BHARAT FORGE

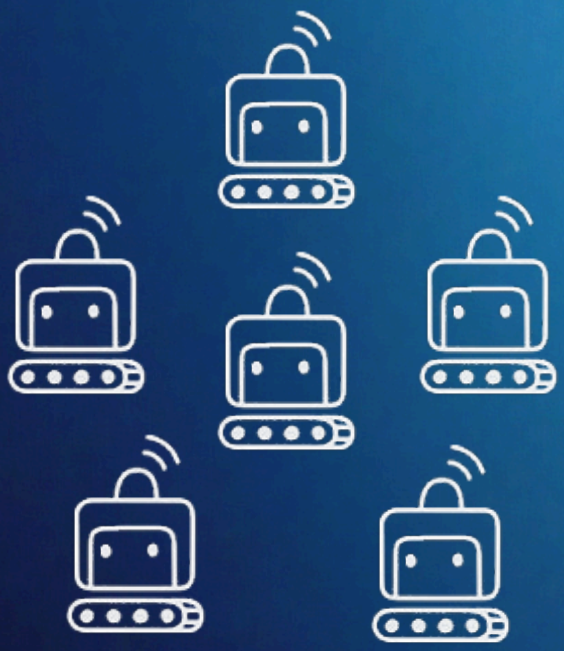
KALYANI

MID PREP PROBLEM STATEMENT

CENTRALIZED INTELLIGENCE FOR DYNAMIC SWARM NAVIGATION

SWARNA

SWarm Autonomous RL-aware Navigation Task Assistant



INTER IIT TECH MEET 13.0

TEAM 30

Contents

1 Problem Understanding

The growing role of automation in various industries has been significantly enhanced by self-driven robots operating in shared environments. A centralized control system ensures efficient task regulation, offering advantages like improved coordination and reduced communication overhead. The focus is optimizing robotic tasks in unknown, dynamic environments through effective mapping, object detection, and task allocation strategies.

1. **Real-Time Mapping and Exploration:** Robots must perform real-time mapping of unknown environments. Efficient exploration algorithms are critical to minimize mapping time while accommodating environmental changes.
2. **Object Detection and Tracking:** Object detection algorithms with minimal training time are essential for monitoring static and dynamic obstacles. Detected object coordinates must be stored in a continuously updating database to reflect real-time changes.
3. **Dynamic Path Planning:** Robust path planning guarantees that robots can effectively handle obstacles while performing tasks in a dynamic environment.
4. **Optimized Task Allocation:** Task allocation algorithms must reduce operating time while increasing output, which ensures efficient resource usage throughout the swarm.
5. **Centralized Control System:** A centralized control system simplifies the coordination of several robots by decreasing communication costs and assuring synchronized task execution.
6. **Chatbot Integration:** Integrating a chatbot simplifies the operation of the entire software suite, making it user-friendly and accessible.

2 System Architecture

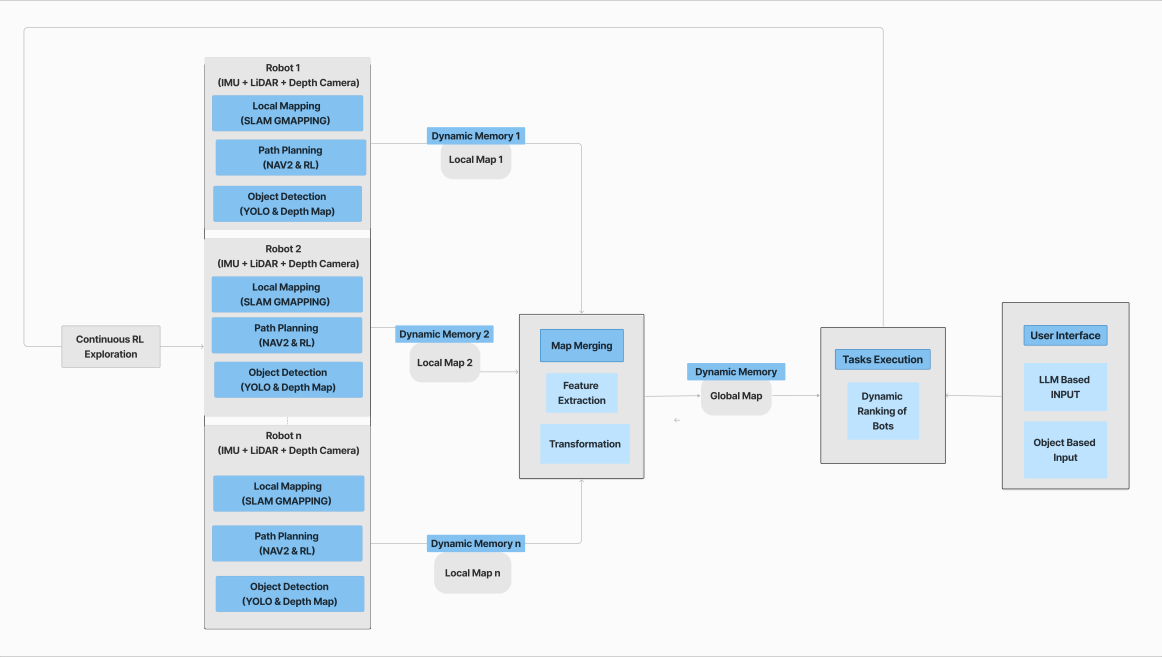
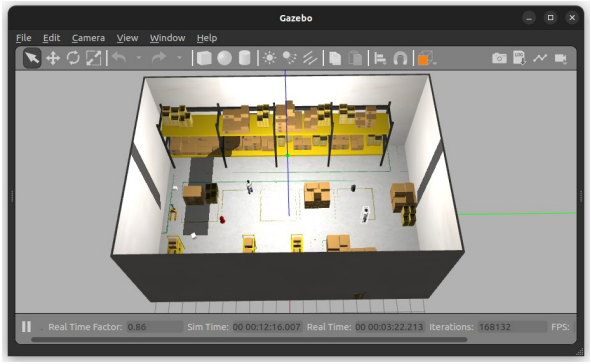


