M2SAAS (Smart Aerospace and Autonomous Systems)

Course: Mission Coordination

Chuanlin ZUO **20235232**

GitHub: https://github.com/shineleft/MC_Lab_SAAS

Part 1: Workout

Section 1: What is ROS in more details?

The Robot Operating System (ROS) is an open-source, flexible framework used in robotics research, development, and industry applications. ROS isn't an actual "operating system" in the traditional sense, but rather a framework that provides essential tools, libraries, and conventions to help develop complex and modular robot software.

It simplifies complex tasks like multi-sensor integration, simulation, and multi-node communication, allowing developers to focus on creating sophisticated, high-performing robotic systems.

Section 2: How is ROS organized?

The ROS filesystem is a hierarchy of directories and files that organize ROS code, configurations, and data. Key elements of the filesystem include: packages, metapackages, workspaces.

The ROS Computation Graph is a peer-to-peer network of processes (nodes) that communicate with each other through various mechanisms, enabling the exchange of data and coordination of tasks. Core components of the Computation Graph include: nodes, master, topics, messages, services, actions.

Section 3: What is the main difference between a ROS package and a ROS stack? Give a stack example and its application domain?

A package focuses on a single functionality or tool and is the smallest ROS component.

A stack is a collection of packages grouped together, often for a larger, application-specific purpose.

Example of a ROS Stack: The Navigation Stack

The ROS Navigation Stack is a widely used stack that contains a set of packages designed to facilitate autonomous navigation for mobile robots. It provides a comprehensive suite of tools for robot localization, mapping, and path planning, making it ideal for creating autonomous, mobile robotic applications. The stack includes several core packages and libraries, each

focused on a specific aspect of navigation.

Section 4: ROS basics definitions

Master: The ROS Master is a central coordinating entity that manages and tracks the network of nodes. It functions as a name service, allowing nodes to locate each other, facilitating communication without the nodes needing direct connections initially. The Master keeps a registry of topics, services, and parameters, helping nodes to find publishers, subscribers, and service servers.

Node: A Node is the most basic process in ROS and represents an individual computation unit within the ROS network. Each node is designed to perform a specific task, such as reading data from a sensor, processing data, or controlling an actuator. Nodes are modular, meaning each node can be developed, deployed, or replaced independently.

Topic: A Topic is a named bus that allows nodes to communicate by publishing and subscribing to data streams. Topics are used in a publish-subscribe model, where nodes can either publish data to a topic or subscribe to it to receive data. Topics are ideal for one-way, continuous data streams like sensor readings (e.g., camera data, LiDAR scans, etc.) and control commands. Each topic has a unique name and a specific message type that defines the data structure for all messages published to that topic.

Publisher and Subscriber: A Publisher is a node or process that publishes data to a specific topic. It creates and sends messages that other nodes can receive by subscribing to the same topic. Publishers and topics are flexible, meaning multiple publishers can send data to the same topic, and multiple topics can be created by the same publisher.

A Subscriber is a node or process that listens to (or subscribes to) a specific topic to receive messages published to that topic. Subscribers are notified and receive data whenever a message is published to the topic they are subscribed to.

Message: A Message is a data structure used by ROS nodes to communicate via topics, services, and actions. Each message type defines a set of fields with specific data types (e.g., integers, floats, strings) and can also contain arrays and nested message types.

Server and Client: The Service Server is a node that advertises and provides a specific service. It waits for requests, processes them, and returns a response to the client.

The Service Client is a node that calls the service. It sends a request to the service server and waits for the response.

Services: Services provide a request-response communication model that differs from the continuous publish-subscribe model of topics. Services allow one node (client) to send a request to another node (server) and receive a response. This model is suitable for actions that are completed in a single call, such as requesting a sensor reset or asking the robot's current position. Each service has a unique name, and a Service Definition specifies the structure of the request and response messages.

Action: An Action in ROS is designed for tasks that take longer to complete or require feedback during execution, such as moving a robot arm to a target position or navigating to a goal point. Actions are an extension of the service concept, allowing for goal-feedback-result interactions, which is particularly useful for tasks that may need to be canceled or monitored over time. Each action has three parts:

- Goal: Specifies what the action server should accomplish (e.g., a target position).
- Feedback: Provides ongoing updates on progress (e.g., current position).

