M2SAAS (Smart Aerospace and Autonomous Systems)

Course: Mission Coordination

Group number

Group member 1: Chahinaz HAMIDECHE 20235715
Group member 2: Nabila TIHAL 20245855

Github link: https://github.com/TIHAL-Nabila/Go_to_flag

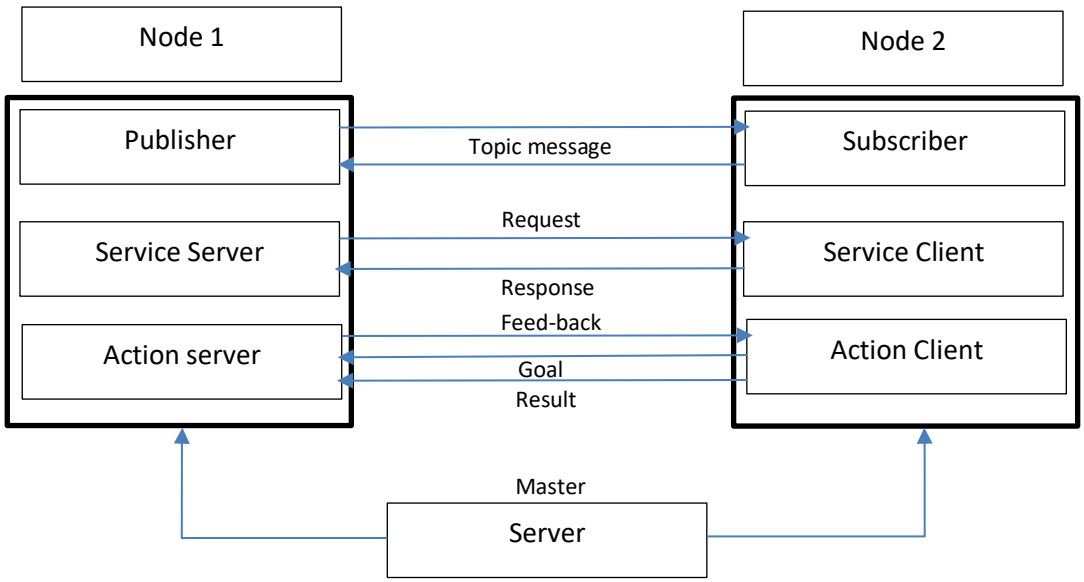
Workout

Section 1: What is ROS in more details?

ROS is sometimes called a meta operating system because it performs many functions of an operating system, but it requires a computer's operating system such as Linux. One of its main purposes is to provide communication between the user, the computer's operating system, and equipment external to the computer. This equipment can include sensors, cameras, as well as robots. As with any operating system, the benefit of ROS is the hardware abstraction and its ability to control a robot without the user having to know all of the details of the robot. For example, to move a robot's arms, a ROS command is issued or scripts in Python or C++ written by the robot designers cause the robot to respond as commanded. The scripts can, in turn, call various control programs that cause the actual motion of the robot's arms. It is also possible to design and simulate your own robot using ROS.

Section 2: How is ROS organized?

Here is a scheme that explains how ROS is organized:



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

the main difference between ROS package and a ROS stack is that a **ROS Package** is the basic building block of ROS software. It contains nodes, libraries, configuration files, message definitions, and other resources needed for a specific functionality or feature. while, **A ROS stack** is a collection of related ROS packages grouped together for a broader functionality.

Example Stack: Navigation Stack

Application Domain:

The Navigation Stack is widely used in mobile robotics, particularly in autonomous vehicles, drones, and service robots. It helps the robot navigate complex environments by allowing it to plan and follow paths, avoid obstacles, and localize itself in real-time.

Section 4: ROS basics definitions

Master, Node, Topic, Publisher and Subscriber, Message, Server and Client, Services, Action, Parameter.