Figure 1: System Architecture Overview

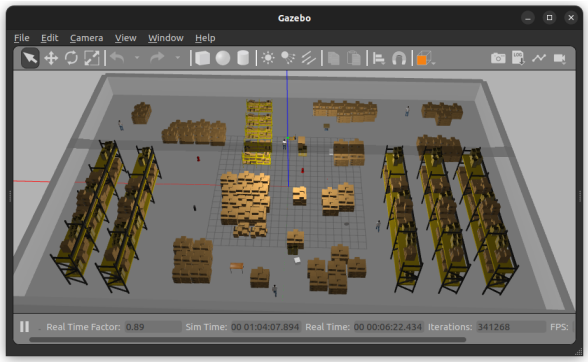
Users can provide LLM or object-based inputs to assign tasks for RL exploration and map merging using dynamic ranking. The final implementation involves local mapping, path planning, and object detection. Our system can efficiently scale up the environment and number of bots due to the use of RL and parameterization.

3 Development Platform Choice

- **ROS2 Humble**: Unlike ROS1, ROS2 [1] introduces real-time support for time-sensitive tasks and replaces custom middleware with Data Distribution Service [9], ensuring more **reliable** and **scalable** communication, especially when dealing with multi-agent systems.
- **Gazebo Classic**: We have tested our solution in the following two environments. Four dynamic elements were introduced in the environment shown in 2(b) and three in 2(a), respectively, to further test the algorithm's robustness.
- **TurtleBot**: Our simulation used the **TurtleBot3 Waffle** along with the **Intel RealSense R200 Depth camera** plugin. Please refer to the appendix for TurtleBot Navigation Stack.



(a) 40m X 20m map



(b) 40m X 60m map

Figure 2: Gazebo simulation worlds

3.1 Master-Client Configuration

In our project, autonomous robots within the same ROS DOMAIN ID are connected via the Master-Client configuration. Each mimicked robot independently carries out the following tasks:

- SLAM for environment mapping,
- YOLO for object detection, and
- Path planning for efficient navigation.

Detected object data and positional information are published to a centralized server, which:

- Combines the data into a unified object map.
- Manages a shared object memory or dictionary to store and synchronize detected objects.

This system employs distributed computation, assigning heavy processing to individual robots while coordinating efficiently via a central server. The technology is extremely scalable, with each robot functioning semi-autonomously. This decreases the processing load on the central server while ensuring synchronous job management. Please refer to the appendix for detailed work on the Master Client Architecture.

4 Exploration of Unknown Environment

An in-depth Literature Survey was conducted to understand various exploration methods used. There are mainly three types of autonomous exploration methods **Random, Frontier based and Learning Based**[?]. The random exploration method was rejected as it can lead to the clustering of robots.

4.1 Frontier-Based Exploration

We have used **explore_lite** package, which provides greedy **frontier-based exploration**. When the node was run in a small environment, the robot greedily explored its environment until no frontiers (the boundary between known and unknown space) could be found. It sends movement commands directly to the move_base package, which handles the robot's navigation. Unlike other exploration tools, explore_lite doesn't create its cost map, making it easier and lighter to set up on system resources.

