Mail Delivery Robot in Carleton Tunnels

Final Report

Denis Cengu 101077902
Douglas Lytle 101147977
Cholen Premjeyanth 101184247

Supervisor: Dr. Babak Esfandiari

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

April 8th 2025

Abstract

By Denis Cengu

The Carleton Mail Delivery Robot is designed to autonomously deliver mail to various locations within the Carleton University tunnel system. The system integrates the iRobot Create 3 along with a Raspberry Pi 4, a LiDAR sensor and a Pi Camera to enhance obstacle detection and navigation capabilities. Navigation relies on strategically placed Bluetooth beacons, using signal gradients to determine proximity and direction. Robust protocols, including obstacle avoidance, collision handling, and dynamic navigation, ensure seamless operation in complex environments.

Building on last year's design, the team is implementing a modular subsumption architecture to simplify the addition and removal of behaviors, enhancing system flexibility and maintainability. This report details the software and hardware components of the system, highlights progress made so far, and discusses future improvements. It expands on the design decisions outlined in the proposal while summarizing the project's current achievements.

See Appendix E for demonstration videos and a link to the project repository.

Table of Contents

List of Tables

List of Figures

1 Objectives

1.1 Project Objectives

By Douglas Lytle

The primary objectives for this project remained largely the same as they were in previous iterations. The robot should be developed such that it is able to autonomously navigate from an arbitrary location in the Carleton tunnels to any of its destination docking stations, while avoiding obstacles. The robot should be able to carry mail and ensure it is not lost in transit, and must have a battery capacity large enough to complete deliveries. This year, the robot should be able to navigate between at least three different locations in the Carleton Tunnels. The web app should also continue to be developed and will remain the primary method for users to interact with the robot.

Early in the development of the project, an additional focus was placed on rectifying the issues outlined in last year's report's sections titled *Documented Issues*, and *Future Work*. These include problems preventing the robot from properly docking, undocking, and stopping. They also include improvements to intersection detection, and the state machine used for navigation should be greatly simplified or potentially removed.

In order to facilitate these goals, another objective which was handled early in development is the implementation of simulations using Gazebo, which is a much more sophisticated way to test the system compared to what was used last year.

1.2 Project Description

By Denis Cengu

This was the fourth year of this project. The main goal is to create an autonomous robot that can deliver mail between different buildings at Carleton University through its tunnels. The project will include a web application that will allow users to easily place their orders and manage deliveries.

Last year's team made improvements to the robot's movement abilities, refining the wall following, turning and intersection navigation. The primary focus for this year's team was to ensure more consistent and reliable behavior. This included the introduction of new sensor systems and reworking or altogether scrapping the state machine approach currently used for navigation as well as making the system more scalable and adaptable.

The key areas for this year's team are to ensure the primary focus are improvements to docking and undocking, which the robot currently struggles due to reliance on the short range IR sensors, intersection detection, a removal of the state machine to simplify the navigation, implementation of Gazebo simulations as well as further improvements to the security of the mailbox.

1.3 Requirements

By Douglas Lytle and Cholen Premjeyanth

1.3.1 Functional Requirements

- **R1**: The robot shall be able to autonomously navigate from any starting location within the Carleton tunnels, to one of the designated destination stations.
- **R2**: The robot shall be able to detect and handle collisions with obstacles.
- **R3**: The robot shall be able to detect and identify intersections within the tunnels.
- **R4**: The robot shall ensure mail is not lost during delivery.
- **R5**: The robot shall be able to notify the system of its current status, including mail delivery progress, battery levels, and docking status.
- **R6**: The robot shall be able to receive signals from Bluetooth beacons placed in the tunnels.
- **R7**: The robot shall be able to dock automatically upon reaching its destination or when the battery is low.
- **R8**: The system shall allow users to register an account using the web application.
- **R9**: The system shall allow users to make delivery requests using the web application.
- **R10**: The system shall be able to notify the robot of new delivery requests.
- **R11**: The system shall be able to provide detailed logs of all robot activities.
- **R12**: The web application shall be able to display the approximate location of the robot.

1.3.2 Non-Functional Requirements

- **R13**: The system shall ensure only administrators have access to administration features.
- **R14**: The robot shall maintain a speed of at least 0.15m/s while navigating through the tunnels.
- **R15**: The system shall be tested thoroughly to ensure reliability of the system.
- **R16**: The robot shall maintain enough of its battery level to reach the nearest docking station at all times.
- **R17**: The web application shall be available to any registered user connected to Carleton University's Wi-Fi.
- **R18**: The project shall remain within the specified budget.
- **R19**: The project timeline and deadlines shall be followed and met.

1.3.3 Final Evaluation of Project Status in Relation to Requirements

By Douglas Lytle

It should be noted that as these requirements were gathered at the start of the project, the actual scope of the project differed slightly from what was expected at that time. The main source of this discrepancy is that the team made the decision due to time constraints and scope concerns to not develop the web app any further. Specifically, R5, R8, R9, R10, R12, R13, and R17 were deemed to be outside of the scope of the project for the current year due to only concerning the web app.

However, as the project evolved throughout the past year, there were a number of unforeseen areas in which significant development took place and new goals and requirements emerged. Most notably, the entire system was rebuilt using a subsumption architecture, as opposed to last year's state-based approach, which the team believes is a huge improvement to the extensibility of the system. A new source of sensor data was added, and intersection detection was improved noticeably as a result of this and tweaks to existing sensors. Robot navigation was decoupled from the control of the robot, which facilitated testing and error correction. Issues preventing the robot from docking and undocking correctly were fixed. A feature was implemented which allows the robot to stop mid-delivery and recharge its battery when necessary before resuming the delivery. Finally, a robust simulation using Gazebo was created, which should also be of great usefulness for any future development.

1.4 Use Cases

By Cholen Premjeyanth

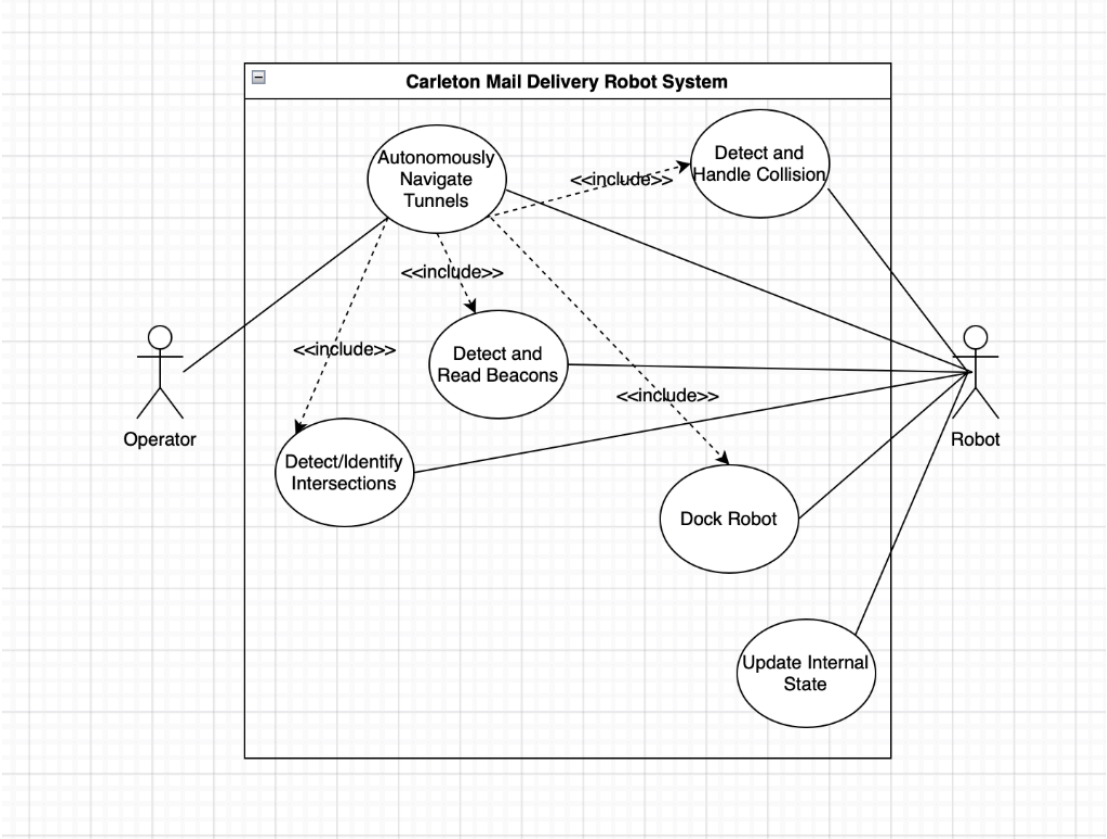


Figure 1: The use case diagram for the Mail Delivery Robot System.

1.4.1 Individual Use Cases

By Cholen Premjeyanth

Table 1: Autonomously Navigate Tunnels use case

Use Case Name	Autonomously Navigate Tunnels
Brief Description	Robot autonomously navigates from one location to another using wall following, beacon data, and internal maps.
Primary Actor	Robot
Secondary Actor	Operator (In this case this refers to the person who initiates the delivery through their terminal)
Precondition	Robot is active, has a battery, beacons are placed, and a delivery is initiated.
Postcondition	Robot successfully navigates to the specified destination and updates internal status.
Basic Flow	1. Robot finds the nearest wall, and starts navigation towards its destination. 2. As the robot moves, it uses LiDAR data to detect and navigate around obstacles. 3. The robot continually seeks and moves towards the next beacon in its path, updating its location periodically. 4. At intersections, the robot uses sensor input to determine and execute routing decisions 5. The robot detects the dock and initiates the docking process. 6. Robot docks successfully and updates internal logs with completion status.
Alternate Flows	a) If the path is blocked, the robot recalculates the route using last known beacon and orientation. b) If the robot detects an obstacle, it attempts to reroute using a built in obstacle avoidance layer.

Table 2: Detect and Handle Collision use case

Use Case Name	Detect and Handle Collision
Brief Description	The robot detects collisions using sensors, and reroutes to avoid obstacles while resuming delivery.
Primary Actor	Robot
Secondary Actor	N/A
Precondition	Robot is in motion within the tunnels
Postcondition	Robot successfully detects and/or avoids obstacles while being able to reorient itself onto its path.
Basic Flow	1. Robot detects a collision via a bumper sensor. 2. The robot's avoidance layer takes control and stops forward movement for a few seconds. 3. The robot attempts to bypass the obstacle. 4. This maneuver is attempted upto 4 times. 5. If the obstacle persists, the robot reroutes using an angular turn. 6. Once a clear path is found, the robot continues its navigation towards the destination.
Alternate Flows	N/A

Table 3: Detect and Read Beacons use case

Use Case Name	Detect and Read Beacons
Brief Description	The robot is able to detect beacon signals traveling through the tunnels.
Primary Actor	Robot
Secondary Actor	N/A
Precondition	Beacons are active, and the robot is turned on within the tunnels.
Postcondition	Beacon signal strength is logged and used to determine navigation updates.
Basic Flow	1. Robot detects the beacon signal.

	2. Signal strength and ID are logged. 3. Robot updates its internal state.
Alternate Flows	N/A

Table 4: Detect/Identify Intersections use case

Use Case Name	Detect/Identify Intersections
Brief Description	Robot identifies intersections using LiDAR and camera, then determines appropriate routing action.
Primary Actor	Robot
Secondary Actor	N/A
Precondition	Robot is in motion and approaches an intersection.
Postcondition	Robot successfully traverses the intersection using the correct path.
Basic Flow	1. Robot's intersection detection unit analyzes LiDAR and camera data. 2. Robot identifies the intersection through data. 3. Robot's navigation unit provides a routing instruction. 4. Robot's turning Layer executes the turn based on instruction. 5. Control is returned to the robot's travel layer once turn is complete.
Alternate Flows	N/A

Table 5: Dock Robot use case

Use Case Name	Dock Robot
Brief Description	The robot initiates docking either after completing a delivery at the destination station or when the battery drops below a critical threshold during a task.
Primary Actor	Robot
Secondary Actor	N/A

Precondition	Robot has either completed a delivery or reached a low battery threshold.
Postcondition	Robot is docked and charging. Internal system status is updated to reflect the charge state and operational readiness.
Basic Flow	1. Robot detects a low battery or is completing a delivery. 2. Robot calculates and navigates the path to the nearest docking station. 3. The robot uses its docking station sensors to align and connect accurately with the docking station. 4. The robot sends a status update to the internal state, indicating that it has reached the docking station and is recharging. 5. Robot initiates the charging process, and the system logs and displays the docking and battery recharge status.
Alternate Flows	a) If docking fails, the robot retries docking or sends an error update to the system.

Table 6: Update Internal State use case

Use Case Name	Update Internal State
Brief Description	Robot updates internal logs with
Primary Actor	Robot
Secondary Actor	N/A
Precondition	Robot is operational and running.
Postcondition	Logs are updated with the most recent activity.
Basic Flow	1. Robot completes an action. 2. State is updated and logged internally. 3. Status is accessible by operator.
Alternate Flows	a) If there is no internet connection, then keep trying to reconnect.

2 The Engineering Project

2.1 Health and Safety

By Cholen Premjeyanth

The testing, development, and future upgrades of the robot was continued in the Canal lab room (CB 5101) and the Carleton University tunnels, as in previous years. All experiments were conducted in controlled environments, ensuring there were no variables that may obstruct the robot's operation.

The safety of both the team and the public were prioritized during testing. In areas where the robot may operate near people, clear communication and safety measures were implemented to inform the surrounding individuals and minimize risks.

When handling the robot's hardware, strict safety protocols were followed. This included removing loose clothing and jewelry, maintaining a safe distance from electrical probes, and ensuring that all power sources are disconnected prior to working on any electronic components. These precautions ensured a safe working environment and protected both the team and equipment.

2.2 Engineering Professionalism

By Cholen Premjeyanth

ECOR 4995 emphasized the importance of professional responsibility throughout every stage of our project. One of the most valuable aspects was learning how to collaborate effectively in a team. From weekly meetings to hands-on testing, we developed strong communication and decision-making skills, which were essential to managing such a large and evolving project.

We prioritized public safety in all design decisions, especially considering the robot would operate in shared spaces like tunnels. Ethical engineering practices were also reinforced, such as citing sources correctly and respecting intellectual property rights when referencing previous work.

We followed an agile development process, which taught us how to adapt and manage unforeseen challenges while keeping progress on track. This experience also highlighted the value of accountability, documentation, and iteration which are key principles in professional engineering environments. Overall, the course provided a strong foundation in both the technical and ethical expectations of the engineering profession.

2.3 Project Management
By Cholen Premjeyanth

The project required the team to implement various project management techniques over the course of the term. Each technique is described in the following sections.

2.3.1 Software Development Methodology
By Douglas Lytle

For the continued development of this year's project, the team worked using an Agile software development model. This allowed any design flaws to be discovered as early as possible, and allowed for the goals of the project to be iterated upon as more work was completed. The many benefits of an Agile methodology were learned in SYSC 4106 (Software Economics and Project Management). The team also tried to follow the practices of Continuous Integration, which were learned about in SYSC 4806 (Software Engineering Lab). Continuous Integration will allow for changes to individual modules in the system to be made more freely and frequently, without fear of a long and difficult integration period.

2.3.2 Revised Timeline
By Denis Cengu and Douglas Lytle

Table 7: Revised project milestones and target completion dates along with current status

Milestone	Target Completion Date	Status
Introduce Gazebo simulations for testing the system	October 16th, 2024	Complete
Complete the implementation of last year's system in to gather performance data	October 25th, 2024	Complete
Proposal	October 28th, 2024	Complete
Redesign mail holder	Nov 1st, 2024	Complete
Complete the design of the new system architecture	Nov 5th, 2024	Complete
Implement stubs for everything so that any new code can be tested in Gazebo in the context of the new architecture	Nov 12th, 2024	Complete
Implement modules for individual behaviors	Nov 26th, 2024	Complete
Implement module responsible for behavior switching	Dec 3rd, 2024	Complete

Refine behavior switching	Jan 21st, 2024	Complete
Oral Presentation	TBD (January 20th-24th)	Complete
Improve performance of modules for individual behaviors	Feb 11th, 2024	Complete
Implement the new web app features	Feb 25th, 2024	Abandoned
New system is able to complete a delivery with performance close to the targets	March 18th, 2024	Complete
Poster Fair	March 28th, 2025	Complete
Final Report	April 8th, 2025	Complete

2.3.3 Gantt Chart

By Douglas Lytle

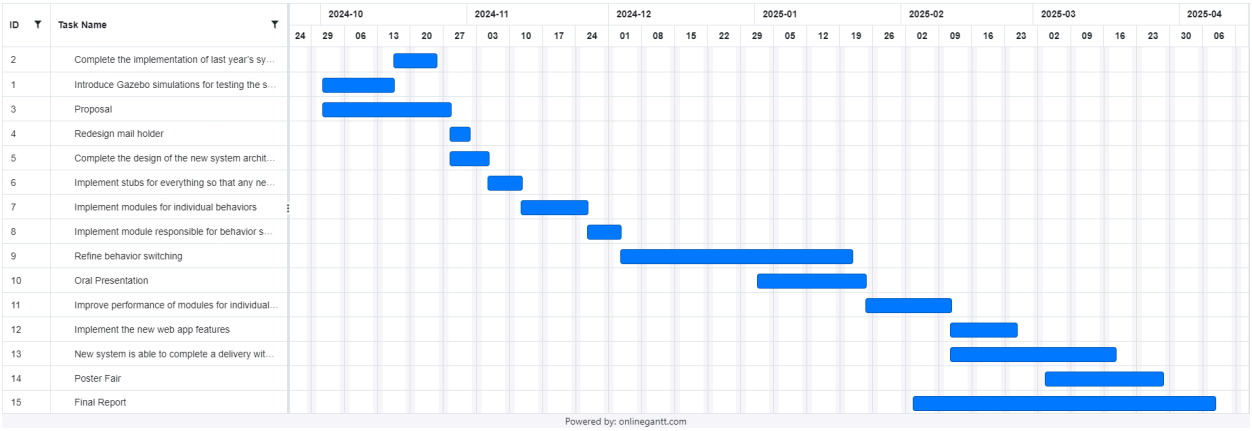


Figure 2: Gantt chart showing project timeline

2.3.4 Risks and Mitigations

By Douglas Lytle

Table 8: Risks to the project and strategies to mitigate them

Project Risk	Mitigation Strategy
The existing Bluetooth beacons might not have sufficient signal strength/reliability or battery capacity to be effective for navigation through the tunnels.	The team will take care to test the beacons and ensure they are functional for their required purpose. If they are found to not be sufficient, enough project budget will be maintained so as to potentially buy more powerful beacons.