- Result: Returns the final outcome when the task is complete or canceled.

The Action Server provides the action, and an Action Client initiates it and can receive feedback or cancel the goal if necessary.

Parameter: The Parameter Server in ROS is a shared, centralized storage that holds various configuration parameters that nodes can use. Parameters are useful for storing data that may need to be accessed by multiple nodes, such as sensor calibration settings, robot dimensions, or PID control gains. Parameters are usually set at startup and can be adjusted dynamically, allowing for flexibility without requiring code changes.

Section 5: What are these Basic execution commands used for? Give the syntax of each.

roscore: The roscore command initializes the ROS system by starting the ROS Master, Parameter Server, and other core components needed for communication between nodes. It is the foundational command that must be executed before running any other ROS command.

Syntax: roscore

rosrun: The rosrun command allows you to run a specific node within a package. It's commonly used to start nodes quickly without needing a launch file.

Syntax: rosrun <package_name> <node_name>

roslaunch: The roslaunch command is used to start multiple nodes and configure parameters at once by executing a launch file. Launch files are written in XML or YAML format and provide a structured way to manage the startup configuration of complex applications with multiple nodes, parameters, and settings.

Syntax: roslaunch <package_name> <launch_file.launch>

rosnode: The rosnode command provides various subcommands to manage and interact with nodes, allowing you to list, display information about, kill, and ping nodes within the ROS system.

Syntax and Subcommands:

rosnode list # Lists all active nodes
rosnode info <node_name> # Displays information about a specific node
rosnode kill <node_name> # Terminates a specific node
rosnode ping <node_name> # Checks connectivity with a node

rostopic: The rostopic command is used to interact with and inspect topics. It allows you to list topics, publish messages, echo (print) message data, display information, and monitor topic performance.

Syntax and Subcommands:

rostopic list # Lists all active topics
rostopic echo <topic_name> # Prints data being published on a topic
rostopic info <topic_name> # Shows information about a topic (e.g., publishers, subscribers)
rostopic pub <topic_name> <message_type> <args> # Publishes a message to a topic
rostopic hz <topic_name> # Monitors the rate at which messages are published on a topic

Section 6: Give two ROS tools and explain what they are used for.

RViz (ROS Visualization): RViz is a 3D visualization tool in ROS that allows users to view and interact with sensor data, robot models, maps, and trajectories. RViz provides an intuitive, graphical interface where users can visualize real-time data, making it an essential tool for robot development, debugging, and simulation.

rqt (ROS Qt-Based GUI Tool): rqt is a GUI tool in ROS based on the Qt framework that

provides an array of plugins for monitoring, controlling, and visualizing various aspects of the ROS system. rqt is modular, meaning it can load different plugins as needed, making it versatile for multiple tasks.

Section 7: Applications Subject:

We have a mobile base robot equipped with two sensors: one lidar and one camera.

1- Define the master(s), and node(s)

Master: In ROS, there is typically a single Master that coordinates all nodes within the ROS network. The Master provides name services, helping nodes find each other and facilitate communication between them. The Master runs on a machine or device within the robot or the network, and the nodes (LiDAR, Camera, etc.) register with it.

Nodes:

LiDAR Node: A node that communicates with the LiDAR sensor, processes data from it, and publishes it to relevant topics.

Camera Node: A node that communicates with the camera sensor, captures images, and processes them to extract visual information.

Obstacle Detection Node: A node that processes data from the LiDAR and camera nodes to detect obstacles and compute the position of obstacles in the environment.

2- What could be the parameters of each node?

LiDAR Node Parameters:

- **scan_frequency:** The frequency at which the LiDAR sensor sends data**.**
- **max_range:** Maximum detectable range of the LiDAR.
- **min_range:** Minimum detectable range of the LiDAR.
- **frame_id:** The reference frame for the LiDAR data.