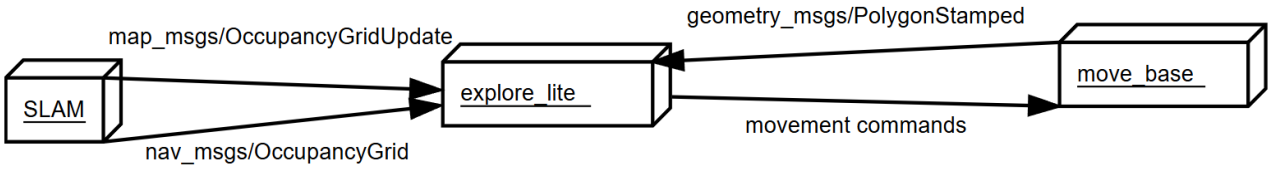


Figure 3: Architecture.

4.1.1 Limitations of the approach

- A greedy frontier selection approach leads to **redundant backtracking** and **increased exploration time for large and complex environments.**
- It lacks a global exploration strategy. This results in **local minima** where the robot oscillates or becomes stuck in a small area instead of exploring new frontiers.

4.2 Learning Based Exploration

RL-based exploration tactics are **more scalabl**e than frontier-based exploration because they adapt to changing conditions over time. Furthermore, it proposes a global exploration technique that eliminates closed places.

4.2.1 Model Testing and Selection

We tested several RL models to determine the most effective approach for exploration tasks in dynamic environments. We considered models such as Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimisation (PPO), and Deep Q-Networks (DQN), including Double DQN, Dueling DQN, Categorical DQN, and Rainbow DQN. Our studies showed that PPO-based models converged the fastest, while others struggled to map the environment within the maximum timesteps. The PPO-based A2C model demonstrated high stability and efficiency during the exploration task. The PPO-based model predicts the next exploration goal for each robot, maximizing total area explored in the lowest period of time. Robots optimize their exploration paths by fine-tuning parameters and reward structures, avoiding obstacles and limiting overlap with other agents.

4.2.2 Defining the Reward Function

For the task of multi-agent exploration, the reward function includes positive and negative rewards to encourage exploration while limiting collisions and inefficient behaviour. The components of the reward structure are:

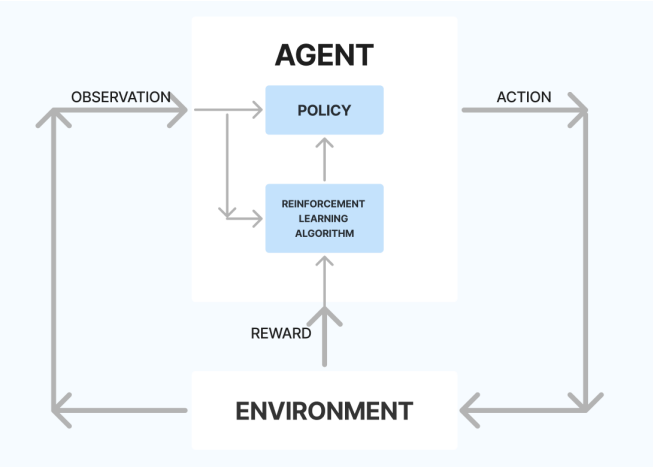


Figure 4: R-L model

- A penalty of **-10000** for moving outside map bounds: *Prevents out-of-bounds movement.*
- A penalty of **-1000** for moving into obstacles: *Ensures safe navigation.*
- A reward based on newly explored area and Euclidean distance: *Encourages exploration with minimal travel.*
- A **-10000** penalty for being within 1 meter of both explored space and obstacle: *Ensures a safe distance from obstacles.*
- A **+10000** reward for being within 1 meter of both explored space and unexplored boundary: *Encourages pushing into unexplored areas.*
- A **-10000** penalty for robots within 1.5 meters, **-1000** for within 5 meters: *Prevents collisions and clustering.*
- A reward based on the ratio of explored-obstacle to explored-unexplored boundary length, multiplied by 10000: *Promotes efficient exploration of boundary regions.*

4.2.3 Training Pipeline and Implementation Details

Random Map Generation for Generalization: We generate a new random map for each training episode featuring random obstacles and free space to promote generalisation. This ensures the agent encounters diverse environments and prevents overfitting to specific scenarios while maintaining ample exploration area.

Final Model and Training Objectives: The final model uses an **Actor-Critic** architecture, where the Actor chooses the robot's actions, and the Critic evaluates their effectiveness for exploration. This setup supports fast, safe exploration and continuous refining based on environmental feedback. The model aims to maximize exploration while minimizing collisions and pauses. Using PPO with A2C stabilizes learning, fostering efficient exploration strategies.

Multi-threading for Command Dispatching: To improve multi-robot exploration efficiency, we implement multi-threading for simultaneous command dispatch. This eliminates delays, ensuring synchronized actions and streamlined exploration by avoiding waiting times between robots.