The robot could be damaged during testing or operation.	The team will take every possible precaution in order to avoid both damage to the robot, as well as its surroundings.
Integration of code could prove difficult and very time-consuming if the team has different or incorrect assumptions about the functions of modules of the system.	Practices of continuous integration will be used to ensure no code can be published to the main branch without first being rigorously tested in the context of the complete system.

2.4 Justification of Suitability for Degree Program
By Denis Cengu, Douglas Lytle, Cholen Premjeyanth

The team has two students in computer systems engineering and a student in software engineering. These programs are similar and both have major focuses on topics in software and computer engineering. Both programs cover topics related to embedded development, requirements engineering, web development, real-time systems and software development among others. The scope of the project encompasses all of the above.

The emphasis of the project revolves around a code base for the robot developed in Python using the Robot Operating System (ROS). The robot itself is driven through use of a Raspberry Pi 4. The Raspberry Pi is used in SYSC 3010 (Computer Systems Development Project) , where students in groups use multiple Pi's that communicate with one another to create an embedded system primarily with a Python code base as well. As well as that course for Computer Systems students, Software Engineering students have also covered at a basic level these concepts in ECOR 1051 (Fundamentals of Engineering I) and ECOR 1052 (Fundamentals of Engineering II). Furthermore, the knowledge gained from SYSC 3303 (Real Time Concurrent Systems) was useful in this project as we had to replace a state-machine based system, with a subsumption based architecture, which are topics all covered in the course.

The hardware required to operate the robotics for this project requires knowledge of circuitry as well experience with hardware and software integration. As previously mentioned, in SYSC 3010 students have learned how to integrate hardware and read data in the project. As well, circuit knowledge as well as electronic theory was learned in ELEC 2501 (Circuits and Signals) along with ELEC 2507 (Electronics I). Additional experience in integration with hardware sensors and software was acquired in SYSC 4805 (Computer Systems Design Lab).

Throughout the project, the material covered in SYSC 4120 (Software Architecture and Design) was useful to manage the structure of the project. Also, testing knowledge gained from SYSC 4101 (Software Validation) was applied here.

2.5 Individual Contributions
By Douglas Lytle

The following sections describe the contributions of each team member to this report, and to the project as a whole.

2.5.1 Report Contributions
By Douglas Lytle

Below each subsection heading in this report the author of that subsection is listed. If there are multiple names, then that indicates multiple people helped with the writing of that section.

2.5.2 Project Contributions
By Denis Cengu, Douglas Lytle, and Cholen Premjeyanth

- Denis Cengu:
    - Testing plan and performance metrics tracked.
    - Gazebo simulator environment with designed sensors to replicate the real device.
    - Implementation of the travel layer node.
    - Design and installation of the mail holder

- Douglas Lytle:
    - Design of the system and its architecture, including the determination of all necessary software components and how they should communicate.
    - Implementation of the camera sensor, navigation unit, intersection detection unit, avoidance layer, docking layer, action translator, music player, and captain nodes.
    - Deconstructing and determining what parts of the codebase from last year could be re-used and how to best do so.
    - Maintaining the project and correcting errors as they arose.

- Cholen Premjeyanth:
    - Implementation of the turning layer, and battery monitor node.
    - Analyzed and tested hardware features of the Create 3, and tested code from last year to understand which aspects were transferable towards subsumption architecture.

3 Group Skills

By Denis Cengu, Douglas Lytle, and Cholen Premjeyanth

The project team consists of three members, with members from both the computer systems engineering and the software engineering programs. This gives the team collectively a broad knowledge base of engineering principles, design experience, and tools which were needed to complete the project.

- **Denis Cengu:** Denis is a fourth-year computer systems engineering student with experience in C++, Java and in particular Python. He has previous experience developing a smart device operated with multiple Raspberry Pi's which involved substantial work in embedded systems and hardware integration. He also has experience using Firebase for backend services and real-time data management. His background in embedded systems and hardware integration served useful for close interaction between software and physical components.

- **Douglas Lytle:** Douglas is a fourth-year software engineering student with a good knowledge of software design and experience in various programming languages such as Java, C, and Python, which provided a great background for working on the robot. Douglas also has experience with web development using tools and languages such as HTML/CSS, JavaScript, and PHP.

- **Cholen Premjeyanth:** Cholen is a fourth-year computer systems engineering student with an experienced background in Java, Python, and C. He has previous experience developing a 3D printer monitor which involved a Raspberry Pi, and focused on hardware integration. He also has experience using Arduino in a snow remover based project. His background in both software design and hardware implementation served useful in this project.

4 Background

4.1 Robot Operating System

By Denis Cengu

The Robot Operating System 2 (ROS 2) is a significant redesign of the original ROS, developed to meet the evolving demands of modern robots. Unlike the predecessor, ROS 2 includes critical features such as enhanced security, real-time capabilities, reliability in non-ideal environments and scalability. [1] Built on the Data Distribution Service (DDS) communication protocol, ROS 2 is compatible with diverse hardware platforms, including support for Ubuntu 20.04 that will be run on the Raspberry Pi 4. ROS 2 includes a unified set of core libraries that support multiple programming languages, such as C++, Python and Java. Moreover, ROS 2 is well suited for integration with the iRobot Create 3 and 2, through microcontroller communication, which will be used to enable effective multi-robot communication and

autonomous navigation. ROS 2's modularity and interoperability make it a powerful tool for deployment of autonomous systems.

4.2 iRobot Create

By Denis Cengu

The iRobot Create 3 is the latest version of the iRobot Create series, offering several improvements over its predecessor. It comes with built-in ROS 2 support which means it will not require an external microcontroller to send instructions resulting in a straightforward integration process.[2] The Create 3 is equipped with six IR obstacle sensors to detect objects and follow walls. Additionally, the custom faceplate allows for easy mounting of additional boards. The Create 3 also offers a storage compartment for the Raspberry Pi and additional power options including a 14.4V/2A battery connection and a 5V/3A USB-C port for powering and connecting components. The built in ROS 2 provides a comprehensive set of commands for sensor data collection, wall following and positional navigation.

4.3 LiDAR Sensor

By Cholen Premjeyanth

The LiDAR sensor is integral to the robot's navigation by allowing it to use an accurate mapping of its surroundings. This sensor emits laser pulses that emit off objects and it measures the time it takes for each pulse to bounce back after it hits the objects. This allows for a highly accurate mapping of the robots surroundings. The data the robot receives through this sensor allows it to detect obstacles, navigate complex environments, and avoid collisions. In this project, the LiDAR sensor will be used to enable an accurate 360 degree view of the environment; which is the Carleton tunnels. This sensor can be paired with the Create 3's built in ROS 2 functionality, which will allow the data to be processed in real time, ensuring the robot adheres to safety and performance requirements. Due to the sensor having compatibility with ROS, it allows for it to be easily integrated.

The specific LiDAR sensor being used is the Slamtec RPLIDAR A1 which measures 8000 times per second with a range of 0.15-12m with a resolution of 0.5mm [7]. This sensor, which was recommended from last year has a scanning frequency of 10Hz and a range of up to 12m.

4.4 Raspberry Pi Camera Module

By Douglas Lytle

The Raspberry Pi Camera Module is a new addition to the robot this year. It is used in conjunction with the other sensors to improve the reliability of intersection detection for the robot. This is achieved by taking photos of the ground the robot passes over at regular intervals, and using computer vision to detect the intersection markers in the tunnels. This process is detailed below in section 5.5.

4.5 Gazebo

By Douglas Lytle

Gazebo is a simulation software developed by Open Robotics, the same company which develops ROS. Gazebo features an extensive set of development libraries made to facilitate simulations and testing for robotics projects [6]. ROS is fully integrated with Gazebo, so it is relatively simple to import the control systems for the robot and to construct an extremely accurate simulation. Gazebo even has libraries for the simulations of a wide variety of sensors which can accurately read information from the simulated environment and pass this information to the control system. The exact LiDAR sensor used in the project, as well as the iRobot models used, are represented in these libraries, which will allow for accurate testing. Using Gazebo for simulation will greatly improve the ability to test any modifications to the robot control system, as it will no longer be necessary to be physically present to test changes in many cases.

4.6 Subsumption Architecture

By Cholen Premjeyanth

Subsumption architecture is a type of robot design made by Rodney Brooks [8] that uses a layered control system as an alternative to traditional robot control methods, which relied on a vertical decomposition of functions in a step-by-step processing pipeline. In these traditional systems, sensor data flows through a series of modules, with higher levels controlling the behaviour of the entire robot. In contrast, in subsumption architecture, behaviors operate in independent modules, each running on its own and communicating with other layers only when needed. Brooks illustrates this with a set of diagrams that compare traditional robot control methods (figure 3) to subsumption behavior based layered structure (figure 4).

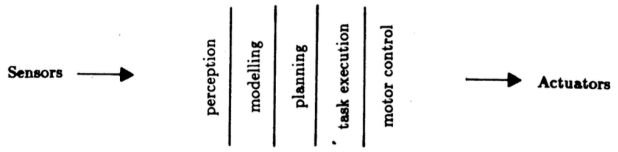


Figure 3: Traditional Robot control methods design [8]

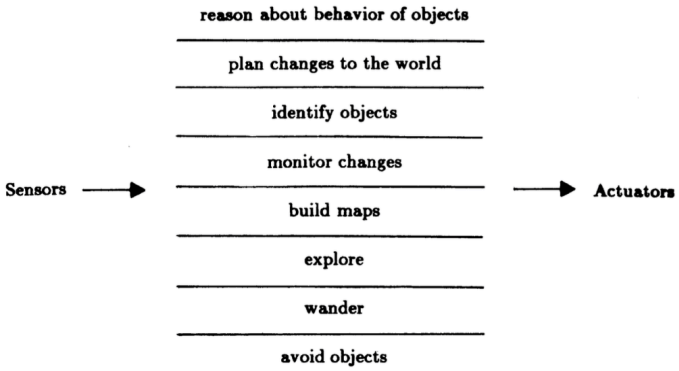


Figure 4: Subsumption Architecture design [8]

Each layer in subsumption architecture has a specific task and a level of behavioral competence, with higher-priority tasks able to override or suppress lower-priority ones when required [8]. This can be done so by using time-based inhibition to temporarily control outputs from other layers. Instead of having one master module making all decisions, subsumption allows each layer to react to sensor data in real time and compete for control when necessary. Modules communicate asynchronously, through message passing, without relying on global memory, which helps keep the system robust [8]. This approach ensures that even if higher-level behaviors fail or don't engage, the lower layers continue to function, allowing the system to operate in unpredictable environments.

In our project, subsumption architecture allows for a structured yet flexible control system. The robot has multiple layers of behaviors, such as basic movement (travel layer), obstacle avoidance (avoidance layer), turning at intersections (turning layer), and docking at its destination (docking layer). Since higher-priority actions (like avoiding obstacles) take precedence over lower-priority tasks (like following a wall), the robot can navigate safely and efficiently. Additionally, new functions can be added without redesigning the entire system, making it scalable for future improvements.

5 Methods

5.1 Replacing the System Architecture

By Douglas Lytle

In order to accomplish the project objective of heavily simplifying the state machine used for navigation, which was a necessary goal for the current and future development of the project, the team transitioned the project away from the state pattern. The benefits and limitations of the state pattern were learned in SYSC 3110 (Software Development Project), and in the context of this project it added more complexity than it alleviated. Thus, the robot control system was modified to use a subsumption architecture, which is extremely suitable for a mobile robot. This made use of principles learned in SYSC 3303 (Real Time and Concurrent Systems).

5.2 Improving Reliability of the Robot Through Simulation

By Cholen Premjeyanth

To enhance the reliability of the autonomous robot system, the team utilized Gazebo for extensive simulation testing. Gazebo is a powerful robotics simulator that allows for realistic 3D environments to be modeled, enabling us to test various navigation algorithms while not putting the robot at risk of damage. By simulating various intersection scenarios, we can analyze the robots behavior in various conditions. Furthermore, by using simulations to test the robot we can analyze the robot more efficiently, rather than consistently trying to set up the same tests

physically. This approach is similar to an approach that was learned in SYSC 3020 (Intro to Software development), which emphasizes the importance of testing and validation within the software development life cycle.

Furthermore, the use of the LiDAR sensor was important to improving the reliability of the intersection detection. We can use sensor fusion even to improve the detection of intersections, as through combining sensor data, the robot will be more accurate. In SYSC 3303 (Real Time and Concurrent Systems) it was learned how to incorporate synchronization for multiple sensor inputs and explored strategies relating to real-time systems and their responses. This was helpful in improving intersection detection. Ultimately, the goal of Gazebo testing was to lead to the development of better algorithms for intersection detection and traversal, ensuring that the robot can navigate complex environments effectively and safely.

5.3 Improving Intersection Detection With Computer Vision
By Douglas Lytle

Intersection detection has remained one of the largest issues for the robot. To improve the reliability of intersection detection, the team has implemented computer vision in order to detect the yellow intersection markers seen at each intersection in the Carleton Tunnels.

Figure 5: Intersection marker in the tunnels (Left), and its yellow mask (Right)

This is implemented by having the camera module continually take photos of the ground the robot passes over. Then, the image is filtered to find any yellow contained in it. If the amount of yellow in the image crosses a certain threshold, then the system can be almost certain it has passed over an intersection marker, and the camera sensor ROS node will publish as such. This was used alongside the LiDAR data to improve intersection detection.

6 Analysis

6.1 Analysis of Project Starting Point

By Douglas Lytle

As this is the fourth year of the Mail Delivery Robot project, there is significant work which has already been completed by past project teams. To begin this year's iteration of the project, the team began by examining in detail the report left behind by last year's project group.

It can be seen that last year at the conclusion of the project, the robot was able to follow walls well, navigate through intersections with reasonable consistency, avoid simple obstacles, follow its internal map to reach its destination, and communicate with the web application. All of this together allowed the robot to complete its first delivery.

6.2 Analyzing the Capabilities of the Equipment

By Denis Cengu, Douglas Lytle, and Cholen Premjeyanth

To better understand the initial capabilities of the robot and equipment, several tests were conducted. The results of these tests can be seen in the following subsections.

6.2.1 Analyzing the LiDAR

By Denis Cengu

The LiDAR was tested in various intersections using RViz to provide a visualization. It was determined to be very reliable in ranges of 3m-5m and still providing useful information upwards of its maximum of 12m. In particular for the scope of the project, the LiDAR is more than capable of providing the robot with sensor data for wall following and intersection detection, with the latter requiring further refinement in its methods. This can be seen in the figures below which depict the robot first coming into an intersection and then roughly 50cm into an intersection. The red lines represent the walls detected by the LiDAR and the blue X represents the robot's location within the visualization.

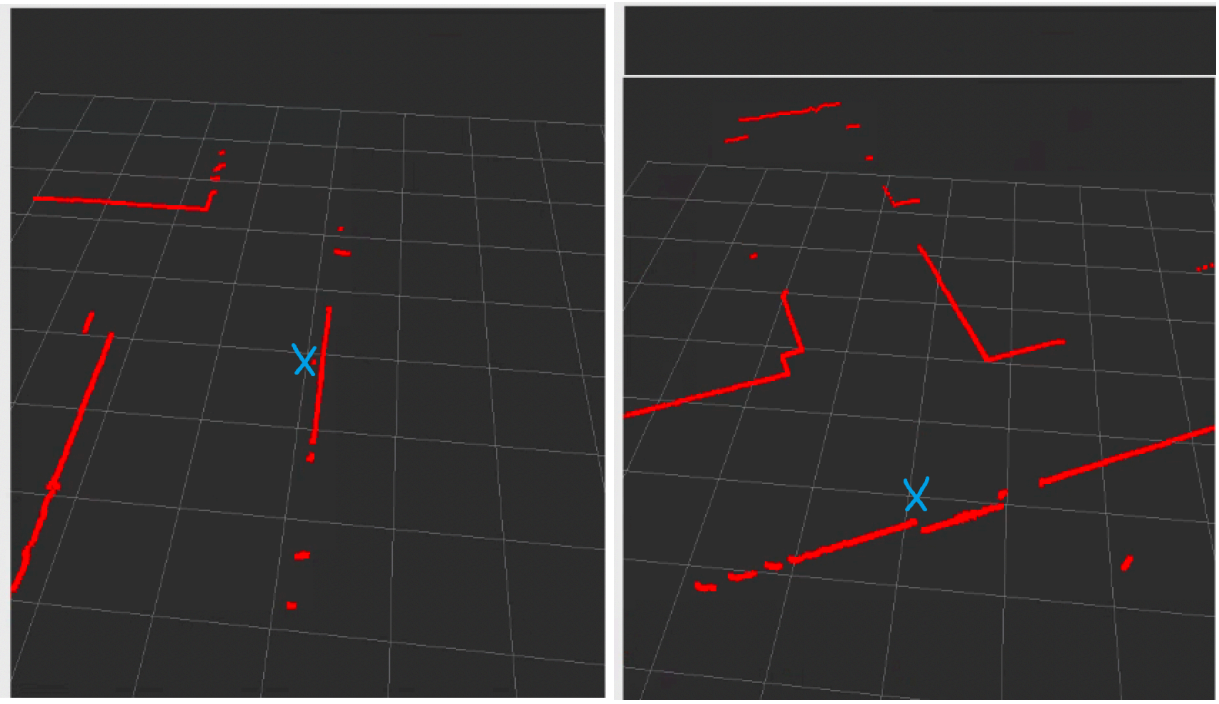


Figure 6: LiDAR Visualization at the Beginning of an intersection (left) and 50cm into an intersection (right)

6.2.2 Analyzing the Beacons

By Denis Cengu

The beacons similarly to last year's team were found to be somewhat unreliable in their readings potentially due to noise, low battery power and distance among other factors. Despite this, the data represents a clear enough trend that should allow the team to further build on last year's method of reading a gradient to determine whether a beacon is being approached or not. The strength of the beacons was measured five times at each distance, and in the following table the averages of those readings can be seen.

Table 9: Measured beacon signal strength at 1 and 10 meters

Beacon Address	**1 Meter Strength**	**10 Meter Strength**
78:46:f6:40:db:5b	-67dB	-84dB
40:83:3d:c7:ec:98	-69dB	-82dB
fb:ef:5c:de:ef:e4	-74dB	-88dB
51:c6:33:c1:41:94	-67dB	-85dB

ee:16:86:9a:c2:a8	-64dB	-82dB
e4:87:91:3d:1e:d7	-73dB	-90dB
df:2b:70:a8:21:90	-72dB	-89dB
ef:08:20:32:8d:a0	-64dB	-91dB

6.2.3 Analyzing the Create 3 Behaviour

By Cholen Premjeyanth

The behaviour implemented by last year's group was analyzed in order to evaluate the effectiveness of the navigation, battery usage, and docking. This ensures the system functions as expected and helps identify areas for enhancement.