Camera Node Parameters:

- **image_resolution:** The resolution of the camera images.
- **frame_rate:** The frame rate for capturing images.
- **camera_intrinsics:** Parameters like focal length, sensor size, and other internal properties used for computer vision processing.
- **frame_id:** The reference frame for the camera data.

Obstacle Detection Node Parameters:

- **threshold_distance:** The distance at which an obstacle is considered to be too close.
- **processing_rate:** How often the obstacle detection node processes incoming data.
- **use_camera_data:** A boolean flag indicating whether the camera data should be used for obstacle detection**.**

3- Define three topics in this example

/lidar/scan:

- Publisher: LiDAR Node.
- Description: This topic publishes data from the LiDAR sensor, such as the distance measurements from the sensor at various angles. The message type could be sensor_msgs/LaserScan.

/camera/image_raw:

- Publisher: Camera Node.
- Description: This topic publishes raw image data captured by the camera. The message type could be sensor_msgs/Image.

/obstacle_position:

- Publisher: Obstacle Detection Node.
- Description: This topic publishes the detected position of any obstacles, including distance and angle. The message type could be geometry_msgs/Pose or geometry_msgs/Point.

4- We would like to get the obstacle's position which is at 2 meters. Give the appropriate communication flow.

To get the obstacle's position at a distance of 2 meters, the communication flow would involve the following steps:

1. LiDAR Node publishes data on the /lidar/scan topic. The data includes range and angle information that represents the distances to obstacles at various angles around the robot.
2. Obstacle Detection Node subscribes to the /lidar/scan topic and processes the LiDAR data. The node analyzes the LiDAR point cloud or range data to detect obstacles. It checks for objects within a specific distance based on the threshold_distance parameter.
3. If an obstacle is detected at the 2-meter range, the Obstacle Detection Node computes the position of the obstacle and publishes the obstacle's location on the /obstacle_position topic. The message could be a geometry_msgs/Point message, which contains the X, Y, and Z coordinates of the obstacle relative to the robot's frame.
4. Any other nodes that are interested in the obstacle position can subscribe to the /obstacle_position topic and take appropriate action, such as planning a new path to avoid the obstacle.

Lab 1

Q1: In this new terminal (the last one), you should see some data. What do this text represent ?

```
process[agent_1-1]: started with pid [12242]
Running ROS..
Robot : robot_1 is starting..
robot_1 distance to flag =  30.0
robot_1 distance to flag =  59.99999237060547
robot_1 distance to flag =  59.107234954833984
```

This text represent the distance between robot_1 and flag.

Q2: What is list command used for and what is the result ?

The rostopic list command is used to display all currently active topics in a ROS environment. Topics are the channels over which nodes communicate, so listing topics helps users see all data channels currently in use. This command is particularly useful for understanding the communication network, verifying if a topic exists, or checking if a specific node is publishing data correctly.

```
user:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
```

Q3: According to you, who are the publisher and the subscriber?

```
user:~$ rostopic info /robot_1/odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://3_xterm:44657/)

Subscribers:
 * /agent_1 (http://3_xterm:46069/)
```

The publisher is /gazebo and the subscriber is /agent_1.

Q4: According to you, what type of messages are published on this topic?

The odometry message are published on this topic

Q5: What is echo command used for and what is the result?

The rostopic echo command is used to display real-time messages published to a specific topic. It allows users to view the exact data being sent on a topic, which is especially helpful for debugging, monitoring sensor output, or verifying that a node is correctly publishing data.

```
user:~$ rostopic echo /robot_1/odom
header:
  seq: 2521
  stamp:
    secs: 507
    nsecs: 204000000
  frame_id: "odom"
child_frame_id: "base_footprint"
pose:
```

STEP 3: MOVE ONE ROBOT

```
127     # Strategy
128     velocity = 0.5
129     angle = 0
130     distance = float(robot.getDistanceToFlag())
131     print(f"{robot_name} distance to flag = ", distance)
```

