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):
        '''
        This function is called every time self.subscription gets a message
        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:
            reflexes_enabled: false
        in your Application Configuration Parameters File!!!
        '''

        # If it wasn't doing anything, there's nothing to cancel
        if self._goal_uuid is None:
            return

        # msg.detections is an array of HazardDetection from HazardDetectionVectors.
        # Other types can be gotten from HazardDetection.msg
        for detection in msg.detections:
            if detection.type == 1:   #If it is a bump
                self.get_logger().warning('HAZARD DETECTED')

                with lock: # Make this the only thing happening
                    self.get_logger().warning('CANCELING GOAL')
                    self._goal_uuid.cancel_goal_async()
                    # Loop until the goal status returns canceled
                    while self._goal_uuid.status is not GoalStatus.STATUS_CANCELED:
                        pass
                    print('Goal canceled.')

                # self.get_logger().warning('OUCH! THAT HURTS!')
                # self._rotate_ac.wait_for_server()
                # # updating dialogue
                # self.get_logger().warning('SERVER AVAILIBLE')
                # self.get_logger().warning('NOW INSIDE SPIN ROUND PROTOCOL')

                # rot_goal = RotateAngle.Goal()
                # rot_goal.angle = (math.pi)

                # 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.

```
def sendDriveGoal(self,goal):
    """
    Sends a drive goal asynchronously and 'blocks' until the goal is complete
    """

    with lock:
        drive_handle = self._drive_ac.send_goal_async(goal)
        while not drive_handle.done():
            pass # Wait for Action Server to accept goal

        # Hold ID in case we need to cancel it
        self._goal_uuid = drive_handle.result()



    while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
        pass # Wait until a Status has been assigned

    # After getting goalID, Loop while the goal is currently running
    while self._goal_uuid.status is not GoalStatus.STATUS_SUCCEEDED:
        if self._goal_uuid.status is GoalStatus.STATUS_CANCELED:
            break # If the goal was canceled, stop looping otherwise loop until finished
        pass

    with lock:
        # Reset the goal ID, nothing should be running
        self._goal_uuid = None

# 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.

```
    # Rotation Goal
    def sendRotationGoal(self, goal):
        """
        Sends a rotation goal asynchronously and 'blocks' until the
     goal is complete
        """
        with lock:
            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:
            pass # Wait until a Status has been assigned

        while self._goal_uuid.status is not GoalStatus.
    STATUS_SUCCEEDED:
            if self._goal_uuid.status is GoalStatus.STATUS_CANCELED
    :
                break # If the goal was canceled, stop looping;
    otherwise, loop until finished
            pass

        with lock:
            # Reset the goal ID; nothing should be running
            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):

# 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]
"""
Undocks the robot and drives out a meter asynchronously.
"""
# Freshly started, undock
    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’)
    self.get_logger().warning(’UNDOCKING’)

# create a new Undock goal object to send to the server
    undock_goal = Undock.Goal()

    self._undock_ac.send_goal(undock_goal)
    self.get_logger().warning(’UNDOCKED’)

# wait for DriveDistance action server (blocking)
    self._drive_ac.wait_for_server()
    self.get_logger().warning(’DRIVING!’)

# create a goal object and specify the distance to drive
    drive_goal = DriveDistance.Goal()
    drive_goal.distance = 1.0

# send goal to async function
    self.sendDriveGoal(drive_goal)
    # repeat the random walk routine for 10 interations
    for i in range(0, 10):
        # send rotation goal
        self.get_logger().warning(’WAITING FOR SERVER’)
        # block until the rotate action server is ready
        self._rotate_ac.wait_for_server()
        # update dialogue
        self.get_logger().warning(’SERVER AVAILABLE’)
        self.get_logger().warning(’ROTATING’)
        """
```

```
            create a new rotation goal object and define the
            rotation (in radians) to spin.
            """
            rot_goal = RotateAngle.Goal()
            """
            using the random library, choose a fractional value
            from the list and multiply by pi (to get radians)
            """
            rot_goal.angle = (random.choice(radians) * math.pi)
            # send the goal to the async rotation function
            self.sendRotationGoal(rot_goal)

            self.get_logger().warning('WAITING FOR SERVER')
            # prepare to drive forward
            self._drive_ac.wait_for_server()
            self.get_logger().warning('SERVER AVAILIBLE')
            self.get_logger().warning('DRIVING!')