6.2.3.1 Analyzing the Odometry Data

By Cholen Premjeyanth

Odometry data was collected to look at the robot's ability to track its position and follow the planned path accurately. Plotting the recorded coordinates, the result confirms that the robot traveled from Mackenzie/Minto to Canal, passing the Nicol Building, following a straight path, then turning right at the intersection, which aligned with the expected tunnel route. The (X,Y) values also remained stable with no change from the path showing that the system's odometry-based movement planning works and does not need much change.

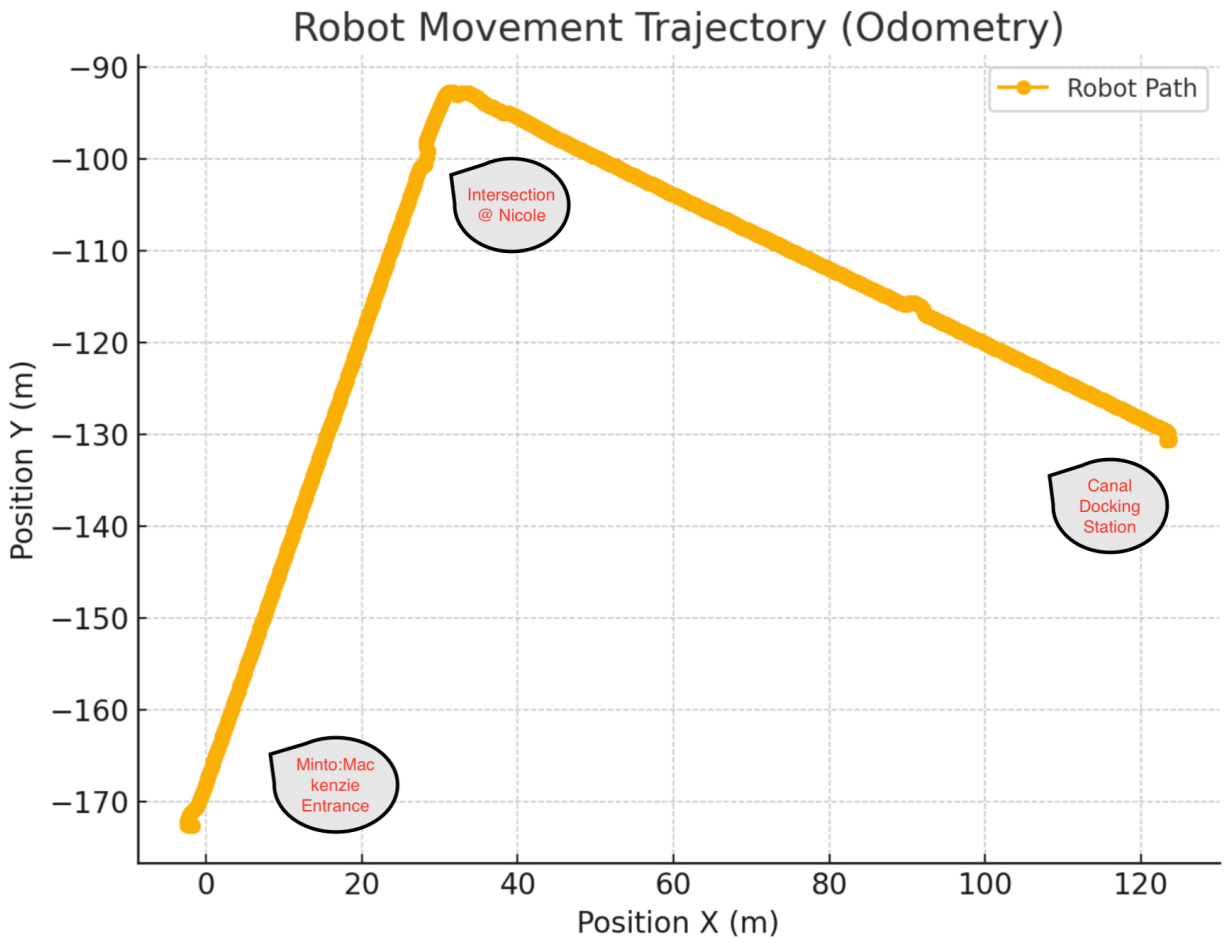


Figure 7: Diagram of the path taken by the robot

The diagram illustrates the path taken by the robot, showcasing the robot going along the designated route. The plotted odometry data was verified using known waypoints, such as intersections, beacon placements, and this confirmed the recorded path accurately represents the tunnel layout.

6.2.3.2 Analyzing the Create 3 Speed

By Cholen Premjeyanth

Speed data was analyzed to evaluate velocity, stability, acceleration patterns, and overall efficiency of the robot. The measurements were taken by tracking the total distance over time and analyzing speed variations across different sections of the route.

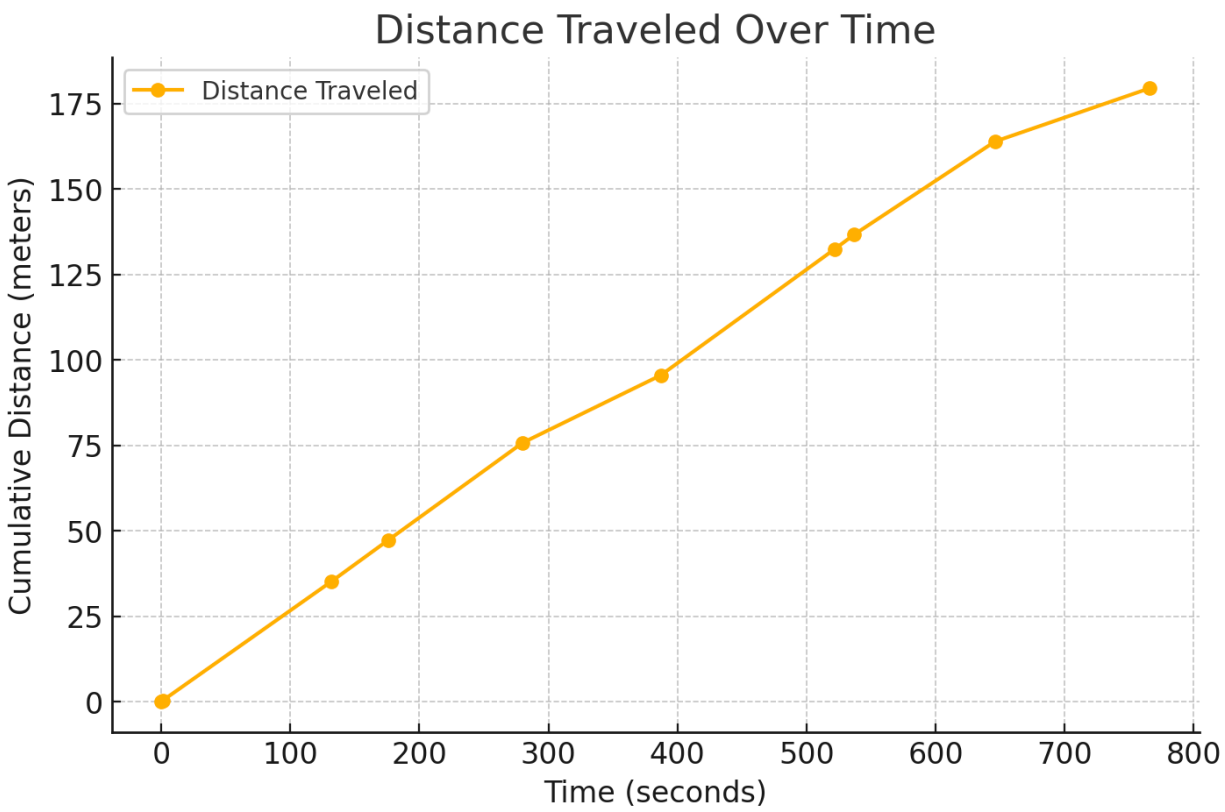


Figure 8: Distance vs Time Graph

The results indicate that the robot maintained consistent motion at an average speed of 0.21 m/s. Velocity was consistent over time, with only minor changes happening at the tunnel doors, where the robot readjusted its wall-following behavior to align with the structure.

6.2.3.3 Analyzing the Create 3 Battery Consumption and Efficiency
By Cholen Premjeyanth

Battery data was analyzed to determine how long a trip takes based on the current energy consumption rate and how many meters the robot can cover before requiring a recharge. The results confirm stable power consumption with predictable patterns, ensuring reliable operation over extended distances. The graphs presented correspond to the Mackenzie/Minto to Canal Run 4, but all conclusions drawn in this section are based on the average values of five test runs, ensuring a representative dataset rather than a single-run outlier.

The battery discharged steadily, decreasing from 26% to approximately 16% over 13.5 minutes of operation, with no sudden voltage drops or irregularities, indicating consistency in battery usage.

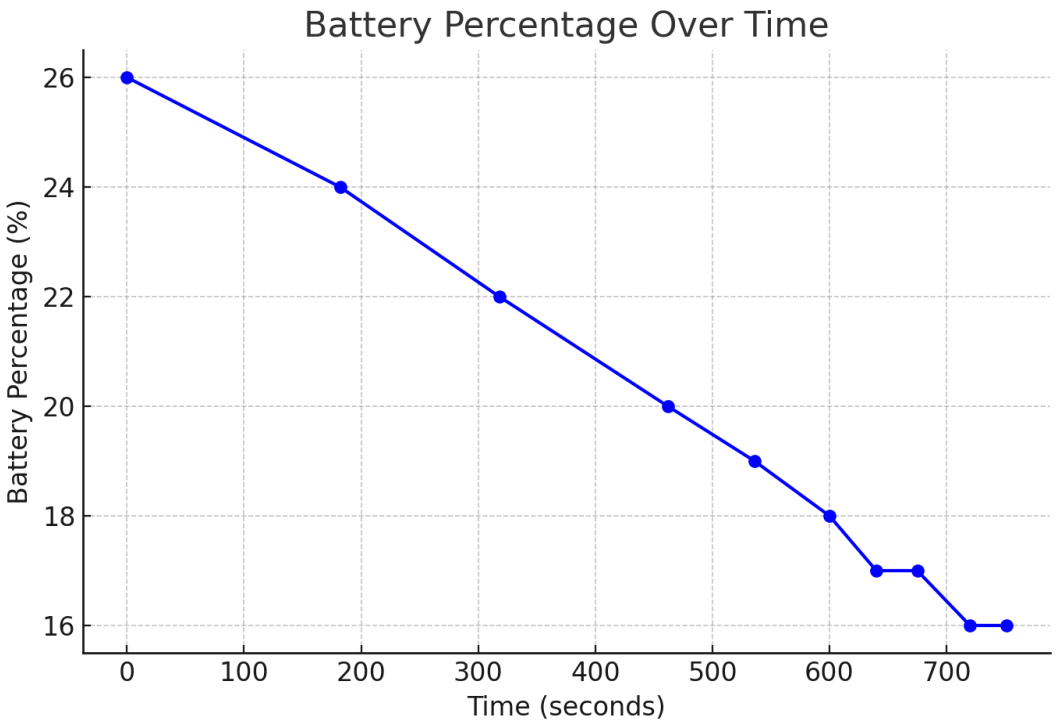


Figure 9: Battery Percentage Over Time

The voltage decreased from approximately 14.2V to 14.0V, with no outlying data, confirming that power regulation and motor efficiency remain stable. The sudden spike in voltage at the end of the run is due to the robot reaching the docking station and beginning the charging process.

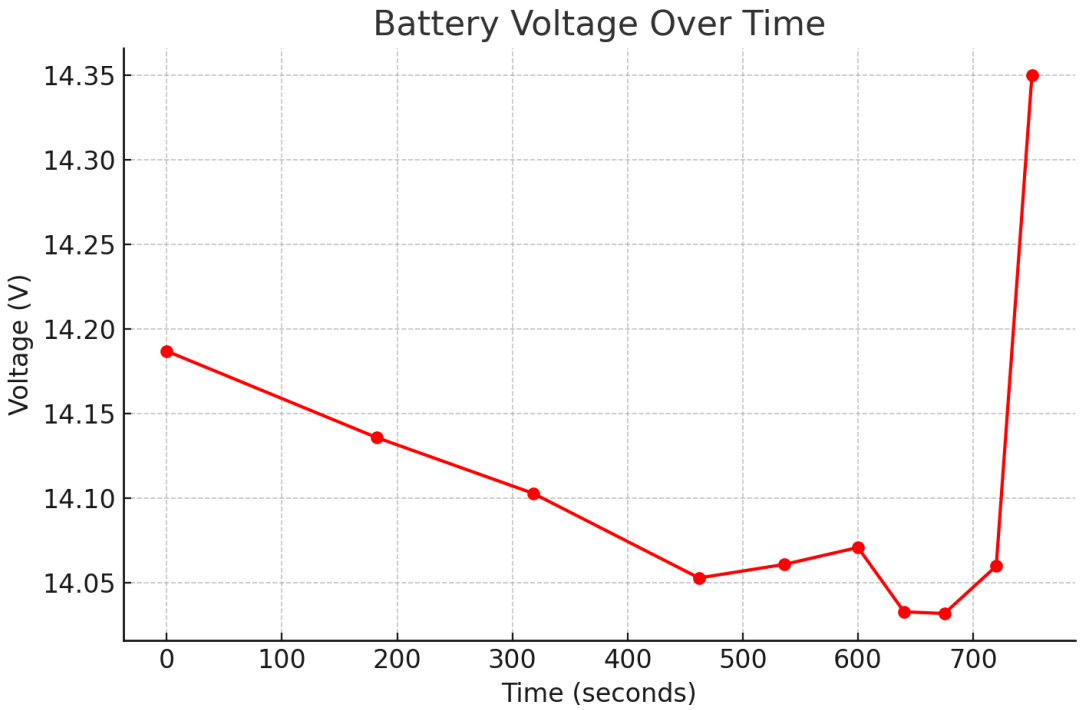


Figure 10: Battery Voltage Over Time

The temperature increased gradually from 33.7°C to 34.2°C, confirming that battery heating remained within safe limits throughout the test.

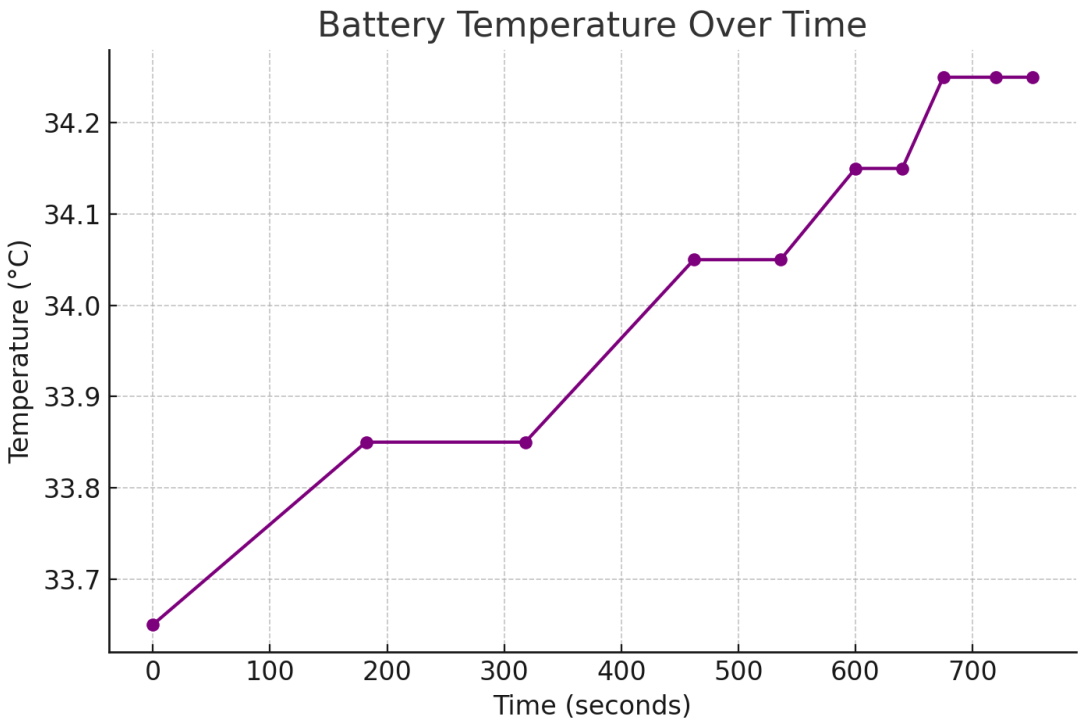


Figure 11: Battery Temperature Over Time

The robot traveled 180 meters consuming 10% of its battery capacity. This shows that a full battery cycle (100%) would allow approximately 1,800 meters of travel before requiring a recharge. However, to prevent complete battery consumption, the system should implement a battery level-based decision-making model. This model, embedded within a layer, would evaluate the remaining charge, distance to the nearest dock, and whether to return, continue, or stop at an intermediate station. This data is helpful for determining optimal docking station placement, as the observed battery depletion rate suggests that docking stations should be positioned at intervals of approximately 800 to 1,000 meters. This distance ensures that the robot can go towards the closest dock for charging before reaching critical power levels.

6.3 Analysis of Previous Codebase

By Cholen Premjeyanth

Before further development on this project, we tested and analyzed the codebase from last year to identify reusable components and further understand the limitations and advantages of the previous system. Much of the earlier implementation was structured around a state machine, which led to increased complexity. After going through the code and testing with the hardware, it became clear that there were a few components that could be directly reused with minor changes. These were mainly the LiDAR, beacon, and bumper sensor nodes. The core logic of the code worked well in testing, so it was carried over with a few minor changes, such as updating naming conventions. The navigation logic from last year was one of the more useful pieces that could be adapted. The logic for reading beacons and determining direction, such as

turning left or right at an intersection, was used and refactored into an individual navigation node that could be used to help other nodes, such as the turning layer, within our system. We also decided against using a lot of the hardcoded logic that was too specific to the state machine. Things like collision handling were re-implemented within their own subsumption layers, where each behavior could be triggered independently. This made the overall system more readable, easier to test, and more adaptable for future upgrades.

