Step 1: Initialize a ROS Workspace

If you haven't already, create and initialize a ROS workspace:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws
catkin init
```

Step 2: Create a ROS Package

Create a ROS package for your multi-robot path planning project:

```
cd ~/catkin_ws/src
catkin create pkg multi_robot_path_planning
```

Step 3: Organize Your Package Structure

Organize cd ~/catkin_ws/src/multi_robot_path_planning

mkdir launch scripts src

your package structure to include necessary folders and files:

```
cd ~/catkin_ws/src/multi_robot_path_planning
mkdir launch scripts src
```

Step 4: Write the ROS Nodes

Implement your multi-robot path planning algorithm in a ROS node. Create a new Python script in the `src` folder. For example, let's call it `multi_robot_path_planning_node.py`:

```
#!/usr/bin/env python

import rospy
```

```
def multi_robot_path_planning_main():
    rospy.init_node('multi_robot_path_planning_node', anonymous=True)

    # Implement your multi-robot path planning algorithm logic here

    rospy.spin()

if __name__ == '__main__':
    multi_robot_path_planning_main()
```

Step 5: Create a Launch File

Create a launch file (e.g., `multi_robot_path_planning.launch`) in the `launch` folder to launch your ROS nodes:

```
<launch>
    <node name="multi_robot_path_planning_node" pkg="multi_robot_path_planning"
type="multi_robot_path_planning_node.py" output="screen"/>
</launch>
```

Step 6: Modify `CMakeLists.txt` and `package.xml`

Update `CMakeLists.txt` in the package root directory to include the necessary dependencies:

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    # Add other dependencies as needed
)
```

Update `package.xml` to declare dependencies:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
# Add other dependencies as needed
```

Step 7: Build Your Package

Build your package:

```
cd ~/catkin_ws
catkin build multi_robot_path_planning
```

Step 8: Source the Workspace

Source the workspace to make the ROS system aware of your new package:

```
source ~/catkin_ws/devel/setup.bash
```

Step 9: Test Your Node

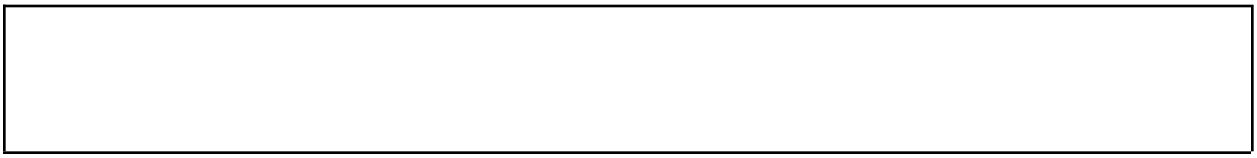
Run your ROS node to test if it works correctly:

```
roslaunch multi_robot_path_planning multi_robot_path_planning.launch
```

Step 10: Create a GitHub Repository

Follow the steps mentioned earlier to create a GitHub repository for your ROS package and push your code.

```
cd ~/catkin_ws/src/multi_robot_path_planning
git init
git add .
git commit -m "Initial commit"
# Add a remote repository on GitHub and push your code
```



```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import PoseStamped
from nav_msgs.msg import Path
from std_msgs.msg import String
from nav_msgs.msg import OccupancyGrid

class MultiRobotPathPlanning:
    def __init__(self):
        rospy.init_node('multi_robot_path_planning_node', anonymous=True)

        # Set up subscribers for receiving robot positions
        rospy.Subscriber('/robot1/pose', PoseStamped, self.robot1_callback)
        rospy.Subscriber('/robot2/pose', PoseStamped, self.robot2_callback)

        # Set up publishers for sending paths to robots
        self.path_pub_robot1 = rospy.Publisher('/robot1/path', Path, queue_size=10)
        self.path_pub_robot2 = rospy.Publisher('/robot2/path', Path, queue_size=10)

        # Add more subscribers/publishers for additional robots as needed

        # Initialize placeholders for robot positions
        self.robot1_pose = None
        self.robot2_pose = None

    def robot1_callback(self, msg):
        # Callback function for receiving robot1 position
        self.robot1_pose = msg.pose

    def robot2_callback(self, msg):
        # Callback function for receiving robot2 position
        self.robot2_pose = msg.pose

    def perform_path_planning(self):
        # Implement your multi-robot path planning algorithm logic here
        # Use self.robot1_pose and self.robot2_pose to access robot positions

        # Placeholder logic: if both robot positions are received, publish paths
        if self.robot1_pose is not None and self.robot2_pose is not None:
            path_robot1 = self.calculate_path(self.robot1_pose)
            path_robot2 = self.calculate_path(self.robot2_pose)

