Autonomous Mail Delivery Robot

Final Report

Max Curkovic 101139937
Cassidy Pacada 101143345
Bardia Parmoun 101143006
Matt Reid 101140593

Supervisor: Dr. Babak Esfandiari

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

April 10th, 2024

Abstract

By Bardia Parmoun

The goal of the Carleton Mail Delivery Robot System is to deliver mail across the Carleton University tunnels. The system is an autonomous robot that can navigate the tunnels without any human interaction. The system is built using an iRobot Create robot, a Raspberry Pi 4, and is equipped with a LiDAR sensor to easily perceive its surroundings. The robot is expected to work reliably in the tunnels. As such, Bluetooth beacons are the only reliable way for the robot to become aware of its location. These beacons are placed strategically near the intersections and docking stations allowing the robot to make appropriate decisions based on the information that it receives from the beacons. Furthermore, the team has designed the system to be as resilient as possible through various fail-safe mechanisms such as collision handling and dynamic navigation. The implementation of these mechanisms are discussed further in the report. The team believes that this system is a simple and cost-effective solution to deliver mail in the Carleton University tunnels. This report contains all the software and hardware information required to develop and assemble the final system, coupled with recommendations on ways that the system can be improved in the future.

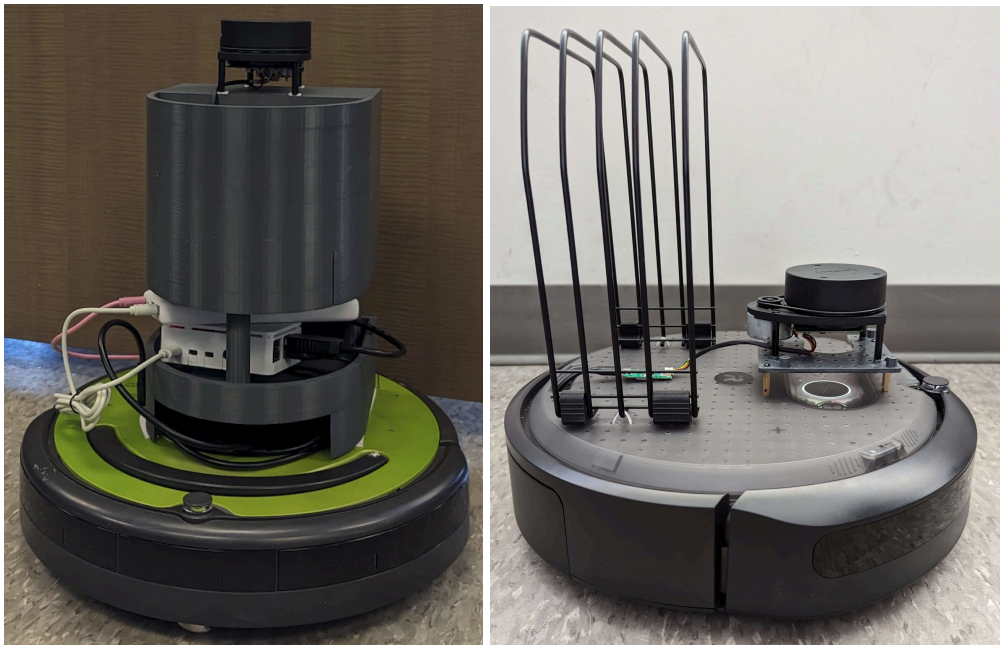


Figure 1: A picture of the final assembled units for the mail delivery system

Table of Contents

List of Tables

List of Figures

1 Objectives

By Max Curkovic, Matt Reid, Cassidy Pacada, and Bardia Parmoun

This section details the objectives of the project that were laid out by the team in the project proposal and progress report, as well as a brief description of the project itself.

1.1 Project Objectives

By Max Curkovic

During this term, the objectives of this project were to develop the robot such that it will be able to autonomously move through the tunnels, be able to detect and avoid anything in its path, (e.g. carts, people) and avoid hitting walls. The robot should be able to maintain enough power to complete its journey to its destination. The robot should have a way to carry mail and should ensure that it does not fall off during the travel time. The robot should also be controlled by a user-friendly web application that is able to set the desired location.

Upon completing these tasks, the project expenses were to remain within the given $500 budget. The team has measured each objective iteratively through a series of tests that can competently consider each case. The team set an objective to have the robot in a position to successfully complete a mail delivery trip to a building by the end of the Winter 2024 term.

1.2 Project Description

By Bardia Parmoun and Matt Reid

This was the third year of this project. As stated in the project's original proposal, the main goal was to create an autonomous robot that can deliver mail between the different buildings at Carleton University through the tunnels. The project also includes a web application that easily allows the users to place their orders and manage the deliveries.

The previous team primarily focused on the functionality aspect of the robot by working on ensuring that the robot moved as expected. This task included implementing and verifying the following: wall following, passing through intersections, right turns, and left turns. Based on the conclusions in last year's report, it can be seen that despite clear improvements in the robot's movements since its first iteration, there were still many upgrades needed. They also emphasized that the existing sensors on the robot needed to be improved. Additionally, there was very little work completed on the web application for the robot. As a result, the team's primary focus for the year was ensuring the functionality of the robot is working perfectly. This involved obtaining new sensors, updating the existing wall following and turn behaviour, improving the collision detection, improving the navigation capabilities, and improving the web application. The goal was to ensure that the robot would have reliable movements and readings.

2 The Engineering Project

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

This section expands on the core principles of engineering professionalism, including health and safety, teamwork skills, ethical regulations and practices, as well as project management techniques and how they apply to the project. The individual contributions (on this report and on the project itself) can also be found here.

2.1 Health and Safety

By Max Curkovic

As specified in the proposal, all experiments and test runs with the robot were conducted in two specified locations: the Canal lab room (CB 5101) and the Carleton University tunnels. The team ensures that all tests were done in secluded areas, whether within the empty lab room or ensuring that there are no people obstructing the path of the robot if in operation. By doing this, the team took every possible precaution to ensure the risk to public safety is minimal.

The team also ensured to take all possible precautions when working with the hardware. When working with voltmeters and converters, all team members ensured that hands stayed away from the probes, and there were no watches or jewelry being worn. It was also extremely important for the Pi to be powered off and unplugged prior to changing the hardware and cables.

2.2 Engineering Professionalism

By Max Curkovic

ECOR 4995 is a course taken by fourth-year undergraduate students that aims to teach the practice of professional engineering. It equips students with knowledge of various topics, such as engineering ethics, professionalism, communication skills via a variety of mediums, and legal obligations. Bearing this in mind, the team vowed to uphold these principles over the duration of the project:

- Communicating effectively, openly, and honestly amongst team members during meetings and work sessions.
- Emphasizing written communication skills when working on the proposal, progress report, and final report.
- Presenting report information in a clear and concise manner to the project supervisor.
- Considering public safety as the highest priority during project development.
- Adhering to all ethical regulations and practices during project development, including respecting intellectual property rights through utilizing correct references.

- Using a project development methodology (in the team's case, an agile development process), and adapting to any changes that may arise within the planning or scheduling of milestones.

The standards maintained in engineering also apply to the project management techniques utilized by the team in order to maintain a level of professionalism.

2.3 Project Management

By Max Curkovic

The size and scope of the project made it necessary for the team to implement several project management techniques over the course of the term. Each technique is described in succeeding sections.

2.3.1 Project Timeline

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

This section provides a summary of the major milestones for this iteration of the progress along with their expected completion dates.

Table 1: Project milestones and proposed target completion dates in table form

Milestone	Target Completion Date	Status
1 - Design a new chassis that accommodates better stands for the components and print it.	September 30th, 2023	Completed
2 - Prepare a prototype using the lidar sensor and compare performance with existing sensors	October 13th, 2023	Completed
3 - Proposal	October 20th, 2023	Completed
4 - Introduce unit tests and integration tests	October 27th, 2023	Completed
5 - Improve the state machine and implement new designs	November 4th, 2023	Completed
6 - Improve wall following	November 18th, 2023	Completed
7 - Fix the existing turn behavior and finalize movements	January 1st, 2024	Completed
8 - Replace the hard-coded navigation with a dynamic one	January 8th, 2024	Completed

9 - Create a simple prototype for the web app with simple commands: start, stop, status log, and delivery	January 22nd, 2024	Completed
10 - Oral Presentation	January 29th, 2024	Completed
11 - Add simple collision handling OR integrate iRobot's collision handling	March 10th, 2024	Completed
12 - Integrate the current codebase into Create 3	March 10th, 2024	Completed
13 - Poster Fair	March 22nd, 2024	In Progress
14 - Final Report	April 10th, 2024	In Progress

2.3.2 Gantt Charts

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

As depicted in the project proposal, a Gantt chart (Figures 2 and 3) was created as an agile, visual representation of when each milestone should be completed for. Each project milestone is divided as its own separate task. Each task is divided into its own "milestone", subsumed by different cycles. The cycle typically starts with implementing any new states that were designed as part of the state machine, then considers all unit and integration testing for each task, and then allows for time to work on the project report. If this cycle is completed, then the team progresses to the next milestone.

The updated Gantt chart is quite large to fit into one figure - as a result, it is separated below into the Fall 2023 and Winter 2024 terms.

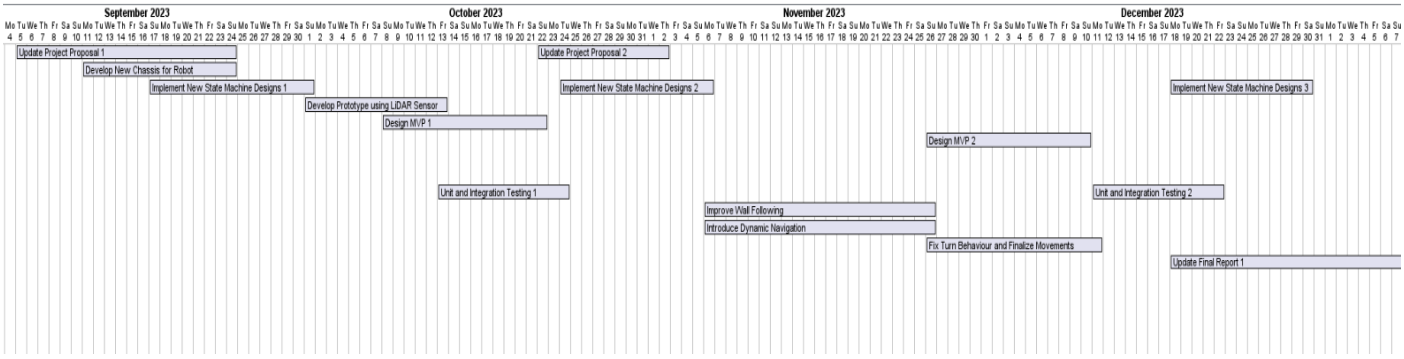


Figure 2: Fall 2023 project milestones and proposed target completion dates, in Gantt form

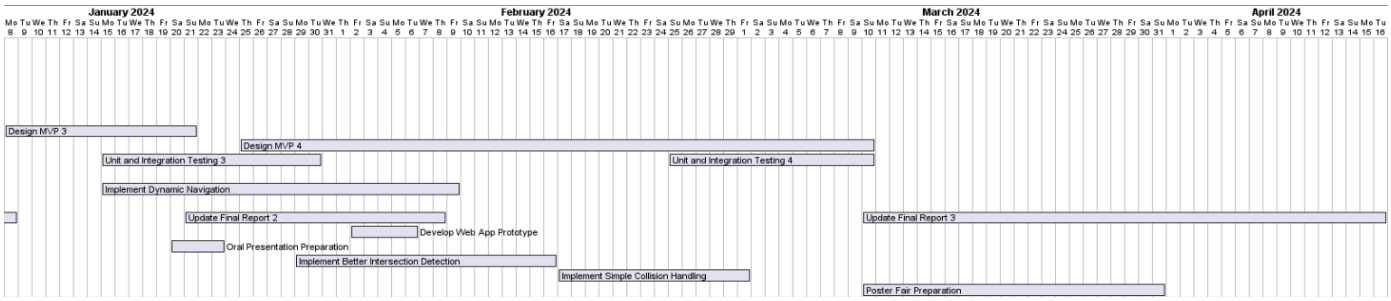


Figure 3: Winter 2024 project milestones and proposed target completion dates, in Gantt form

2.3.3 Risks and Mitigation Strategies

By Cassidy Pacada

Table 2: Possible risks and their mitigation strategies

Project Risk	**Mitigation Strategy**
1. Potential hardware failure could slow project progress as it takes time to re-solder circuitry. Should a larger, more important piece fail, such as the iRobot itself, it may have taken an extended period of time to replace.	The current faulty circuitry was replaced with new equipment to minimize the risk of failure. Additionally, the team has extra pieces of critical hardware on hand in case a failure does occur.
2. Making incorrect changes to the code could potentially set the project back if there is no way to recover previously functional code.	The team used a version control system, Github, to ensure that changes can be reverted if necessary.
3. There was a risk that the existing sensors will not be sufficient to navigate the tunnels effectively. Should this implementation strategy have failed with no backup, it would leave the project directionless.	The team came up with alternatives to fall back on if the initial navigational implementation did not work. These include using LiDAR sensors and computer vision to navigate.
4. Any test of functionality where hitting an obstacle can cause damage to the robot or its hardware can potentially lead to costly replacement of said hardware.	The team supervised and monitored all tests of functionality whenever they are performed, as someone can step in and stop the robot if absolutely necessary. It was also important for the robot testing to be taken in a location with an ample amount of Wi-Fi, as falling out of Wi-Fi range would prevent the team from being able to control the robot, contributing to said risk. All precautions to protect the robot (e.g. ensuring the chassis is in place) were also taken prior to testing.