6.3.1 Analysis of Last Year's Docking and Undocking Behavior

By Douglas Lytle

It was observed during testing with the code inherited from last year's project that the robot was not actually able to do anything after having automatically docked or undocked itself. This is caused by the usage of system calls within the python code to manually send an action goal to dock or undock the robot. This functions, but causes the program to hang and cease execution after the docking or undocking has occurred. The result of this is that the robot can dock itself after having completed a delivery, but will then become unresponsive and the program must be restarted in order for the robot to resume operation. If a delivery is started while the robot is initially docked, it will undock itself, but then do nothing else.

To fix this issue, the node responsible for docking and undocking the robot must be properly configured as a ROS2 Action Client. This allows the built-in docking and undocking functions to be used while still allowing the robot to arbitrarily begin a delivery while docked or not docked, and allow the robot to complete consecutive deliveries.

7 Testing and Metrics

7.1 Testing Plan

By Denis Cengu

A robust testing plan was employed for the shift to a subsumption model to verify all modules function correctly in various scenarios and that the subsumption framework itself properly handles task prioritization, such as obstacle avoidance preempting navigation.

The testing plan was divided into two stages. Unit tests will be used for individual modules to ensure that each meets their functional requirements. For example, the wall following module is tested in isolation to verify the ability of that individual module. Afterwards, integration tests were used to verify how the modules will work with each other within the subsumption framework. These tests are meant to verify how well modules interact with one another while simulating a real world scenario mainly focusing on proper behavior switching and task execution. Each test was automated where possible, using ROS 2 available tools such as pytest for unit tests. The goal of the testing plan was to create a simple, yet comprehensive test system which simplifies deployment of upgrades and changes.

7.2 Performance Metrics
By Denis Cengu

In addition to verifying functionality, it is imperative to measure how well the robot performs each task, whether it is done efficiently or effectively. Performance metrics play a key role in this evaluation, providing objective and subjective measures of success. The focus was on creating and using factual metrics along with qualitative metrics.

Factual metrics include objective measurements such as task completion time, accuracy in navigation, battery consumption and how often the task is successfully completed. These metrics allow the measurement of efficiency in the system and pinpoint areas which may require improvement.

Qualitative metrics assess less tangible aspects, such as how smoothly the system transitions behaviors, how reliably dynamic environments are handled as well as how well a turn is made, among other things. These qualitative assessments help determine how the system is performing in a real world application, beyond just satisfying technical requirements.

Through a combination of both factual and qualitative performance metrics, it gives a comprehensive understanding of not just whether the system can achieve its goals but how well it achieves them, allowing for diligent identification of areas where improvements can be made.

7.2.1 Specific Metrics
By Douglas Lytle and Denis Cengu

Detailed below are some empirical metrics which were used to evaluate the performance of the system, by comparing the actual performance with these targets. These metrics are designed to be able to be automatically tested and evaluated in Gazebo. The dashboard view which will automatically display an evaluation of system performance against these metrics can be seen below in figure 12.

- **M1**: The variance of the robot's distance from the wall as it is wall following should be minimized.
    - This metric's purpose is to ensure the robot follows as straight of a path as possible as it travels and does not waste a lot of motion.

- **M2**: The robot should detect intersections after having traveled no more than 1 meter into the intersection.
    - This value is selected based on tests done with the LiDAR sensor and PiCam. The camera provides the system with a very reliable way to confirm what the LiDAR sees, through tests performed the system should be reliably detecting intersections quickly due to the intersection markers being detected.

- **M3**: The robot should complete deliveries in an amount of time proportional to $t = d/v$, where t is the time elapsed during delivery, d is the total distance from the starting location of the delivery to the destination, and v is the operating speed of the robot, currently estimated to be around 0.2 m/s.
    - This metric's purpose is to ensure the robot wastes no movement. As much as possible, all movements the robot takes should bring it closer to its destination.

- **M4**: The robot should be able to automatically undock to begin a delivery, and automatically dock itself again after having completed a delivery.
    - These are both required behaviors, so if either docking or undocking fails, this metric will be considered not met.

- **M5:** The number of path corrections the robot has to make during a delivery should be minimized.
    - Tracking this is important to determine any possible errors in our navigation or intersection handling, if we know how often and where we make path corrections we can effectively locate where the errors may be.

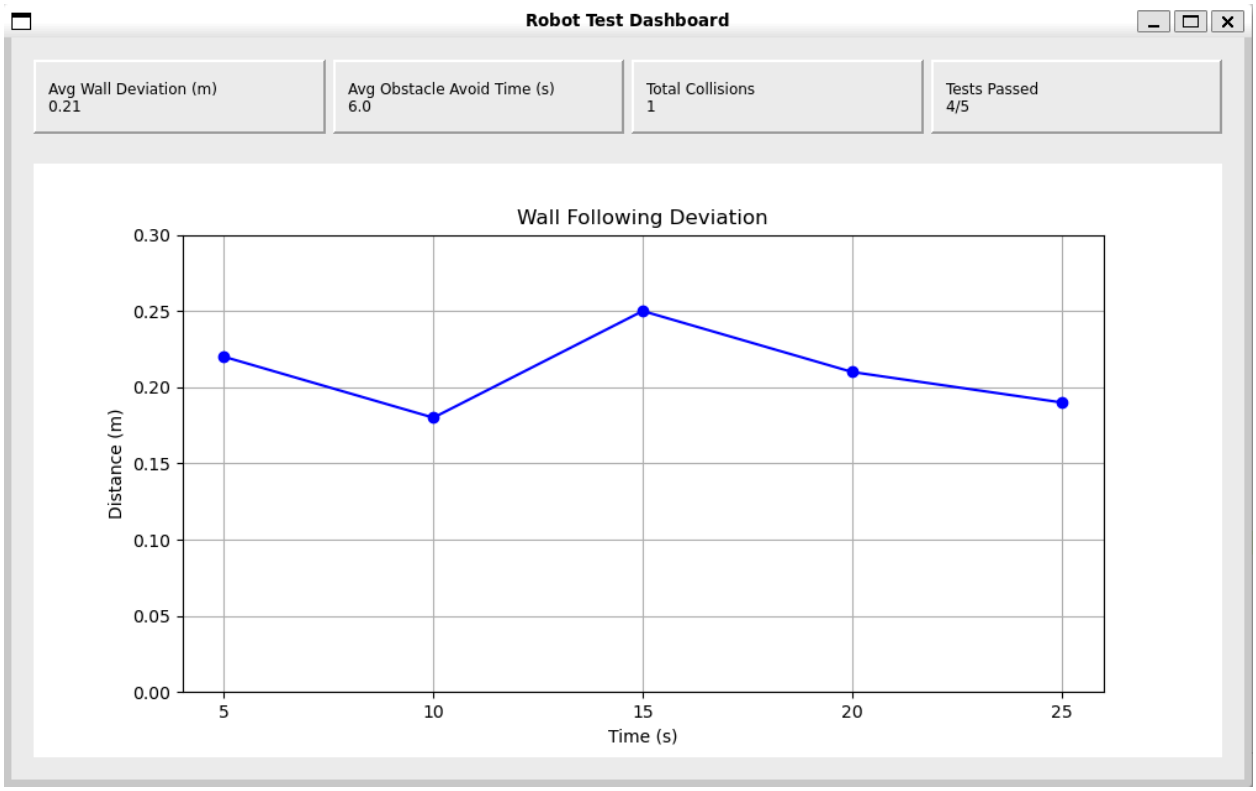


Figure 12: Testing dashboard displaying system performance

7.3 Implementation of Tests

By Denis Cengu

To implement the desired testing design, separate ROS2 packages were created. Tests are structured for each subsumption layer, Avoidance, Docking, Travel and Turning enabling isolated validation of individual functions as well as gathering the specific metrics discussed earlier to pinpoint performance strengths and weaknesses. The tests function similarly to the layers themselves the only key difference being additional sensor data is collected and computed, such as the average deviation from the desired distance from the wall for the Travel layer test among others. To accompany the tests a basic PyQT GUI dashboard was created which will display the test results in a simple and neat manner allowing for easy interpretation and validation of the robot's performance.

As well as implementation of analytical tests, there are automated unit and integration tests using the launch_testing framework in ROS 2. These tests are built and executed using colcon test allowing behavior validation during development. The primary goal of these tests is to ensure each node subscribes and receives the correct data from each topic, processes them correctly and publishes the expected outputs. Each test launches a single ROS 2 node in isolation using launch_ros.actions.Node. Python's Unit test library is then used to publish controlled inputs to the node's subscribed topics, monitor and verify the outputs to the corresponding published topics and assert that the outputs match expected values within a given time frame.

8 Design

8.1 Subsumption Architecture

By Douglas Lytle

The new system architecture which was developed for the robot is a subsumption architecture. A subsumption architecture consists of independent behavioral layers which each receive data from the sensors, and output instructions for the robot. In the system being developed for this project, there are four layers which each have a priority associated with them. There is a central control module (called the captain) which will listen to all instructions, and forward the instructions coming from the layer with the highest priority to the robot for execution.

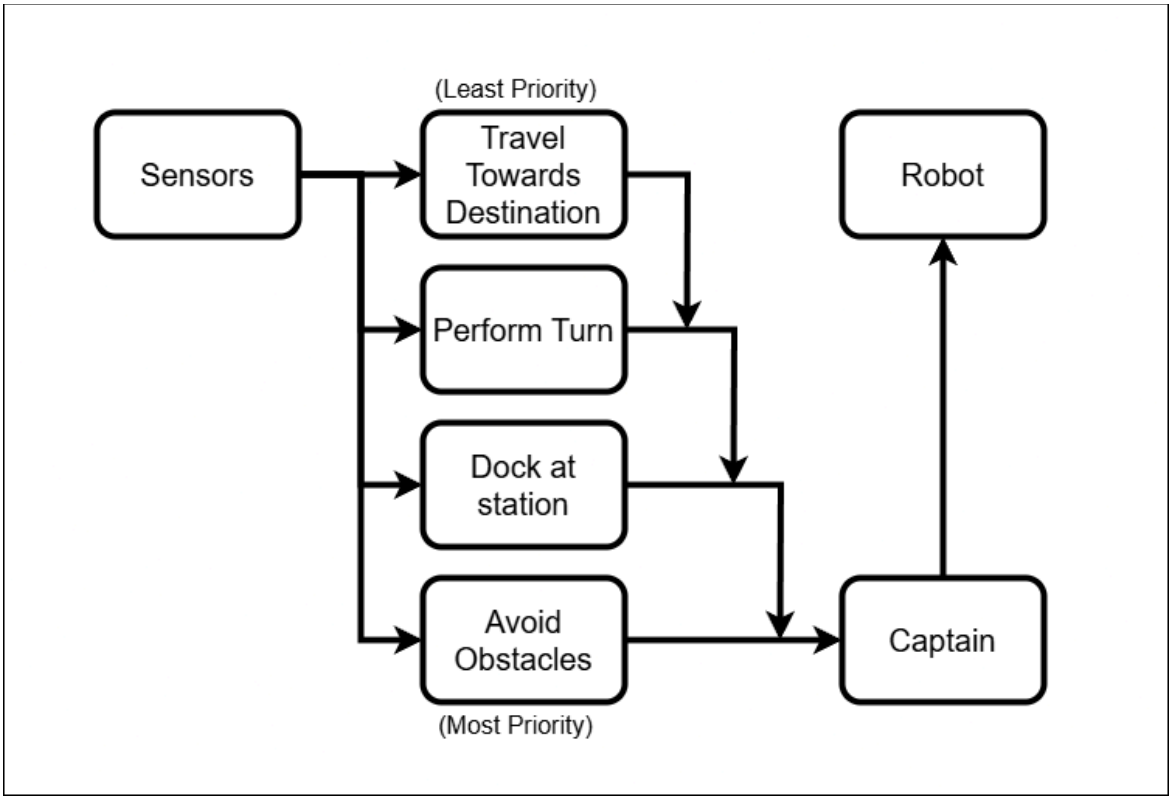


Figure 13: Diagram of the subsumption layers

Seen above is a diagram of the subsumption architecture being developed. The different behavioral layers are explained in detail below. In a typical subsumption system, the layers are built from the lowest level (highest priority) first, thus it makes sense to explain them in this order:

- **Obstacle Avoidance Layer:** This is the lowest layer in the diagram above. It is the highest priority behavior, and thus will override all other behaviors. If the sensors detect that a collision has occurred, this layer will begin sending instructions to the captain until the collision is resolved, at which point the robot will resume listening to other instructions.
- **Docking Layer:** This is the second layer in the diagram above. If the sensors indicate the robot has reached its destination, this layer will begin sending instructions to the captain to dock the robot at the destination's associated docking station. Also, this layer is responsible for undocking the robot at the start of a delivery. This process can still be interrupted if a collision is detected, but will override all other behaviors.
- **Turning Layer:** This is the third layer in the diagram above. If the sensors indicate that the robot has entered an intersection, then this layer will begin sending instructions to the captain on how to execute the intersection traversal. This can be interrupted if either a collision is detected, or it is detected that the robot has reached its destination. This behavior can override only the travel layer.

- **Travel Layer:** This is the top level layer in the diagram above. Being the top level layer, this is the robot's default behavior. This layer is responsible for moving the robot towards its destination by finding a wall to follow and following it in the correct direction. Since this is the top level layer, there are no conditions for the activation of this layer. This behavior can be overridden by any other if their conditions for activation are met.

The main benefit of a subsumption architecture is that it is largely stateless. If, for instance, a collision occurs and a turn through an intersection is interrupted, after the collision is resolved the turn will immediately resume without any memory of what came before the collision. This can happen because the conditions for the turning layer to output instructions are still met. In fact, in this case it would never have stopped outputting instructions on how to execute the turn, the captain just prioritized obstacle avoidance until that was no longer necessary.

8.2 Deployment Diagram
By Douglas Lytle

Seen below is the deployment diagram for the system. Each artifact seen in the mail_delivery_robot node in the diagram represents a ROS node, and the communication paths between them are labeled with the name of the ROS topic they communicate through.

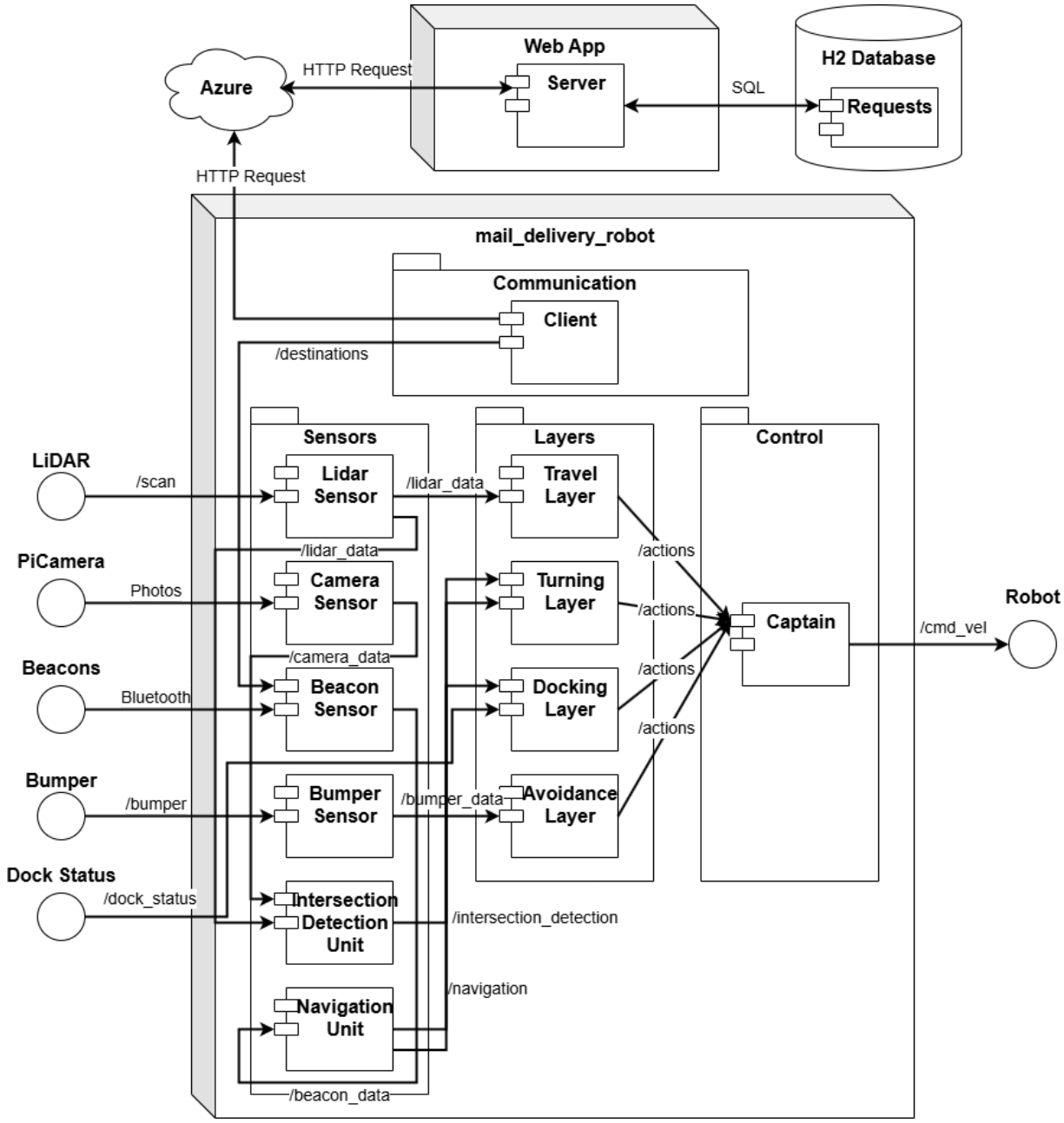


Figure 14: Deployment Diagram

As can be seen in the deployment diagram, there are five sources of sensor data: the LiDAR, the PiCamera, the beacons, the bumper, and the dock status. The data they collect is passed to the appropriate ROS nodes, seen in the package called Sensors. These nodes will then perform any necessary processing before passing their output to the next set of nodes, representing the layers of the subsumption model. Then, these publish their recommendations for actions along with their associated priority, to be read by the captain. The captain will make a decision about which actions to perform, and translate these into actual commands for the robot. The client node, seen at the top of the diagram, is responsible for communication with the web app and passing requests forward to the rest of the system.

8.3 Inclusion of the Navigation and Intersection Detection Units

By Douglas Lytle

In an effort to decouple navigation and intersection detection from the turning layer, it was decided that two additional sensor nodes would be required, bringing the total up to six. These are the navigation unit, and the intersection detection unit. The navigation unit takes input from the beacon sensor, and makes use of the navigation map developed last year to produce navigation messages. These messages are published to a topic called /navigation, which is used by the turning layer to decide what to do at an intersection. These navigation messages are also used by the docking layer, so that instead of reading beacons, it can simply dock as soon as a dock is seen, provided it has received a 'DOCK' message from navigation.