1.**Master**: The ROS Master facilitates communication by managing node registration, topics, and services, enabling nodes to locate and interact with each other.
2. **Node**: A node is an independent process in ROS that performs a specific task and communicates with other nodes.
3. **Topic**: A topic is a named communication channel that allows nodes to exchange messages asynchronously in a publish-subscribe model.
4. **Publisher**: A publisher is a node that sends messages to a specific topic for other nodes to receive.
5. **Subscriber**: A subscriber is a node that listens to a specific topic to receive messages published by other nodes.
6.**Message**: A message is a data structure used for communication between nodes via topics, containing specific information like sensor data or control commands.
7. **Server**: A server is a node that provides a service, processing requests and sending back responses.
8.**Client**: A client is a node that sends requests to a service server and waits for a response.
9. **Service**: A service is a synchronous communication mechanism in ROS, where a client sends a request and receives a response from a server.
10. **Action**: An action is an asynchronous communication mechanism in ROS used for long-running tasks, providing feedback and allowing cancellation.
11. **Parameter**: A parameter is a configurable value stored on the ROS Parameter Server, accessible by nodes for configuration or runtime adjustments.

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

1. Master: The ROS Master coordinates communication between nodes and maintains a registry of topics, services, and parameters.

   Syntax: roscore
2. Nodes: A node is an executable program that communicates with other nodes to perform a specific task.

   Syntax: rosrun <package_name> <node_name>
3. Topics: Topics are communication channels used for data exchange between nodes via a publish-subscribe mechanism.

   Syntax: rostopic list (List all topics)
4. Publisher and Subscriber: A publisher sends messages to a topic, and a subscriber listens to those messages.

Syntax: rostopic info <topic_name>

5. Messages: Messages are structured data sent between nodes through topics.

   Syntax: rosmsg show <message_type>

6. Server and client: A server provides a service, and a client sends requests to that server to perform specific tasks.

   Syntax: rosservice call <service_name> <arguments>

7. Services: Services provide synchronous communication between nodes, enabling request-response interactions.

   Syntax: rosservice info <service_name>

8. Action: Actions enable asynchronous communication for long-running tasks, providing feedback and the ability to cancel.

   Syntax: rostopic list

9. Parameter: Parameters are configurable values stored on the Parameter Server, accessible by nodes at runtime.

   Syntax: rosparam set <parameter_name> <value> (set a parameter)

   Or rosparam get <parameter_name> (get a parameter)

Section 6: Give two (02) ROS tools and explain what they are used for.

1.Rviz (ROS Visualization): is a graphical tool in ROS for real-time visualization of robot data and system status. It is used for:

- Visualizing robot models and tracking their position and orientation.
- Providing an interactive interface for tasks like setting goals or analyzing system performance.
- Displaying sensor outputs and debugging algorithms.

2. Gazebo: is a simulation tool for replicating real-world physics and testing robots virtually. It is used for :

- Simulating robot dynamics, movements, and sensor data in realistic environments.
- Developing and testing control, perception, and planning algorithms without physical hardware.
- Experimenting with robot designs and behaviors before deployment in real-world systems.

Section 7: Applications

Subject:

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

1- <u>the master is</u>: The ROS Master is the central software process running on the robot's onboard computer.

<u>and nodes:</u>

Lidar Node: Processes data from the lidar sensor.
Camera Node: Processes data from the camera sensor.
Localization Node: Combines sensor data to determine the robot's position and map the environment.
Obstacle Detection Node: Processes data to identify obstacles and their positions.

2- the parameters of each node could be:

Lidar node: Frequency of lidar scans / Maximum range / Angular resolution of the lidar.
Camera node: Camera resolution / Frame rateof the camera / Camera intrinsic parameters for processing.

3- three topics in this example are:

/lidar_scan:
- Published by the Lidar Node, providing a laser scan message with distance readings to nearby objects.

/camera_image:
- Published by the Camera Node, providing image data from the camera sensor.

/obstacle_position:
- Published by the Obstacle Detection Node, providing the detected obstacle's position in a structured message.

4- The appropriate communication flow to get the obstacle's position which is at 2 meters is:

1.**Lidar Node**:
- Publishes distance data to the /lidar_scan topic.

2.**Camera Node**:
- Publishes image data to the /camera_image topic.

3. **Obstacle Detection Node**:
- Subscribes to /lidar_scan and /camera_image.
- Processes the lidar and camera data to detect obstacles.
- Publishes the obstacle's position to /obstacle_position.