2.4 Justification of Suitability for Degree Program

By Max Curkovic and Bardia Parmoun

The team for the 2023-2024 iteration of the project consists of four people. Three of the team members, Bardia Parmoun, Cassidy Pacada, and Max Curkovic are software engineering students. Matt Reid, the fourth member, is a computer systems engineering student. Both programs are closely related and focus on major areas in the field of software and computer engineering. These are namely: embedded development, requirements engineering, web development, real-time systems, etc. The development of this project will span all of these areas.

The majority of the code base for the robot has been developed in Python and the Robot Operating System (ROS), with the web application written in Java. The code for the robot itself is run on a Raspberry Pi. These align very closely with two of the engineering courses that all engineering students take in their first year, namely ECOR 1051 (Fundamentals of Engineering I) and ECOR 1052 (Fundamentals of Engineering II). In addition, there are various state machines involved in implementing the main functionalities of the robot. This is closely related to the curriculum for SYSC 3303 (Real Time Concurrent Systems).

The robotics hardware for the project requires knowledge of circuit design as well as integration between hardware and software systems. A fundamental understanding of circuit design and electronics theory was acquired in ELEC 2501 (Circuits and Signals) and ELEC 2507 (Electronics I). Experience with communicating between hardware sensors and software including communication protocols and end-to-end testing was acquired through various design projects and labs in the Computer Systems Engineering courses SYSC 3010 (Computer System Development Project) and SYSC 4805 (Computer Systems Design Lab).

Furthermore, the web application portion of the project is related to the material covered in SYSC 4504 (Fundamentals of Web Development) and SYSC 4806 (Software Engineering Lab). It is also worth mentioning that requirement analysis and software architecture were covered in every step of the project, which are the main focus of the following courses, SYSC 3020 (Introduction to Software Engineering), SYSC 3120 (Software Requirements Engineering), and SYSC 4120 (Software Architecture and Design). Finally, some 3D design work was done for the physical aspects of the robot, including the chassis, which was covered in ECOR 1054 (Fundamentals of Engineering IV).

As illustrated in the project proposal, the mail delivery system project spans various aspects of the software engineering and computer systems engineering programs at Carleton University. The individual contributions support the justification of suitability for the degree program.

2.5 Individual Contributions
By Max Curkovic

The sections below detail the contributions made from team members, on both the overall project and the final report.

2.5.1 Project Contributions
By Max Curkovic

- Max Curkovic
    - Implementing the navigation aspect of the robots.
    - Adding and maintaining the Github workflows.
    - Implementing the web app to control the robot.
- Cassidy Pacada
    - Implementing the partial and full integration tests.
    - Creating thorough unit tests.
    - Creating and maintaining the testing framework.
    - Implementing the web app to control the robot.
- Bardia Parmoun
    - Implementing the robot's state machine, robot driver, and action translator.
    - Developing the middleware for controlling the robot.
- Matt Reid
    - All work related to the development of the robot's hardware, electrical components, and chassis.
    - Developing the middleware for controlling the robot.

2.5.2 Report Contributions
By Max Curkovic

Contributions for each report section are given in the sub-heading beneath each section title. Each section is divided equally amongst team members.

3 Background

By Matt Reid

The autonomous mail delivery service robot project is intended to streamline the transportation of mail across Carleton University, through the use of autonomous robots driving through the tunnels. Professors often send mail through manual methods, usually by means of walking across campus to another building. The robot is designed to operate without any interference from humans which will reduce the overall mail delivery time.

Currently, the robot relies on Bluetooth connectivity to traverse the tunnels. Various sectors of the tunnels do not have cellular service, thus the robot cannot utilize typical navigation methods such as a GPS. The robot utilizes Bluetooth-supported beacons to support navigation by creating a graph representation of the tunnels. Additionally, the robot uses LiDAR sensors to sense the walls and obstacles in the tunnels. This allows for reliable dynamic navigation.

3.1 Robot Operating System

By Matt Reid

The Robot Operating System (ROS) is a set of open-source libraries and tools used for robotic applications [1]. The provided drivers and algorithms for a wide range of sensors and actuators allow for much faster development and simpler source code. ROS2, an updated version of the original ROS1, can be used on a microcontroller to communicate with supported robots. The ROS library is documented in both C++ and Python. There are many pre-made ROS packages to support a wide variety of sensors and actuators. For this project, the team is using the Python version of this library, rospy, as it works easily with the iRobot Create driver.

3.2 iRobot Create

By Matt Reid

The iRobot Create platform provides an educational version of an autonomous iRobot vacuum. The iRobot Create 2 connects over a serial cable to a microcontroller to receive commands [2]. It has components from an iRobot vacuum including two wheels that can be driven on command, a bumped sensor to detect hitting objects, IR cliff detectors to prevent falling off edges, and a vacuum brush. A microcontroller such as a Raspberry Pi or Ardunio can send commands to the serial port on the iRobot Create 2 using UART. The microcontroller receives and processes all of the sensor data, and then sends the robot commands to drive. It also supports the use of the “Virtual Wall” which works like a Bluetooth beacon and creates a wall that the robot will not pass.

A new version of the platform, called the iRobot Create 3 has ROS2 built-in and can connect with WiFi or Bluetooth instead of using a microcontroller [3]. The robot has seven IR obstacle sensors built in to detect objects and follow walls. It also has a custom faceplate which allows for easy mounting of boards through many screw holes, and cable passthrough to the storage compartment. Unlike the Create 2 platform, the Create 3 does not have a vacuum brush. The built-in ROS2 includes a full interface of ROS commands to take sensor data, follow walls, and facilitate positional navigation. Inside the storage component, it provides a 14.4V/2A battery connection as well as a 5V/3A USB-C port that can be used to power and can connect to a microcontroller if needed. This microcontroller connectivity allows for easy porting of code used on the Create 2 platform to the Create 3 without major architectural changes in the software. This means that both platforms can be developed at once without having two fully separate code bases.

3.3 LiDAR Sensor

By Matt Reid

A Light Detection and Ranging (LiDAR) sensor rapidly emits many light pulses that reflect off of objects and measures data such as the time elapsed and reflection angle in order to generate a 3D map [4]. This map can be used by the robot to navigate and avoid obstacles in the tunnels. By using a LiDAR sensor, the robot will be able to detect the tunnel walls at a much farther distance (~30cm with the IR sensors vs. 12m with LiDAR), allowing for navigation straight through an intersection without losing the walls of the tunnel. A LiDAR sensor has a much higher cost than other distance sensors, however with just one LiDAR sensor, an accurate 360-degree map of the robot's surroundings can be made. For the robot, the team is using the Slamtec RPLIDAR A1 which can measure 8000 times per second with a range of 0.15-12m with a resolution of 0.5mm [5]. It outputs the distance and heading measurements over UART meaning that it can be connected to the Raspberry Pi microcontroller currently being used to control the robot. A premade, ROS compatible, SDK is also provided for the device by Slamtec.

3.4 Spring Framework and Spring Python

By Max Curkovic

The Spring Framework is a modern framework that is used for many web applications developed using Java. It is extremely popular as it provides many services that are relevant for modern software engineering, including integration testing support, object-relational mapping for the back-end (ORM), Spring Model-View-Controller, dependency injection, and aspect-oriented programming (AOP) [6]. The team has extensive experience using the Spring Framework from SYSC 4806 (Software Engineering Lab) along with the above concepts, and should be a seamless transition towards using it to develop a Python-based web application.

As discussed above, the Spring Framework also extends to Python, in the form of Spring Python. Spring Python makes use of the features provided with the Spring Framework, and allows for the use of Inversion of Control (IoC) between classes, which can be quite useful for the numerous ROS nodes within the project [7]. Spring Python consists of extensive documentation [7] that will be useful for the setup and execution of the web application.

3.5 Microsoft Azure

By Max Curkovic

The team has utilized Microsoft Azure [8] to deploy the web application. This is mainly due to the prior experience that the team has from SYSC 4806, and Microsoft Azure works seamlessly with continuous deployment on Github [8]. Microsoft Azure also provides many services for web app deployment that are free, albeit with limited resources, which also influenced the team's decision to choose Azure as a deployment tool.

3.6 Thymeleaf

By Max Curkovic

Thymeleaf is a framework used in correlation with Java and Spring Boot. It is used to apply Java to HTML templates [9] and provides numerous different methods for dynamically applying properties to the view of a MVC platform. All Thymeleaf are identified with "th:" within the HTML templates, and Thymeleaf has its own documentation [9] for reference.

3.7 JUnit 5.0

By Max Curkovic

JUnit is a unit testing framework developed for Java. The web application uses JUnit 5, making use of the submodule JUnit Jupiter [10] in order to work seamlessly with Spring Boot. Students working on this project in future years should have extensive experience with JUnit from courses such as SYSC 2004 and SYSC 3110. JUnit has its own documentation for reference [10].

4 Group Skills

By Bardia Parmoun

As previously mentioned, the team for this iteration of the project consists of members both from the software engineering and computer systems engineering program. This allowed for the team to collectively obtain all the skills required to complete the project. Below is a summary of the skills for each team member:

- **Bardia Parmoun**: As a 4th-year software engineering student, Bardia has a lot of experience working with different languages such as Python, Java, C, and C++. Bardia also has some experience working with web development tools and languages such as HTML/CSS and JavaScript. Bardia also has some embedded software development experience and some electronics knowledge. Finally, Bardia also has some experience working with 3D designs and modeling.

- **Cassidy Pacada**: Cassidy is a 4th year software engineering student with a lot of experience with Python and Java. Cassidy also has experience with front-end development using HTML/CSS and JavaScript. She also has some experience working in the field of cybersecurity which was useful with regards to the security of the robot. Finally, Cassidy has extensive experience working in Linux environments.

- **Max Curkovic**: Max is a 4th year software engineering student with knowledge of programming languages such as Python, Java, and C. Max also has a lot of experience with full-stack web development, which helped with the development of the web application associated with the robot. Max also has experience with Raspberry Pis and Linux.

- **Matt Reid**: Matt is a 4th year computer systems engineering student. As such he has a lot of experience working with embedded systems and hardware design. Matt has also done some projects with Raspberry Pis and Arduino microcontrollers. In addition, Matt has previously worked with ROS which is the operating system that is used to control the iRobot's movement, and has electrical engineering knowledge for any required circuitry.

Using each member's individual skill sets, different methods were undertaken during the development of the robot, and many ideas were combined and taken into consideration.

5 Methods

By Cassidy Pacada

Firstly, to accomplish a project objective of improving the robot's navigational capabilities, the group chose to implement several changes to the robot. Primarily, the existing sensors were replaced with LiDAR providing more accurate data for the robot to work with. During the initial testing phase, it was noted that the IR sensors were ineffective after a distance of 30cm. This would have been problematic in larger sections of the tunnels as the robot would have lost the wall and been unable to navigate to its destination. The robot's original IR sensors were both situated on its right side, which was also problematic. As a result, the team determined that the robot would not be able to follow the wall effectively when the wall is on its left side. It also struggled with making turns consistently and was unable to prevent collisions with obstacles directly in front of it. The switch to LiDAR provides the robot with a 360° view, remediating its sensing limitations and allowing for greater progress in the robot's navigational capabilities.

Additionally, the team implemented a simple web application so that users can interact with the robot. This expanded upon the user interface portion of the web application that was created during the previous year's project, which originally did not have any back-end. The team created the back-end of the application using Spring Boot, as it is taught in SYSC 4806 (Software Engineering Lab) and will be easier for future students to manage.

The group used an Agile methodology throughout the duration of the project to allow for continuous improvement. This aided in validating the project design and implementation choices, as each iteration had a testing phase. Using an Agile methodology helped to prevent the team from discovering critical design failures towards the end of the project, especially as the team transitioned from the Create 2 to the Create 3 platform. The team learned about the benefits of this method in SYSC 4106 (The Software Economy and Project Management) and decided to follow it for this project.

To complete this project, the team put many problem-solving methods into practice. The knowledge acquired during the team's degree programs is essential and was applied throughout the entire project. Requirements analysis, which was learned in SYSC 3120 (Software Requirements Engineering) was crucial in determining what the main focus of the project should be. As well, the team has implemented the new features, including the state machine and the architecture of the robot itself, in a way that is clean and maintainable for future groups, which was done using skills obtained in SYSC 4120 (Software Architecture and Design).

6 Analysis

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

This section summarizes all the analysis work the team conducted before preparing the design for the system. Note that the Create 2 and Create 3 are specified for each category.

6.1 Analyzing the Equipment

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

As the system relies on a variety of different components, a thorough analysis of the components was conducted to properly understand the system's capabilities. This involved measuring different aspects of the robots, measuring external equipment such as the Bluetooth beacons, and measuring the environment such as the Wifi speed.