            # Publish paths to robots
```

```
            self.path_pub_robot1.publish(path_robot1)
            self.path_pub_robot2.publish(path_robot2)

    def calculate_path(self, robot_pose):
        # Placeholder path calculation logic
        # You should replace this with your actual path planning algorithm
        # For now, just create a straight path as an example
        path = Path()
        path.header.frame_id = 'map'
        path.poses.append(PoseStamped(header=path.header, pose=robot_pose))
        path.poses.append(PoseStamped(header=path.header, pose=robot_pose))
        return path

    def run(self):
        rate = rospy.Rate(1)  # 1 Hz
        while not rospy.is_shutdown():
            self.perform_path_planning()
            rate.sleep()

if __name__ == '__main__':
    multi_robot_path_planning = MultiRobotPathPlanning()
    multi_robot_path_planning.run()
```

- The `robot_callback` function is a placeholder for the callback that will handle the received positions of each robot. You should implement the logic to process and store the positions.
- Subscribers are set up for each robot to listen for their positions. Adjust the topic names (`'/robot1/pose'`, `'/robot2/pose'`, etc.) and message types (`PoseStamped`) based on your robot configuration.
- Publishers are set up for each robot to send paths. Adjust the topic names (`'/robot1/path'`, `'/robot2/path'`, etc.) and message types (`Path`) based on your robot configuration.
- The `multi_robot_path_planning_main` function is where you implement your multi-robot path planning algorithm. It should use the received robot positions, perform the path planning, and publish the calculated paths back to the robots.
- The `MultiRobotPathPlanning` class is introduced to encapsulate the functionality. The `__init__` method initializes the ROS node and sets up

subscribers and publishers. The `run` method is used to run the path planning loop.

- The `perform_path_planning` method is where you should implement your multi-robot path planning algorithm. It is called in the main loop.
- The `calculate_path` method is a placeholder for your actual path planning algorithm. It currently creates a simple straight path for demonstration purposes.

To integrate your multi-robot path planning project with ROS and Gazebo, you'll need to create Gazebo models for your robots and potentially the environment. Below is an example SDF (Simulation Description Format) file for a simple differential drive robot. This is a basic starting point, and you may need to customize it based on the actual specifications of your robots.

Let's assume you have a differential drive robot called `my_robot`. Create an SDF file named `my_robot.sdf` for this robot:

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <model name="my_robot">
    <static>false</static>
    <link name="chassis">
      <pose>0 0 0.1 0 0 0</pose>
      <collision name="chassis_collision">
        <geometry>
          <box>
            <size>0.2 0.2 0.1</size>
          </box>
        </geometry>
      </collision>
      <visual name="chassis_visual">
        <geometry>
          <box>
            <size>0.2 0.2 0.1</size>
          </box>
        </geometry>
        <material>
          <script>
```

```
        <uri>file://media/materials/scripts/gazebo.material</uri>
        <name>Gazebo/Wood</name>
      </script>
    </material>
  </visual>
</link>

<joint name="left_wheel_hinge" type="revolute">
  <parent>chassis</parent>
  <child>left_wheel</child>
  <axis>
    <xyz>0 1 0</xyz>
    <limit>
      <lower>-1</lower>
      <upper>1</upper>
    </limit>
  </axis>
  <physics>
    <ode>
      <use_fully_actuated>true</use_fully_actuated>
    </ode>
  </physics>
</joint>

<joint name="right_wheel_hinge" type="revolute">
  <parent>chassis</parent>
  <child>right_wheel</child>
  <axis>
    <xyz>0 -1 0</xyz>
    <limit>
      <lower>-1</lower>
      <upper>1</upper>
    </limit>
  </axis>
  <physics>
    <ode>
      <use_fully_actuated>true</use_fully_actuated>
    </ode>
  </physics>
</joint>

<link name="left_wheel">
  <pose>0 0 0 0 0 0</pose>
  <inertial>
```

```
    <mass>1</mass>
    <inertia>
      <ixx>0.0001</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>0.0001</iyy>
      <iyz>0</iyz>
      <izz>0.0001</izz>
    </inertia>
  </inertial>
  <collision name="left_wheel_collision">
    <geometry>
      <cylinder>
        <radius>0.05</radius>
        <length>0.05</length>
      </cylinder>
    </geometry>
  </collision>
  <visual name="left_wheel_visual">
    <geometry>
      <cylinder>
        <radius>0.05</radius>
        <length>0.05</length>
      </cylinder>
    </geometry>
  </visual>
</link>

<link name="right_wheel">
  <pose>0 0 0 0 0 0</pose>
  <inertial>
    <mass>1</mass>
    <inertia>
      <ixx>0.0001</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>0.0001</iyy>
      <iyz>0</iyz>
      <izz>0.0001</izz>
    </inertia>
  </inertial>
  <collision name="right_wheel_collision">
    <geometry>
      <cylinder>
```

```
        <radius>0.05</radius>
        <length>0.05</length>
       </cylinder>
      </geometry>
    </collision>
    <visual name="right_wheel_visual">
     <geometry>
      <cylinder>
       <radius>0.05</radius>
       <length>0.05</length>
      </cylinder>
     </geometry>
    </visual>
   </link>
  </model>
</sdf>
```

This example assumes a simple differential drive robot with two wheels (`left_wheel` and `right_wheel`). Adjust the dimensions and properties according to your actual robot's specifications.

To use this SDF file, you can include it in your ROS package and launch it in Gazebo. Create a Gazebo launch file (`my_robot.launch`):

```
<launch>
 <gazebo>
  <world name="default">
   <!-- Include your environment or other models here -->
   <include>
    <uri>model://ground_plane</uri>
   </include>
   <include>
    <uri>model://my_robot</uri>
    <pose>0 0 0 0 0 0</pose>
   </include>
  </world>
```

```
  </gazebo>
</launch>
```

This launch file includes both the ground plane and your `my_robot` model.

Launch Gazebo with your robot using:

```
roslaunch multi_robot_path_planning my_robot.launch
```

Replace `multi_robot_path_planning` with the name of your ROS package. This should launch Gazebo with your robot model. Adjust the file paths, model names, and launch configurations based on your actual robot and environment setup.