IITB Mars Rover Team
Freshie Induction 2024
Software Subsystem

Assignment - II : ROS Tutorial, Deadline: 27th October, 2024

In this assignment, you will develop a ROS 2-based system to simulate the operations of a Mars rover. You will explore essential ROS 2 concepts, including publisher-subscriber communication, custom message types, action servers, and service clients. You will also visualize node interactions using rqt graph and create launch files to facilitate the simultaneous start of all nodes. As you work on this project, remember to apply object-oriented programming (OOP) principles to enhance the design and maintainability of your code.

Exercise 1: Rover Status

Create a ROS 2 package `rover_status` to simulate and communicate the rover's status:

- Implement a publisher that sends `random` values for `battery_level` (0 to 100%) and `temperature` (-20 to 80°C) using the message type `std_msgs/Float32MultiArray`.
- Implement another publisher that sends a `string` field for `health_status` (e.g., "Healthy", "Critical", "Warning") based on the `battery_level`.
- Implement a subscriber to receive the data published by both nodes and print all the received data in a well-formatted manner.
- Use `rqt_graph` to visualize and inspect the interactions between nodes and topics in your system.
- Create launch files to launch all the nodes simultaneously, ensuring that the publishers and subscribers are properly connected.

Exercise 2: Rover Odometry

Create a ROS 2 package `rover_odometry` to simulate and communicate basic odometry data for a rover.

- Define a custom message type `mars_msgs/RoverOdometry` with the following fields:

  - `int32 rover_id` (unique identifier for each rover)
  - `float32 orientation` (the rover's orientation in radians)
  - `geometry_msgs/Twist linear_velocity` (the rover's linear speed in m/s)
  - `float32 angular_velocity` (the rover's angular speed in rad/s)
- Implement a publisher that simulates sending odometry data from the rover every second. Randomly generate values for `linear_velocity`, and `angular_velocity`. The orientation can be fixed or changed in small increments.
- Implement a subscriber that receives the odometry data and updates the current coordinates using `linear_velocity`, `angular_velocity` and `orientation`.
- Finally, print the updated odometry data (including position coordinates), also a warning if the `linear_velocity` exceeds a specified threshold (e.g., 3 m/s).
- Use `rqt_graph` to visualize the interactions between all nodes and topics in your ROS 2 system.

Exercise 3: Rover Navigation and Obstacle Avoidance

Create a ROS 2 package comprising `obstacle_avoidance` and `rover_navigation` nodes that simulate the rover's movement towards a target destination while incorporating obstacle avoidance.

- Develop an `obstacle_avoidance` node that implements argument parsing functionality to interpret a `m*n` grid matrix indicating occupancy. For example:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

  Here, a value of 1 indicates an occupied cell (obstacle present), while a value of 0 indicates an empty cell (no obstacle). This node will take the current coordinates of the rover from the `rover_navigation` node and publish the list of coordinates containing obstacles, treating the current position as the origin.
- The `obstacle_avoidance` node will publish this information to the topic `/obstacle_coordinates` with the message type `std_msgs/Float32MultiArray`.
- Create a `rover_navigation` node that utilizes argument parsing to accept a starting point and an end goal. This node will subscribe to the `/obstacle_coordinates` topic to receive

obstacle information using the current position as a reference and efficiently navigate towards the goal.

- The `rover_navigation` node will publish its navigation status (e.g., current position, steps taken) to the topic `/navigation/status` using the message type `std_msgs/String`. This status will also be used by the `obstacle_avoidance` node to calculate the positions of obstacles using the current position as the origin.
- Print "Goal Reached!" upon successfully reaching the target with a message displaying "N Steps," indicating the total number of steps taken. If the path is not feasible, output "Path not possible."
- Ensure that the rover moves only one cell at a time and is capable of moving in all eight directions.
- Develop a launch file that facilitates input from the command line interface (CLI) for seamless operation of both nodes.
- Analyze and summarize the entire system using `rqt_graph` for visual representation of node communication and interactions.

Exercise 4: Rover Soil Collection System

Create a ROS 2 system where the rover autonomously collects soil samples from specified coordinates.

- Implement a **service** `collection_service` that allows users to send collection requests to the rover. The service should accept the following parameters:
  - `float64 target_x` - X-coordinate of the collection location.
  - `float64 target_y` - Y-coordinate of the collection location.
- Create a **client node** `collection_client` that:
  - Takes user input for a list of target coordinates (e.g., `[(x1, y1), (x2, y2), ...]`).
  - Iteratively sends each target coordinate to the `collection_service`.
  - Handles any errors in service communication, including timeouts and connection issues.
  - Prints the response from the service after each request, indicating whether the collection was successful or failed.
- Develop a **node** `rover_collection` that:
  - Listens for collection requests from the `collection_service`.
  - Navigates to the specified coordinates while avoiding obstacles. Use a simulated obstacle detection mechanism to adjust the rover's path as needed.
  - Implements a logging mechanism to record the success or failure of each collection attempt.

  - Publishes the collection status (e.g., "Collection Successful", "Collection Failed") to the topic `/soil_collection/status`.

- The rover should also handle return requests to the starting point if the collection fails due to obstacles or other issues.
- Create a launch file `soil_collection.launch.py` that starts all the nodes simultaneously.

ALL THE BEST!