6.1.1 Analyzing the Create 2 Robot's speed

By Max Curkovic

Firstly, the speed of the Create 2 robot was measured. On average, it traveled approximately 0.19 m/s at its default setting.

Table 3: Measuring the speed of the robot

Distance (m)	Time (s)	Speed (m/s)
1	7.18	0.14
2.5	14.89	0.17
5	25.94	0.19

Overall, it can be concluded that at its current average speed, the robot is quite slow which could affect delivery times. The current metric for the delivery time of the robot was designed by this speed; however, the speed of the Roomba may be increased. The team has developed a reliable navigation system that the robot can safely follow, and there is room for a minor increase in robot speed if necessary.

6.1.2 Analyzing both robots' power output

By Matt Reid

Secondly, the robot power systems for external components were analyzed. The external components consist of a Raspberry Pi 4B and a Slamtec RPLIDAR A1. The recommended power input for a Raspberry Pi 4 Model B is 5V with 3A (15W), but a 2.5A supply can be used

if downstream components do not exceed 500mA [11]. The Slamtec RPLIDAR A1 draws 5V with 100mA (0.5W).

The Create 2 robot platform has two power outputs of interest, the serial port which provides two power pins, and the main brush motor driver which is powered directly from the battery. The serial port is rated to provide 2W of power and the main brush motor driver is rated to provide 17W of power [12]. The main brush motor driver requires a 2.2mH, 1.5A inductor since the motor driver is designed to power an inductive load. The power output from each of the sources (with the main brush motor driver connected to an inductor) was measured using a multimeter to verify the rated powers.

Table 4: Measuring the power outputs from the Create 2 robot

Power Source	**Rated Power (W)**	**Measured Power (W)**
Serial Port	2 (10V, 0.2A)	3.2 (16V, 0.2A)
Main Brush Motor Driver	17 (~12V, 1.45A)	16.94 (12.1V, 1.4A)

It can therefore be concluded that the Create 2 main brush motor driver should be used as a power source for the external components that require 15W of power for stable operation. The output can be converted to 5V, 3A using a DC to DC converter. Since the main brush motor driver power output can only be turned on when the robot is on, an external battery will be required to start the system.

The Create 3 robot platform has a USB-C power output rated at the required 5V, 3A of power. The power output from this output was measured using a multimeter to verify the rated power.

Table 5: Measuring the power outputs from the Create 3 robot

Power Source	**Rated Power (W)**	**Measured Power (W)**
USB-C Port	15 (5V, 3A)	15 (5V, 3A)

This shows that the USB-C output is as rated and can be used without issue to power the external components of the robot. The Create 3 therefore does not require any external power circuitry other than a USB-C cable to connect the robot to the Raspberry Pi.

The hardware design of the Create 2 and Create 3 systems, including hardware schematics, can be seen in Section 7.6.

6.1.3 Analyzing the Create 2 robot's existing behaviour

By Max Curkovic and Cassidy Pacada

Thirdly, the strength of the Create 2 robot's behaviour implemented by last year's team was evaluated. This included: wall finding, right turn, and a left turn. The team's testing showed that the robot's current ability to find the wall and follow it depends entirely on the distance of the robot from the wall. Testing showed that, once the robot was further than 30cm away from the wall, its wall-following capabilities were not sufficient. The team believed that this is an issue that a LiDAR sensor would be able to resolve (see Background, section 2.3).

Table 6: The ability of the Create 2 robot to find the wall and maintain it

Distance	Description (Missed, OK, GOOD, PERFECT)
10cm	Good
30cm	Good
50cm	Missed (sensor distance may not be high enough)
1m	Missed (sensor distance may not be high enough)

Similarly, the strength of the robot's right turns was tested. Overall, the right turns were solid. The robot can currently maintain a safe distance from a wall, and perform the right turn as it enters an intersection state. Trials 3 and 4 were suspected to also be an issue with the sensors, which was corrected when the LiDAR sensor was put in place.

Table 7: The ability of the Create 2 robot to make a successful right turn consistently

Trial #	Description (Missed, OK, GOOD, PERFECT)
1	Perfect
2	Good
3	Missed (too close to the wall)
4	Missed (too far from wall)

Next, the strength of the robot's left turns was tested. This is the Create 2 robot's biggest weakness: it was unable to perform a single left turn in any trial. The team believed that, once again, this stemmed from an issue of having two IR sensors, both on the right side of the robot. Once the LiDAR sensor was in place, the left turns became significantly more reliable.

Table 8: The ability of the Create 2 robot to make a successful left turn consistently

Trial #	Description (Missed, OK, GOOD, PERFECT)
1	Missed
2	Missed
3	Missed
4	Missed

Overall, it could be observed that the turn behaviour for the Create 2 robot needed a lot of improvement. The inaccuracy that is being seen in the behaviour is mostly due to the fact that the IR sensors are a bit unstable and have a small range. This is why the team was hoping to improve the behaviour using LiDAR. On the other hand, the robot's wall following behaviour is very stable whenever it is within a short distance of the wall meaning that the PID controller is quite effective and with a better input such as LiDAR the wall following behaviour will be perfect. As a result, the team re-implemented the IR distance module to use LiDAR but copied over the PID controller to improve the output.

6.1.4 Analyzing the beacon strengths

By Max Curkovic

The team also measured the strength of the beacons at different distances. There are no concerns or reliability issues with the current beacons, and they would suffice for the team's implementation of the autonomous mail delivery service robot.

Table 9: Measuring the beacons at known distances

Beacon ID	Beacon Mac Address	Distance	RSSI
1	E2:77:FC:F9:04:93	1m	-73
1	E2:77:FC:F9:04:93	10m	-90
2	EA:2F:93:A6:98:20	1m	-69
2	EA:2F:93:A6:98:20	10m	-87
3	FC:E2:2E:62:9B:3D	1m	-71
3	FC:E2:2E:62:9B:3D	10m	-88.4
4	E4:87:91:3D:1E:D7	1m	-67

4	E4:87:91:3D:1E:D7	10m	-87
5	EE:16:86:9A:C2:A8	1m	-71
5	EE:16:86:9A:C2:A8	10m	-86.2
6	D0:6A:D2:02:42:EB	1m	-91
6	D0:6A:D2:02:42:EB	10m	-85
7	DF:2B:70:A8:21:90	1m	-69
7	DF:2B:70:A8:21:90	10m	-93
8	FB:EF:5C:DE:EF:E4	1m	-61
8	FB:EF:5C:DE:EF:E4	10m	-89

In conclusion, the beacons all seem functional and have a really good range. Some beacons are certainly better than others - 1ED7 and 2190, in particular, are the most reliable. They can still be detected from a 10m range which was more than enough for the applications of this project; however, for the full scale of the project, more beacons will be required at different sections of the tunnels. The exact beacon placements can be seen in Figure 4.

6.1.5 Analyzing the Wifi strengths in the tunnels

By Max Curkovic and Bardia Parmoun

Finally, the strength of the Wi-Fi signal of different sections of the Carleton University tunnels were measured. This was done to allow the team to determine what is the best location for the various docking stations for the robot, and figure out estimates for the beacon locations.

Table 10: Measuring the strengths of the internet signal in Carleton University tunnels

Location	Wi-Fi Speed (None/Bad/Good)	Comments
Entrance to Athletics	Good	40-50 mbps.
Athletics-Tunnel A	None	Measured at intersection.
Nesbitt-Pigiarniq	None	Minimal Wi-Fi speed, 10 mbps.
Entrance to Maintenance	Good	35-40 mbps.
Nicol-B-A	Good	50-60 mbps (use the intersection)

Minto-Mackenzie	Bad	Minimal Wi-Fi (10 mbps)
St Patrick's-Residence Commons	None	Decent Wi-Fi (30-40 mbps)
Architecture-Canal	Good	50-60 mbps. Main area.
Mackenzie-Canal	Good	70-80 mbps. Main study area.
Tunnel E	None	No Wi-Fi in tunnel E
Tory-UC-Azrieli Entrance	Good	70-80 mbps.
Steacie Entrance	None	No service..
Richcraft-Loeb	None	No service.
Entrance to Steacie/Richcraft	Good	50-60 mbps.
Entrance to Herzberg	Good	70-80 mbps.
Entrance to Loeb	None	No Wi-Fi in tunnel F
Entrance to Southam	Bad	10-20 mbps.
Entrance to Library	Good	70-80 mbps. Main study area.

6.2 Beacon and Dock Station Placement

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

The final project utilizes numerous docking stations scattered across campus grounds. When placing the beacons and the docking stations, it was important to account for various factors such as the range of the beacons, Wi-Fi strengths, and possible traffic on campus. To overcome these issues, the team decided to group certain buildings together. This approach helps reduce the cost of the beacons and docking stations. It also ensured that the robot will always have reliable internet connections at its destinations so it can easily communicate with the web app. Since the team already established that the beacons are easily recognizable within a 10m radius, they can be easily detected in the tunnels. In addition, the robot's built-in wall following behaviour should be able to overcome the slight turns in the tunnel so there is no need to place any beacons there. However, it is important to have beacons at every intersection that that robot might encounter even if there is no destination there. This will allow the robot to pass through the intersection. Finally, the team considered a beacon for each docking station so the robot could recognize them upon getting close to them. This will allow the robot to easily dock. As previously explained, in order to simplify the map, the team is only concerned with the main routes in the tunnels. As such, the robot will have intersections where multiple main routes connect.

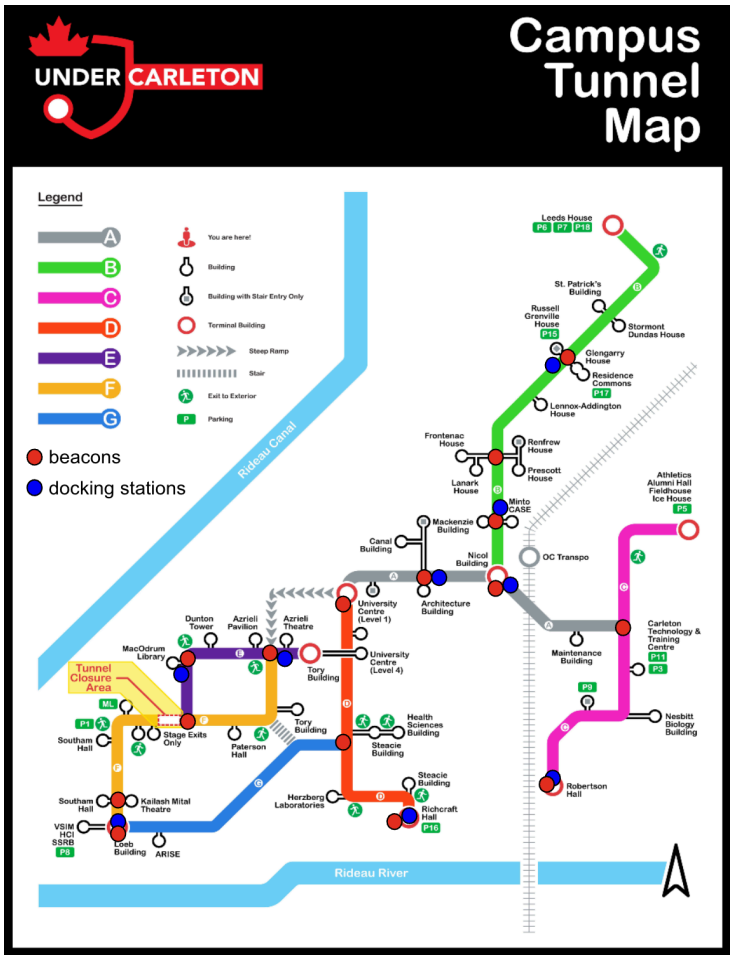


Figure 4: Proposed locations for the beacons and docking stations in the tunnels

6.2.1 Placement of Docking Stations

By Bardia Parmoun and Max Curkovic

As shown in the diagram, the beacons used to identify the docking stations will also double as intersection indicators for the robot. In other words, upon reaching the intersection, the robot will check to see if it needs to dock using its built-in map and the ID of the docking station (or the beacon). If not, the robot will only note the intersection. Based on the Wi-Fi analysis that the team conducted, it can be guaranteed that the robot will have a reliable connection at every place that a docking station has been placed. The team believes that docking stations should be placed in the following locations:

1. Residence common intersection for St. Patrick's Building and Residence Commons.
2. In front of Minto entrance, Minto Building.
3. At the Nicol Building intersection for the Nicol Building.
4. Near Robertson Hall for Maintenance, CTTC, and Nesbitt Buildings and Robertson Hall.
5. In front of the Architecture building for Mackenzie, Canal, and Architecture buildings.

6. In front of Richcraft hall for Steacie, Health Sciences, Herzberg, and Richcraft buildings.
7. At the tunnel E/F intersection for the Tory and Azrieli Buildings and University Centre
8. In front of the MacOdrum library for the library and Dunton Tower
9. In front of the Loeb building for Southam Hall, Loeb Building, and Paterson Hall.

6.2.2 Placement of Beacons

By Bardia Parmoun and Max Curkovic

In each individual tunnel, there should be numerous beacons to assist the movement of the robot as it navigates. The team intended to place one beacon at each intersection so the robot can detect if it has approached the intersection. The robot also utilizes the previous beacon that the robot passed to figure out exactly where the robot is entering the intersection from. This implementation is known as dynamic navigation, which is discussed more in detail later in the report.

