses the message from the Robot
7      and if its
8      a 'bump' message, cancel the current action.
9
10     For this to work, make sure you have:
11     ros__parameters:
12         reflexes_enabled: false
13     in your Application Configuration Parameters File!!!
14
15     # If it wasn't doing anything, there's nothing to cancel
16     if self._goal_uuid is None:
17         return
18
19     # msg.detections is an array of HazardDetection from
20     HazardDetectionVectors.
21     # Other types can be gotten from HazardDetection.msg
22     for detection in msg.detections:
23         if detection.type == 1: #If it is a bump
24             self.get_logger().warning('HAZARD DETECTED')
25
26         with lock: # Make this the only thing happening
27             self.get_logger().warning('CANCELING GOAL')
28             self._goal_uuid.cancel_goal_async()
29             # Loop until the goal status returns canceled
```

```

28         while self._goal_uuid.status is not GoalStatus.
STATUS_CANCELED:
29             pass
30             print('Goal canceled.')
```

```

31
32             # self.get_logger().warning('OUCH! THAT HURTS!')
33             # self._rotate_ac.wait_for_server()
34             # # updating dialogue
35             # self.get_logger().warning('SERVER AVAILIBLE')
36             # self.get_logger().warning('NOW INSIDE SPIN ROUND
PROTOCOL')
```

```

37
38             # rot_goal = RotateAngle.Goal()
39             # rot_goal.angle = (math.pi)
40
41             # self.sendRotationGoal(rot_goal)
```

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 uuid is reset until this function is called again.

```

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

```

25         with lock:
26             # Reset the goal ID, nothing should be running
27             self._goal_uuid = None
28
29     # Rotation Goal

```

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 uuid 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.

```

1     # Rotation Goal
2     def sendRotationGoal(self, goal):
3         """
4         Sends a rotation goal asynchronously and 'blocks' until the
5         goal is complete
6         """
7         with lock:
8             rotation_handle = self._rotate_ac.send_goal_async(goal)
9             while not rotation_handle.done():
10                 pass
11             self._goal_uuid = rotation_handle.result()
12
13         while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
14             pass # Wait until a Status has been assigned
15
16         while self._goal_uuid.status is not GoalStatus.
17             STATUS_SUCCEEDED:
18             if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
19             :
20                 break # If the goal was canceled, stop looping;
21                 otherwise, loop until finished
22             pass
23
24         with lock:
25             # Reset the goal ID; nothing should be running
26             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.

```
1  def drive_away(self):
2
3      # list of rotations by radian values for the rotateAngle action
      list
4      radians = [1/6, 1/4, 1/3, 1/2, 2/3, 3/4, 5/6, 1, 7/6, 5/4,
5                  4/3, 3/2, 5/3, 7/4, 11/6, 2,
6                  -1/6, -1/4, -1/3, -1/2, -2/3, -3/4, -5/6, -1, -7/6,
7                  -5/4, -4/3, -3/2, -5/3, -7/4, -11/6, -2]
8      """
9      Undocks the robot and drives out a meter asynchronously.
10     """
11     # Freshly started, undock
12     self.get_logger().warning('WAITING FOR SERVER')
13     # wait until the robot server is found and ready to receive a
      new goal
14     self._undock_ac.wait_for_server()
15     self.get_logger().warning('SERVER AVAILABLE')
16     self.get_logger().warning('UNDocking')
17
18     # create a new Undock goal object to send to the server
19     undock_goal = Undock.Goal()
20
21     self._undock_ac.send_goal(undock_goal)
22     self.get_logger().warning('UNDocked')
23
24     # wait for DriveDistance action server (blocking)
25     self._drive_ac.wait_for_server()
26     self.get_logger().warning('DRIVING!')
27
28     # create a goal object and specify the distance to drive
29     drive_goal = DriveDistance.Goal()
30     drive_goal.distance = 1.0
31
32     # send goal to async function
33     self.sendDriveGoal(drive_goal)
34     # repeat the random walk routine for 10 iterations
35     for i in range(0, 10):
36         # send rotation goal
37         self.get_logger().warning('WAITING FOR SERVER')
38         # block until the rotate action server is ready
39         self._rotate_ac.wait_for_server()
40         # update dialogue
41         self.get_logger().warning('SERVER AVAILABLE')
42         self.get_logger().warning('ROTATING')
43     """
```

