Coordination Robot ROS package

CS269 Multi Robot Systems / 2021 Spring / PA-3

Mingi Jeong / mingi.jeong.gr@dartmouth.edu

1. General

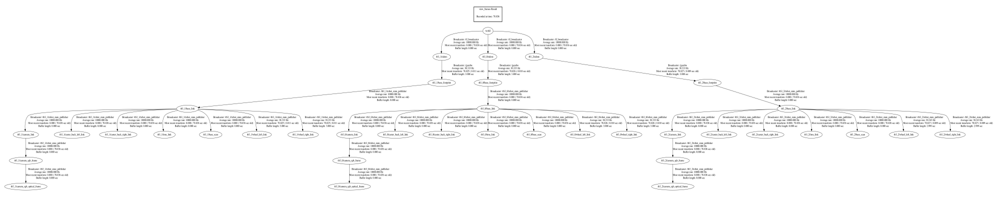
- This package achieves coordination of robots for reaching goal positions by three robots.
- There are two launch files: one launch file (`roslaunch coordination_robot_pkg coordination_environment.launch`) can run all the environments and connection nodes, while the other launch file (`roslaunch coordination_robot_pkg waypoint_send.launch waypoint_task:=NUMBER`) has a role of sending intended waypoints.
- It is robust to achieve diverse waypoints sent by ROS topic from rosparam passed via .yaml file.
- It also pops up Rviz node automatically and users can easily monitor the task performance.

2. Method

1. Assumption
    - The *turtlebot3_waffle_pi* model has been modified to use **encoder** for odometry data. This is included in **custom_turtlebot3_description** package.
    - The user starts robots with prior knowledge of initial positions and orientations of three robots with respect to the world frame. This is included in **coordination_environment.launch**. For example, the followings are first robot's argument for that pose.
    - The user can modify this on the terminal while launching the nodes; however, it is advisable to modify in the launch file as there are total 3*3 = 9 arguments (except for z) to test spawn.

    ```
    <arg name="first_tb3_x_pos" default="-3.0"/>
    <arg name="first_tb3_y_pos" default="0.0"/>
    <arg name="first_tb3_z_pos" default=" 0.0"/>
    <arg name="first_tb3_yaw"   default=" 1.57"/>
    ```

2. Start
    - Based on the above initial position arguments, (1) they are used for spawning robots in Gazebo for those poses, (2) they are passed to **coordination_robot** node (`robot_load.launch`) to make connection of TF hierarchy.
    - This is aligned with an real environment example where an user put three robots in indoor, e. g., in Sudikoff, set the reference frame, measure their poses, and pass them as arguments to do the real tests.
    - Specifically, in **coordination_robot** node, the robot saves the initial pose from Rosparam and publishes `/initialpose` topic with message type `geometry_msgs/PoseWithCovarianceStamped` after transforming Euler to quaternion.
    - Then, **tf_broadcastor** node receives `/initialpose` topic and broadcast TF whenever (every call back) the message is received.
    - The connection of TF hierarchy is given as follows.



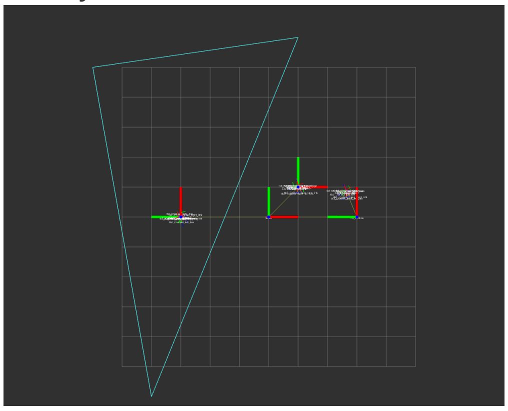
3. Task-related

    1. Instantiation

    - A leader and followers are initiated by respective **coordination_robot** nodes. Depending on arguments (leader flag), it will instantiate a relevant class despite the same reusable node code
    - Another good implementation design is to use inheritances by super class (*coordination_robot_super.py*). All the common features are included in super class, while specific features from either leader (*coordination_robot_leader.py*) or follower (*coordination_robot_follower.py*) are implemented in the descendant class.

    2. waypoint inputs

    - Note that after `coordination_environment.launch`, you will see robots in Gazebo and Rviz in addition to robot load and TF connection on the terminal with ros logwarn (visualized in yellow).
    - Then, on another terminal, if you initiate `waypoint_send.launch`, it passes the waypoints saved in .yaml files. Arguments as `waypoint_task` can be selected from 1 to 6, but feel free to change any of them from the pre-made ones, if you want.
    - The waypoints are sent via `waypoints` topic with message type `PolygonStamped`, which made it easier to visualize in Rviz. The example is as follows from the task number 6 with destination points in Cyan.