            drive_goal2 = DriveDistance.Goal()
            # drive .75 meters in the direction of the previously selected angle
            drive_goal2.distance = 0.75

            self.sendDriveGoal(drive_goal2)
            self.get_logger().warning(f'ITERATION:{i+1}')

        self.get_logger().warning('WAITING FOR SERVER')
        self._dock_ac.wait_for_server()
        self.get_logger().warning('SERVER AVAILIBLE')
        self.get_logger().warning('DOCKING!')
        dock_goal = Dock.Goal()
        self._dock_ac.send_goal(dock_goal)
```

2 Block Diagram

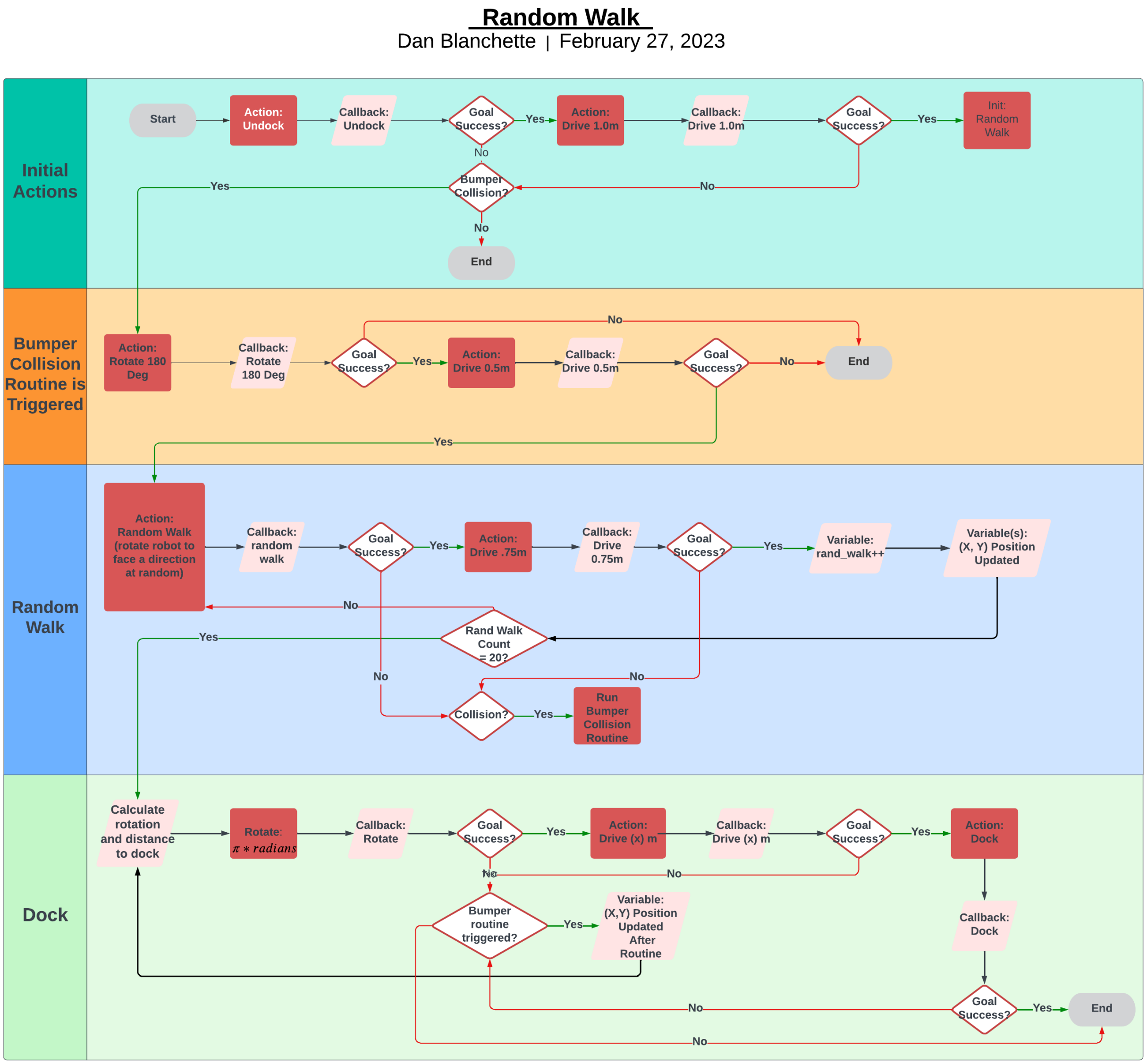

Figure 1: Random Walk Block Diagram

3 Source Code

