0.1 Agent Movement in Gazebo

The robot considered for our project is a differential drive robot. A differential drive robot is one that uses two independently driven wheels placed on either side of its body to move. This design allows for precise control over the robot's motion. Additionally, a third wheel termed as the 'caster wheel' is present for support, as can be seen in 1.

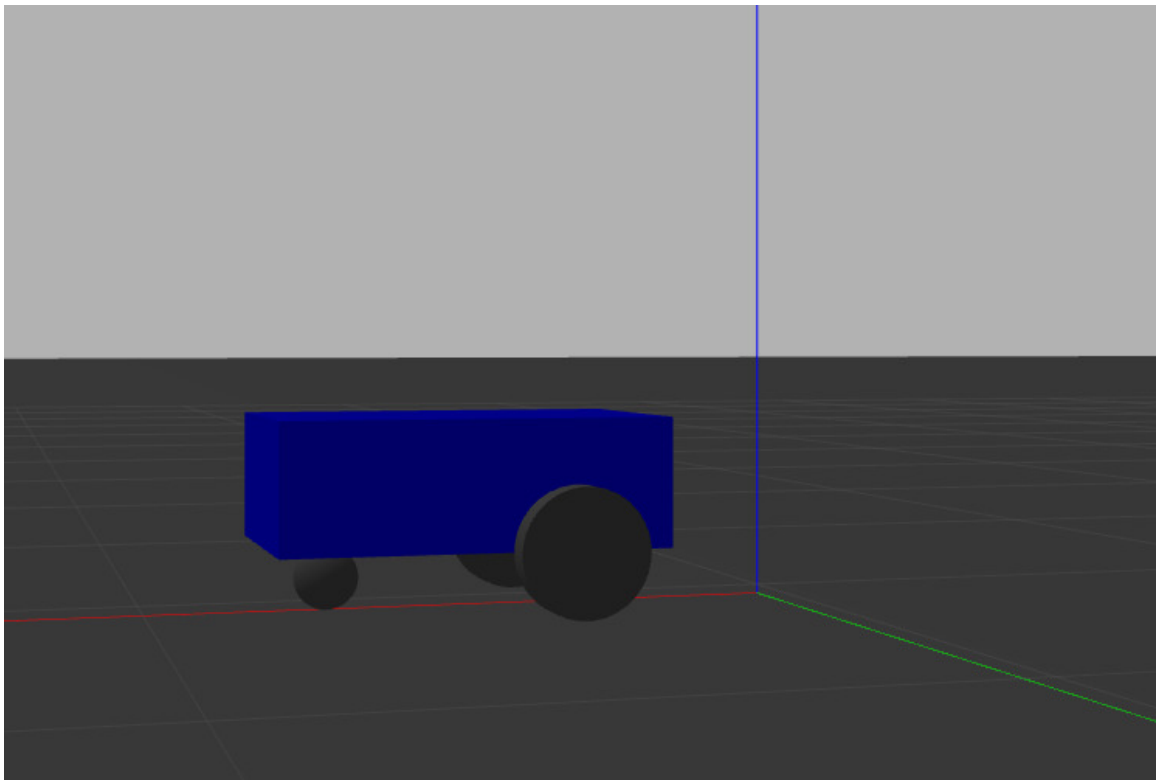

Figure 1: Our Differential Drive Robot Model

The agent listens to the instructions given to it by the path planner and executes them one by one to reach its destination in Gazebo's simulated environment. Instructions to the agent are given as an array of global cardinal instructions ('north', 'south', 'east' and 'west').

Since the agent is a differential drive robot, it assesses its current orientation and the given instruction and rotates by 90 degrees accordingly, if it needs to move along its perpendicular axis. This comparison and subsequent rotation (if required) is periodically done before each linear movement.

This project focuses on a grid-based warehouse structure, where the storage and operational areas are divided into a regular pattern of squares, forming a grid. Hence, it is essential for the agents to navigate such compact spaces precisely, and without deviating too much from its specified path. Additionally, the robot's movements should be immune to any external disturbances. For this, we have integrated PID control along with the script for the agent movement, to ensure smooth and accurate

agent movement.

Controllers help in regulating a system's behaviour to achieve a desired state or output. They do this by taking the output as feedback and comparing it with the desired output. The error, or the difference between the desired output and the system output, is processed to ensure an accurate output.

PID controllers process the error using three control actions:

1. **Proportional (P) Control:**

   It multiplies the proportional constant Kp with the error term. This helps adjust the output in proportion to the error.

2. **Integral (I) Control:**

   It multiplies the integral constant Ki with the integral of the errors encountered in the system over time. This addresses steady-state, or persistent errors in the system.

3. **Derivative (D) Control:**

   It multiplies the derivative constant Kd with rate of change of the error. This prevents abrupt control actions and anticipates future errors, providing stability.

We have implemented PID control for controlling both linear and angular velocity of the agent.

The agent subscribes to the 'nav_msgs' topic in order to gauge its position and orientation in Gazebo. Based on the error, which is the difference between the agent's desired destination and current position, the linear PID control calculates the desired linear velocity for the robot and publishes this information to the 'cmd_vel' topic, which is responsible for directing the agent's movements.

Similarly, angular PID control is implemented to ensure that the robot does not deviate from the axis it is required to move on, especially after rotation. If the robot deviates even slightly from its desired orientation, control action is taken in the opposite direction to orient it correctly before any linear movement.