4. Leader and Follower

    - In principle, I am using a finite state machine to check a certain state is met (e. g., a certain prior message is received so that a robot can head to the next step).

    1. Custom message for intention of coordination behavior

        - Once the leader receives `waypoints` topic (save in self._wp_msg), it sends the custom message via `wp_allocate_intention` topic to followers.
        - The custom message (`Waypoint_init.msg`) includes (1) leader robot name (2) some string information (3) waypoints coordinated represented in a format `geometry_msgs/Polygon`, which again makes it easier to extract waypoint coordinates.

    2. Registration Service

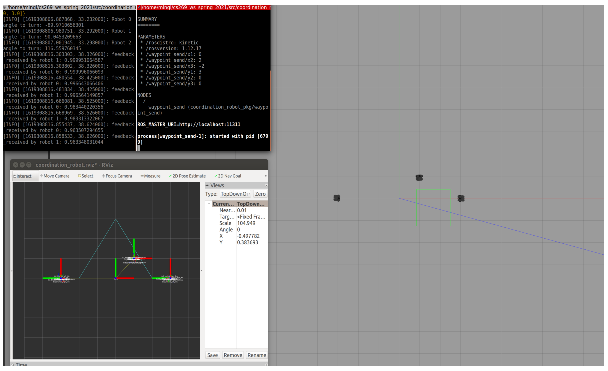
        - Once the follower robot receives `wp_allocate_intention` topic, the follower (**client**) robot sends a request via Service proxy designated on that specific robot.
        - The custom service `ServiceRegistration.srv` includes (1) follower robot name (2) global position transformed by TF listener implemented in Superclass (see self._look_up_transform, self.transform_local_psn_wrt_global).

    - The leader (**server**) receives the request, registers that specific follower robot, then, gives a response back with Bool.
    - The follower noticed it is registered and the result is logged with rospy.logwarn for the user to see.
3. Allocation of waypoints
    - After the leader robot checks the condition (line 112), the leader allocates waypoints to robots (including the leader) based on the greedy approach (see self.waypoint_allocate).
    - For each waypoint, Euclidean distance is calculated and the waypoint is assigned to a robot. Then, that robot is deleted from the candidate lists to find another robot, which will be assigned to the next waypoint.
    - Using hash map data structure, I was able to efficiently handle this allocation (key: robot name, value: to-go waypoint)
4. Action Goal
    - The custom action (`Coordination_Destination.action`) takes care of making robots reach waypoints. It includes (1) Goal position as a list of x,y coordinates (2) feedback as distance to go (Euclidean distance) (3) result as boolean to indicate the action goal is completed.
    - The leader has a role of **client** connected to a respective robot (**server**). Note that the leader itself is a **server**, too. For this reason, the server is implemented in Superclass (line 134-178).
    - Once the **server** receives the action goal, the robot (1) matches its heading towards its allocated destination (line 167 in Superclass) and (2) translate to the destination (line 171 in Superclass). During the action, the robot gives a feedback callback in terms of the remaining distance. It is printed out as rospy.loginfo.
    - Finally, when the robot reached the waypoint, the **client** receives the result and the result is also logged as logwarn.
5. Flush out
    - There were some updates of class properties during the above steps. The main purpose is to block unnecessary repetition of previous steps. However, in order to perform the next waypoint task, they should be initialized.
    - Once the leader found out that all the robots reached the allocated waypoints, the leader publishes `flushout` topic for a certain period (5 sec in my case). Then, (1) the leader itself initializes the necessary properties, (2) the followers initializes, (3) waypoint_send node shuts down.
    - Note that I found out the server and client of 'service' and 'action' can be maintained, as they work as connection bridge.
    - After the robots flushed out, if the user execute `waypoint_send.launch` with another task number, the robots resume the steps above: allocated to the new waypoints, go to the destination again.

3. Evaluation

Please see the additional material video.

1. Task performance
    - My code performs well with being well organized and commented.
    - It is robust in achieving diverse waypoints with initialized positions.
    - The code fully achieves the PA-3 criteria such as custom topic, action, service, TF broadcast, etc.
    - You can see the visualization together with rviz running automatically.