The team believes that beacons should be placed in the following locations:

- Tunnels A and C intersection
- Tunnels A and B intersection
- Frontenac and Renfreew house intersection
- Tunnels A and D intersection
- Tunnel E and long ramp intersection: 3 beacon
- Tunnels F and E intersection
- Tunnels D and G intersection
- Southam Hall and Kailash Mital Theatre intersection

This brings the total number of required beacons to (8 + 9) 17.

7 Design

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

This section outlines all of the functional and non-functional requirements for the robot, as well as each individual use case and the overall use case diagram.

7.1 Requirements

By Max Curkovic and Bardia Parmoun

Below is a list of all functional and non-functional requirements that the robot should accomplish in order to satisfy the project objectives.

7.1.1 Functional Requirements

- **R1:** The robot shall be able to send mail from one destination to another using the Carleton University tunnels.
- **R2:** The robot shall be able to retrieve mail from one destination to another using the Carleton University tunnels.
- **R3:** The robot shall be able to navigate the tunnels with great precision.
- **R4:** The robot shall be able to notify the systems of its status.
- **R5:** The robot shall be able to handle any collisions with the people and objects in the tunnels.
- **R6:** The robot shall be able to communicate with the Bluetooth beacons installed in the tunnels.
- **R7:** The system should allow the administrator to monitor its behavior by providing detailed logs.
- **R8:** The robot shall be able to dock itself upon reaching its destination or whenever it is low on battery.
- **R9:** The system should allow the users to make delivery requests.
- **R10:** The system shall notify the robot of new requests and provide it with a path.
- **R11:** The system shall notify the user with information regarding their deliveries.

7.1.2 Non-Functional Requirements

- **R12:** The robot control software shall be interoperable with the web display application.
- **R13:** The robot shall be able to travel at a speed of at least 0.15m/s through Carleton University's tunnels.
- **R14:** The beacons shall be able to be interfaced with by the robot within 14.2 meters.
- **R15:** The robot's battery shall maintain at above 50 percent while it performs a delivery.
- **R16:** The robot shall ensure that only intended recipients can access mail.
- **R17:** The project shall remain within the specified $500 budget.

- **R18:** The expected features of the project, as proposed in the timeline, shall be completed on their respective dates.
- **R19:** The web application shall only be accessible for users connected to the Carleton University Wi-Fi or VPN.

7.2 Use Cases

By Max Curkovic and Bardia Parmoun

For each functional requirement, a use case was designed in order to illustrate how each requirement will be performed, whether by the robot, the system, or the user.

7.2.1 Use Case Diagram

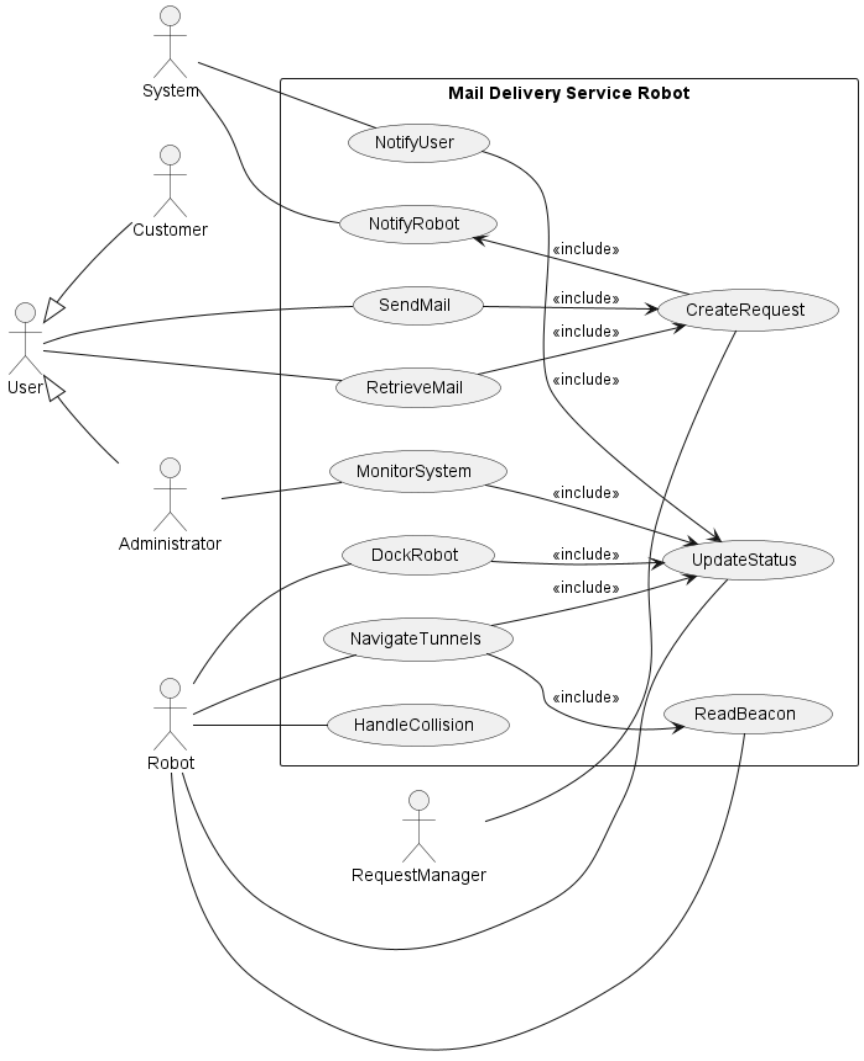


Figure 5: The use case diagram for the Mail Delivery Robot System

7.2.2 Individual Use Cases

Table 11: Send Mail use case

Use Case Name	Send Mail
Brief Description	The user sends a request to send mail to a destination.
Primary Actor	User
Secondary Actor	RequestManager
Precondition	The system is running, the robot is turned on, and the beacons are installed.
Dependency	INCLUDE USE CASE Create Request
Basic Flow	Steps: 1. The sender opens a request to send mail. 2. The RequestManager receives and validates the request. 3. RequestManager asks for an available robot. 4. The sender places the package in the robot. 5. The system notifies the recipient of the mail. **PostCondition:** Mail is in the robot being delivered.
Specific Alternative Flows	RFS Basic Flow 2. 1. IF There are no available robots THEN the request will be placed in a pending queue ENDIF **PostCondition:** Request will be placed in a pending queue.

Table 12: Retrieve Mail use case

Use Case Name	Retrieve Mail
Brief Description	The user sends a request to retrieve mail from the robot.
Primary Actor	User
Secondary Actor	RequestManager
Precondition	The system is running, the robot is turned on, and the beacons are installed.
Dependency	INCLUDE USE CASE Create Request
Basic Flow	Steps: 1. Users (sender and recipient) are notified by the System that mail has arrived at the destination. 2. The recipient retrieves the mail from the robot. **Postcondition:** The recipient User has successfully received mail from the Robot, sent by the sender User.
Specific Alternative Flows	RFS Basic Flow 2. Steps: 1. IF The Recipient User does not retrieve the mail from the robot THEN record the error and provide the robot status ENDIF **Postcondition:** The robot returns to its home dock and proceeds with other requests.

Table 13: Navigate tunnels use case

Use Case Name	Navigate Tunnels
Brief Description	The robot is able to navigate the tunnels based on its given path. This includes navigating various intersections and turns.
Primary Actor	Robot
Secondary Actor	System
Precondition	The system is running, the robot is turned on and the beacons are installed.
Dependency	INCLUDE USE CASE Read Beacon INCLUDE USE CASE Update Status
Basic Flow	Steps: 1. The system provides the robot with a destination and a navigation path. 2. The robot uses its sensors to receive information from its surroundings. 3. The robot uses the beacon sensors to determine its required next action. 4. The robot calculates its next move 5. The robot reaches its destination. **Postcondition:** The robot successfully navigated the hallway.
Specific Alternative Flows	RFS Basic Flow 2. Steps: 1. IF The robot is unable to detect the sensor THEN update the system ENDIF **Postcondition:** The robot notifies the system.
Specific Alternative Flows	RFS Basic Flow 3. Steps: 1. IF The robot is unable to detect the beacons THEN update the system ENDIF **Postcondition:** The robot notifies the system.

Table 14: Update Status use case

Use Case Name	Update Status
Brief Description	The robot is able to send a status update to the System.
Primary Actor	Robot
Secondary Actor	System
Precondition	The system is running, the robot is turned on, the beacons are installed and a delivery must be in progress.
Dependency	N/A
Basic Flow	Steps: 1. The sender user requests the System for a status update. 2. The robot sends a positional update, as well as a delivery status, to the System. 3. The system notifies the sender user. **Postcondition:** The sender user and the System are now both aware of the delivery status.
Specific Alternative Flows	RFS Basic Flow 3. Steps: 1. IF there is no Internet in the Carleton tunnels, THEN save it as a cache and try again ENDIF. **Postcondition:** The sender user will still be updated on the delivery status.

Table 15: Handle Collisions use case

Use Case Name	Handle Collisions
Brief Description	The Robot collides with an object (wall, cart, etc.) and re-orients itself.
Primary Actor	Robot
Secondary Actor	N/A
Precondition	The system is running, the robot is moving through the tunnels.
Dependency	N/A
Basic Flow	Steps: 1. The Robot collides with an object. 2. The Robot bumpers detect that a collision has occurred. 3. The Robot reverses. 4. The Robot turns away from the object and re-orients in the right direction. **Postcondition:** The robot has successfully navigated through the collision.
Specific Alternative Flows	N/A

Table 16: Read Beacons use case

Use Case Name	Read Beacons
Brief Description	The robot detects a beacon signal traveling through the tunnels.
Primary Actor	Robot
Secondary Actor	N/A
Precondition	The system is running and the robot is moving.
Dependency	N/A
Basic Flow	Steps: 1. The robot detects the beacon and determines what intersection it is coming from. 2. The robot logs the information on the system. **Postcondition:** The robot has successfully provided the system with its beacon data.
Specific Alternative Flows	N/A

Table 17: Monitor System use case

Use Case Name	Monitor System
Brief Description	An administrator monitors the current state of the system.
Primary Actor	Admin
Secondary Actor	Robot
Precondition	The system is running, and the robot is operational.
Dependency	INCLUDE USE CASE Update Status
Basic Flow	Steps: 1. USE CASE Update Status is initiated. 2. The administrator requests to view the current state of the system. 3. The system replies with the current state. **Postcondition:** The administrator is informed of the current system state.
Specific Alternative Flows	RFS Basic Flow 2. Steps: 1. IF the system is not running OR the robot is not operational, THEN send an error ENDIF **Postcondition:** The administrator is informed of the current system state.

Table 18: Dock Robot use case

Use Case Name	Dock Robot
Brief Description	The robot is near a docking station and determines that it needs to dock.
Primary Actor	Robot
Secondary Actor	N/A
Precondition	The system is running and a docking station has been encountered.
Dependency	INCLUDE USE CASE Update Status
Basic Flow	Steps: 1. The robot gets near a docking station. 2. The robot determines that it needs to dock at the specific docking station. 3. The robot mounts the docking station. 4. The robot updates the system about its behavior. **Postcondition:** The robot is charging and the system is notified.
Specific Alternative Flows	RFS Basic Flow 3. Steps: 1. IF The robot is unable to mount the docking station THEN send an error ENDIF **Postcondition:** The system is notified of the error.

Table 19: Create Request use case

Use Case Name	Create Request
Brief Description	The request manager creates and validates a request for the sender user.
Primary Actor	RequestManager
Secondary Actor	Robot
Precondition	The system is running, and the robot is operational.
Dependency	INCLUDE USE CASE NotifyRobot
Basic Flow	Steps: 1. The sender user creates a request to the request manager to send or receive mail. 2. The request manager validates the user's request. 3. USE CASE Notify Robot is initiated to inform the robot of the request. **Postcondition:** The robot is notified of the request and will proceed to fulfill it.
Specific Alternative Flows	RFS Basic Flow 3. Steps: 1. IF the robot is not operational, THEN inform the user to try again at a later time. ENDIF **Postcondition:** The sender user is notified that the robot is not currently operational and will have to wait in order to put in a request.

Table 20: Notify Robot use case

Use Case Name	Notify Robot
Brief Description	The system notifies the robot of new updates. This includes new requests, navigation data, etc.
Primary Actor	System
Secondary Actor	Robot
Precondition	The system is running and the robot is operational.
Dependency	N/A
Basic Flow	Steps: 1. The system prepares a new update for the robot. 2. The system sends the update to the robot. **Postcondition:** The robot receives the update and acts accordingly.
Specific Alternative Flows	RFS Basic Flow 2. Steps: 1. IF The system is unable to contact the robot THEN inform the user and terminate the request ENDIF **Postcondition:** The system notifies the user of the error.

Table 21: Notify User use case

Use Case Name	Notify User
Brief Description	The user is notified of any updates to the robot, their delivery, or the system itself.
Primary Actor	System
Secondary Actor	Robot
Precondition	The system is running and the robot is operational.
Dependency	INCLUDE USE CASE Update Status
Basic Flow	Steps: 1. USE CASE Update Status is initiated. 2. The system posts the updated results for the user. **Postcondition:** The user is now informed of any updates to the robot.
Specific Alternative Flows	RFS Basic Flow 2. Steps: 1. IF The system is unable to contact the robot THEN inform the user ENDIF **Postcondition:** The system notifies the user of the error.