The intersection detection unit takes input from the LiDAR and camera sensors to determine if the robot is currently in an intersection. The messages published by this unit are listened to by the turning layer, which simplifies the processing in that node. Overall, this change has greatly reduced unnecessary coupling in the various nodes, and also increased the cohesion of the code units.

8.3.1 Design of the Navigation Unit

By Douglas Lytle

The navigation unit makes use of the navigation map developed by last year's team, which, when given a destination, an intersection, and the direction the intersection is being approached from gives the action which must be taken at that intersection. These actions are either a left turn, right turn, intersection pass, dock, or u-turn.

Every time a beacon is encountered and that beacon is not the same as the previous beacon which was encountered, the navigation layer will publish a single message to the /navigation topic, containing an instruction. If the instruction is to u-turn (which usually indicates something went wrong), it is executed immediately by the turning layer so as to not waste any time. If the instruction is to dock, it is executed by the docking layer as soon as a dock is seen by the robot. Otherwise, the instruction will be executed by the turning layer the next time an intersection is entered, as indicated by the intersection detection unit.

8.3.2 Design of the Intersection Detection Unit

By Douglas Lytle

The intersection detection unit makes use of data from the LiDAR and camera sensors to determine if the robot is currently in an intersection. The intersection detection unit requires both the LiDAR sensor and the camera sensor to indicate that the robot is in an intersection before the robot will be considered to be in an intersection. On a timer, a message is published to the /intersection_detection topic indicating whether or not the robot is currently in an intersection.

In addition to the camera data being considered, a number of changes were made to the way LiDAR data is interpreted in order to improve the reliability of intersection detection

compared to last year. In order to explain these changes, it is important to understand what exactly the LiDAR sensor is measuring.

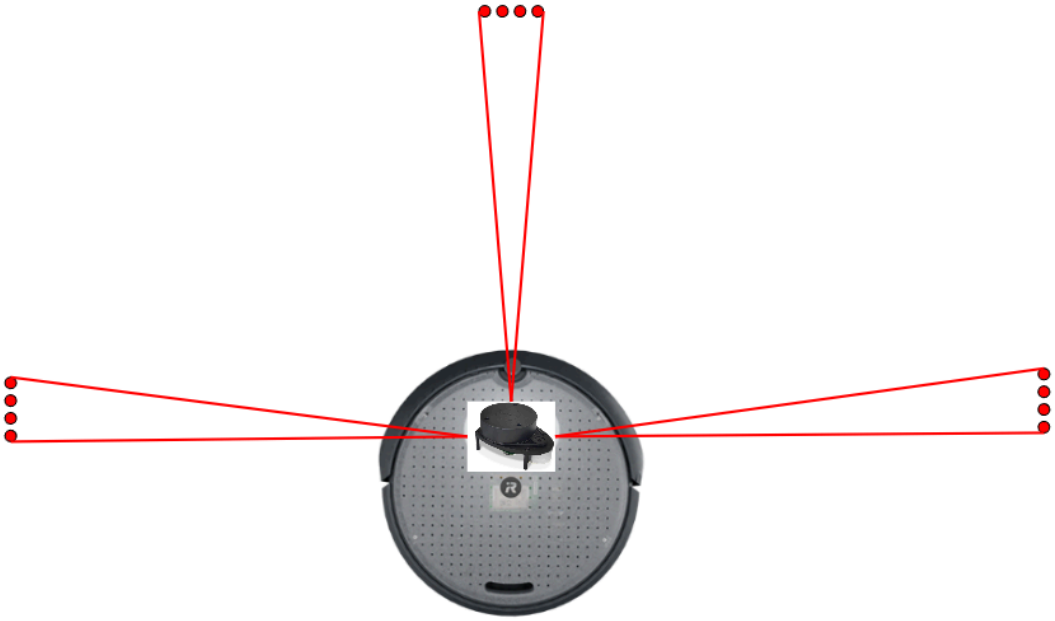


Figure 15: Visualisation of Measured LiDAR Readings

Unchanged from last year, the LiDAR sensor reads a cluster of points in three regions around the robot: to its left, right, and in front. On a timer, the sensor will publish a message containing the following:

- The smallest distance to any of the measured points
- The angle of the robot relative to the closest point
- The smallest distance to any of the measured points in the right side cluster
- The smallest distance to any of the measured points in the left side cluster
- The smallest distance to any of the measured points in the frontal cluster

However, if the sensor cannot get a reading in one of the regions, or if the reading is giving a distance greater than a predefined threshold, or if there is a high degree of variance among measured distances to points within a cluster, then that side will report that the wall has been lost. With this in mind, the methods used for intersection detection last year and in the current year (only considering LiDAR data) are compared below in Table 10.

Table 10. Comparison of Intersection Detection methods using LiDAR data

Old	**New**
Lost Wall Thresholds	
• Right: 2 meters	• Right: 4 meters

• Front: 4 meters • Left: 10 meters NOTE: It is unclear why the thresholds are so dramatically different between the different sides of the robot. Initial testing was done with these thresholds and they produced very inconsistent results.	• Front: 4 meters • Left: 4 meters NOTE: The value of 4 meters is chosen based on the approximate size of the Carleton tunnels.
Lost Wall Requirements	
The robot is considered to be in an intersection if: • Front wall is lost, AND • Left OR Right wall is lost NOTE: It is unclear why the front wall must be lost in order to be 'in an intersection', since this does not account for 'T' intersections. Initial testing was done using this method and it produced (expectedly) inconsistent results.	The intersection detection unit considers the LiDAR sensor to indicate an intersection if: • Any two of the three walls are lost

In addition to improving the way LiDAR data is used for intersection detection, camera data is also used as a second factor for further improvement. As discussed earlier, whenever an intersection marker is crossed, the camera sensor node will publish a message indicating as such. When the intersection detection unit receives this message, it will take note of the robot's current position with the help of the built-in odometry functions available. Then, the intersection detection unit considers the camera to indicate that the robot is in an intersection until the current position of the robot reaches a certain distance away from where the intersection marker was observed. The distance the robot is allowed to travel before this happens was determined experimentally based on the largest intersections which could be found in the tunnels.

To summarize, the robot is considered to be in an intersection if two conditions are met: The LiDAR must be reporting that at least two out of the front, right, and left walls are lost, and the robot must not have travelled too far since the last intersection marker was crossed.

8.4 Design of ROS components
By Douglas Lytle

Seen below is a diagram depicting the main ROS nodes and topics in the system. As much as possible, the sensor nodes should be responsible only for reading data from their respective sensors, and transforming that data into a format which can be easily transmitted and read by the subsumption layer nodes. The layer nodes should be responsible only for processing incoming data from the sensor nodes, and outputting actions when needed. The captain should be responsible only for reading incoming actions, and then translating into robot commands the action with the highest priority before sending those forward.

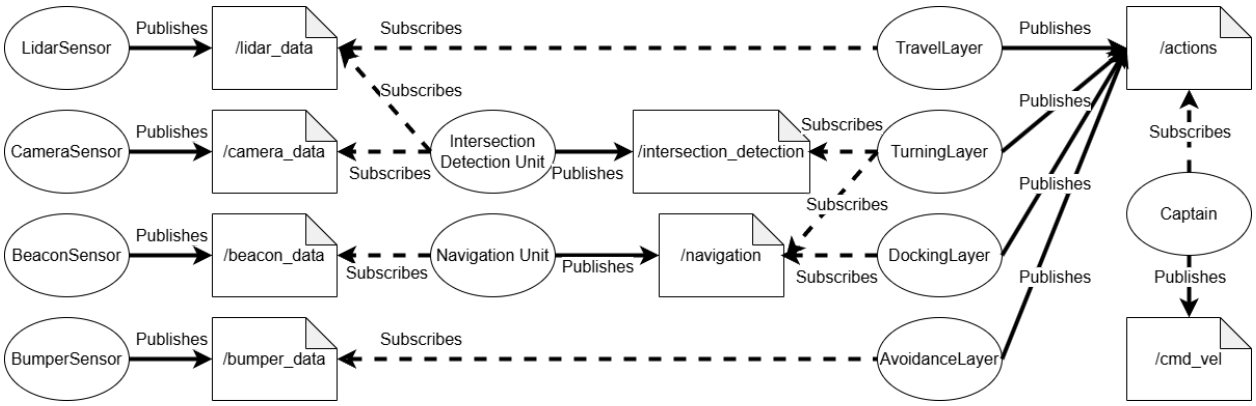


Figure 16: Main ROS Nodes and Topics

8.5 Inclusion of Internal Finite State Machines

By Douglas Lytle

It has been determined after implementation began in earnest that some notion of state is necessary for the system to be effective. The team wants to avoid an extremely large and complicated state machine which models the entire robot, and believes in a much more effective and easy to understand method to handle this which combines the benefits of the subsumption architecture with last year's state pattern based approach.

Each of the subsumption layer nodes now contains an internal finite state machine, and as such the actions they publish to the captain are a function of not just their most recently received sensor data, but also their current state. This allows for more complex behaviors composed of a series of actions to be output by the layers without compromising the key benefit of the subsumption architecture, which is the ability to switch between and combine elements of these behaviors at will to produce a more reactive and "intelligent" robot.

8.6 Determination of Internal State Machines

By Douglas Lytle

In order to determine the necessity of state machines for the various layers and to define the states that will be modeled, in this section all possible inputs which could elicit a response from the four layers of the system will be considered, along with their corresponding possible outputs. It should be noted that a certain level of abstraction will be applied here to make a more comprehensive analysis possible. For instance, every possible LiDAR reading will not be considered, but higher level interpretations of the sensor data (e.g. LiDAR indicates an intersection has been entered) will be considered. This is sufficient for this analysis, since the processing and interpretation of the sensor data is a separate concern from the behavioral

control of the subsumption layers. It should also be noted that inputs or combinations of inputs which do not produce an output are not necessarily considered here.

Table 11: Inputs and Outputs of the Avoidance Layer

Avoidance Layer		
Input ***(Source)***	**Possible Outputs** ***(Dependencies)***	**Response to Input Is State-Based?**
Bumper data = Bump_Event.PRESSED.value *(/bumper_data)*	• Wait for 5 seconds *(Less than 3 bump events in the past 45 seconds)* • Begin obstacle avoidance subroutine *(3 or more bump events in the past 45 seconds)*	Yes
Bumper data != Bump_Event.PRESSED.value *(/bumper_data)*	• No output *(obstacle avoidance subroutine is not currently being executed)* • Continue Waiting *(wait is currently being executed)* • Continue obstacle avoidance subroutine *(obstacle avoidance subroutine is currently being executed)*	Yes

Table 12: Inputs and Outputs of the Docking Layer

Docking Layer		
Input ***(Source)***	**Possible Outputs** ***(Dependencies)***	**Response to Input Is State-Based?**
An UNDOCK navigation message is received *(/navigation)*	• No output *(/dock_status indicates robot is not docked)* • Undock the robot *(/dock_status indicates robot is docked)*	Yes
The current destination is reached *(/navigation, /dock_status)*	• Dock the robot *(none)*	No

Table 13: Inputs and Outputs of the Turning Layer

Turning Layer		
Input ***(Source)***	**Possible Outputs** ***(Dependencies)***	**Response to Input Is State-Based?**
Intersection entered *(/intersection_detection)*	• Execute left turn *(last_nav_message = 'LEFT_TURN')* • Execute right turn *(last_nav_message = 'RIGHT_TURN')* • Execute intersection pass *(last_nav_message = 'STRAIGHT')* • No output *(last_nav_message = anything else)*	Yes
'U_TURN' message received *(/navigation)*	• Execute U-Turn *(none)*	No

Table 14: Inputs and Outputs of the Travel Layer

Travel Layer		
Input ***(Source)***	**Possible Outputs** ***(Dependencies)***	**Response to Input Is State-Based?**
LiDAR data is received *(lidar_data)*	• No output *(the robot has no current destination)* • Find and follow the nearest wall forwards *(the robot has a destination)*	Yes

8.7 Design of Internal Finite State Machines and Behavioral Flows
By Douglas Lytle

Explanations of the designs of the subsumption layers' internal finite state machines can be seen in the following subsections. Each of the state machines has only a few states, which helped to reduce the complexity of the project. In addition to these state machines, since not all of the robot's behaviors are state-based, some UML activity diagrams have been included to further explain the behaviors of the layers.

8.7.1 Avoidance Layer State Machine and Behavior
By Douglas Lytle

As can be seen in Figure 17 below, the state machine for the avoidance layer consists of just two states. The current implementation begins in the NO_COLLISION state, which represents normal operation for the robot. Whenever the layer enters this state, it publishes a NONE action to the /actions topic. If a collision is detected, the layer moves into the COLLISION state, which causes the layer to begin publishing instructions on how to resolve the collision to /actions. Once the collision resolution process has completed, the layer moves back into the NO_COLLISION state and another NONE action is sent.

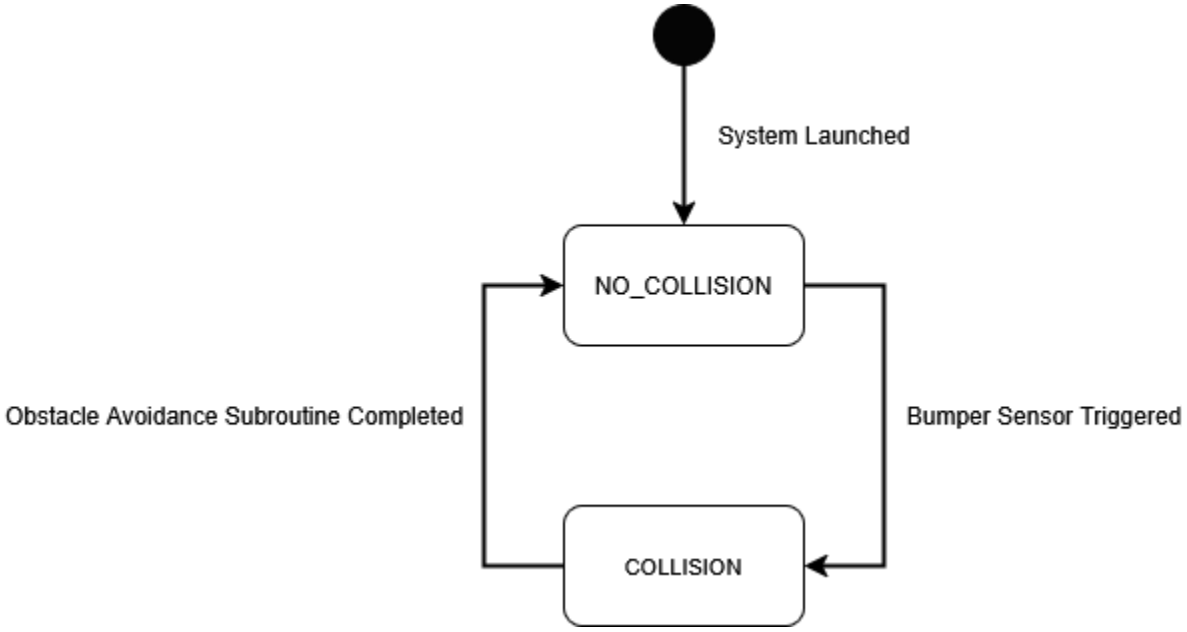


Figure 17: Avoidance Layer State Diagram

When the layer enters into the COLLISION state, two different things can happen. Usually, the robot will simply wait for a period of time, since the team believes that the safest way for the robot to resolve a collision with a student walking through the tunnels is to simply stop moving, as they are likely to go around the robot themselves. However, in the case of collisions with stationary objects, this will not be sufficient, and the robot will need to intelligently navigate around these obstacles. In order to handle these two types of collisions without any

information on the nature of the object collided with, the robot keeps track of the number of times it has collided with an obstacle recently. If the number of collisions is greater than a predefined threshold, then the robot will attempt to navigate around the obstacle instead of simply waiting. A UML activity diagram explaining this behavior can be seen below in Figure 18.

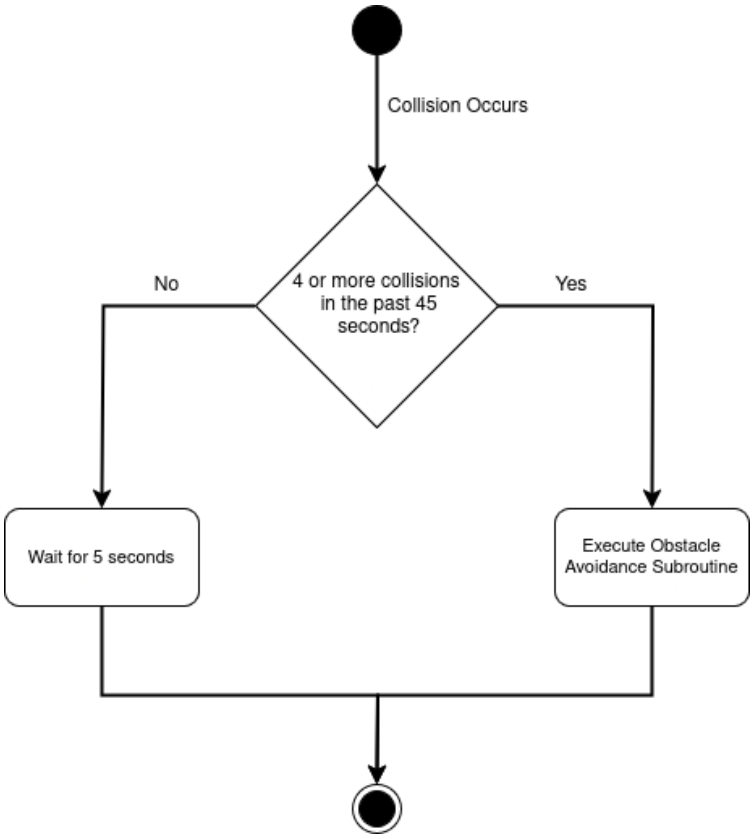


Figure 18: Avoidance Layer Activity Diagram

8.7.2 Docking Layer State Machine
By Douglas Lytle

As can be seen below in Figure 19, the docking layer's state machine consists of two states. Whenever the system is launched, the robot should not do anything (except resolve collisions if they occur) until a destination is received, thus this layer and the two other remaining layers begin in a state called NO_DEST. When a destination is received, if the robot is docked, this layer will send commands to /actions to undock the robot. After undocking, this layer will move into the HAS_DEST state, where it will remain until the robot reaches its destination. Alternatively, if the robot is not docked and a destination is received, the layer moves directly into HAS_DEST without sending additional instructions. While in the HAS_DEST state, this layer does not output anything to /actions, but will publish a single NONE action when the transition occurs. Once the sensor data indicates the robot has reached its destination, this layer will send commands to /actions to dock the robot and finish its delivery, at which point the NO_DEST state will be entered once more and the robot is prepared to make another delivery.