4.**Localization or Control Node**:
- Subscribes to /obstacle_position to access the detected obstacle's position for navigation or planning purposes.

1. LAB01:
   1.1. Environment set up:
   1.2. Get started:

Question1:

In a second terminal, we run the python script containing the existing strategy by executing:
$roslaunch evry_project_strategy agent . launch nbr_robot :=1

```
process[agent_1-1]: started with pid [5109]
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 =  59.06129837036133
robot_1 distance to flag =  58.2584114074707
robot_1 distance to flag =  57.05408477783203
robot_1 distance to flag =  56.251197814941406
robot_1 distance to flag =  55.04686737060547
```

The results show the distance between Robot number 1 and its corresponding flag.

Question2:

To list the current topics, we enter this instruction: $rostopic list

```
user:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/robot_1/cmd_vel
/robot_1/odom
/robot_1/sensor/sonar_front
/robot_2/cmd_vel
/robot_2/odom
/robot_2/sensor/sonar_front
/robot_3/cmd_vel
/robot_3/odom
/robot_3/sensor/sonar_front
/rosout
/rosout_agg
/tf
```

This is the list of all the topics for each actual nodes running in ROS

Question3:

We execute the folloxing commande : $rostopic info / robot_1 / odom

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

Publishers:
 * /gazebo (http://3_xterm:35955/)

Subscribers:
 * /agent_1 (http://3_xterm:43203/)
```

Question4:

The type of messages are published on this topicOdometry message is published

Question5:

```
header:
  seq: 776
  stamp:
    secs: 156
    nsecs: 572000000
  frame_id: "odom"
child_frame_id: "base_footprint"
pose:
  pose:
    position:
      x: -10.260577239784215
      y: 10.260758789885548
      z: 0.0
    orientation:
      x: 6.933552354168249e-05
      y: -2.872360509719466e-05
      z: 0.9238470996532588
      w: 0.3827617154703449
  covariance: [1e-05, 0.0, 0.0, 0.0, 0.0, 0.
000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000
twist:
  twist:
    linear:
      x: 1.536563172817716e-07
      y: 5.669454836521542e-07
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: -1.1861620850824004e-06
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
---
```

Question6: We modified our code to move one robot safely to its corresponding flag and

stop it at this position, where we put the velocity equals to distance so that the robot stops when the distance is equal to zero.

```
126     while not rospy.is_shutdown():
127         # Strategy
128         velocity = 2
129         angle = 0
130         distance = float(robot.getDistanceToFlag())
131         velocity =distance
132         print(f"{robot_name} distance to flag = ", distance)
```

Question 7: to adapt it to real life. For this purpose, we implemented a PID controller. It means that, when the robot is far from its goal, it moves with the highest velocity values and and as it gets closer, it slows down to a stop. The code is shown in the figure below

```
# Timing
Kp=1.2
Ki=0.05
Kd=0.6
d_des=0
err_old_01=0
angle=0
dt=0.05
V0=6
robot.set_speed_angle(V0, angle)
rospy.sleep(3*int(robot_name[-1]))

while not rospy.is_shutdown():
    # Strategy
    # Write here your strategy..


    distance = float(robot.getDistanceToFlag())
    print(f"{robot_name} distance to flag = ", distance)
    err1=distance-d_des
    vt=Kp*err1+Ki*(err_old_01+err1)+Kd*(err1-err_old_01)/dt
    robot.set_speed_angle(vt, angle)
    err_old_01=err1
```

Question 8: We implemented 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. The code is shown in the figure below.