5 Navigation

As shown in Figure 4, the model sends navigation commands to individual robots for exploration. We have deployed the SLAM algorithm to build a local cost map and report it back to the RL model to receive appropriate incentives. The implementation details are as explained below:

5.1 Path-Planning

Path planning is finding a collision-free path from starting to the goal point. We have used Nav2, the navigation stack for mobile robots designed for ROS 2. It has 3 main steps. **1.** Giving a goal position **2.** Finding a path to it using plugins based on the environment. **3.** Deploying a controller to follow the path.

How it is implemented.

- Environment Representation: Nav2 uses maps to represent the environment. These maps can be: Occupancy grids: 2D maps showing free, occupied, or unknown areas. Costmaps: Maps with cost gradients based on proximity to obstacles.
- Path Planning Plugins: Planner server uses plugins like **NavFnPlanner** (Dijkstra based) to compute paths.
- Global Planning: It uses costmaps and map data to calculate the optimal path using algorithms like **Dijkstra, A*,** or **hybrid A***.
- Local Planning: The controller server generates velocity commands to follow the global path and responds to obstacles using planners like **DWB**(Dynamic-Window Based) or custom controllers.
- Behaviour Trees: Nav2 utilizes **Behaviour Trees** (BTs) for task coordination. The NavigatePathToPose tree includes nodes like ComputePath and FollowPath for path calculation and execution.

Please refer to the appendix for detailed work on the Navigation Stack.

An approach using the PPO-based RL model was tested for path planning predicting successive adjacent pixels to plan an efficient path, using proximity to the goal as a reward. This turned out to be less efficient than regular A*/Dijksta-based path planning algorithms.

5.2 Costmaps and Their Role in SLAM

Costmaps are 2D grid-based representations of a robot's environment, where each cell encodes a cost value representing the traversability of the terrain or the presence of obstacles.
There are two primary types of costmaps:

- **Global Costmap:** Represents the known environment and is used for long-term path planning.
- **Local Costmap:** Focuses on the robot's immediate vicinity, dynamically updating with sensor inputs to handle real-time obstacle avoidance.

The accuracy of these costmaps depends on the SLAM algorithm used to generate the map. We have tested and integrated the following SLAM packages compared below:

ROS2 is compatible with `slam_toolbox`, while `slam_gmapping`, which mainly supports ROS1, needs a wrapper to implement it in ROS2.

5.3 Comparison of SLAM Toolbox and Gmapping

Aspect	SLAM Comparison	
	SLAM Toolbox	**Gmapping**
Algorithm Type	Pose graph-based SLAM with graph optimization.	Particle filter-based SLAM.
Dynamic Environment Handling	Supports lifelong mapping, enabling adaptation to changes in the environment.	Limited to static maps, with poor handling of moving obstacles.
Localization Accuracy	Robust localization through continuous map refinement and pose graph optimization.	Suffers from drift and inaccuracies in dynamic or complex environments.
Map Updates	Real-time and incremental updates via online and lifelong mapping modes.	Static maps created at runtime; no adaptation to environmental changes.
Memory Management	Optimized for large-scale maps with multi-session support, allowing incremental merging of new data.	Higher memory usage for large maps; no multi-session capability.
Unknown Environment Exploration	Efficient mapping of unknown areas using robust pose graph adjustments.	Slower and less efficient due to reliance on particle filters.
Dynamic Costmap Integration	Seamlessly integrates with dynamic costmaps for accurate real-time obstacle avoidance.	Limited dynamic costmap support, unsuitable for high-speed navigation.

Table 1: Comparison of SLAM Toolbox and Gmapping

5.4 Error Minimization

During SLAM, yaw values from odometry drift due to accumulated errors in IMU angular velocity integration. To address this, an **Extended Kalman Filter (EKF)** was integrated to fuse noisy sensor data. This improved yaw estimation, leading to **accurate orientation, better SLAM performance**, and handling of Gaussian noise in IMU data.

5.5 Emergency Node

The Emergency Node enhances dynamic obstacle avoidance. It monitors the environment using sensor data (**LaserScan topic**) and triggers an emergency stop upon detecting a dynamic obstacle within a safety threshold. After stopping, it identifies the direction with maximum free space and moves the robot to a safer location. Finally, it replans the path to the original goal. It caters to mapping delays and faults in perception, thereby significantly improving the system's precision.

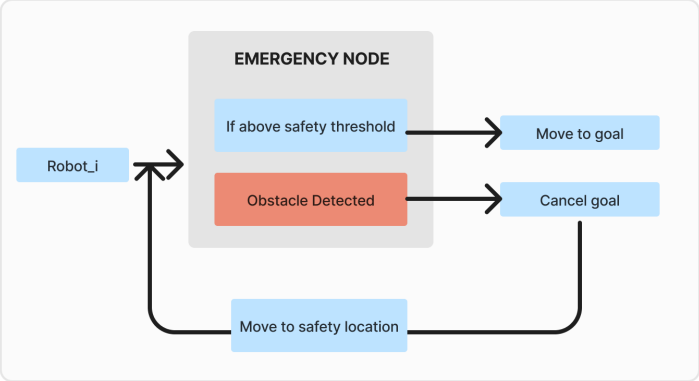


Figure 5: Emergency Node

6 Map Merging

Map Merging includes aligning and integrating numerous pictures or grids into a unified map. Important procedures include **warping** (using affine transformations like scaling or rotation), **feature matching** (identifying key spots, extracting descriptors, and finding matches), and utilizing a **grid compositor** to combine and polish the converted grids into a single representation.