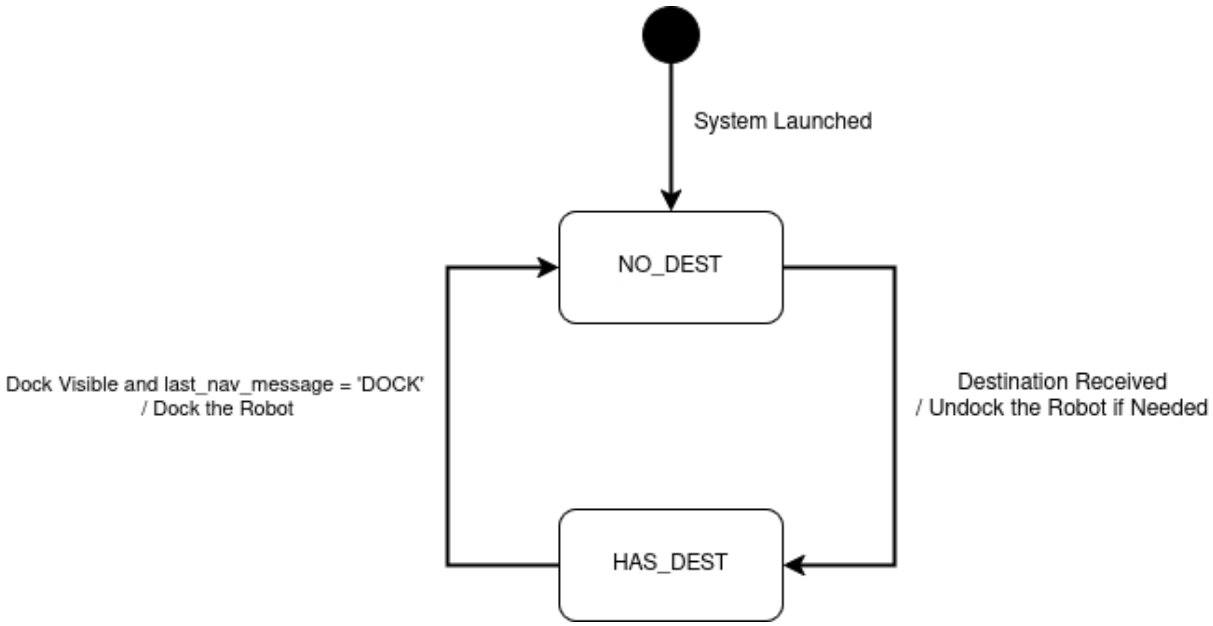


Figure 19: Docking Layer State Diagram

8.7.3 Turning Layer Behavior

By Douglas Lytle and Cholen Premjeyanth

The behavior of the turning layer is to handle navigation decisions when the robot encounters intersections along its path. As illustrated in Figure 20, this layer operates based on incoming navigation messages from the navigation unit and detects intersections using input from the intersection detection unit. Once the system is launched, the robot begins in an IDLE state. When a navigation message (stored as last_nav_msg) is received the system first checks if the message indicates a u-turn. If yes, the layer publishes the action 2:U_TURN to the /actions topic, which indicates a u-turn being required, and returns to an idle state after the action is completed. If the message is not a u-turn, and the robot detects an intersection the system evaluates the last_nav_msg and determines if the robot should turn left, right, or continue straight. Based on what is required, the layer will publish the actions 2:LEFT (for left turn), 2:RIGHT (for right turn), or 2:PASS (for straight). After the action is published and complete, the layer returns to an IDLE state, ready to handle any further intersections or navigation instructions.

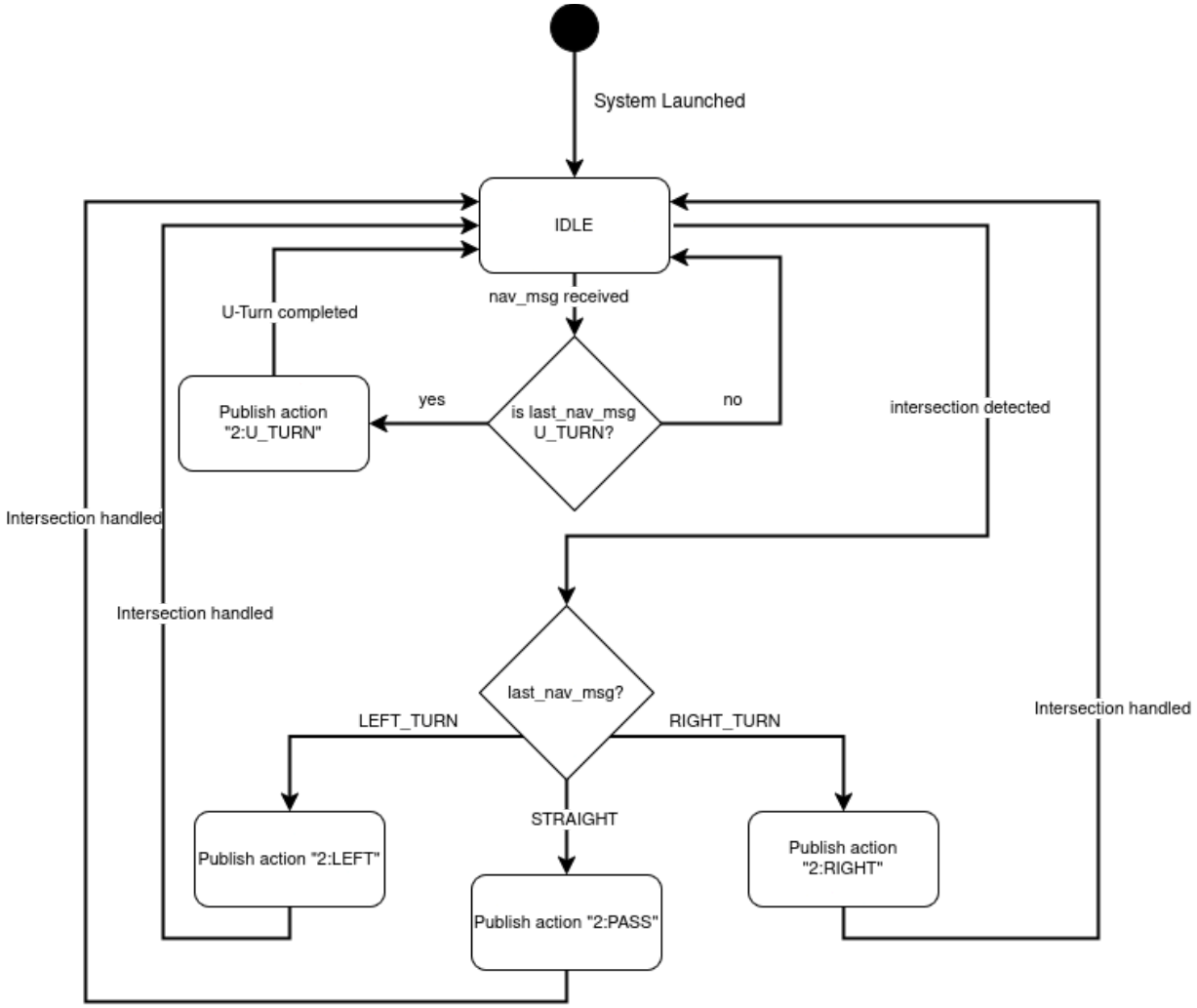


Figure 20: Turning Layer Activity Diagram

8.7.4 Travel Layer State Machine

By Douglas Lytle

As seen in Figure 21 below, the travel layer's state machine has two states. As mentioned earlier, when the system is launched, the robot should not do anything except resolve collisions if necessary until a destination is received. When a destination is received by the robot, this layer will transition into the HAS_DEST state, in which this layer publishes instructions to /actions to find and follow the nearest wall forwards. When the robot reaches its destination, this layer will transition back to the NO_DEST state and the robot will wait for further instructions.

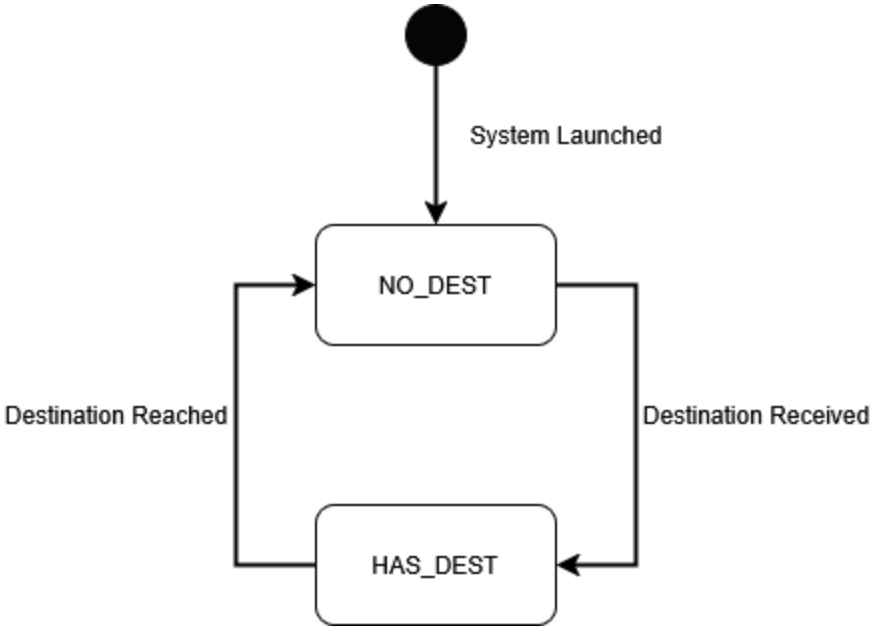


Figure 21: Travel Layer State Diagram

8.8 Design of the Captain Node
By Douglas Lytle

The captain node is responsible for reading the actions sent forward by the subsumption layer nodes. On a timer, it will determine which pending action has the highest priority and translate that action into concrete commands for the robot. The only exception to this is in the case of the highest priority action being a DOCK on UNDOCK command. To execute these commands, instead of using Twist objects sent to /cmd_vel, the captain is configured as an ROS2 Action Client. This allows the captain to send an Action Goal to the corresponding Action Server for docking and undocking, which will make use of the robot's built-in docking and undocking functionality.

8.9 List of Nodes and Topics
By Douglas Lytle

Below is a list of ROS nodes and topics used by the system. The topics are separated into two categories: those which are defined by default in ROS but are used by the robot, and those which are custom topics created only at runtime. The nodes are organized by package. Below this list in Table 15, it can be seen which topics are published to and subscribed to by each node.

Nodes:

- Sensors:
    - **lidar_sensor**: Node responsible for interpreting lidar scans.
    - **beacon_sensor**: Node responsible for reading signals from the Bluetooth beacons.

    - **bumper_sensor**: Node responsible for interpreting hazards.
    - **camera_sensor**: Node responsible for interpreting camera data.
    - **navigation_unit**: Node responsible for providing navigation instructions.
    - **intersection_detection_unit**: Node responsible for determining if the robot is in an intersection.
    - **battery_monitor:** Node responsible for monitoring the battery level of the robot.
- Layers:
    - **travel_layer**: Node responsible for wall following behavior.
    - **turning_layer**: Node responsible for intersection handling and u-turn behavior.
    - **docking_layer**: Node responsible for docking and undocking behavior.
    - **avoidance_layer**: Node responsible for collision handling behavior.
- Control:
    - **captain**: Node responsible for determining the robot's next action among those published by the subsumption layers.
- Communication:
    - **music_player**: Node responsible for playing a song so that bystanders are aware of the robot's presence.
    - **client**: Node responsible for communication with the web app. (not implemented)

Topics:

- Default ROS topics used by the system:
    - **/cmd_vel**: Messages sent here must contain a linear and angular velocity, and the robot will move accordingly. This is the main method of controlling the robot.
    - **/cmd_audio**: Messages sent here must contain a sequence of notes made up of a frequency and duration, the robot will play those notes through built-in speakers.
    - **/odom**: Contains a stream of odometry data coming from the robot.
    - **/dock_status**: Contains a stream of information about whether the robot is currently docked and if a dock is visible to the robot.
    - **/hazard_detection**: Contains a stream of data about hazards detected by the robot (bumper, cliff sensor, kidnap sensor, etc).
    - **/battery_state**: Contains a stream of information about the current status of the robot's battery (charge level, voltage, temperature, etc)
- Custom-defined ROS topics:
    - **/scan**: Contains a stream of raw LiDAR data. Created by the sllidar_ros2 package which is a dependency of this project.
    - **/actions**: Subsumption layer nodes publish their actions here for the captain to read.
    - **/destinations**: The robot's trips should be published here. Currently this is done manually but eventually should be integrated with the web app.
    - **/lidar_data**: Contains interpreted LiDAR data.
    - **/beacon_data**: Contains data about Bluetooth beacons encountered by the robot.
    - **/bumper_data**: Contains interpreted bumper readings from the robot.

- **/camera_data**: Contains interpreted camera data.
- **/navigation**: Contains navigation instructions.
- **/intersection_detection**: Contains data about whether or not the robot is in an intersection.

Table 15: Nodes, Subscriptions, and Publications

Node	Subscribes	Publishes
lidar_sensor	• /scan	• /lidar_data
beacon_sensor	None	• /beacon_data
bumper_sensor	• /hazard_detection	• /bumper_data
camera_sensor	None	• /camera_data
navigation_unit	• /destinations • /beacon_data	• /navigation
intersection_detection_unit	• /lidar_data • /camera_data • /odom	• /intersection_detection
battery_monitor	• /battery_state • /beacon_data • /destinations	• /destinations
travel_layer	• /lidar_data • /destinations • /dock_status	• /actions
turning_layer	• /navigation • /intersection_detection	• /actions
docking_layer	• /navigation • /dock_status	• /actions
avoidance_layer	• /bumper_data	• /actions
captain	• /actions	• /cmd_vel
music_player	None	• /cmd_audio

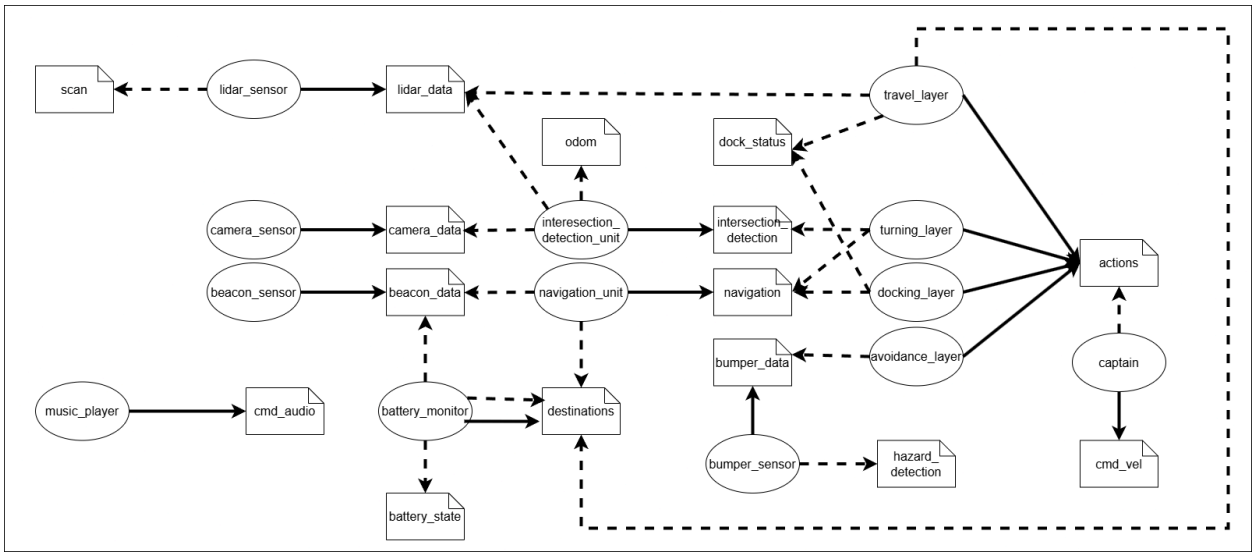


Figure 22. All Nodes and Topics Used by the System

9 Implementation

9.1 Implementation of the Physical Sensor Nodes

By Douglas Lytle

All six of the sensor nodes responsible for interpreting data from the physical sensors have been implemented and are functional for the system. As the LiDAR sensor, beacon sensor, and bumper sensor nodes have been modified only trivially from last year, mostly just changing the names of some things to fit with this year's established naming conventions, their implementations will not be discussed in detail here.

9.1.1 Implementation of the Camera Sensor Node

By Douglas Lytle

The camera sensor node is implemented as follows: on a timer, a command is sent to the attached PiCamera to take a photo. Then, the image is loaded and converted into HSV color space before it is determined what ratio of pixels in the image can be considered yellow. If the ratio crosses a specified threshold, then the node publishes a message to /camera_data indicating that the robot has crossed an intersection marker. If the photo taken does not cross the specified threshold, a message indicating such will also be published. Currently, the timer fires every 0.2 seconds, the range of values considered to be yellow is HSV(10, 50, 50) to HSV(45, 255, 255), and the required threshold ratio is 10%. These values can be adjusted through the config file to account for different lighting conditions or other factors.

9.1.2 Implementation of the Battery Sensor Node

By Cholen Premjeyanth

In addition to this a battery monitor sensor node has been implemented to support automatic docking when battery levels are low. This node subscribes to /battery_status published by the robots battery management system, where it receives periodic updates on the current battery level. On a timer, it checks whether the battery level has dropped below a specific threshold. If so, it publishes a new destination to the /destinations topic, setting the robot up to go to the nearest available docking station, while storing the destination the robot was going towards and only republishing that when the robot has reached the higher threshold, which would indicate to the robot to continue its original trip. The nearest docking station is determined based on the last beacon detected. A configuration file is used to map each beacon location to its closest docking station. When the battery drops below the threshold, the system queries this mapping using the most recently detected beacon and selects the corresponding docking station as the new destination. The threshold values were selected based on experimental analysis of the robot's battery consumption over time. The lower threshold of 15% was chosen after determining that it provides approximately 20 minutes of operational time, which was sufficient for the robot to navigate to the nearest docking station.

```
"""
IF battery_level <= LOW_BATTERY_THRESHOLD AND low_battery_triggered is False THEN
    Set low_battery_triggered to True
    Save current destination
    Lookup dock based on last detected beacon
    IF dock is found THEN
        Publish dock as new destination
    ELSE
        Log warning: "No valid beacon found"

ELSE IF low_battery_triggered is True AND battery_level >= HIGH_BATTERY_THRESHOLD THEN
    IF saved destination exists THEN
        Publish saved destination
        Clear saved destination
    ENDIF
    Set low_battery_triggered to False
ENDIF
"""
```

Figure 23: Pseudocode for Battery Monitor Node

This pseudocode summarizes the basic behavior of the Battery Monitor Node. It depicts what the robot will do, depending on which threshold is reached.