```python
    # Timing
    rospy.sleep(3*int(robot_name[-1]))
    while not rospy.is_shutdown():
        # Strategy
        velocity = 2
        angle = 0

        # Write here your strategy..
        if (robot.robot_name== 'robot_1'):
                distance = float(robot.getDistanceToFlag())
                print(f"{robot_name} distance to flag = ", distance)
                velocity =distance

        if (robot.robot_name== 'robot_2'):
                distance = float(robot.getDistanceToFlag())
                print(f"{robot_name} distance to flag = ", distance)
                velocity =distance
        if (robot.robot_name== 'robot_3'):
                distance = float(robot.getDistanceToFlag())
                print(f"{robot_name} distance to flag = ", distance)
                velocity =distance
```

2. LAB02:
    1. Now that we are going to write a robust strategy where each robot is be able to reach its goal, obstacles are added as follow

```xml
<include>
  <uri>model://basic_box</uri>
  <name>box_1</name>
  <pose>0 -10 0 0 0 0</pose>
</include>

<include>
  <uri>model://basic_box</uri>
  <name>box_2</name>
  <pose>-11.21320344 11.21320344 0 0 0 0</pose>
</include>

<include>
  <uri>model://basic_box</uri>
  <name>box_3</name>
  <pose>11.21320344 11.21320344 0 0 0 0</pose>
</include>
```

Strategy Explanation :

Our approach involves guiding robots toward their respective flags using calculated heading angles while implementing an obstacle avoidance mechanism to ensure safe navigation.

1. **Flag-Directed Navigation**

- The robot calculates the **heading angle** toward its assigned flag using the coordinates of its current position and the flag's position.
- This ensures the robot maintains a direct path toward the target when no obstacles are present.

2. **Obstacle Avoidance**

- The robot continuously monitors the distance to obstacles using its ultrasonic sensor.
- When the distance to an obstacle falls below a predefined **threshold**, the robot initiates an avoidance maneuver by:

- Adjusting its heading with a predefined **avoidance angle** to steer away from the obstacle.
- Once clear of the obstacle, it recalculates its heading angle and resumes its path toward the flag.

3. **PID Controllers**

- A **PID controller** is used to fine-tune the robot's **linear velocity** and **angular velocity**:
    - The robot moves at maximum velocity when far from its target and gradually reduces speed as it approaches the flag, ensuring precise stopping.
    - The angular velocity is dynamically adjusted to align the robot with the desired heading angle, resulting in smoother navigation and accurate orientation.
    -

By integrating these elements, the strategy effectively combines goal-oriented navigation with robust obstacle avoidance, ensuring collision-free and efficient movement to the destination.

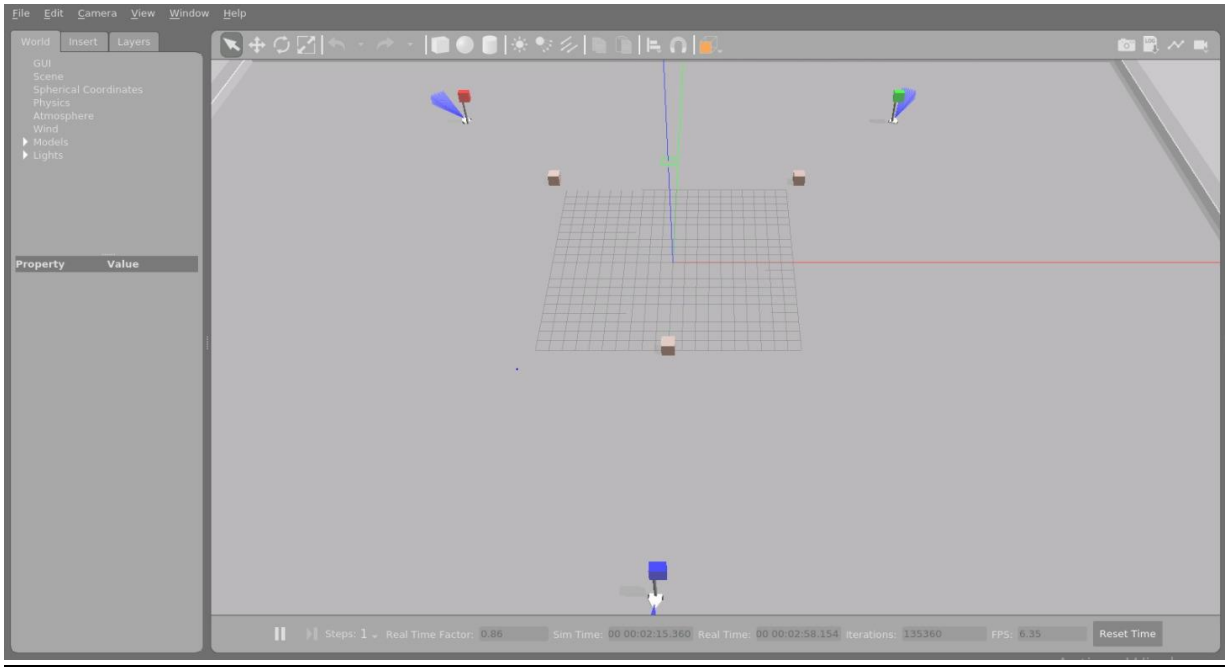
Our strategy is represented in the code bellow:

```
def run_demo():
    """Main loop for robust strategy with obstacle avoidance."""
    robot_name = rospy.get_param("~robot_name")
    robot = Robot(robot_name)
    rospy.loginfo(f"Robot {robot_name} is starting...")

    # PID Parameters for goal tracking
    Kp = 1.2
    Ki = 0.05
    Kd = 0.6
    d_des = 0.1  # Acceptable distance threshold to the flag
    dt = 0.05

    # Obstacle avoidance parameters
    obstacle_distance_threshold = 2.0  # Avoid objects within 1 meter
    avoidance_speed = 0.5
    avoidance_angle = math.pi / 2
    avoidance_turn_time = 1.0

    # Initial speed and timing strategy
    rospy.sleep(3 * int(robot_name[-1]))  # Staggered start based on robot ID

    # Get initial distance to the flag
    distance = float(robot.getDistanceToFlag())

    # Calculate flag position relative to the robot's initial pose
    #flag_x = robot.x + distance * math.cos(robot.yaw)
    #flag_y = robot.y + distance * math.sin(robot.yaw)
    flag = [robot.getDistanceToFlag().flag_x, robot.getDistanceToFlag().flag_y]
```

```
        # PID control for angular velocity
        angle = Kp * angle_error + Ki * (angle_error * dt) + Kd * (angle_error / dt)

        # PID control for linear velocity
        err1 = distance - d_des
        vt = Kp * err1 + Ki * (err1 * dt) + Kd * (err1 / dt)

        # Set speed and maintain direction
        robot.set_speed_angle(vt, angle)
        rospy.sleep(dt)


if __name__ == "__main__":
    print("Running ROS..")
    rospy.init_node("Controller", anonymous=True)
    run_demo()
```

```python
flag = [robot.getDistanceToFlag().flag_x, robot.getDistanceToFlag().flag_y]
flag_x=flag[0]
flag_y=flag[1]
while not rospy.is_shutdown():
    # Get distance to the flag and obstacle distance
    distance = float(robot.getDistanceToFlag())
    obstacle_distance = float(robot.get_sonar())  # Assuming get_sonar() provides distance to the closest obstacle

    rospy.loginfo(f"{robot_name} distance to flag: {distance}")
    rospy.loginfo(f"{robot_name} obstacle distance: {obstacle_distance}")

    # Obstacle avoidance logic
    if obstacle_distance < obstacle_distance_threshold:
        rospy.logwarn(f"{robot_name} is avoiding an obstacle!")
        robot.set_speed_angle(avoidance_speed, avoidance_angle)  # Turn to avoid obstacle
        rospy.sleep(avoidance_turn_time)
        continue  # Skip goal tracking for this iteration

    # Stop the robot if it reaches the flag
    if distance <= d_des:
        robot.set_speed_angle(0, 0)
        rospy.loginfo(f"{robot_name} has reached its flag.")
        break

    # Calculate angle to the flag
    flag_angle = math.atan2(flag_y - robot.y, flag_x - robot.x)
    angle_error = flag_angle - robot.yaw
    angle_error = math.atan2(math.sin(angle_error), math.cos(angle_error))  # Normalize to [-π, π]
```

Running the following code gave the results shown in the figure below :



Conclusion

This project successfully implemented a multi-robot navigation system using ROS and Gazebo. Robots navigated to their flags while avoiding obstacles by calculating heading angles and using a PID controller for precise motion. The strategy proved robust and effective, ensuring collision-free navigation.