STEP 4: MOVE ONE ROBOT TO THE CORRESPONDING FLAG

Q6 and Q7: to modify your program to move one robot safely to its corresponding flag and stop it at this position. To adapt it to real life. For this purpose, you can implement a PID controller. It means that, when the robot is far from its goal, it moves with the highest velocity values and as it gets closer, it slows down to a stop.

```
126     # Constants for the PID controller
127     KP = 0.4  # Proportional gain
128     KI = 0  # Integral gain
129     KD = 0.05  # Derivative gain
130     # PID state variables
131     previous_error = 0.0  # The error at the last time step
132     integral = 0.0  # Accumulated error over time
133
134     while not rospy.is_shutdown():
135         # Strategy
136         # velocity = 0.5
137         angle = 0
138         distance = float(robot.getDistanceToFlag())
139         print(f"{robot_name} distance to flag is ", distance)
140
141         # Write here your strategy..
142         # PID Control calculations
143         error = distance  # The goal is to reduce distance to 0
144         integral += error  # Accumulate the error over time
145         derivative = error - previous_error  # Rate of change of
146         previous_error = error  # Update the last error
147
148         # Calculate the control signal (velocity)
149         velocity = KP * error + KI * integral + KD * derivative
```

Q8: We will implement one of the simplest strategy: timing strategy. The timing strategy consists on starting each robot at different time. In this way, we will avoid collision. Implement it at line 124.

```
124     # Timing
125     if robot_name == "robot_1":
126         delay_time = 0.0  # The first robot starts immediately
127     elif robot_name == "robot_2":
128         delay_time = 50.0  # The second robot starts after a 50-second delay
129     elif robot_name == "robot_3":
130         delay_time = 100.0  # The third robot starts after a 100-second delay
131     else:
132         delay_time = 1.0  # Default delay is 1 second
133 
134     rospy.sleep(delay_time)  # Delay before starting
135     print(f"{robot_name} is starting after {delay_time} seconds delay.")
```

Q9: Write a launch file for this strategy.

```
<launch>
  <arg name="nbr_robot" default="3"/>

  <!--MAIN CODE-->
  <node pkg="evry_project_strategy" type="agent_with_timing.py" name="agent_1" output="screen" if="$(eval arg('nbr_robot') > 0)">
    <param name="robot_name" value="robot_1"/>
  </node>

  <node pkg="evry_project_strategy" type="agent_with_timing.py" name="agent_2" output="screen" if="$(eval arg('nbr_robot') > 1)">
    <param name="robot_name" value="robot_2"/>
  </node>

  <node pkg="evry_project_strategy" type="agent_with_timing.py" name="agent_3" output="screen" if="$(eval arg('nbr_robot') > 2)">
    <param name="robot_name" value="robot_3"/>
  </node>
```

Lab2

1.Now that you can write a simple and non-robust strategy, you can build on this knowledge to implement your robust strategy.

Firstly, I improved the *getDistanceToflag* function, which is used to obtain the relative distance dx, dy between the robot and the flag.

```
99      def getDistanceToFlag(self):
100         """Get the distance separating the agent from a flag. The service 'distanceToFlag' is called for this p
101         The current position of the robot and its id should be specified. The id of the robot corresponds to th
102 
103 
104         Returns:
105             float: the distance separating the robot from the flag
106         """
107         rospy.wait_for_service('/distanceToFlag')
108         try:
109             service = rospy.ServiceProxy('/distanceToFlag', DistanceToFlag)
110             pose = Pose2D()
111             pose.x = self.x
112             pose.y = self.y
113             # int(robot_name[-1]) corresponds to the id of the robot. It is also the id of the related flag
114             result = service(pose, int(self.robot_name[-1]))
115             return result.distance
116         except rospy.ServiceException as e:
117             print("Service call failed: %s" % e)
```

Then, two strategies are proposed.