```python
# Project: ROS2 Random Walk irobot Create3
# Date Started: 2/22/2023
# Date Modified: 2/27/2023
# Author Dan Blanchette
# Credit: Jordan Reed for catching my indent errors on a couple of my functions

import rclpy
from rclpy.node import Node
from rclpy.action.client import ActionClient
from rclpy.qos import qos_profile_sensor_data
from rclpy.callback_groups import MutuallyExclusiveCallbackGroup
from geometry_msgs.msg import PoseStamped
from action_msgs.msg._goal_status import GoalStatus

import irobot_create_msgs
from irobot_create_msgs.action import DriveDistance, Undock, Dock, RotateAngle
from irobot_create_msgs.msg import HazardDetectionVector

from pynput.keyboard import KeyCode
from key_commander import KeyCommander
from threading import Lock
from rclpy.executors import MultiThreadedExecutor

import random
import math


# To help with Multithreading
lock = Lock()

class Chocobo(Node):
    """
    Class to coordinate actions and subscriptions
    """

    def __init__(self, namespace):
        super().__init__('chocobo')

        # 2 Seperate Callback Groups for handling the bumper Subscription and Action Clients
        cb_Subscripion = MutuallyExclusiveCallbackGroup()
        #cb_Action = cb_Subscripion
        cb_Action =MutuallyExclusiveCallbackGroup()

        # Subscription to Hazards, the callback function attached only looks for bumper hits
        self.subscription = self.create_subscription(
            HazardDetectionVector, f'/{namespace}/hazard_detection', self.listener_callback, qos_profile_sensor_data, callback_group=cb_Subscripion)

        # Action clients for movements
        self._undock_ac = ActionClient(self, Undock, f'/{namespace
}/undock',callback_group=cb_Action)
        # added dock action
        self._dock_ac = ActionClient(self, Dock, f'/{namespace}/dock', callback_group=cb_Action)
        self._drive_ac = ActionClient(self, DriveDistance, f'/{namespace}/drive_distance',callback_group=cb_Action)
        # added rotate angle topics
        self._rotate_ac = ActionClient(self, RotateAngle, f'/{namespace}/rotate_angle',callback_group=cb_Action )

        # Variables
        self._goal_uuid = None


    # bumper stuff
    def listener_callback(self, msg):
        '''
        This function is called every time self.subscription gets a message
        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:
            reflexes_enabled: false
        in your Application Configuration Parameters File!!!
        '''

        # If it wasn't doing anything, there's nothing to cancel
        if self._goal_uuid is None:
            return

        # msg.detections is an array of HazardDetection from HazardDetectionVectors.
        # Other types can be gotten from HazardDetection.msg
        for detection in msg.detections:
            if detection.type == 1:   #If it is a bump
                self.get_logger().warning('HAZARD DETECTED')

                with lock: # Make this the only thing happening
                    self.get_logger().warning('CANCELING GOAL')
                    self._goal_uuid.cancel_goal_async()
                    # Loop until the goal status returns canceled
                    while self._goal_uuid.status is not GoalStatus.STATUS_CANCELED:
                        pass
                    print('Goal canceled.')