7.3 Metrics

By Bardia Parmoun and Cassidy Pacada

The following section details the metrics which were used to evaluate the system:

- **M1:** The system should cover all the major tunnel paths: A, B, C, D, E, F, and G:

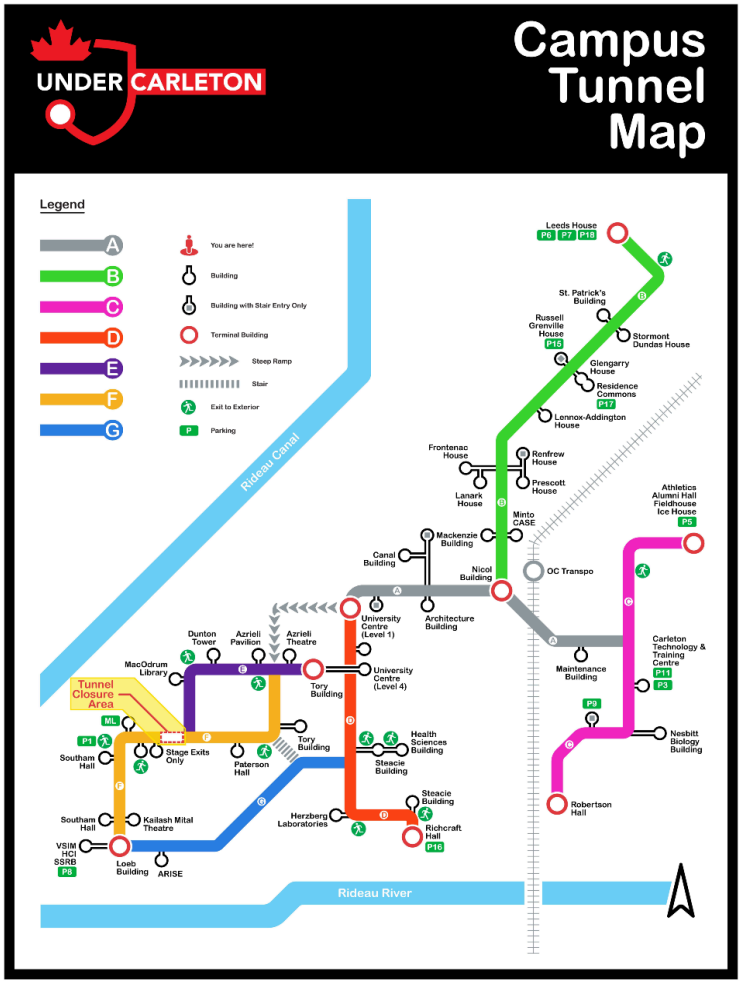


Figure 6: A map of the Carleton University tunnels

To make the implementation easier for this iteration of the project, the robot would only traverse the mentioned paths, meaning it would only recognize the major intersections and avoid the smaller detours along the path. For example, if the robot has deliveries with destinations at the Canal and Mackenzie buildings, it will only dock at the entrance of the detour leading to these buildings on path A.

- **M2:** The robot speed should be at least 0.2m/s and delivery time at most an hour.

- **M3:** The robot shall be able to deliver envelopes of size A3: 29.7cm x 42 cm

- **M4:** The robot shall be able to deliver packages with a total weight of 1 kg or less.

7.4 State Machine

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

This section details the structure of the state machine for the project. The state machine is has undergone significant restructuring over the duration of the project, as a result of numerous new states and events that have been implemented. The main reason for the shift was the fact that the team observed the real time nature of the system which meant various sources of inputs and events could occur simultaneously.

7.4.1 List of States, Events, and Actions

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

Sources of Inputs:

The system contains four separate sources of inputs: the bumper sensor, the beacons, the LiDAR sensor, and any possible errors. Together these will help determine the states for the system. The team believes that a state in the system is simply a snapshot of the status of all these events. This ensures that the states are completely independent and account for every single combination of events in the system.

States:

- OPERATIONAL: The overall state containing all the valid states that the robot can have.
    - NO_DEST: The robot is moving but has not received a beacon.
    - SHOULD_TURN_LEFT: The robot is moving and detected a beacon to turn left.
    - SHOULD_TURN_RIGHT: The robot is moving and detected a beacon to turn right.
    - SHOULD_U_TURN: The robot is moving and detected a beacon to perform a U-Turn.
    - SHOULD_PASS: The robot is moving and has received a beacon to pass.
    - SHOULD_DOCK: The robot has received a signal to dock.
    - HANDLE_INTERSECTION: The robot has arrived at an intersection.
    - DOCKED: The robot has reached a docking station.
    - COLLISION_NO_DEST: The robot received a collision with no signal.
    - COLLISION_TURN_LEFT: The robot received a collision with a left signal.
    - COLLISION_TURN_RIGHT: The robot received a collision with a right signal.
    - COLLISION_U_TURN: The robot received a collision with a U-Turn signal.
    - COLLISION_PASS: The robot received a collision while having a pass signal.
    - COLLISION_DOCK: The robot received a collision while having a dock signal.

    - COLLISION_INTERSECTION: The robot received a collision at an intersection.
- NOT_OPERATIONAL: The robot has encountered a fatal error and has stopped.

Events:

Since this is a real time system, it is expected that all the different inputs can occur at the same time. As a result, an event in the state machine is a snapshot of these input values.

Table 22: List of the transitions for the state machine based on the different inputs.

TRANSITION	ERROR	BUMPER	BEACON	LiDAR
1	FALSE	FALSE	NONE	FALSE
2	FALSE	FALSE	NONE	TRUE
3	FALSE	FALSE	LEFT	FALSE
4	FALSE	FALSE	LEFT	TRUE
5	FALSE	FALSE	RIGHT	FALSE
6	FALSE	FALSE	RIGHT	TRUE
7	FALSE	FALSE	PASS	FALSE
8	FALSE	FALSE	PASS	TRUE
9	FALSE	FALSE	U-TURN	FALSE
10	FALSE	FALSE	U-TURN	TRUE
11	FALSE	FALSE	DOCK	FALSE
12	FALSE	FALSE	DOCK	TRUE
13	FALSE	TRUE	NONE	x
14	FALSE	TRUE	LEFT	x
15	FALSE	TRUE	RIGHT	x
16	FALSE	TRUE	PASS	x
17	FALSE	TRUE	U-TURN	x
18	FALSE	TRUE	DOCK	x
19	TRUE	x	x	x

As shown in the table, the different sensors in the system could have different values. The bumper sensor used for collision detection could either be activated (TRUE) or deactivated (FALSE). Similarly, the beacons could trigger six separate navigation events (NONE, LEFT, RIGHT, U-TURN, PASS, and DOCK). Finally, the LiDAR sensor could either indicate if a wall exists (TRUE) or not (FALSE). This table summarizes every combination of these inputs and marks them as transitions for the state machine. For certain transitions, some inputs are marked with x, meaning their value is not important in the transition. This was done to simplify the transitions in the state machine. As illustrated in the state machine diagram, every state is expected to account for every single one of these transitions. This will ensure that the system is properly defined and there are no sneak paths.

Actions:

- WALL_FOLLOW: The robot will move at the angle given by the lidar sensor at 0.4m/s.
- L_TURN: The robot will make a 30 degree left turn with a 0.4m/s speed.
- U_TURN: The robot will make a 180 degree reverse with a 0.4m/s speed.
- R_TURN: The robot will make a 90 degree right turn with a 0.4m/s speed.
- FORWARD: The robot will go forward with no angle change with a 0.4m/s speed..
- DOCK: The robot will go to the docking station.
- UNDOCK: The robot leaves the docking station.
- KILL: Stops the system.

To better demonstrate the relations between the various states, a state machine diagram was prepared as follows. Please note that in order to facilitate reading the state machine, the diagram was split into two separate diagrams. The first diagram shows all the events that do not involve collisions namely T1-T12 and T19. The second diagram shows the remaining T13-T18.

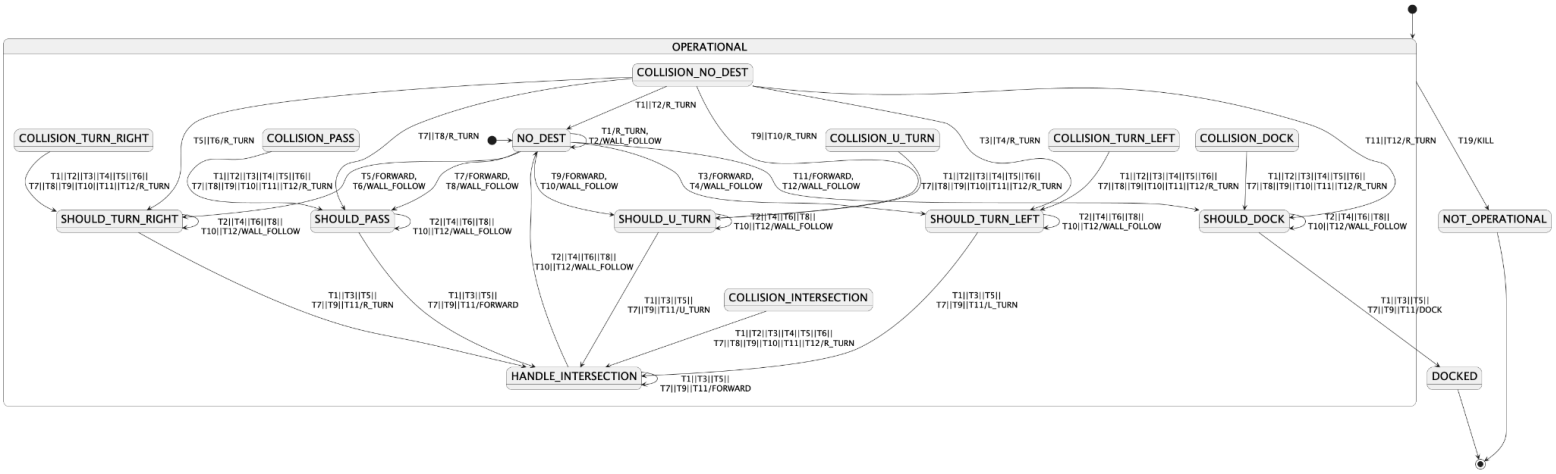


Figure 7: The state machine for the system without the collision events

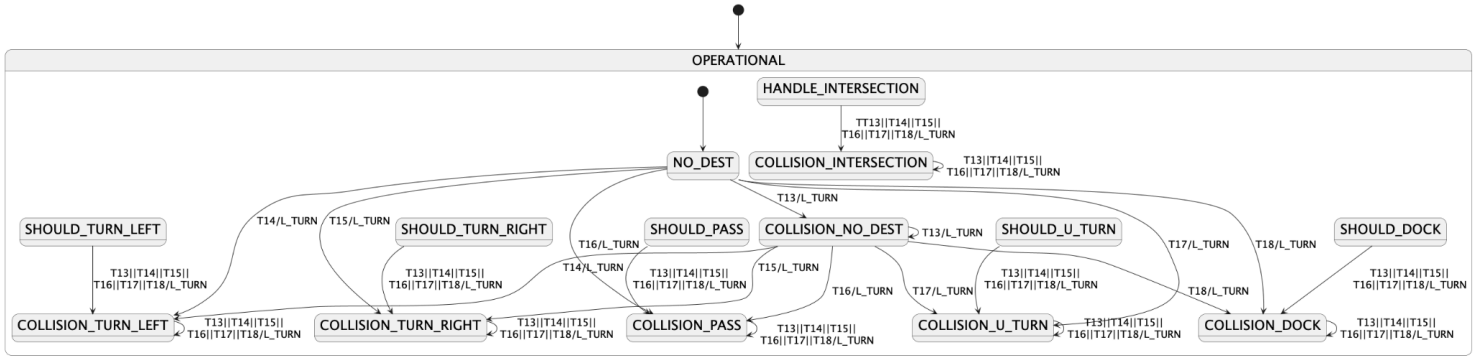


Figure 8: The state machine for the system with the collision events

7.4.2 Justifications

By Max Curkovic and Bardia Parmoun

The typical flow of the system:

The robot starts at the NO_DEST state until it receives a navigation notification. Based on the navigation notification that it gets, the robot moves to one of the following states: SHOULD_DOCK for a docking, SHOULD_PASS for pass, SHOULD_TURN_LEFT for turning left, SHOULD_U_TURN for a U-Turn and SHOULD_TURN_RIGHT for turning right. Afterwards, from the SHOULD_PASS, SHOULD_TURN_LEFT, SHOULD_U_TURN and SHOULD_TURN_RIGHT states the robot will enter the HANDLE_INTERSECTION state upon losing the wall. These transitions will be done with their proper actions: FORWARD, L_TURN, U_TURN and R_TURN accordingly. During the HANDLE_INTERSECTION, the robot will keep going FORWARD until it has found a wall. Finally, when the robot gets a wall, it will go back to the NO_DEST event to start a different intersection. Similarly, when the robot loses the wall at a SHOULD_DOCK state, the robot will go to the DOCKED state which concludes its trip. Note that WALL_FOLLOW is an expected behaviour for the NO_DEST, SHOULD_TURN_LEFT, SHOULD_U_TURN, SHOULD_TURN_RIGHT, SHOULD_PASS, and SHOULD_DOCK states since the robot is expected to be following a wall during those states. In other words, during the states, if the robot gets a transition which has a wall, it will perform wall following. Also note that events that include losing the wall should not be expected for the NO_DEST state. In the case that it occurs, the robot is instructed to turn right (in case this event was triggered by the robot getting too far from the wall accidentally). Finally, it is important to mention that at any time an event with error can lead the robot to enter NOT_OPERATIONAL with a KILL action.