Strategy 1: reactive obstacle avoidance mechanism

The robot moves toward the flag using PID controllers to adjust its speed and direction.

```
        distance = robot.getDistanceToFlag()
        dx = distance[0]
        dy = distance[1]
        # print(f"{robot_name} distance_x to flag = {dx}")
        # print(f"{robot_name} distance_y to flag = {dy}")

        # calculate distance and target angle
        distance_to_flag = math.sqrt(dx**2 + dy**2)
        target_angle = math.atan2(dy, dx)

        # get robot's pose
        robot_pose = robot.get_robot_pose()
        current_angle = robot_pose[2]

        # calculate angle error
        angle_error = target_angle - current_angle
        angle_error = (angle_error + math.pi) % (2 * math.pi) - math.pi

        # calculate velocity and angle
        velocity = speed_pid.compute(distance_to_flag)
        angle = angle_pid.compute(angle_error)
```

At the same time using a sonar sensor to measure the distance to the front obstacle. If the sonar reading is less than *safe_distance*: The robot enters a reactive avoidance mode.
The robot slows down to *avoid_speed* and turns by *avoid_angle*. After turning, the robot skips the rest of the logic for that iteration (continue) and rechecks for obstacles in the next loop.

```
    # Obstacle avoidance parameters
    safe_distance = 2.5  # Minimum safe distance to avoid obstacles
    avoid_speed = 2.0    # Speed when avoiding obstacles
    avoid_angle = math.pi/3    # Angle to turn when avoiding obstacles


    while not rospy.is_shutdown():
        # Get the sonar reading (distance to the closest obstacle)
        sonar_distance = robot.get_sonar()

        if sonar_distance < safe_distance:
            # Obstacle detected: reactive avoidance
            print(f"{robot_name} detected obstacle at distance = {sonar_distance}. Avoiding...")

            # Reactive avoidance: turn and reduce speed
            velocity = avoid_speed
            angle = avoid_angle  # Positive angle turns left; adjust as needed

            robot.set_speed_angle(velocity, angle)
            rospy.sleep(0.8)
            continue  # Skip further processing and continue avoidance
```

Strategy 2: Artificial Potential Field

In this part, I implements a robot navigation system based on the Artificial Potential Field method combined with PID control. The robot is guided toward a target (flag) using attractive forces while avoiding obstacles using repulsive forces.
The robot computes the attractive force pulling it toward the flag:

$$\text{Fx} = \text{gian} * \text{dx} \qquad \text{Fy} = \text{gian} * \text{dy}$$

$$F_{obstacle} = gian * \left(\frac{1}{sonar_{distance}} - \frac{1}{repulsive_{threshold}}\right) * \left(\frac{1}{sonar_{distance}}\right)^2$$

If the obstacle is beyond the threshold, the repulsive forces are zero.
This combined force determines the direction and strength of the robot's motion.

```
        # Distance and direction to the flag
        distance = robot.getDistanceToFlag()
        dx, dy = distance[0], distance[1]
        distance_to_flag = math.sqrt(dx**2 + dy**2)
        target_angle = math.atan2(dy, dx)

        # Current robot pose
        robot_pose = robot.get_robot_pose()
        current_angle = robot_pose[2]

        # **Attractive Force (to Flag)**
        attractive_force_x = attractive_gain * dx
        attractive_force_y = attractive_gain * dy


        # **Repulsive Force (from Obstacles)**
        sonar_distance = robot.get_sonar()
        if sonar_distance < repulsive_threshold and sonar_distance > 0.01:
            # Compute repulsive forces only within the threshold
            repulsive_force_x = repulsive_gain * (1 / sonar_distance - 1 / repulsive_threshold) * (1 / sonar_distance**2)
            repulsive_force_y = 0  # Assume only frontal obstacles (adjust as needed)
        else:
            repulsive_force_x = 0
            repulsive_force_y = 0

```

```python
        # **Combine Forces**
        total_force_x = attractive_force_x - repulsive_force_x
        total_force_y = attractive_force_y - repulsive_force_y

        # Calculate desired angle and distance
        desired_angle = math.atan2(total_force_y, total_force_x)
        desired_distance = math.sqrt(total_force_x**2 + total_force_y**2)

        # **PID Control**
        angle_error = desired_angle - current_angle
        angle_error = (angle_error + math.pi) % (2 * math.pi) - math.pi  # Normalize angle to [-π, π]

        velocity = speed_pid.compute(desired_distance)  # Velocity controlled by target distance
        angle = angle_pid.compute(angle_error)  # Angle controlled by angle error

        print(f"{robot_name} distance_x to flag = {dx}")
        print(f"{robot_name} distance_y to flag = {dy}")
        print(f"{robot_name} distance to flag = {distance_to_flag}")
        print(f"{robot_name} sonar distance = {sonar_distance}")
        print(f"Calculated total force_x = {total_force_x}, total force_y = {total_force_y}")
        print(f"Desired angle = {math.degrees(desired_angle)}, Velocity = {velocity}, Angle = {angle}")
```