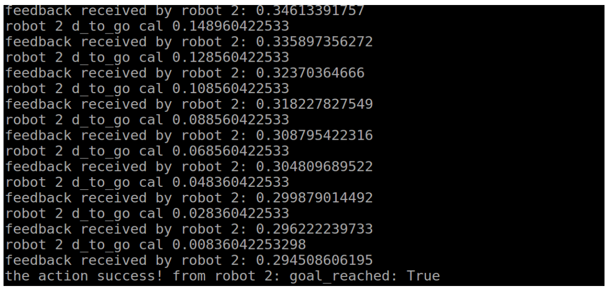


```
/home/mingi/cs269_ws_spring_2021/src/coordination_r
0, 3.0]}
[INFO] [1619308806.867868, 33.232000]: Robot 0
angle to turn: -89.9710656301
[INFO] [1619308806.989751, 33.292000]: Robot 1
angle to turn: 90.0453209663
[INFO] [1619308807.001945, 33.298000]: Robot 2
angle to turn: 116.559760345
[INFO] [1619308816.303303, 38.326000]: feedback
 received by robot 1: 0.999951064587
[INFO] [1619308816.303802, 38.326000]: feedback
 received by robot 0: 0.999996066093
[INFO] [1619308816.480554, 38.425000]: feedback
 received by robot 0: 0.996643066406
[INFO] [1619308816.481834, 38.425000]: feedback
 received by robot 1: 0.996564149857
[INFO] [1619308816.666081, 38.525000]: feedback
 received by robot 0: 0.983440220356
[INFO] [1619308816.668969, 38.526000]: feedback
 received by robot 1: 0.983313322067
[INFO] [1619308816.855437, 38.624000]: feedback
 received by robot 0: 0.963507294655
[INFO] [1619308816.858533, 38.626000]: feedback
 received by robot 1: 0.963348031044
```

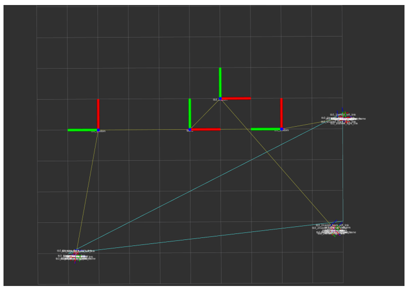
```
/home/mingi/cs269_ws_spring_2021/src/coordination_r

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17
 * /waypoint_send/x1: 0
 * /waypoint_send/x2: 2
 * /waypoint_send/x3: -2
 * /waypoint_send/y1: 3
 * /waypoint_send/y2: 0
 * /waypoint_send/y3: 0

NODES
  /
    waypoint_send (coordination_robot_pkg/waypo
int_send)

ROS_MASTER_URI=http://localhost:11311

process[waypoint_send-1]: started with pid [679
9]
```

2. Discussion

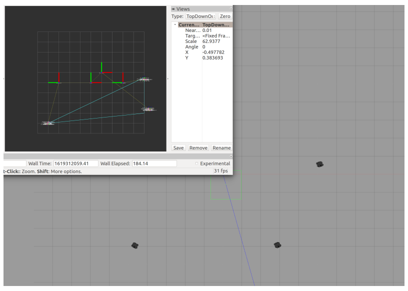
- **Error of position**: I have observed errors as offsets from the intended destinations. There are several reasons as PA-1, but here are the main ones:
  - Dynamic and physics model of the robot: control of the robot is not always sharp as our intention due to friction, inertia, etc.
  - Odom based on encoder also brings about more errors. Odom is weak to drifting and odom data themselves include noise. Therefore, even when the robot is sharply at a starting position, I was able to observe the position slightly increasing as time elapsed.
  - This indicates that, in general, the more the robots move, the more errors arise.
  - One example is shown as the following two figures. *feedback* on the terminal is based on Euclidean distance between the current transformed position with respect to the world and the goal position with respect to the world. On the other hand, *robot 2 d to go* is calculated based on time passed * linear velocity. As a result, the latter seems to have not much error (simulated as a wheel encoder) while the actual global positions have larger offsets.
  - Another interesting error is between Gazebo and Rviz. I assume that it is caused by some physics-related error such as friction, plus odometry. Specifically, the right two robots are aligned almost vertically in Rviz, while they are aligned diagonally in Gazebo.



```
feedback received by robot 2: 0.34613391757
robot 2 d_to_go cal 0.148960422533
feedback received by robot 2: 0.335897356272
robot 2 d_to_go cal 0.128560422533
feedback received by robot 2: 0.32370364666
robot 2 d_to_go cal 0.108560422533
feedback received by robot 2: 0.318227827549
robot 2 d_to_go cal 0.088560422533
feedback received by robot 2: 0.308795422316
robot 2 d_to_go cal 0.068560422533
feedback received by robot 2: 0.304809689522
robot 2 d_to_go cal 0.048360422533
feedback received by robot 2: 0.299879014492
robot 2 d_to_go cal 0.028360422533
feedback received by robot 2: 0.296222239733
robot 2 d_to_go cal 0.00836042253298
feedback received by robot 2: 0.294508606195
the action success! from robot 2: goal_reached: True
```



error example on terminal and Rviz



error example between Gazebo and Rviz

- **Action server connection** : I was about to re-instantiate action server connection during the flushout process. I realized that the original action server/client is still alive and it gave some error such as adding the same action server/client is not valid. Therefore, I used a finite state machine to block unnecessary procedures during the call back of the action, even if the action goal had been completed.