9.2 Implementation of the Subsumption Layer Nodes

By Cholen Premjeyanth

The subsumption layer nodes were implemented using a modular design, where each layer responds to sensor input and sends actions along with their associated priority.

9.2.1 Implementation of the Avoidance Layer Node

By Cholen Premjeyanth

The avoidance layer is implemented to handle getting beyond obstacles using data from the bumper sensor. This design implements the usage of a bump counter, which tracks the number of recent collisions. This allows the robot to distinguish between temporary and persistent obstacles. If the number of recent collisions exceeds the threshold set by *MAX_BUMPS_BEFORE_AVOID*, the node begins publishing a sequence of directional actions designed to maneuver around the obstacle. The predefined sequence is hardcoded directional actions managed by a delay counter named *AVOIDANCE_DELAY*. The sequence is then executed over a time step value labeled *AVOIDANCE_STEP*. This allows how long each action is sent for before transitioning to the next, before reorienting the robot in a position where it can continue its trip bypassing the obstacle. Another timer, *BUMP_COUNTER_REDUCE_TIME,* also reduces the bump counter so that the robot can recover if there was only a temporary obstacle.

```
IF bumper_sensor_triggered THEN
    bump_counter ← bump_counter + 1
    IF bump_counter < MAX_BUMPS_BEFORE_AVOID THEN
        Publish "0:WAIT"
    ELSE
        Execute predefined avoidance sequence
        // Sequence involves timed LEFT, GO, RIGHT, etc.
    ENDIF

ELSE IF currently_executing_sequence THEN
    Continue publishing next step in sequence
    delay_counter ← delay_counter − 1

ELSE
    Publish "0:NONE"
    Reset delay_counter
ENDIF

// Timer every few seconds:
IF bump_counter > 0 AND NOT currently_executing_sequence THEN
    bump_counter ← bump_counter − 1
ENDIF
```

Figure 24: Pseudocode for Avoidance Layer Node

This pseudocode illustrates how this node uses the bump counts to determine whether to wait or execute an avoidance sequence.

9.2.2 Implementation of the Docking Layer Node

By Cholen Premjeyanth

The docking layer is implemented to manage the robots ability to undock at the beginning of a delivery and redock upon reaching its destination, or at a point where further charge is required to continue its trip. This layer receives input from the /navigation topic which determines if a delivery is in progress, and the /dock_status topic to check if a docking station is visible, and whether the robot is currently docked. If the robot is docked during this, the action *1:UNDOCK* is immediately published to the /actions topic. This ensures the robot begins its delivery once a destination is acquired. During navigation, if the navigation message becomes

DOCK and the dock is visible (as confirmed by the dock sensor), the node publishes the docking command *1:DOCK*. The command is only issued when both these conditions are true, ensuring a safe and valid docking procedure. If the robot successfully docks (confirmed by the is_docked from the /dock_status topic), the state returns to *NO_DEST*, ready for the next request.

```
"IF nav_msg == "UNDOCK" AND is_docked THEN
    Publish "1:UNDOCK"
    Transition to HAS_DEST

ELSE IF nav_msg == "DOCK" AND dock_visible THEN
    Publish "1:DOCK"

ELSE
    Publish "1:NONE"
ENDIF
"""
```

Figure 25: Pseudocode for Docking Layer Node

This depicts how the docking layer reacts to navigation messages and sensor data to determine when to initiate docking or undocking actions.

9.2.3 Implementation of the Turning Layer Node

By Cholen Premjeyanth

The turning layer node is implemented to execute directional actions at intersections based on the incoming navigation data. To manage the execution of the turns, the node uses three internal counters. The first one is *TURN_CYCLES*, which determines how long the robot performs a left or right turn after entering an intersection. The next one is *U_TURN_CYCLES*, which controls how long the robot executes a U-turn action for once the instruction is received. Finally the *TURNING_GO_CYCLES*, determines how long the robot should go forwards following a turn, so the robot can thoroughly identify a wall, and continue its navigation towards its destinations. These values are configurable and ensure that each movement, such as a turn or forward motion after a turn, is sustained for an appropriate duration. To ensure each navigation instruction is only acted upon once, a flag *NAV_MESSAGE_HANDLED* is used to prevent re-execution of already-processed messages. This avoids issues where an instruction could be continuously re-triggered due to persistent sensor input. When a turn instruction has finished executing, the node resets its counters, restoring the node to an idle state.

```
IF nav_msg == "U_TURN" AND not_handled THEN
    Publish "2:U_TURN" for U_TURN_CYCLES
    Mark nav_msg as handled

ELSE IF in_intersection AND nav_msg not_handled THEN
    Publish "2:<TURN_DIRECTION>" for TURN_CYCLES
    THEN publish "2:GO" for GO_CYCLES
    Mark nav_msg as handled

ELSE
    Publish "2:NONE"
ENDIF
```

Figure 26: Pseudocode for Turning Layer Node

This figure summarizes how the turning layer applies turns depending on if it is in an intersection, or if a U-turn is required.

9.2.4 Implementation of the Travel Layer Node

By Denis Cengu

The Travel Layer node implements the robots basic movement instructions, primarily wall-following behavior. The node subscribes to three topics, /lidar_data, /destinations and /dock_status. The /lidar_data topic provides the necessary measurements needed to compute the wall-following behavior, namely the robot's current distance and angle relative to the wall. Based on this information the node calculates a linear and angular speed which keeps the robot at a fixed distance to the wall as it travels. Parameters such as desired distance and speed are handled via the config file. The node maintains an internal state to track whether the robot is currently docked or has an active destination, if the robot is docked or has no destination, it will not attempt to move and publish 3:NONE messages to the /actions topic. When the robot is undocked and a destination is published, the node will attempt to compute and publish a 3:WALL_FOLLOW<l_s><a_s> format message to /actions. If the wall following cannot be calculated, we fallback to a 3:GO message instead. These action messages are picked up by the captain node which decides whether they should be executed based on priority. A timer running at 0.2 second intervals ensures that the travel layer checks for new data and updates its commands.

```
IF is_docked OR no_destination THEN
    Publish "3:NONE"

ELSE IF wall_detected THEN
    Compute linear_speed and angular_speed based on LiDAR data
    Publish "3:WALL_FOLLOW <linear> <angular>"

ELSE
    Publish "3:GO"
ENDIF
```

Figure 27: Pseudocode Travel Layer Node

9.3 Implementation of the Captain Node

By Douglas Lytle

The captain node is no longer responsible for navigation, as that is handled by the navigation unit and turning layer node. As such, the captain node is relatively simple, and is implemented as follows. In order to ensure that only the highest priority actions are the ones that are executed while avoiding constant interruptions, and due to the nature of how publish/subscribe relationships work, the captain cannot simply execute action commands as they are received. In order to solve this issue, the captain maintains an array with four slots, one for each of the different priority levels, indexed according to their priority. Whenever a new message is published to the /actions topic by one of the layers, the captain updates the corresponding slot in the array. Then, on a timer, the captain will iterate over the array until it encounters an action which is not “NONE”. When an action is found, the captain queries the action translator, which returns a Twist object representing that action. Then, the returned Twist object is published to /cmd_vel, which moves the robot.

10 Simulation

10.1 Gazebo & Integration with ROS2

By Denis Cengu

As discussed in earlier segments, Gazebo is a powerful physics-based simulation tool that provides us with a realistic environment for testing our system. A big focus point is that Gazebo incorporates a full physics engine which allows for dynamic interactions, sensor emulation and realistic world modelling. A key advantage of Gazebo is its seamless integration of ROS2. In the setup used for the simulation environment, the simulated Create 3 acts as a ROS2 node, publishing and subscribing to topics just as the real Create 3 robot does. This ensures the software developed for the real robot can easily be tested in Gazebo without modification.

Beyond software compatibility, Gazebo offers extensive environment customization allowing for the creation of realistic testing conditions. Using the tools offered in Gazebo it’s possible to design tunnels similar to the real world environment as well as stationary and dynamic obstacles to further replicate the real world challenges that might appear in deployment. This also enables testing of edge cases that would be difficult or impractical to recreate in physical trials, such as sensor occlusions and navigation failures due to unexpected obstacles. Furthermore the simulation can be automated to run multiple scenarios in parallel allowing for efficient evaluation of results and metrics from testing.

10.2 URDF/Xacro Modeling

By Denis Cengu

To create an accurate representation of the Create 3 Robot in simulation, URDF (Unified Robot Description Format) and Xacro (XML Macros) are used. These formats provide a structured way to define a robot's physical properties, joint constraints and sensor attachments. For the Create 3 robot and docking station simulation, the team used the create3_sim package provided by iRobot. The package includes the URDF model of the Create 3 complete with realistic dynamics, collision models and full ROS2 interface compatibility. However, the create3_sim package does not include the additions of the LiDAR and PiCam to the robot model; these additional sensors required custom URDF definitions to accurately replicate the real world counterpart. This process involves defining new links for the sensors, attaching them to the correct joints of the robot's base as well as configuration of physics properties to ensure realistic interactions in Gazebo. The use of Xacro allows for modularity, easing the process of adjusting sensor positions and parameters without extensive rewrites to the provided URDF structure.

Through extension of the provided Create 3 package with the added sensors we can accurately create a one-to-one mapping between the simulated and physical Create 3. This consistency is important to ensure that perception based algorithms such as the LiDAR wall following, or the PiCams intersection marker detection function correctly in both environments without need for adjustments between the two.

10.3 LiDAR Module Simulation

By Denis Cengu

The Create 3 used for the project features an added Slamtec RPLiDAR A1. To replicate this within Gazebo, a ray-based sensor with a full 360-degree field of view was implemented, this allows it to continuously scan the environment as the real LiDAR does. The range in simulation is set to 6 meters as we found our most consistent ranges to be in the range of 3-5m in analysis, the update rate was set to 5.5Hz to match the sensor's default scanning frequency. Additionally, the simulated LiDAR provides 360 data points per revolution to align with the level of detail provided by the physical device. Beyond raw performance, the LiDAR simulation accounts for real environmental interactions including occlusion from obstacles, reflections and noise that could potentially be present in real world scenarios. Furthermore the LiDAR acts as the real part does by creating a /scan topic which the LiDAR publishes its results to, this ensures that the software is one-to-one and simplifies the testing process further.

10.4 PiCam Module Simulation

By Denis Cengu

The Create 3 robot also features a Raspberry Pi Camera Module V2 which features an approximate 62.2 degree horizontal field of view and supports image capture at a resolution of 3280x2464. To accurately simulate this in the Gazebo implementation, the camera was

configured to capture images at a resolution of 640x480 while maintaining a horizontal field of view that matches the real camera at that resolution. 480p is chosen to reflect the real part's value used on the robot. The camera was mounted 20 cm above the base of the robot, angled 60 degrees downward at the rear to mimic its position atop the mail holder. The mass and inertia were adjusted to be negligible as it caused some issues with jittering the robot in simulation, the camera module weighs around 3 g and in real world practice has a negligible effect on the physical properties of the robot. As with the real module, the simulated camera publishes to the camera_data topic to ensure a one-to-one model to facilitate ease of testing the real robot's software.

10.5 Beacon Simulation

By Denis Cengu

For the purpose of simulating the AprilBeacon N04 in Gazebo the team opted to build a beacon publisher that takes information of the position of the beacons and the robot using the /model_states topic published by Gazebo to ROS2. The difference in position is taken and then a log-distance path loss model is applied to get an accurate RSSI similarly accurate to what the team used in the real deployment.

11 List of Required Components and Facilities

By Douglas Lytle

Table 16: List of required components/facilities, purposes, and estimated costs

Required Component/Facility	Purpose	Estimated Cost
Tunnel Access	As the purpose of the robot is to deliver mail in the tunnels, tunnel access will be needed at times to test the robot in a real environment.	$0
Robots/Sensors	Already purchased in past years.	$0
Raspberry Pi Camera Module https://www.amazon.ca/dp/B01ER2SKFS	To improve intersection detection by using computer vision to detect the intersection markers in the Carleton Tunnels.	$30
Longer Ribbon Cable for Raspberry Pi Camera Module https://www.amazon.ca/dp/B07DNYM8KC	The cable included with the camera module was not long enough to reach from the interior of the robot to the exterior.	$15
Protective Case for Raspberry Pi Camera Module https://www.amazon.ca/dp/B09GG1MR6P	To prevent the camera module from being damaged, and to make it easier to affix to the robot.	$15

11.1 Justification of Purchases

By Douglas Lytle

In order to improve the reliability of intersection detection of the robot, the team made the decision to purchase and install the Raspberry Pi Camera Module V2.1. This camera module was chosen since it is specifically designed for use with a Raspberry Pi, making setup easier than it would have been with a USB camera. The Raspberry Pi Camera Module V2.1 was chosen over other Raspberry Pi camera modules since it is the cheapest option which is still available for purchase, and factors such as photo quality are not important for this application.

Also, a longer cable had to be purchased for the camera, as the one included was far too short to reach from the interior of the robot to the exterior. The specific cable purchased was chosen because it was the cheapest option available.

Finally, as the camera module is quite small and fragile, a protective case was purchased for it. Similarly to the cable, the specific case purchased was chosen only because it was the cheapest option available.

12 Known Issues

By Douglas Lytle

Detailed below are some issues with the robot which have been identified and should be fixed if possible, but were not able to be fixed due to time constraints.

12.1 Consecutive Docking/Undocking

By Douglas Lytle and Denis Cengu

The robot is able to successfully dock and undock itself, but only once for each of those actions before restarting the program. Upon attempting to either dock the robot when it has already docked previously since being restarted, or to undock the robot when it has already been undocked previously since being restarted, the robot will freeze, all program execution will halt, and the light ring on the robot will become solid red, which indicates an error. The robot will then become completely unresponsive for 30 seconds to a minute before automatically rebooting itself.

The exact cause of this issue is unknown, but it may arise due to some conflict between the two different ways of controlling the robot (1. Publishing a Twist message to /cmd_vel, or 2. Sending an action goal from an action client to a corresponding action server). Action goals are used to make use of the robot's built-in docking and undocking features, while /cmd_vel is used for everything else.

Another possible cause, or at least a related phenomenon, is that the action goals sent to perform docking and undocking are not resolving properly. This is known to be the case, since a callback function was created which should have been able to receive feedback from the action goals as they are performed and completed. However, no feedback is ever received.

This phenomenon is not replicated in Gazebo, which leads us to believe that potentially it could be an error related to CycloneDDS, in which case a potential solution could involve updating the robot's firmware or switching to FastRTPS.

12.2 Robot stalling upon temporary WiFi signal loss

By Douglas Lytle

Another issue is one which causes the robot to rarely stop in place, and freeze like that for a few seconds before resuming. This happens whenever the robot temporarily loses its WiFi connection, but the actual cause of the freeze is unknown, since the code is all running on the robot and it should not require communication to continue its execution. There are certain specific points in the tunnels where the robot will almost always stop for a few seconds as it changes routers.

12.3 LiDAR node startup issue

By Douglas Lytle

Sometimes, when the system is launched, an error will occur with a message something like 'SL_RESULT_OPERATION_TIMEOUT', or 'SLLIDAR NODE HAS DIED', and the robot will be unable to act. When this happens, the LiDAR sensor must be physically unplugged from the robot and then plugged back in. It is unknown why this issue happens, and the only solution found is as described above.

12.4 Inconsistent wall following behavior at intersections

By Douglas Lytle

Rarely, as the robot approaches an intersection, instead of the intended behavior of the robot first entering the intersection and then performing its turn, the robot will simply never stop following the wall to its right, causing it to take a right turn at the intersection regardless of what should have been done. In theory, this is not a catastrophic failure, as if the bluetooth beacons are placed at every intersection, then as soon as the robot reads the next one it will realize a mistake has been made and it will promptly perform a u-turn to get back on course. However, it is still far from ideal, and a very inefficient way to travel. It should be noted that this does only happen rarely, the exact shape of the intersection and angle of approach likely have a large impact on the likelihood of this error occurring.

One possible solution to this issue is to take a gradient of the distance to the right wall over time, and if the wall suddenly starts getting much further from the robot then it is likely to be entering an intersection.

12.5 Beacon RSSI Inconsistency

By Denis Cengu

The beacons cannot be turned off and thus will consistently drain their batteries, resulting in their emitted signals growing weaker and noisier over time. While the team did not

change the batteries when they performed their first analysis of the beacons, it is highly recommended that the batteries are switched upon project start prior to analysis, as this will greatly increase the RSSI output and consistency of the beacons.

13 Future Work

By Douglas Lytle

While great progress was made on the project this year, a number of areas still remain in which there exists significant improvements to be made. Some of these areas for future work which have been identified are summarized below.

13.1 Improvements to the Accuracy of the Navigation Map

By Douglas Lytle

Currently, the navigation map used by the robot considers each destination to be located at an intersection in the tunnels, when in reality most buildings do not have their tunnel entrances located directly at an intersection. In the future, if this project were to actually be used to deliver mail, it would be a prerequisite that the navigation map was updated to account for the actual location of buildings in the tunnels, so that the docking stations/mailboxes could be placed in more sensible locations.

13.2 Dynamic Turn Calculation for Smoother Intersection Handling

By Cholen Premjeyanth

In the current implementation, the robot uses a hardcoded turning motion defined by TURN_CYCLES, U_TURN_CYCLES, and TURNING_GO_CYCLES. While this has proven reliable, it can still result in inconsistent performance depending on the layout and angle of the intersection. In the future, the system can further leverage sensor data to calculate the actual turn angle at runtime. Instead of using fixed length turns, the robot would be able to determine when a turn is complete or what angle of turn is required based on analyzing real time LiDAR data, and the navigation map. This not only creates a smoother handling of intersections, but also allows the robot to stay aligned with the wall while turning. Furthermore, this creates a more robust system as it can adapt to non right angle intersections.