```

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

```

2 Block Diagram

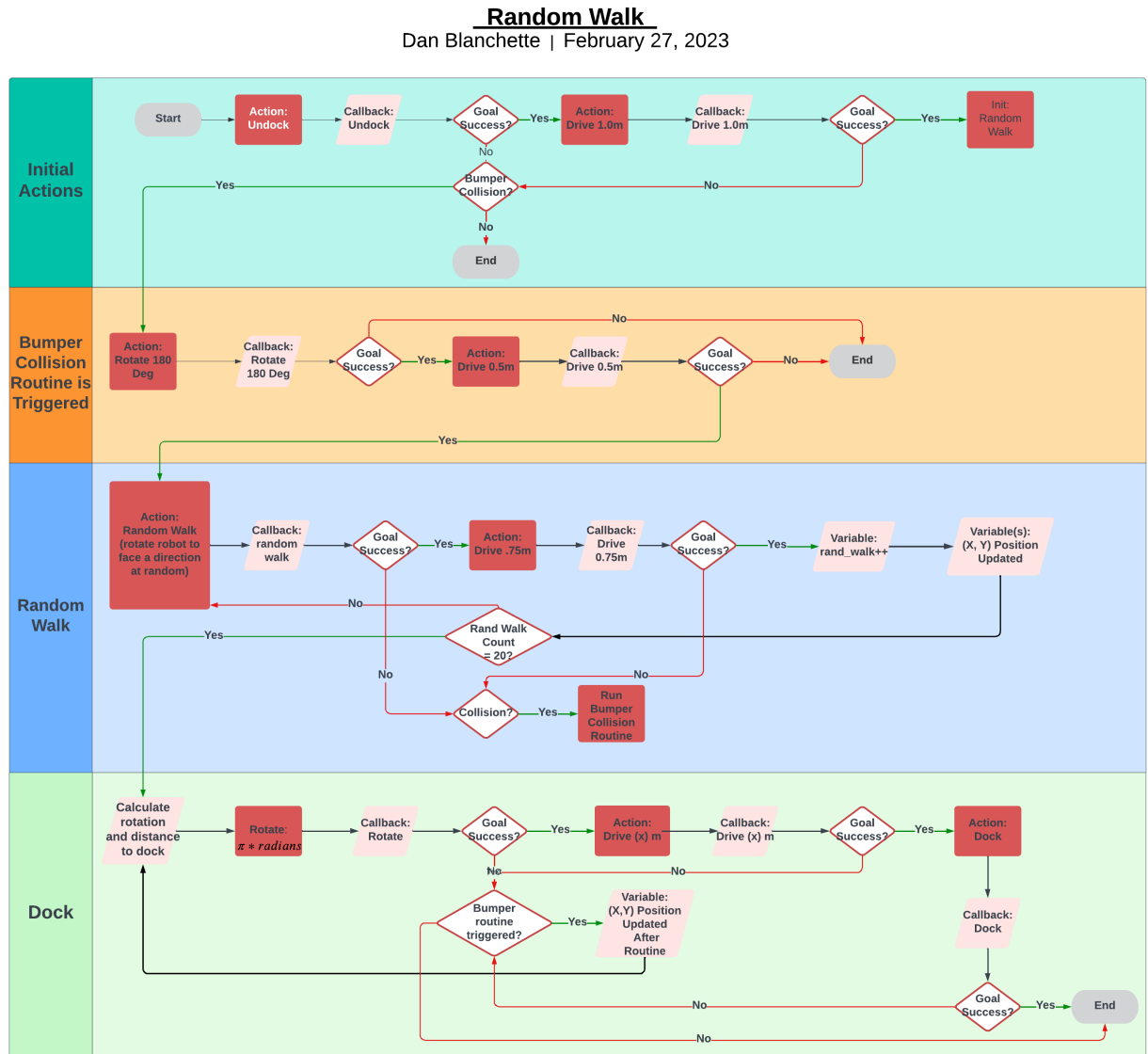


Figure 1: Random Walk Block Diagram

3 Source Code