2. Bonus: You can add some obstacles and move the flags to test the robustness of your strategy.

Modify flag position and add obstacles

```xml
    <!-- ********************FLAGS****
    <!-- 3 flags positionned at 30m (d
    <include>
      <uri>model://red_flag</uri>
      <name>flag_1</name>
      <pose>-20 20 0 0 0 0</pose>
    </include>

    <include>
      <uri>model://green_flag</uri>
      <name>flag_2</name>
      <pose>10 20 0 0 0 0</pose>
    </include>

    <include>
      <uri>model://blue_flag</uri>
      <name>flag_3</name>
      <pose>-10 -20 0 0 0 0</pose>
    </include>
```

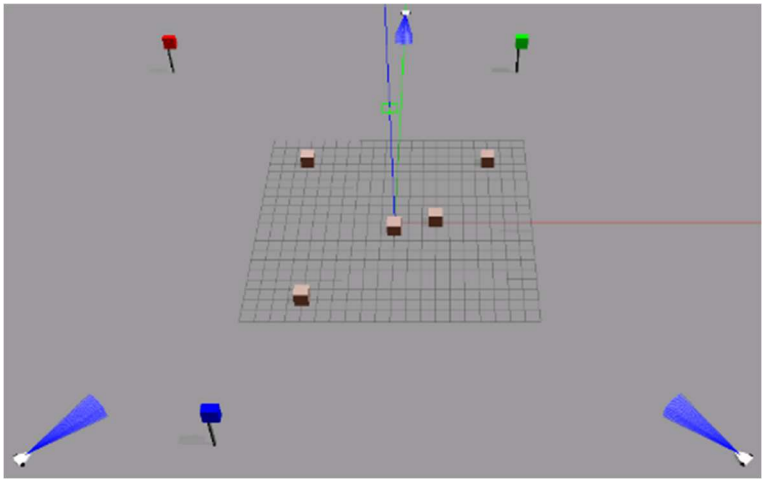
```xml
    <!-- ********************BLOCK**
    <include>
      <uri>model://basic_box</uri>
      <name>box_1</name>
      <pose>3 0 0 0 0 0</pose>
    </include>

    <include>
      <uri>model://basic_box</uri>
      <name>box_2</name>
      <pose>-3 -1 0 0 0 0</pose>
    </include>

    <include>
      <uri>model://basic_box</uri>
      <name>box_3</name>
      <pose>7 7 0 0 0 0</pose>
    </include>

    <include>
```

New map like this:



Conclusion

Strategy 1: reactive obstacle avoidance mechanism

When encountering an obstacle, the obstacle avoidance behavior is activated first; when there

is no obstacle, the target tracking behavior is activated. Obstacle avoidance and target tracking are decoupled, and the behaviors will not interfere with each other, avoiding method conflicts. In complex scenarios, the hierarchical mechanism can ensure the priority of obstacle avoidance and improve the ability to respond to unexpected situations. Based on specific trigger conditions, different behaviors are clearly switched and can quickly adapt to the environment.

However, if the distribution of obstacles is dense, the switching between behaviors may be frequent, resulting in path jitter or stagnation.

Strategy 2: Artificial Potential Field

Artificial potential field method uses attractive and repulsive forces to control the robot movement at the same time. The combined force determines the direction and speed of the robot. The target and obstacle avoidance do not need to be processed hierarchically, but are uniformly modeled through the superposition of forces, and the control flow is more natural and real-time.

However, in some cases, the robot may be trapped in a local minimum and cannot reach the target. When obstacles are dense or close to the target point, the repulsive force may become too large, causing the robot path to jitter.