Collision Handling:

In order to preserve the expected direction for the robot, a collision state has been dedicated for every state that can have a collision. Those states are as follows: COLLISION_NO_DEST, COLLISION_DOCK, COLLISION_U_TURN, COLLISION_TURN_RIGHT, COLLISION_PASS, COLLISION_INTERSECTION, and COLLISION_TURN_LEFT. The typical flow of collision handling is the same for all of these states. After the robot receives an event with a bumper sensor, it will do a L_TURN and enter its specific collision state. The U_TURN follows very similar logic to that of a L_TURN with a bumper sensor event. Once it receives an event without the bumper, it will do a R_TURN to get back to its original direction and will go back to the state that called it. The idea is that through these constant left and right turns the robot will slowly move past that obstacle while always maintaining its original direction and angle. Please note that this method of collision detection only works for the obstacles that are either attached to the wall on one side or are removed after the collision (ie a human). With the current design, if there is empty space all around a static obstacle, the robot will constantly circle around it. This will be addressed in future designs. Finally, the current design of the robot is capable of detecting any obstacle in front of it and

performing a turn accordingly. This means it should be capable of avoiding most obstacles that are in front of it, by default.

7.5 Deployment Diagram

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

As previously mentioned, the source code for the project is primarily written in ROS. As such, every component for the project was developed as a separate ROS node. This helped keep the various components fully decoupled and independent from each other which facilitates testing. This section details the structure of the ROS nodes within the project, all connected with the external components such as the sensors and the web application.

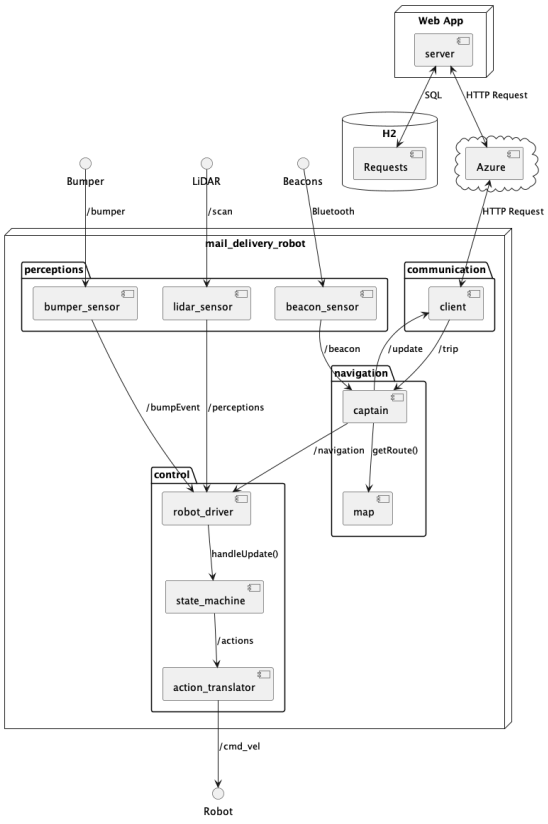


Figure 9: The deployment diagram for the system

7.5.1 List of Nodes, Messages and Entities

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

Nodes:

- client: Defines a node which describes the client.
- captain: Defines a node which describes the captain - controlling the robot en route.
- beacon_reader: Describes a node which allows for reading the detected beacons.
- lidar_sensor: Describes a node for the LiDAR sensor.

- bumper_sensor: Describes a node for the bumper sensor.
- robot_driver: Describes a node for the robot driver, which takes in state machine actions.
- Action_translator: Describes a node for the action translator, which translates state machine actions into valid serial commands.

Paths:

- /trip: Sends a trip message from the client node to the navigation_plot node.
- /update: Sends an update message from the captain node to the client node.
- /navigation: Sends a navigation message from the captain node to robot_driver node.
- /actions: Sends an action message from the robot_driver node to the action_translator node.
- /cmd_val: Valid command that represents a state machine action sent to the robot.
- /bump_event: Bumper sensor event message sent to the robot_driver node.
- /perceptions: LiDAR sensor event message sent to the robot_driver node.
- /beacon: Beacon reader message sent to the captain node.

Entities:

- Server: Defines the web server used in the web application.
- Tunnel_map: Defines a map of the tunnels.
- Route: Defines a route for the robot to take.
- Beacons: Defines beacons used by the robot when traveling.
- LiDAR sensor: Defines the LiDAR sensor used on the robot.
- Bumper sensor: Defines the bumper sensors used on the robot.
- State_machine: Defines the state machine used by the robot.

7.5.2 Justifications

By Max Curkovic

The typical flow:

The above figure depicts the proposed node structure, which would be detailed using ROS. Essentially, a client node would receive a request, which would then be sent through the captain during the trip action. The captain node utilizes a beacon reader node, which aids in sensing the direction of the robot, and utilizes a robot driver to assist in sensing for obstacles. The state machine entity (see Figure 3) uses an action translator which tells the robot what state to enter. The navigation_plot node utilizes a tunnel map entity to assist in the robot's proper navigation through the tunnels. The beacons assist in ensuring that the robot follows this particular route. There are two sensor nodes that are accounted for: the LiDAR sensor and the bumper sensor. The LiDAR sensor sends perception messages to the robot driver depending on where the robot is on its path. The bumper sensor should recognize if the robot has hit a wall, and should perform the expected "handle collision" state.

7.6 Hardware Design
By Matt Reid

The project hardware is built on the iRobot Create platform including the first iteration of the robot based on the Create 2 platform as well as a new second iteration of the robot based on the Create 3 platform. The three main hardware components used for the robots are an iRobot Create 2 or 3, a Raspberry Pi 4B, and a Slamtec RPLIDAR A1. The iRobot Create is the robot platform used as the base for the project (see Background, Section 3.2), which provides power to the Raspberry Pi 4B that controls and powers the Slamtec RPLIDAR A1. The LiDAR plugs into the USB port of the Pi in order to receive power and send data. The communication between the Pi and the iRobot Create robot is dependent on the version of the robot (2 or 3).

7.6.1 Create 2 Hardware Design
By Matt Reid

The Create 2 has a UART Serial port for receiving commands and sending sensor data. For the robot, the Create 2 is connected to the Raspberry Pi 4B microcontroller using a provided Serial-to-USB cable.

As found in the power output analysis done in Section 6.1.2, the Create 2 provides approximately 17W of power from its main motor brush driver which can be converted to the 5V, 3A (15W) input needed by the Pi to power itself and the LiDAR. In order to provide the power required from the main motor brush driver, a 2.2mH, 1.4A inductor is required since the circuit is designed to power the inductor vacuum brush motor. The motor brush driver only turns on once the motor has started, so it is connected to a power bank which starts the robot and charges while the robot drives (similar to a car battery). The hardware schematic showing the power and data connections between components for the Create 2 is as follows:

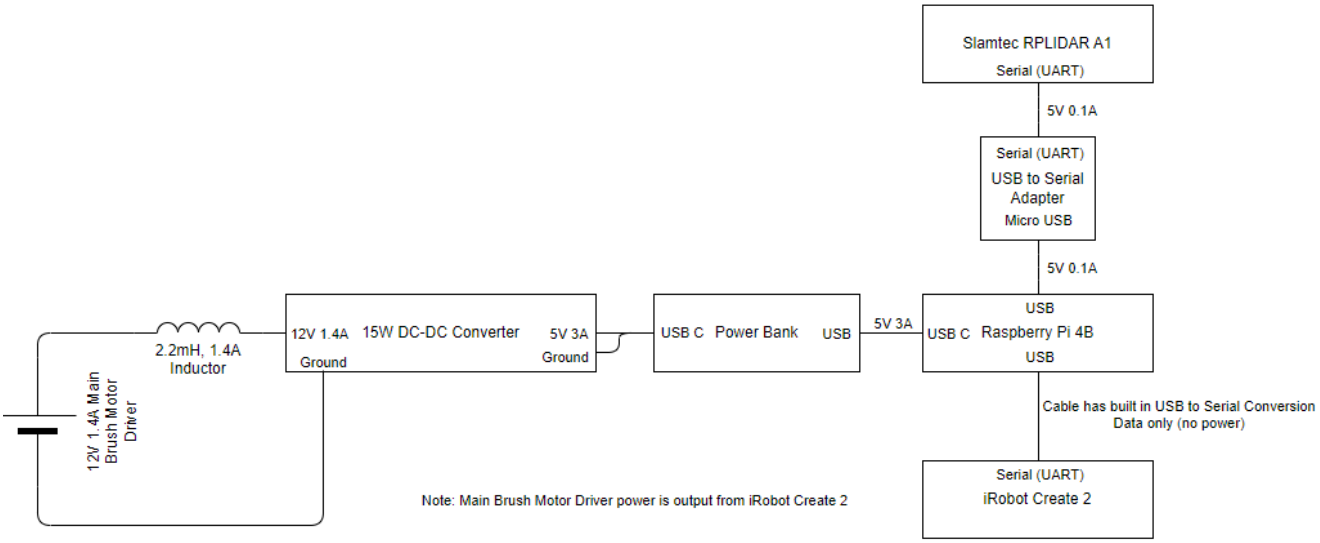


Figure 10: Autonomous Mail Delivery Robot Hardware Schematic (Create 2)

7.6.2 Create 3 Hardware Design
By Matt Reid

The Create 3 has a USB-C OTG port which can communicate with and power the Pi at the same time. It has built in ROS2 support providing a node and command topics to the Raspberry Pi for easier integration. This means that instead of having a ROS node on the Pi sending serial commands to the robot, the robot instead is the ROS node that receives the ROS actions from the state machine. The power output is 5V, 3A, which is exactly as required by the Raspberry Pi, so no external power circuitry is required. The USB-C OTG port allows for a network connection to be made between the Create 3 and the Pi so that ROS2 can be used to communicate between the nodes.

The hardware schematic showing the power and data connections between components for the Create 3 is as follows:

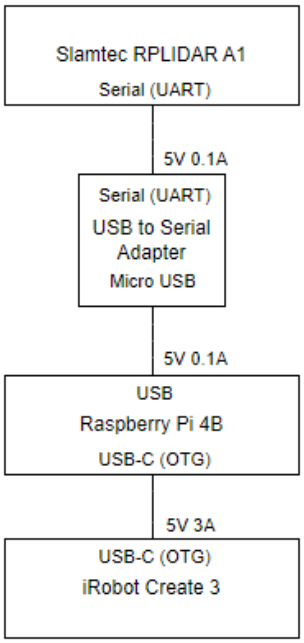


Figure 11: Autonomous Mail Delivery Robot Hardware Schematic (Create 3)

7.7 Navigation Design
By Max Curkovic

Navigation is a core element of the robot. In order to successfully satisfy many of its non-functional requirements, it must be able to travel from Point A to Point B with minimal issues. In order for this to be able to occur, the robot must invoke the use of dynamic navigation to be able to navigate its way through the tunnels and follow its correct path to its destination. This is a major shift from the original proposal, as the robot's use of LiDAR now makes dynamic navigation a viable possibility to control its paths.

Generally, to allow the robot to navigate, multiple beacons must be placed at each branch of an intersection. For example, an intersection with four possible paths would require four beacons. This gives the robot more coverage and be able to detect the path significantly easier than with a singular beacon at each point. The exact beacon placements can be found in Figure 3.

7.7.1 Dynamic Navigation Design
By Matt Reid

Dynamic navigation means that every intersection is aware of the robot's destination. It is considered "dynamic" because the starting point is changing constantly depending on the robot's location. The robot always tries to go from Point A to Point B, where Point A changes depending on the intersection it is closest to, and Point B is always the final destination. If it enters a particular intersection, that intersection must be able to take an input destination, and be able to update the robot's map based on the given input. This intersection would be considered Point A.