13.3 Additional Safety Features for Tunnel Traversal

By Douglas Lytle

Another change which should be made to ensure the safety of students in the tunnels and the safety of the robot itself is for the robot to stop momentarily and announce its presence before entering an intersection, much in the same way that the tunnel carts which are frequently seen in the Carleton tunnels do. This could be accomplished by having a node which publishes a high priority STOP action to /actions whenever an intersection marker is encountered, as well as publishing an audio message to /cmd_audio so that the robot beeps audibly. The only reason this was not already implemented is that only closely monitored trials of the robot's delivery

capabilities have been performed to date. However, this change would be a necessity before any fully autonomous or ‘unmanned’ trials should be attempted.

13.4 Re-Introduction of the Web App

By Cholen Premjeyanth and Douglas Lytle

The original project proposal included a web based interface that allowed users to create delivery requests, and track robot progress. While there were plans to re-implement the ability for the robot to receive requests from the web app, this was not done due to time constraints. It should be noted that the web app is still completely functional and has not been touched since its implementation by last year’s project team, and in order for it to be live once more it simply needs to be deployed using a service such as Microsoft Azure. Then, a client node would need to be developed for the robot which can receive delivery requests from the web app and publish a destination to /destinations in response. This node would also periodically post updates about the current status of the robot to the web app.

13.5 Improvements to Gazebo Simulation & Testing

By Denis Cengu

While the Gazebo simulation provides an accurate representation of the robot, sensors and beacons, it lacks similar polish for the measurement of performance metrics. The team defined metrics to be tracked but the implementation is lacking refinement and will require further work to provide a more complete analysis of the overall performance of the robot. The groundwork is done however, and it will just require changes to the GUI and data gathered through topics.

14 Reflections

By Denis Cengu, Douglas Lytle, and Cholen Premjeyanth

As discussed in section 1.3.3, the project’s goals evolved significantly as the year progressed. Overall, the team fell considerably behind in the first term due to unfamiliarity with the ROS2 platform as well as the overall complexity of last year's design, combined with a number of tricky undocumented issues. Despite this, significant progress was made in the second term, albeit at the cost of polish and refinement in some specific areas, in particular testing specific metrics and the web app. The broad goals of shifting the system over to a modular and extensible subsumption architecture as well as providing an accurate simulator were met. The team believes these advancements will significantly aid in the future development of this project in particular with maintaining a rigorous continuous integration workflow.

15 References

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, Architecture, and Uses In The Wild," *Science Robotics*, 2024. Available: https://www.science.org/doi/10.1126/scirobotics.abm6074.

[2] "iRobot Create 3 Hardware Overview," iRobot Education, Accessed: Oct. 8, 2024. [Online]. Available: https://iroboteducation.github.io/create3_docs/hw/overview/

[3] "iRobot Create 2 Programmable Robot," iRobot, Accessed: Oct. 8, 2024. [Online]. Available: https://www.irobot.ca/en_CA/irobot-create-2-programmable-robot/RC65099.html#:~:text=Create%202%20is%20ready%20to,controlled%20via%20Open%20Interface%20Commands

[4] “Thymeleaf”. Thymeleaf Team. Accessed: Oct. 13, 2024. [Online]. Available: https://www.thymeleaf.org/

[5] “Azure App Service”. Microsoft. Accessed: Oct. 13, 2024. [Online]. Available: https://azure.microsoft.com/en-ca/products/app-service/

[6] “Gazebo”. Open Robotics. Accessed: Oct. 12, 2024. [Online]. Available: https://gazebosim.org/home

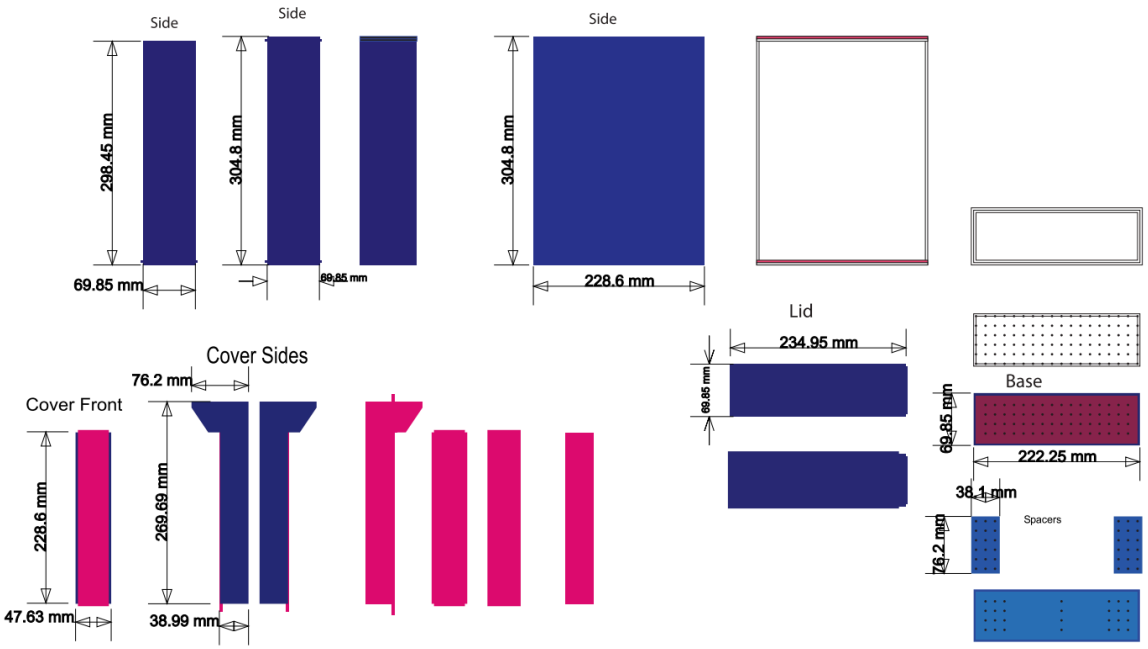
[7] RPLIDAR A1: “RPLIDAR A1 Datasheet.” SLAMTEC. Accessed: Oct. 15, 2024. [Online]. Available: https://bucket-download.slamtec.com/d1e428e7efbdcd65a8ea111061794fb8d4ccd3a0/LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf

[8] R. A. Brooks, *A Robust Layered Control System for a Mobile Robot*, MIT Artificial Intelligence Laboratory, A.I. Memo 864, Sept. 1985. [Online]. Available: https://people.csail.mit.edu/brooks/papers/AIM-864.pdf

Appendix A: Mail Holder Design

Create 3 Mail Holder Design

The Create 3 mail holder was designed to hold 8.5x11in Letter size papers with a sliding lid for security. It was designed using Adobe Illustrator and then ToolPath was used to ready it for the CNC machine. All pieces are fit together and glued, the camera bracket on the back can be removed with a star head screwdriver while the entire holder can be removed with a square head screwdriver.



Appendix B: Hardware Setup

Create 3 Hardware Setup

1. Connect the micro-USB into the LiDAR's USB adapter board and to the Raspberry Pi 4B.
2. Connect the PiCameras ribbon cable to the Raspberry Pi 4B.
3. Connect the USB-C cable between the USB-C port in the robot's cargo bay to the Raspberry Pi 4B's USB power input port.
4. Ensure the LiDAR Serial Cable is connected to the USB adapter board.
5. Ensure the mail holder and camera bracket are firmly in place, tighten with screwdriver if needed.

Appendix C: Software Setup

Setting up the project for the Create 3

For additional help or any unmentioned circumstances, refer to Create 3 or ROS documentation located at the following https://iroboteducation.github.io/create3_docs/ & https://docs.ros.org/en/humble/index.html.

Setting up the Create 3 robot firmware (skip if the Create 3 firmware was already set up).

1. Download the latest version of the Create 3 firmware for Humble: https://iroboteducation.github.io/create3_docs/releases/overview/.
2. Power the robot on by placing it onto a plugged in charging dock.
3. Press and hold both of the side buttons until the lights around the middle button turn blue.
4. Connect to the Create-[xxx] Wi-Fi network on your device.
5. Navigate to 192.168.10.1 to access the Create 3 web interface.
6. Navigate to the Update tab and follow the steps to upload the downloaded firmware.
7. Wait until the robot chimes and then navigate to the Application → Configuration tab.
8. Ensure the RMW_IMPLEMNTATION dropdown is set to rmw_cyclonedds_cpp.
9. Restart the application if the RMW implementation is changed.
10. Navigate to Beta Features → NTP sources tab and add "server 192.168.186.3 iburst" so the robot can receive NTP info from the Raspberry Pi.
11. Reboot the robot and disconnect from the Wi-Fi network

Installing the Ubuntu Image and setting up Wi-Fi (Requires a micro-SD slot)

1. Download the Raspberry Pi imager: https://www.raspberrypi.com/software/
2. Using the Raspberry Pi imager, select Ubuntu Server 22.04x LTS (64-bit).
3. Select the gear icon and ensure SSH is enabled, also set a username and password. The wireless LAN setup included will not work with the Create 3 and Carleton network.
4. Write the image to the Raspberry Pi 4B's microSD card.
5. Eject and reinsert the microSD card into your device to access the system-boot partition.
6. Edit the "config.txt" file to add "dtoverlay=dwc2,dr_mode=peripheral" at the end.
7. Edit the "cmdline.txt" file to add "modules-load=dwc2,g_ether" after "rootwait".
8. Edit the "network-config" file to add Wi-Fi information

```
version: 2
ethernets:
    eth0:
      dhcp4: true
      optional: true
    usb0:
      dhcp4: false
      optional: false
      addresses: [192.168.186.3/24]
wifis:
    renderer: networkd
    wlan0:
      dhcp4: true
      optional: true
      access-points:
        "eduroam":
           auth:
              key-management: eap
              method: peap
              identity: <school wifi login username>
              password: <wifi password>
```

9. Insert the microSD card into the Raspberry Pi 4B.

Creating an account on the image (An account should have been created when the image was first created)

1. ***sudo su - root***
2. ***hostname ctl set-hostname***
3. ***adduser [accountname]***
4. ***su - [accountname]***

Setting up the NTP Server

1. Go to https://iroboteducation.github.io/create3_docs/setup/compute-ntp/ and follow the steps to configure an NTP server on the Raspberry Pi. Steps 6 and 7 were completed during firmware setup.
2. Run timedatectl and check if “System clock synchronized” has the value “yes”. If it does, move on the locale configuration.
3. If not, run ***sudo nano /etc/system/timesyncd.conf***
4. Add “**NTP=ntp.ubuntu.com**” and “**FallbackNTP=0.us.pool.ntp.org**” if not already configured.
5. Run ***systemctl restart systemd-timesyncd.service***

ROS2 Humble Installation

1. Install ROS Humble using this link as a guide: https://docs.ros.org/en/humble/Installation.html
2. Run ***sudo apt install -y ros-humble-irobot-Create-msgs***
3. Run ***sudo apt install -y build-essential python3-colcon-common-extensions python3-rosdep ros-humble-rmw-cyclonedds-cpp***
4. Run ***sudo rosdep init***
5. Run ***echo “source /opt/ros/humble/setup.bash” >> ~/.bashrc***
6. Run ***echo “export RMW_Implementation=rmw_cyclonedds_cpp”>> ~/.bashrc***
7. Reboot the robot by holding the power button for 10 seconds and then place it onto the charging dock.
8. Run ***ros2 topic list*** and ensure the Create 3 API topics are displayed.

Libcamera and rpicam-apps Installation for the camera

1. Install Libcamera using this link as a guide: https://github.com/raspberrypi/libcamera
2. First, install the dependencies:
3. Run **sudo apt install -y libboost-dev**
4. Run **sudo apt install -y libgnutls28-dev openssl libtiff-dev pybind11-dev**
5. Run **sudo apt install -y qtbase5-dev libqt5core5a libqt5widgets**
6. Run **sudo apt install -y meson cmake**
7. Run **sudo apt install -y python3-yaml python3-ply**
8. Run **sudo apt install -y libglib2.0-dev libgstreamer-plugins-base1.0-dev**
9. Then, install libcamera:
10. Run **git clone https://github.com/raspberrypi/libcamera.git**
11. Run **cd libcamera**
12. Run **meson setup build --buildtype=release -Dpipelines=rpi/vc4,rpi/pisp -Dipas=rpi/vc4,rpi/pisp -Dv4l2=true -Dgstreamer=enabled -Dtest=false -Dlc-compliance=disabled -Dcam=disabled -Dqcam=disabled -Ddocumentation=disabled -Dpycamera=enabled**
13. Run **ninja -C build install**

14. Then, install rpicam-apps:
15. Run **git clone https://github.com/raspberrypi/rpicam-apps.git**
16. Run **cd rpicam-apps**
17. Run **meson setup build**
18. Run **ninja -C build install**

Creating a workspace and cloning the necessary code.

1. Run ***mkdir -p ~/cmdr_ws/src***
2. Run ***cd ~/cmdr_ws/src***
3. Clone the repository:
   git clone https://github.com/deniscengu/carleton-mail-delivery-robot.git
4. Clone the LiDAR package: ***git clone https://github.com/Slamtec/sllidar_ros2.git***
5. Install bluepy by running: ***sudo pip install bluepy***
6. Install requests by running: ***sudo pip install requests***
7. Run ***cd ~/cmdr_ws***
8. Run ***colcon build --symlink-install***

Running the Code

1. Run ***sudo -s***
2. Run ***cd ~/cmdr_ws***
3. Run ***source /opt/ros/humble/setup.bash*** (This was set to run on boot but just in case)
4. Run ***source install/setup.bash***
5. Run ***ros2 launch mail-delivery-robot robot.launch.py***
6. In another terminal, publish a destination in the following format:
   ros2 topic pub --once /destinations std_msgs/msg/String '{data: Nicol:Canal}'

NOTE: if you encounter an error while running the code which states something like 'lidar node has died' , or 'SL_RESULT_OPERATION_TIMEOUT', try unplugging the lidar from the robot and plugging it back in.

Setting up Firebase for IP address access

The CU Wireless network will allocate a new IP to the Pi every boot, therefore to get the assigned IP for SSH access a Firebase database will be used. A tool for running a systemd service that sends the IP on boot is included in the tools folder of the project repository and can be setup as follows:

1. Go to https://console.firebase.google.com/ and login to a Google account (can make a new one or use an existing account)
2. Click on the "add project" button and give the project a name (ex. Create-ip)

3. Create the project
4. Add a realtime database and anonymous authentication to the project
5. Find the API key from the project settings page
6. Go to the tools folder of the project GitHub repo and open the Firebase_Ip_Service folder
7. Edit the fwdip.py file and add your Firebase information to the CONFIG dictionary. The apiKey value should be your project's API key, the authDomain value should be ".firebaseapp.com", the database URL value should be "https://-default-rtdb.firebaseio.com/", and storageBucket should be ".appspot.com". Replace values with the project name chosen in step 2 (ex. "authDomain: "Create3-ip.firebaseapp.com").
8. Configure the "CHILD" value with the name you want sent to the firebase. When using multiple Create 3 robots, change this value so that you know the associated IP of each robot.
9. Install the Pyrebase library using “pip3 install Pyrebase4”.
10. Copy the fwdip.service file to the /etc/systemd/system folder and edit the ExecStart path to the path of your fwdip_helper.sh and fwdip.py files (fwdip_helper.sh and fwdip.py must be in the same folder).
11. Initialize the service by running “sudo systemctl daemon-reload” and then “sudo systemctl start fwdip.service”.
12. The service will now run on boot and post the IP address to the Firebase. The status of the service can be checked using “sudo systemctl status fwdip.service”.

Appendix D: Gazebo Setup

To set up and run Gazebo you will need an Ubuntu 22.04x system, this can be an emulator (VirtualBox, WSL).

Installing ROS2 Humble

1. Install ROS Humble Desktop and Dev tools using this link as a guide: https://docs.ros.org/en/humble/Installation.html
2. Run ***sudo apt install -y ros-humble-irobot-Create-msgs***
3. Run ***sudo apt install -y build-essential python3-colcon-common-extensions python3-rosdep ros-humble-rmw-cyclonedds-cpp***
4. Run ***sudo rosdep init***
5. Run ***echo “source /opt/ros/humble/setup.bash” >> ~/.bashrc***
6. Run ***echo “export RMW_Implementation=rmw_cyclonedds_cpp”>> ~/.bashrc***

Installing Gazebo

1. Run ***curl -sSL http://get.gazebosim.org | sh***
2. Try running ***gazebo***
3. If that doesn't work follow the instructions from the following page: https://classic.gazebosim.org/tutorials?tut=install_ubuntu
4. Follow the instructions on this page to install Ignition Fortress: https://gazebosim.org/docs/fortress/install_ubuntu/
5. Run ***sudo apt-get install ros-humble-ros-gz***

Creating a workspace and cloning the necessary code.

1. Run ***mkdir -p ~/testing_ws/src***
2. Run ***cd ~/testing_ws/src***
3. Clone the repository: ***git clone https://github.com/deniscengu/carleton-mail-delivery-robot-gazebo.git***
4. Move dashboard.py from external_files into your home directory, move the demo_video.world file to your .gazebo/worlds/ directory, and finally move all the mac address folders and intersection marker folder to .gazebo/models.
5. Run ***cd ~/testing_ws***
6. Run ***colcon build --symlink-install***

Robot Bringup Process

1. Run **cd ~/testing_ws**
2. Run **source /opt/ros/humble/setup.bash**
3. Run **source install/setup.bash**
4. Run **ros2 launch irobot_create_gazebo_bringup create3_gazebo.launch.py**

Running Code in Gazebo

Code is run as you would run it normally, just make sure that Gazebo is running and the robot is correctly behaving. There are some bugs with the system though most are fixed by closing Gazebo and running the bring up process once more.

To run the metric analyzer, make sure to use the launch argument as follows:

ros2 launch mail-delivery-robot robot.launch.py enable_metrics:=true

Following the end of your delivery simulation, shut down the ROS2 node and run your dashboard.py as follows from the home directory:

python3 dashboard.py

Appendix E: Github Repo and Demo Video Links

- Github repo: https://github.com/deniscengu/carleton-mail-delivery-robot/
- Github repo for Gazebo: https://github.com/deniscengu/carleton-mail-delivery-robot-gazebo
- Robot Demo 1: https://www.youtube.com/watch?v=HrFrQdZ8Tf8
- Robot Demo 2: https://www.youtube.com/watch?v=2Pq49RQnNI0
- Robot Demo 3: https://www.youtube.com/watch?v=L7cs-OAEkkU
- Gazebo Demo: https://www.youtube.com/watch?v=34wwi-BHo8g