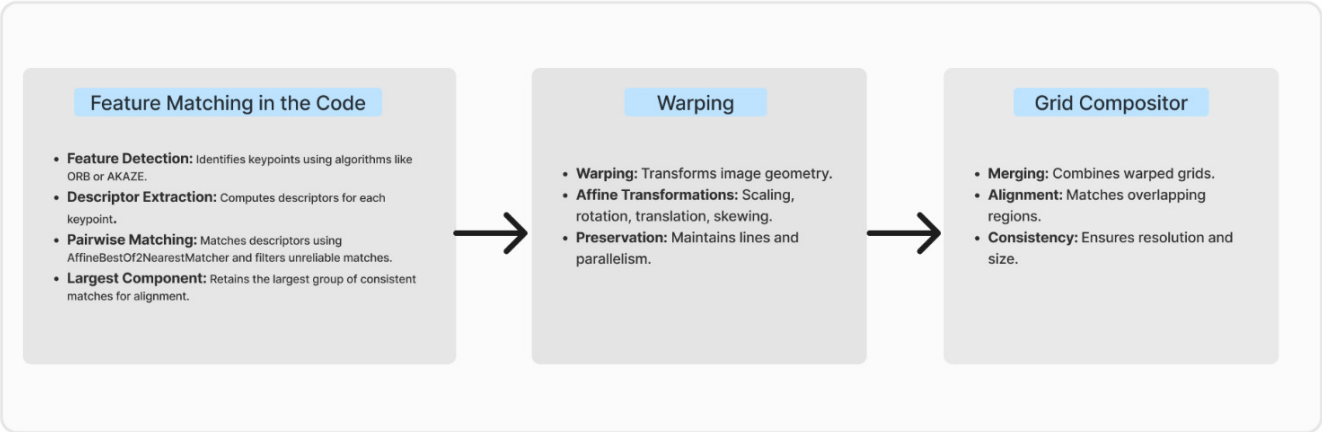


Figure 6: Map Merge Pipeline

7 Object Identification and Dynamic Memory

In highly dynamic environments where objects are being re-arranged, our robots need to identify and track their location. Each turtle bot is enabled with a Depth camera that provides RGBD values from the environment. We used **YOLOv3** for object detection during exploration to update their positions in the global map, marking the object's position with its central coordinate.

7.1 Object Localization

7.1.1 3D Coordinates of Object in Robot Frame

The 3D coordinates of a point in the scene in the robot's frame can be calculated as:

$$P = (X, Y, Z) = \left(\frac{(x - c_x)Z}{f_x}, \frac{(y - c_y)Z}{f_y}, Z \right)$$

where: $Z = D(x, y)$ is the depth at pixel (x, y), (c_x, c_y) is the principal point, and (f_x, f_y) are the focal lengths in pixel units.

7.1.2 Conversion from Robot to Global Frame

$$\text{object}_x = \text{odom\_position}_x + \text{dist} \cdot \cos(\text{orientation}) + \text{point}_x \cdot \sin(\text{orientation})$$

$$\text{object}_y = \text{odom\_position}_y + \text{dist} \cdot \sin(\text{orientation}) - \text{point}_x \cdot \cos(\text{orientation})$$

- $\text{object}_x, \text{object}_y$: Global coordinates of the object.
- $\text{odom\_position}_x, \text{odom\_position}_y$: Robot's global coordinates.
- $\text{point}_x, \text{point}_y$: Object's x-coordinate and y-coordinate in the robot's frame.
- dist: Distance from robot to object.
- orientation: Robot's orientation in the global frame.

7.2 Memory Persistence

A ROS 2 node tracks and maintains a persistent memory of objects and their locations in a dynamic environment. It subscribes to robot odometry and object detection topics, using the data to update the global map and store object positions in dictionaries (object_positions for current locations and object_history for historical movement). Object positions are updated only if they are significantly different, and the global map is updated with both robot and object positions. Every second, the node publishes the updated map and object history, ensuring the system retains and shares the dynamic memory of objects.

8 Large Language Model (LLM) Interface

We have implemented an LLM interface for the user's input, making it easy to use. The Google Gemini API [12] is deployed to give the process the input and identify the intended task. The output coordinates are then calculated for where the user wants the robot to move.