                # self.get_logger().warning('OUCH! THAT HURTS!')
                # self._rotate_ac.wait_for_server()
                # # updating dialogue
                # self.get_logger().warning('SERVER AVAILIBLE')
                # self.get_logger().warning('NOW INSIDE SPIN ROUND PROTOCOL')

                # rot_goal = RotateAngle.Goal()
                # rot_goal.angle = (math.pi)

                # self.sendRotationGoal(rot_goal)


#--------------------Async send goal calls -----------------------------
    def sendDriveGoal(self,goal):
        """
        Sends a drive goal asynchronously and ’blocks’ until the goal is complete
        """

        with lock:
            drive_handle = self._drive_ac.send_goal_async(goal)
            while not drive_handle.done():
                pass # Wait for Action Server to accept goal

            # Hold ID in case we need to cancel it
            self._goal_uuid = drive_handle.result()



        while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
            pass # Wait until a Status has been assigned

        # After getting goalID, Loop while the goal is currently running
        while self._goal_uuid.status is not GoalStatus.STATUS_SUCCEEDED:
            if self._goal_uuid.status is GoalStatus.STATUS_CANCELED:
                break # If the goal was canceled, stop looping otherwise, loop until finished
            pass

        with lock:
            # Reset the goal ID; nothing should be running
            self._goal_uuid = None

    # Rotation Goal
    def sendRotationGoal(self, goal):

        with lock:
            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:
            pass # Wait until a Status has been assigned

        while self._goal_uuid.status is not GoalStatus.STATUS_SUCCEEDED:
            if self._goal_uuid.status is GoalStatus.STATUS_CANCELED:
                break # If the goal was canceled, stop looping;
    otherwise, loop until finished
            pass

        with lock:
            # Reset the goal ID; nothing should be running
            self._goal_uuid = None

# ----------------------------------------------------------------------



    def drive_away(self):
        """
        Undocks the robot and drives out a meter asynchronously.
        """
        # 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]

    # Freshly started, undock
        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')
        self.get_logger().warning('UNDOCKING')

    # create new Undock goal object to send to server
        undock_goal = Undock.Goal()

        self._undock_ac.send_goal(undock_goal)
        self.get_logger().warning('UNDOCKED')

    # wait for DriveDistance action server (blocking)
        self._drive_ac.wait_for_server()
        self.get_logger().warning('DRIVING!')

    # create a goal object and specify the distance to drive
        drive_goal = DriveDistance.Goal()
        drive_goal.distance = 1.0

    # send goal to async function
        self.sendDriveGoal(drive_goal)
        for i in range(0, 10):
            # send rotation goal
            self.get_logger().warning('WAITING FOR SERVER')

            self._rotate_ac.wait_for_server()
            # updating dialogue
            self.get_logger().warning('SERVER AVAILIBLE')
            self.get_logger().warning('ROTATING')

            rot_goal = RotateAngle.Goal()
            rot_goal.angle = (random.choice(radians) * math.pi)

            self.sendRotationGoal(rot_goal)

            self.get_logger().warning('WAITING FOR SERVER')
            self._drive_ac.wait_for_server()
            self.get_logger().warning('SERVER AVAILIBLE')
            self.get_logger().warning('DRIVING!')

            drive_goal2 = DriveDistance.Goal()
            drive_goal2.distance = 0.05

            self.sendDriveGoal(drive_goal2)
            self.get_logger().warning(f'ITERATION:{i+1}')

        self.get_logger().warning('WAITING FOR SERVER')
        self._dock_ac.wait_for_server()
        self.get_logger().warning('SERVER AVAILIBLE')
        self.get_logger().warning('DOCKING!')
        dock_goal = Dock.Goal()
        self._dock_ac.send_goal(dock_goal)



if __name__ == '__main__':
    rclpy.init()

    namespace = 'create3_05B9'
    c = Chocobo(namespace)

    # 1 thread for the Subscription, another for the Action Clients
    exec = MultiThreadedExecutor(2)
    exec.add_node(c)

    keycom = KeyCommander([
        (KeyCode(char='r'), c.drive_away),
        ])

    print("r: Start drive_away")
    try:
        exec.spin() # execute slash callbacks until shutdown or destroy is called
    except KeyboardInterrupt:
        print('KeyboardInterrupt, shutting down.')
        print("Shutting down executor")
        exec.shutdown()
        print("Destroying Node")
        c.destroy_node()
        print("Shutting down RCLPY")
        rclpy.try_shutdown()
```