The Kp, Ki and Kd values are collectively termed as 'PID gains'. 'PID tuning' is the process of adjusting these gains to achieve optimal performance. This involves finding the right balance between responsiveness, stability, and accuracy. We manually tuned the PID gains by observing the system's behaviour and by making precise modifications, to arrive at the following gain values:

- Linear velocity gains:
  - **Kp** = 0.2
  - **Ki** = 0.0002
  - **Kd** = 0.08

- Angular velocity gains:
  - **Kp** = 0.8
  - **Ki** = 0.005
  - **Kd** = 0.15

Thus, by implementing PID control for linear and angular velocity, we achieved precise and stable motion for our differential drive robot in the simulated warehouse. This control system effectively minimized errors in path following and ensured accurate navigation through the grid-based environment.

0.1.1 Architecture

The integration of task scheduling, path planning, and collision avoidance is critical to our system, ensuring efficient and safe operations. This section details how these functionalities are cohesively brought together through a central coordination mechanism.

An entity known as the 'Central Coordinator' oversees the system's state, which includes the positions of all agents and packages. When a new package arrives at the dropping station, the Central Coordinator is immediately notified. It then relays all relevant information to the State-map Publisher, which maintains an up-to-date map of the entire system's state. The package scheduler, path planner, and collision avoidance modules subscribe to this state-map to assess the system's current status and determine the next actions required for each module.

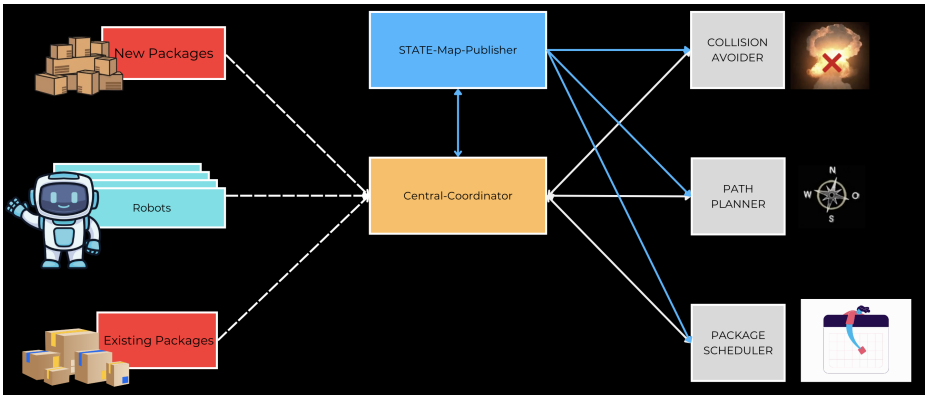

Figure 2: System Architecture

1. **Package Scheduler:**

   The Package Scheduler is responsible for processing requests for placing and dropping packages. It assigns specific goals to different agents within the system, ensuring that every agent is allocated a package. This module plays helps in balancing the workload among agents and optimizing the overall package handling process. By efficiently distributing tasks, the package scheduler enhances the system's throughput and minimizes idle times for agents.

2. **Path Planner:**

   The Path Planner employs algorithms such as Dijkstra and A* to determine the most efficient route for an agent to reach its assigned goal. When dealing with environments involving multiple levels, the path planning process is divided into two segments:

   - Planning a path from the agent's current position to the elevator on the first level.
   - Planning a path from the elevator to the goal on the next level.

3. **Collision Avoider:**

   The Collision Avoider is tasked with ensuring that agents do not collide with each other while executing their paths. It continuously monitors the planned paths of all agents, checking for any intersections. If a common point is detected between the paths of two agents, the module first verifies if either agent has already passed this point. If neither has, the path planner is signaled to generate a new path for the second agent, ensuring no points are shared with the first agent's path.

As a result of this seamless integration, the system can thus dynamically adapt to changes, handle multiple tasks concurrently, and prevent potential collisions, ultimately leading to a robust operation framework.

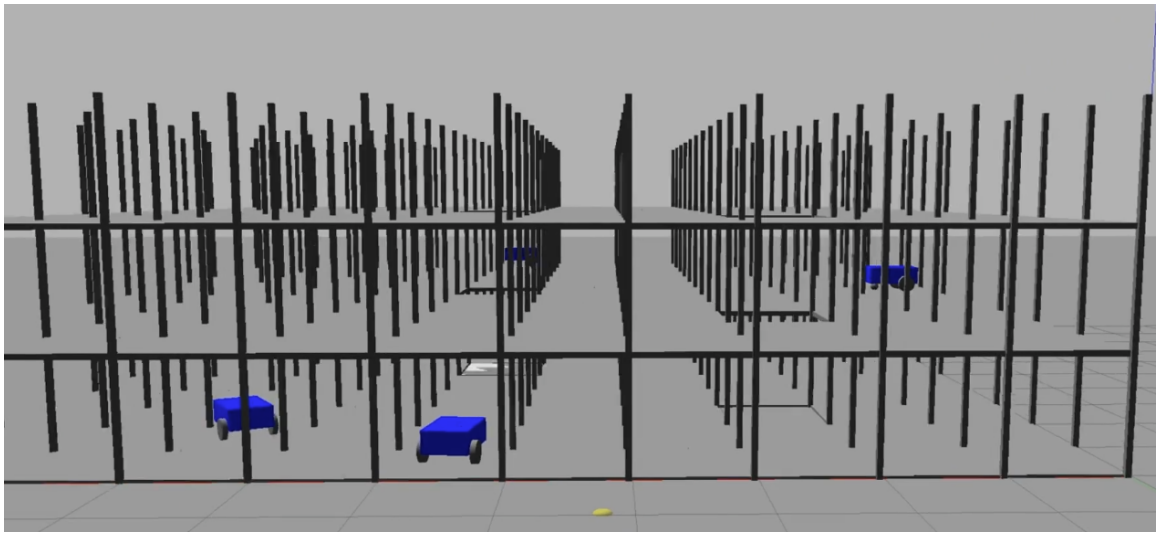

Figure 3: Dynamic Multi-Agent System Simulation in Gazebo