9 Dynamic Ranking of Tasks and Allocation

9.1 Task Allocation

To coordinate multi-robot task execution, we designed our optimization algorithm, which is described below:

- A **priority queue** is defined, where each task is a pair of the form (Urgency, Time of Arrival). The Urgency indicates the task's priority (1 for urgent, 0 for non-urgent), and Time of Arrival corresponds to the index at which the task arrives. The queue is sorted using the following criteria:
  - Urgent tasks (Urgency = 1) have higher priority than non-urgent ones (Urgency = 0).
  - If two tasks have the same urgency, the task with the earlier arrival time is given higher priority.

- Three types of bots are defined:
  - **Free** bots: Not assigned any task yet, typically exploring or updating the environment.
  - **Non-Urgently busy** bots: Bots assigned to non-urgent tasks.
  - **Urgently busy** bots: Bots assigned to urgent tasks.
- Task types are handled as follows:
  - For **non-urgent** tasks, the algorithm checks all free bots to find the one nearest to the task's object location.
  - For **urgent** tasks, both free and non-urgently busy bots are considered. The nearest bot is selected.
- If a busy bot is selected for a new task, the previously assigned task is pushed into the priority queue, and the new task is assigned. Thus, the most optimal robot and task coordinates are then executed.

9.2 Task Execution

The execution of the task is done by Nav2, which plans the path using Global Path Planning Algorithms like A* and Dijkstra's Algorithms. The execution of tasks is carried out following the concepts mentioned in Exploration of Unknown Environment and Navigation sections.

10 Outlook

10.1 Challenges

The pipeline involved running Gazebo simulations, individual costmaps, and map merging algorithms alongside RL-based exploration while simultaneously executing YOLO object detection for each robot. These computationally heavy modules required multiple devices connected via LAN to distribute the workload and ensure smooth operation. Replicating real-world robot behaviour presented considerable hurdles since we had to account for faults in the odometry data, which frequently caused inaccuracies. To address this, we carefully designed and optimized our algorithms to minimize the impact of these errors and improve the system's overall reliability.

10.2 Future Scope

- Autonomous Exploration for Search and Rescue: Swarm robotics is suited for exploring unknown, complex, or hazardous environments, such as space, disaster-hit areas, or challenging terrains, enabling efficient management and intervention.
- Industrial and Agricultural Automation: The system can boost coordination and precision in industries such as warehousing, agriculture, mining, and construction through task optimization.
- Urban Infrastructure and Environmental Monitoring: Swarm robots can tackle urban challenges like traffic monitoring and waste collection while also ensuring the safety of animals and the environment.

10.3 Recommendations

- The solution is currently relying on Ubuntu 22.04 and ROS2 Humble. Dockerizing this system will remove this constraint while also simplifying the implementation process for embedded computing platforms.

- Integrating **OpenRMF** improves swarm navigation scalability by allowing additional robots to be easily included into the system without requiring large reconfigurations. Its modular architecture facilitates dynamic task distribution, efficient communication, and resource management, ensuring that performance stays steady and responsive as the number of robots grows, allowing for large-scale swarm operations.

10.4 Lessons Learnt

The project provided an excellent opportunity to learn about designing and simulating swarm robotics systems. We gained important insights into mapping, localization, and navigation challenges while also focusing on the system's scalability and efficiency. Building a modular architecture helped us to work on different sections of the pipeline more easily.
The technical challenges of the problem statement required assessing various SLAM methods and improving their accuracy and dependability. A deeper insight into scalability in swarm systems was also made possible by the implementation of dynamic task allocation for multi-robot cooperation, which highlighted the difficulties of resource management under computing restrictions.

11 Performance Analysis

11.1 Analysis of RL Exploration

No. of bots vs iterations taken to map the environment by RL exploration. The efficiency of RL exploration is directly proportional to the number of bots.