```
1 # 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
6
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
11 from rclpy.callback_groups import MutuallyExclusiveCallbackGroup
12 from geometry_msgs.msg import PoseStamped
13 from action_msgs.msg._goal_status import GoalStatus
14
15 import irobot_create_msgs
16 from irobot_create_msgs.action import DriveDistance, Undock, Dock,
   RotateAngle
17 from irobot_create_msgs.msg import HazardDetectionVector
18
19 from pynput.keyboard import KeyCode
20 from key_commander import KeyCommander
21 from threading import Lock
22 from rclpy.executors import MultiThreadedExecutor
23
24 import random
25 import math
26
27
28 # To help with Multithreading
29 lock = Lock()
30
31 class Chocobo(Node):
32     """
33     Class to coordinate actions and subscriptions
34     """
35
36     def __init__(self, namespace):
37         super().__init__('chocobo')
38
39         # 2 Seperate Callback Groups for handling the bumper
40         Subscription and Action Clients
41         cb_Subscription = MutuallyExclusiveCallbackGroup()
42         #cb_Action = cb_Subscription
43         cb_Action =MutuallyExclusiveCallbackGroup()
44
45         # Subscription to Hazards, the callback function attached
46         only looks for bumper hits
47         self.subscription = self.create_subscription(
48             HazardDetectionVector, f'/{namespace}/hazard_detection',
49             self.listener_callback, qos_profile_sensor_data,
50             callback_group=cb_Subscription)
51
52         # Action clients for movements
53         self._undock_ac = ActionClient(self, Undock, f'/{namespace}
```

```

50     }/undock', callback_group=cb_Action)
51     # added dock action
52     self._dock_ac = ActionClient(self, Dock, f'/{namespace}/
dock', callback_group=cb_Action)
53     self._drive_ac = ActionClient(self, DriveDistance, f'/{
namespace}/drive_distance', callback_group=cb_Action)
54     # added rotate angle topics
55     self._rotate_ac = ActionClient(self, RotateAngle, f'/{
namespace}/rotate_angle', callback_group=cb_Action )
56
57     # Variables
58     self._goal_uuid = None
59
60     # bumper stuff
61     def listener_callback(self, msg):
62         '''
63         This function is called every time self.subscription gets a
64         message
65         from the Robot. Here it parses the message from the Robot
66         and if its
67         a 'bump' message, cancel the current action.
68
69         For this to work, make sure you have:
70         ros__parameters:
71             reflexes_enabled: false
72         in your Application Configuration Parameters File!!!
73         '''
74
75         # If it wasn't doing anything, there's nothing to cancel
76         if self._goal_uuid is None:
77             return
78
79         # msg.detections is an array of HazardDetection from
80         HazardDetectionVectors.
81         # Other types can be gotten from HazardDetection.msg
82         for detection in msg.detections:
83             if detection.type == 1: #If it is a bump
84                 self.get_logger().warning('HAZARD DETECTED')
85
86                 with lock: # Make this the only thing happening
87                     self.get_logger().warning('CANCELING GOAL')
88                     self._goal_uuid.cancel_goal_async()
89                     # Loop until the goal status returns canceled
90                     while self._goal_uuid.status is not GoalStatus.
91                     STATUS_CANCELED:
92                         pass
93                     print('Goal canceled.')
94
95                     # self.get_logger().warning('OUCH! THAT HURTS!')
96                     # self._rotate_ac.wait_for_server()
97                     # # updating dialogue
98                     # self.get_logger().warning('SERVER AVAILIBLE')
99                     # self.get_logger().warning('NOW INSIDE SPIN ROUND
100                     PROTOCOL')
101
102                     # rot_goal = RotateAngle.Goal()

```



```

98         # rot_goal.angle = (math.pi)
99
100        # self.sendRotationGoal(rot_goal)
101
102
103    #-----Async send goal calls
104    -----
105    def sendDriveGoal(self,goal):
106        """
107        Sends a drive goal asynchronously and 'blocks' until the
108        goal is complete
109        """
110
111        with lock:
112            drive_handle = self._drive_ac.send_goal_async(goal)
113            while not drive_handle.done():
114                pass # Wait for Action Server to accept goal
115
116            # Hold ID in case we need to cancel it
117            self._goal_uuid = drive_handle.result()
118
119
120            while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
121                pass # Wait until a Status has been assigned
122
123            # After getting goalID, Loop while the goal is currently
124            running
125            while self._goal_uuid.status is not GoalStatus.
126            STATUS_SUCCEEDED:
127                if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
128                :
129                    break # If the goal was canceled, stop looping
130            otherwise, loop until finished
131            pass
132
133            with lock:
134                # Reset the goal ID; nothing should be running
135                self._goal_uuid = None
136
137    # Rotation Goal
138    def sendRotationGoal(self, goal):
139
140        with lock:
141            rotation_handle = self._rotate_ac.send_goal_async(goal)
142            while not rotation_handle.done():
143                pass
144            self._goal_uuid = rotation_handle.result()
145
146            while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
147                pass # Wait until a Status has been assigned
148
149            while self._goal_uuid.status is not GoalStatus.
150            STATUS_SUCCEEDED:
151                if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
152                :
153                    break # If the goal was canceled, stop looping;

```