The figure below depicts a general flow of how dynamic navigation works:

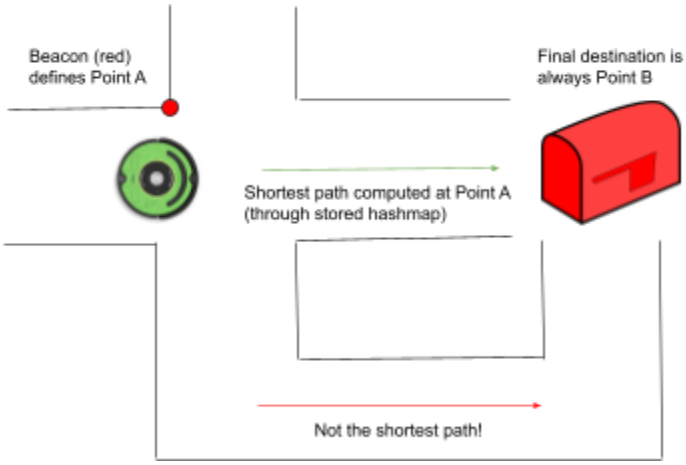


Figure 12: Demonstrating Dynamic Navigation

The team has discovered that it is not necessary to implement an algorithm that calculates the shortest path, such as Dijkstra, due to the sheer linearity within the tunnels. An algorithm can be implemented at a later date to account for any route closures, but it is not required at this time. Therefore, the team has developed a routing table that will act as the robot's hard-coded map. This map is parsed by the navigation parser into a hashmap of locations. The key is the starting location (which has multiple entries depending on the orientation of your arrival at the location) and the value is another hashmap that consists of the possible destinations and their required directions that are passed into the robot's captain.

	Loeb	Southam	TunnelJunction	Library	Azrieli/Tory	Steacie	Richcraft	UC	Canal	Nicol	Mackenzie/Minto	Frontenac	ResComms	CTTC	Robertson
Loeb1	DOCK	U-TURN	U-TURN	U-TURN	U-TURN	LEFT	LEFT	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
Loeb2	DOCK	RIGHT	RIGHT	RIGHT	RIGHT	U-TURN	U-TURN	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT
Southam1	FORWARD	DOCK	U-TURN	U-TURN	U-TURN	FORWARD	FORWARD	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
Southam3	U-TURN	DOCK	FORWARD	FORWARD	FORWARD	U-TURN	U-TURN	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
TunnelJunction1	RIGHT	RIGHT	DOCK	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
TunnelJunction2	FORWARD	FORWARD	DOCK	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT
TunnelJunction4	U-TURN	U-TURN	DOCK	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT
Library2	FORWARD	FORWARD	FORWARD	DOCK	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
Library3	U-TURN	U-TURN	U-TURN	DOCK	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
Azrieli/Tory1	RIGHT	RIGHT	RIGHT	RIGHT	DOCK	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
Azrieli/Tory3	LEFT	LEFT	LEFT	LEFT	DOCK	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
Azrieli/Tory4	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT
Steacie1	RIGHT	RIGHT	U-TURN	U-TURN	U-TURN	DOCK	FORWARD	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
Steacie3	LEFT	LEFT	FORWARD	FORWARD	FORWARD	DOCK	U-TURN	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
Steacie4	U-TURN	U-TURN	LEFT	LEFT	LEFT	DOCK	RIGHT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT
Richcraft1	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
UC2	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	FORWARD	FORWARD	DOCK	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
UC3	LEFT	LEFT	LEFT	LEFT	LEFT	U-TURN	U-TURN	DOCK	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
UC4	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	RIGHT	RIGHT	DOCK	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT
Canal2	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD
Canal4	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN
Nicol1	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	DOCK	U-TURN	U-TURN	U-TURN	LEFT	LEFT
Nicol2	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK	RIGHT	RIGHT	RIGHT	U-TURN	U-TURN
Nicol4	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	LEFT	LEFT	LEFT	RIGHT	RIGHT
Mackenzie/Minto1	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK	U-TURN	U-TURN	U-TURN	U-TURN
Mackenzie/Minto3	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	FORWARD	FORWARD	FORWARD	FORWARD
Frontenac1	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK	U-TURN	U-TURN	U-TURN
Frontenac3	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	FORWARD	FORWARD	FORWARD
ResComms1	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK	U-TURN	U-TURN
ResComms3	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	FORWARD	FORWARD
CTTC1	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	DOCK	FORWARD
CTTC3	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	LEFT	DOCK	U-TURN
CTTC4	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	U-TURN	DOCK	RIGHT
Robertson1	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	FORWARD	DOCK

Figure 13: Routing table for the robot's navigation

With this routing table, the robot is also required to understand which orientation it must face in correlation with the beacon placements. Very similarly to the routing table, a hashmap was created with the orientations of the beacons that the robot would have to follow in order to actually execute these directions when it picks up the beacon ID: the key, again, is the starting destination, and the value is a hashmap of all possible destinations and their orientations. Note that, for each beacon, the robot only requires the adjacent beacons to determine their orientation, leading to all non-adjacent beacons using dashes as their entries.

	Loeb	Southam	TunnelJunction	Library	Azrieli/Tory	Steacie	Richcraft	UC	Canal	Nicol	Mackenzie/Minto	Frontenac	ResComms	CTTC	Robertson
Loeb	0	1	-	-	-	2	-	-	-	-	-	-	-	-	-
Southam	3	0	1	-	-	-	-	-	-	-	-	-	-	-	-
TunnelJunction	-	4	0	1	3	-	-	-	-	-	-	-	-	-	-
Library	-	-	3	0	2	-	-	-	-	-	-	-	-	-	-
Azrieli/Tory	-	-	3	4	0	-	-	1	-	-	-	-	-	-	-
Steacie	4	-	-	-	-	0	3	1	-	-	-	-	-	-	-
Richcraft	-	-	-	-	-	1	0	-	-	-	-	-	-	-	-
UC	-	-	-	-	4	3	-	0	2	-	-	-	-	-	-
Canal	-	-	-	-	-	-	-	4	0	2	-	-	-	-	-
Nicol	-	-	-	-	-	-	-	-	4	0	1	-	-	2	-
Mackenzie/Minto	-	-	-	-	-	-	-	-	-	3	0	1	-	-	-
Frontenac	-	-	-	-	-	-	-	-	-	-	3	0	1	-	-
ResComms	-	-	-	-	-	-	-	-	-	-	-	3	0	-	-
CTTC	-	-	-	-	-	-	-	-	-	-	-	-	-	0	3
Robertson	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0

Figure 14: Beacon orientations in correlation with their respective IDs

Once the two hashmaps are created, the robot's captain determines its orientation from the retrieved beacon ID. As a result, it is able to detect where to go to reach the next beacon. Ultimately, the robot will be able to determine where exactly it is using this orientation, and be able to follow its corresponding routing table to determine where it needs to go (depending on

the current orientation) and the directions it needs to take to get there. This can be done at any Point A, so long as the robot remains within a beacon threshold in the tunnels at all times.

The implementation of dynamic navigation utilized the robot's captain in correlation with its respective map class. More details can be found in section 8.7.

8 Implementation

By Bardia Parmoun, Max Curkovic, and Matt Reid

This section describes the implementation of the robot, and details how each member's work on a particular aspect contributes to achieving the requirements of the robot.

8.1 Project Progress

By Max Curkovic, Cassidy Pacada, Matt Reid, and Bardia Parmoun

To evaluate all the design choices in the report, a minimum viable product was designed at the beginning of the project. Here are the use cases that were considered for the MVP:

1. The ability for the robot to reliably wall follow using LiDAR. This includes detecting and finding a wall and maintaining the distance with the wall using the PID controller. That is, if the robot starts off closer or further from the expected distance, it should autocorrect itself to account for the set distance.
2. The ability for the robot to reliably make left and right turns. This includes recognizing an intersection, making a turn, and returning to the wall following.
3. The ability for the robot to properly detect beacons. In this iteration, all the robot needs to do is to log the beacons it has seen. If possible, this behaviour can be further improved by having a test intersection and an expected beacon so that the robot will use the beacon to properly identify the intersection.
4. The system should implement the state machine proposed in Section 1.4 and act accordingly. For this iteration, the system could bypass certain states such as docking, charging, and collision handling and instead provide stubs for them.

Every week, the team focused on implementing the use cases while keeping track of the robot's current states. This method helped the team ensure that the robot's functionality is always tested. Here is the summary of the team's progress so far:

Table 23: A summary of all the MVPs implemented for Create 2 and Create 3

Create 2 Progress Summary	Create 3 Progress Summary	Completion Date
1. Implemented the initial version of the state machine and developed the general node structure. During this iteration the team also connected the LiDAR sensors and the Bluetooth beacons to the robot.	Robot not available	October 30th, 2023
2. In this iteration, the team updated the robot's PID controller to use the new wall following strategy using the algorithm specified in 8.3. The robot can now reliably wall follow from any distance.	Robot not available	November 7th, 2023
3. The team added the latest iteration of the state machine, specified in 7.4, to the project. The team also verified that the robot can now reliably handle multiple events at once.	Robot not available	November 14th, 2023
4. The robot's bumper sensor was connected to the project. The robot can now detect and avoid intersections as specified in 8.6. The team also attempted to implement intersection handling as specified in 8.5. There is still some work needed to reliably implement this task.	Robot not available	December 1st, 2023
5. The robot's state machine was improved to account for long actions. In other words, the state machine now sends a specific action multiple times. This improved the robot's existing wall following, intersection detection, and collision handling. The robot's collision handling was also improved to keep track of the left turns for better orientation.	Robot not available	January 24th, 2024
6. Fully implemented dynamic navigation for the system. The robot is now capable of detecting a given beacon and navigating the tunnels using it.	6. Transferred the given codebase to the Create 3. Attached the mailbox and the LiDAR chassis to the robot and confirmed that the robot is moving as expected. Started working on and undocking	February 21st, 2024
7. Created a simple proof of concept for the web app. Verified that the robot is able to communicate with the web app.	7. Started work on automatic docking and undocking for the Create 3. Split the codebase to support all version specific commands.	March 5th, 2024

8. Fully implemented the main functionalities of the website and connected it to the robots and the simulator. Verified that the robot can easily get its trips and send status updates.	8. Implemented automatic docking and undocking on the Create 3. Also added code to properly detect collisions using the bumper sensor.	March 12th, 2024

8.2 State Machine Implementation

By Bardia Parmoun

The state machine is a core part of the robot's behaviour. As such, it is important to provide a proper implementation for it that closely resembles the state machine diagram. A file named "state_machine.py" was created under the control module to encapsulate all the state behaviours. The state design pattern was utilized to implement the state machine. This allows the robot_driver class to simply include a state parameter that holds the current state of the robot. That state parameter will include an action_publisher which is the same as the action publisher for the robot_driver node. This action_publisher will be used by the state for publishing its actions. As a result, the robot_driver can simply send updates to the current state. Upon receiving each update, the current state will publish the corresponding action and will return a state (either itself or a new state indicating a transition).

As explained in 7.4.1, the robot has various sources of perception that can be triggered at various times. As such, the team decided to define events as a snapshot of these perception events. To achieve this, the robot_driver includes specific variables for each source of perception (LiDAR sensor, bumper sensor, and the beacon). These variables are constantly updated by the call back functions for each source of perception. To send an event to the state machine, the robot_driver includes a timer that will trigger every 0.1 seconds. Once the timer triggers, the value of all three variables are passed to the state machine. From there, the state will use these values to figure out which transition has been triggered and it will call it.

The state machine implementation was implemented in a way for repeating actions. This was done due to the nature of the Twist of command which the state machine actions rely on. Since the Twist command is a velocity (angular & linear), the robot requires it to be sent multiple times. This way the robot can continue with the same movement. To achieve this, a central clock was defined in the robot_driver. Every clock cycle, the robot_driver sends an update to the state machine with the current state of the system. The state machine itself keeps track of its long action and the number of required clock cycles for it. This way every update, the state machine checks to see if the long action is still in progress or not. If the long action is in progress, the state machine will simply ignore the update from the robot_driver and send the same action. It should be noted that although the state machine ignores the update, it will still remember critical information such as the last wall and navigation update. This way once the long action is done, the state machine will still have the important updates.

8.3 Implementing Wall Following

By Bardia Parmoun

The previous team for this project implemented a simple PID controller that was able to maintain a specific angle with the wall. This controller worked perfectly when the robot was close to the wall, however, the team noticed some issues when the robot was placed further from the wall. This was mostly due to the fact that the PID was only maintaining the distance with the wall by sending specific rotation commands for the *Twist* command in ROS. Since the PID was not aware of the current orientation of the robot, the effects of these rotation commands would easily compound and result in the robot completely missing the wall or doing a U-Turn.

To overcome this issue, the team came up with a much easier implementation that simply involved having the robot maintain a specific angle with the wall. To implement this solution, the team first finds the robot's angle and distance with the wall using the LiDAR sensor. This is achieved by reading the positive side of the sensor with a big range such as 60◦ to 170◦. The robot's angle with the wall is calculated as follows:

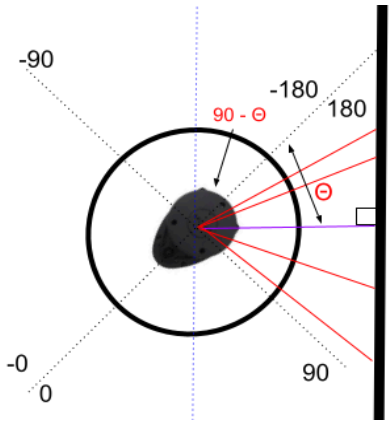


Figure 15: Calculating the robot's angle with the wall using LiDAR scan

As shown in the diagram, the program will first look through the scans to find the smallest distance. That indicates that the scan was perpendicular to the wall (the purple scan in Figure 10). The program will then take a look at the angle that resulted in the shortest scan, Θ, and calculates 90 - Θ. That determines the angle of the robot with the wall. This is due to the fact that the robot and the wall create a right angle triangle.