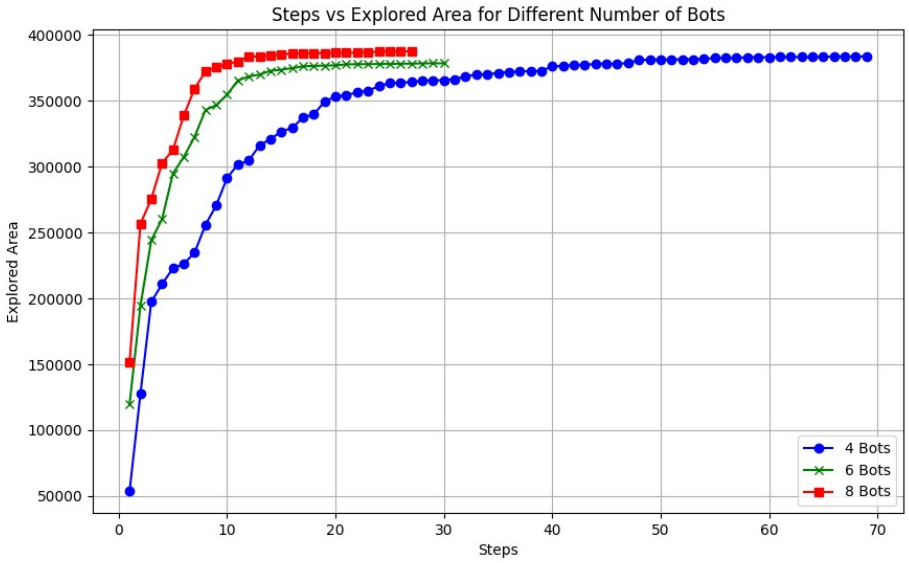


Figure 7: Number of Pixels per Step

11.2 Analysis of Task Allocation Module

The use of a priority queue allows for fast task sorting based on urgency and arrival time, ensuring that high-priority tasks are handled first. The algorithm efficiently assigns tasks to free and non-urgently busy bots, minimizing delays in urgent task execution. The time complexity for each task assignment is $O(K \log N)$, where K is the number of robots and N is the number of tasks, which is manageable for moderate swarm sizes. .

Appendix

References

[1] ROS2 Humble Documentation

[2] Nav2 Documentation

[3] slam gmapping ROS2

[4] m-explore ROS2

[5] P.-Y. Lajoie and G. Beltrame, "Swarm-SLAM: Sparse Decentralized Collaborative Simultaneous Localization and Mapping Framework for Multi-Robot Systems," *IEEE Robotics and Automation Letters*, vol. 9, no. 1, pp. 475–482, Jan. 2024. DOI: 10.1109/LRA.2023.3333742.

[6] S. Sharma and R. Tiwari, "A survey on multi robots area exploration techniques and algorithms," *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, New Delhi, India, 2016, pp. 151–158. DOI: 10.1109/ICCTICT.2016.7514570.

[7] De Rose, M. (2021). *LiDAR-based dynamic path planning of a mobile robot adopting a costmap layer approach in ROS2* (Master's thesis). Politecnico di Torino. https://webthesis.biblio.polito.it/21253/1/tesi.pdf.