```

147         otherwise, loop until finished
148             pass
149
150         with lock:
151             # Reset the goal ID; nothing should be running
152             self._goal_uuid = None
153     # -----
154
155
156     def drive_away(self):
157         """
158         Undocks the robot and drives out a meter asynchronously.
159         """
160         # list of rotations by radian values for the rotateAngle
161         # action list
162         radians = [1/6, 1/4, 1/3, 1/2, 2/3, 3/4, 5/6, 1, 7/6, 5/4,
163                    4/3, 3/2, 5/3, 7/4, 11/6, 2,
164                    -1/6, -1/4, -1/3, -1/2, -2/3, -3/4, -5/6, -1, -7/6,
165                    -5/4, -4/3, -3/2, -5/3, -7/4, -11/6, -2]
166
167         # Freshly started, undock
168         self.get_logger().warning('WAITING FOR SERVER')
169         # wait until the robot server is found and ready to receive a
170         # new goal
171         self._undock_ac.wait_for_server()
172         self.get_logger().warning('SERVER AVAILABLE')
173         self.get_logger().warning('UNDocking')
174
175         # create new Undock goal object to send to server
176         undock_goal = Undock.Goal()
177
178         self._undock_ac.send_goal(undock_goal)
179         self.get_logger().warning('UNDocked')
180
181         # wait for DriveDistance action server (blocking)
182         self._drive_ac.wait_for_server()
183         self.get_logger().warning('DRIVING!')
184
185         # create a goal object and specify the distance to drive
186         drive_goal = DriveDistance.Goal()
187         drive_goal.distance = 1.0
188
189         # send goal to async function
190         self.sendDriveGoal(drive_goal)
191         for i in range(0, 10):
192             # send rotation goal
193             self.get_logger().warning('WAITING FOR SERVER')
194
195             self._rotate_ac.wait_for_server()
196             # updating dialogue
197             self.get_logger().warning('SERVER AVAILABLE')
198             self.get_logger().warning('ROTATING')
199
200             rot_goal = RotateAngle.Goal()

```

```

197         rot_goal.angle = (random.choice(radians) * math.pi)
198
199         self.sendRotationGoal(rot_goal)
200
201         self.get_logger().warning('WAITING FOR SERVER')
202         self._drive_ac.wait_for_server()
203         self.get_logger().warning('SERVER AVAILABLE')
204         self.get_logger().warning('DRIVING!')
205
206         drive_goal2 = DriveDistance.Goal()
207         drive_goal2.distance = 0.05
208
209         self.sendDriveGoal(drive_goal2)
210         self.get_logger().warning(f'ITERATION:{i+1}')
211
212         self.get_logger().warning('WAITING FOR SERVER')
213         self._dock_ac.wait_for_server()
214         self.get_logger().warning('SERVER AVAILABLE')
215         self.get_logger().warning('DOCKING!')
216         dock_goal = Dock.Goal()
217         self._dock_ac.send_goal(dock_goal)
218
219
220
221 if __name__ == '__main__':
222     rclpy.init()
223
224     namespace = 'create3_05B9'
225     c = Chocobo(namespace)
226
227     # 1 thread for the Subscription, another for the Action Clients
228     exec = MultiThreadedExecutor(2)
229     exec.add_node(c)
230
231     keycom = KeyCommander([
232         (KeyCode(char='r'), c.drive_away),
233     ])
234
235     print("r: Start drive_away")
236     try:
237         exec.spin() # execute slash callbacks until shutdown or
238         # destroy is called
239     except KeyboardInterrupt:
240         print('KeyboardInterrupt, shutting down.')
241         print("Shutting down executor")
242         exec.shutdown()
243         print("Destroying Node")
244         c.destroy_node()
245         print("Shutting down RCLPY")
246         rclpy.try_shutdown()

```