[8] Dynamic collision avoidance using ProximitySensors

[9] DDS Implementations

[10] https://roboticsbackend.com/ros2-multiple-machines-including-raspberry-pi/

[11] Arrow_SensorFusion_turtlebot3

[12] https://ai.google.dev/gemini-api/docs

System FlowCharts

1. Master Client Architecture

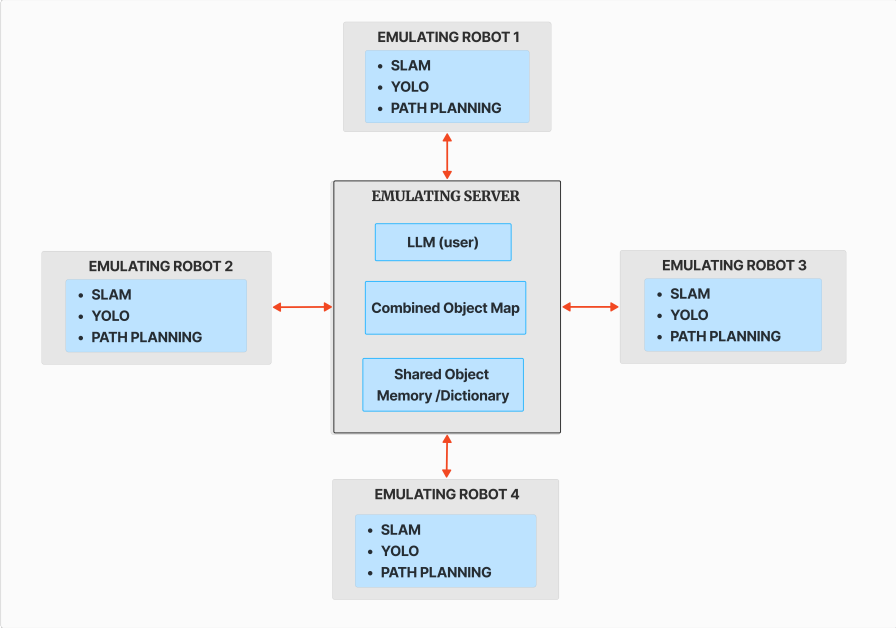


Figure 8: Master Client Architecture

2. TurtleBot Navigation Stack

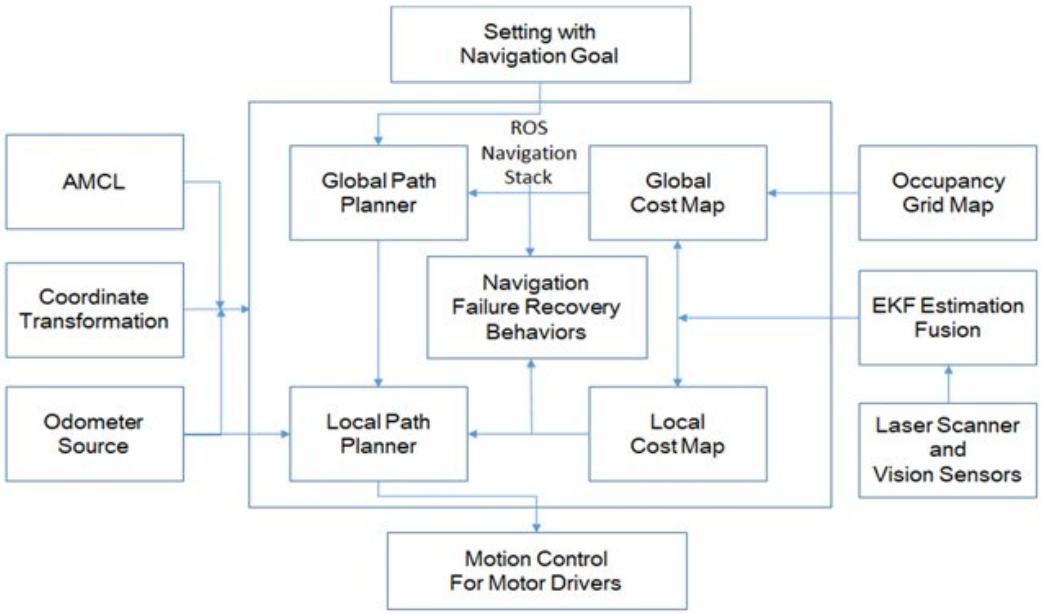


Figure 9: TurtleBot Navigation Stack

3. Nav2 Architecture

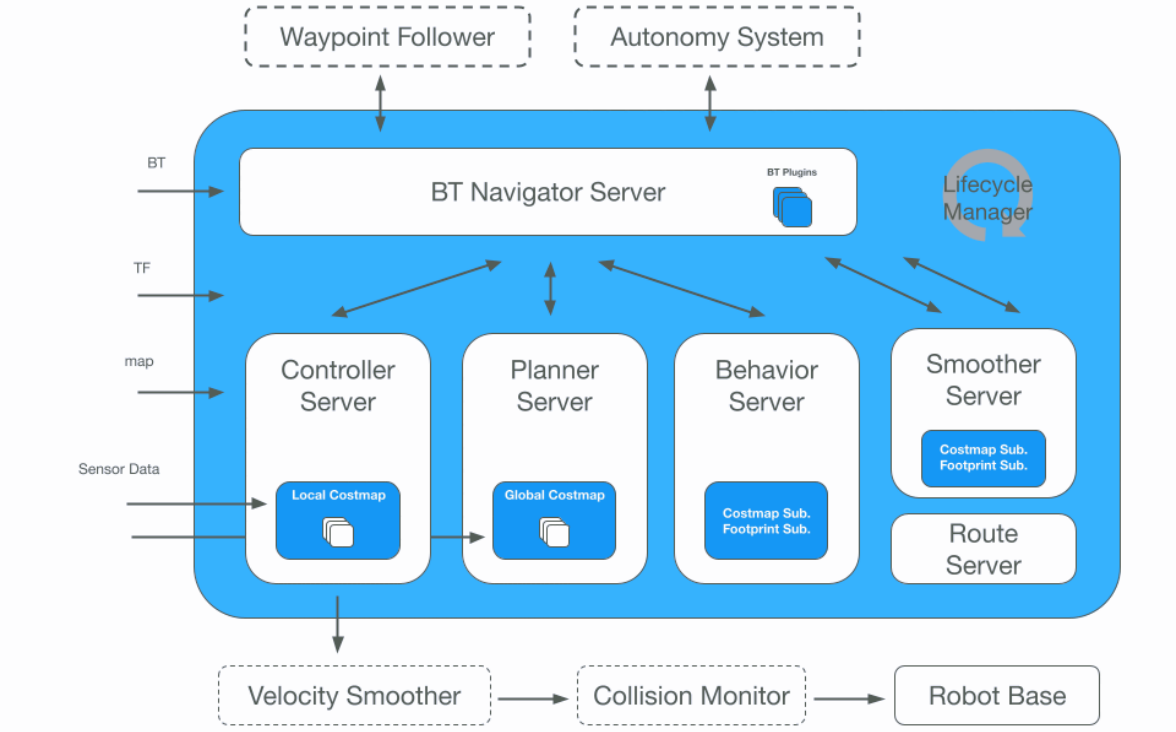


Figure 10: Navigation Stack