Now using this angle, the wall following behaviour can easily make the robot maintain a set angle with the wall. The main approach is to constantly calculate the robot's angle with the wall using the method described above. The robot will then be provided with an angular velocity to correct its current angle so that it will match the preset angle. This correction process will continue until the robot reaches a certain threshold with the wall at which point it will just follow the wall in parallel until it once again leaves that threshold and needs more correction. Please

note that to avoid constant minor adjustments around the threshold, an "acceptable distance from the wall" error region was defined on either side of the threshold. This prevents the robot from doing any angle corrections in that region. That error region is calculated based on the robot's maximum speed and the set angle. The error formula is as follows:

$$error = (speed \cdot sin(angle) \cdot time) / 2$$

This formula allows for a small error region around the threshold. This region is equivalent to the distance that the robot will travel during a given interval. The error value is half of that area since it is being applied to both ends of the threshold. Although this solution has produced great results in the current implementation of the robot, the robot still produces some oscillations. To alleviate this issue, the team is planning on adding back the PID controller for when the robot is closer to the wall. This will help smooth out the wall following.

8.4 Robot Simulator

By Bardia Parmoun

To simulate wall following and the robot's other core behaviours, the team decided to create a simple simulator. This simulator was developed using the *Turtle* library in Python. This helped the team save a lot of development time since testing the robot was not needed as often.

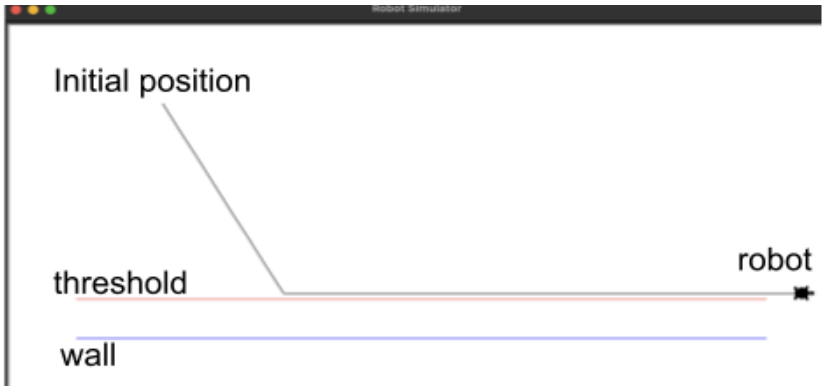


Figure 16: Simple simulator showcasing the robot's wall following behaviour

As illustrated in the diagram, the simulator allows for the robot to be initialized at a custom distance and angle with the wall. Similarly, the simulator also allows the user to set a threshold for the distance that the robot should maintain with the wall. From there the simulator will calculate the error region and give the robot the proper commands. The simulator also simulates the ROS twist command by giving appropriate linear and angular velocities to simulate its behaviour. The source code for the simulator along with instructions on how to use it was included in the source code for the project under the "tools" directory.

Since there is *ROS Gazebo* support for iRobot Create 3, the team is planning on implementing a more advanced simulator to properly simulate the state machine and the remaining core functionalities of the robot.

8.5 Intersection Handling

By Bardia Parmoun

Intersection detection and intersection handling is a crucial step for the robot. To achieve this, the team implemented a simple algorithm based on the type of the intersection and how the robot is entering the intersection.

The robot is aware of the approximate shape for the intersection that it is about to enter. It then uses the lidar information to determine when it has reached an intersection and when it has left it. To achieve this, the robot will repeatedly perform a 360° scan. It then uses the scan to determine if it can reliably detect a wall on its three sides (left, right, and front). A reliable wall is detected in a direction if the closest particle in the region is within a set threshold and if all the particles are close to each other.

In addition, the robot maintains a memory of its past ten readings for each direction and checks to see if there was a sudden jump in the readings. Inconsistent data or a sudden jump in the data results in the robot considering the wall as lost in that direction. The robot will then use the expected map as a guide to determine whether it has reached the desired intersection or not. For example in a normal four-way intersection, the robot will simply need to know whether it has lost the wall on its right and front sides. As shown in the state machine, the main strategy for intersection handling is for the robot to make an initial left or right turn and then repeatedly go forward until it is able to find the wall again.

An example of this implementation is shown in the diagram below (the green and red arrows show whether the robot has found or lost a wall respectively.

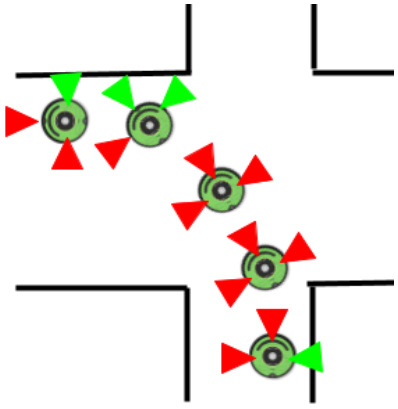


Figure 17: An example of the current implementation for intersection handling for a left turn

Here you can notice that the robot initially started with a wall on its right so it continued its normal wall following behaviour; however, upon reaching the intersection, the wall on the right of the robot was lost so the robot started handling the intersection. This resulted in the robot making an initial left turn. It then kept going forward in that direction until it was able to find a wall on its right side. It then used that wall to go back to its wall following state.

In order to improve the intersection detection and reduce the chance of false positives, the team specified a threshold for the RSSI of the beacon. This allows the robot to only pick up the beacon when it is close to it. At the moment the team selected a beacon threshold of -60.

8.6 Collision Handling

By Bardia Parmoun

As illustrated in the state machine, the robot is already programmed to handle simple collisions. The moment the bumper sensor is triggered, the state machine is programmed to transition the robot into the proper collision handling state. This allows the system to avoid losing crucial information such as its next destination. The current logic for the collision handling is to attempt a left turn until the sensor is deactivated. Then the robot will turn right before leaving the state. This results in the robot making a series of left and right turns while gradually passing the obstacle. Eventually once the robot has passed the obstacle it will go back to its normal flow. Here is an example of this algorithm in action.

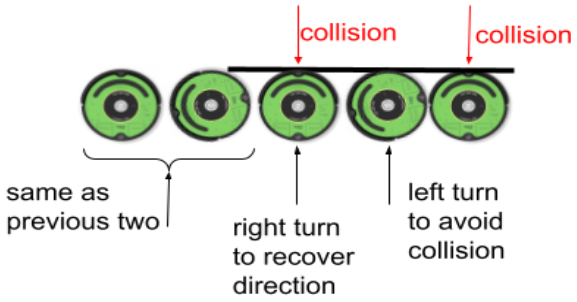


Figure 18: Demonstrating the robot's current collision handling mechanism

As shown in the diagram, the robot might repeatedly go back and forth between the normal state and the collision until it is able to successfully pass an obstacle. This strategy allows the robot to always maintain its current direction. Although this strategy is very effective for normal obstacles, it can lead to unexpected results for really large obstacles that can block the robot from different angles. An example of this is a large L-shaped example. In other words, if the robot hits another side of the obstacle, it might start following it thinking it is a wall which will lead to the robot losing its original direction. Here is a demonstration of this problem:

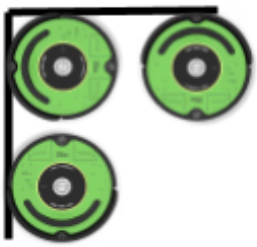


Figure 19: Demonstrating the robot's problem when handling L-shaped collisions

In order to make sure that the robot is able to maintain its original orientation, the state machine keeps track of the number of left turns that it makes so it can make the same number of right turns accordingly. This way the robot will not easily lose its direction.

The team is hoping the future implementations of the project will prepare a better collision handling mechanism to account for cases like this. Due to the refactored design of the project, this task can be achieved by simply updating the collision states. The group also recommends a small sub state machine dedicated to just handling the collision events.

Furthermore, it should be noted that due to the wall following algorithm that was described earlier, it is quite unlikely for the robot to run into collision situations. This is due to the fact that the robot has a very wide range when it comes to detecting the wall. So it will try to avoid any obstacles that it detects on its front and right.

8.7 Implementing Navigation

By Max Curkovic

To implement the aspects of dynamic navigation, there are two components of the robot that work in correlation with each other to send the correct navigation action, entirely dependent on the beacon in which the robot is closest to and the orientation in which the robot is facing.

8.7.1 Implementing Dynamic Navigation

By Max Curkovic

When the robot is initialized, it defines a hashmap that it uses to store the routes that it goes on. This hashmap is pre-defined in accordance with a parsed CSV file, where the key is the source destination and the value is a nested hashmap of all potential final destinations, with the immediate next direction in which it needs to go to get to its destination in the shortest amount of time. Each source destination also accounts for the beacon orientation in which the robot could be facing. The following beacon "compass" is used for orientation:

1

4 0 2

3

For instance, routing_map[“Loeb1”][“Southam”] means that the source destination is that the robot is going towards the beacon at Loeb from the north. Therefore, in order to get to Southam, it would have to perform a U-TURN action.

The beacon connections follow a very similar process. Again, a hashmap of the beacons is created, where the key is the starting location, and the value is a nested hashmap of its nearby beacon connections. For instance, beacon_connections[“Loeb”][“Southam”] would be 1, since the beacon at Southam is north of the beacon from Loeb.

The robot’s captain is where the determination of the starting and final destinations take place, in real-time. The captain stores a previous beacon, a destination, and the two hashmaps from above (the map and beacon connections). The captain reads for the beacon that it is closest to it using the closest RSSI value. Once the current beacon is obtained, it uses that beacon, along with the previous beacon stored, to determine the orientation it is facing. If it cannot successfully obtain the orientation, it is assumed that the robot has been moved and it must perform a U-Turn to return to its original point (see section 8.7.2). Using a combination of the current beacon, the beacon orientation and the destination, a direction can be defined by the captain and can be casted as a navigation event for the map publisher node. Afterwards, the message is published by the captain to the robot, and will take that corresponding navigation event (NAV_L_TURN, NAV_R_TURN, NAV_U_TURN, NAV_DOCK, or NAV_PASS). This process repeats at every new beacon the robot reaches, until it has docked (reached its destination). A new previous beacon will always be stored at that iteration’s “current beacon”.

There are fail-safe flows within the captain to account for any “bad flows”. Firstly, if there is no trip defined, the robot cannot proceed on a route and will not utilize any navigation events. Secondly, if the current beacon of the robot is equal to the previous beacon, again there cannot be a route occurring. The beacon_orientation is always assumed to be at 0 if it is not previously defined, i.e. the route is just starting.

8.7.2 Implementing U-Turns

By Max Curkovic

Another feature that the robot utilizes, in relation to dynamic navigation, is the ability to complete a U-turn. U-Turns are a necessary fail-safe mechanism for the robot. The robot is able to recognize its path due to its given map from the LiDAR sensor. If it veers off the correct path for any reason, the robot shall perform a 180° turn immediately when it recognizes it is no longer going in the right direction and make the necessary adjustments to return to the path. The implementation of the U-Turn is integrated as part of the robot’s state machine and action translator, and can be found more in detail in section 7.4 of the report.

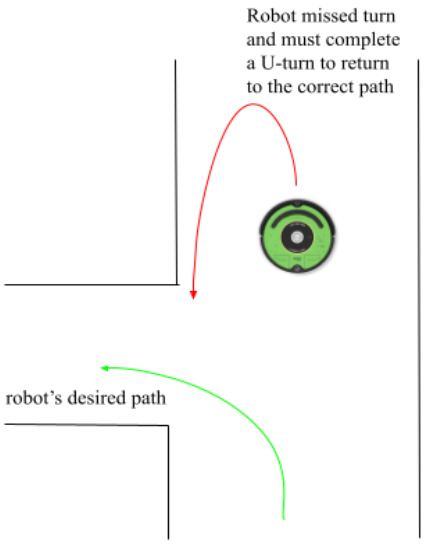


Figure 20: Demonstrating a U-Turn

8.8 Constants and Magic Numbers

By Bardia Parmoun

The current system relies on a variety of different constants. These include the speed of the robot, the distance thresholds for detecting the wall, the timer frequencies, etc. To ensure that these constant numbers are easily accessible and modiable, the team created a CSV file which includes all the constants for the system. A CSV parser was also included in the project to parse the config files and provide a way for them to be accessible to the different nodes.

Similarly, another CSV file was developed which includes all the beacon IDs and their locations in the system. This way the beacons can easily be added/removed and they are accessible to the various nodes that might need them (i.e captain and beacon_reader).

8.9 Hardware Implementation

By Matt Reid

The project implements the designed hardware for the Create 2 and Create 3 platforms discussed in Section 7.6. The Create 2 hardware implementation is discussed in Section 8.9.1, and the Create 3 hardware implementation is discussed in Section 8.9.2.

8.9.1 Create 2 Hardware Implementation
By Matt Reid

The Create 2 implementation requires connecting the main brush motor driver under the robot to the external components on top of the robot, connecting the Raspberry Pi 4B microcontroller to the iRobot Create 2 Serial port and LiDAR for sending commands and receiving data, and mounting the 3D printed chassis to hold the mail and the LiDAR so it has a clear view.

The main brush motor driver which is located underneath the iRobot Create 2 is connected to an inductor and a DC to DC converter. The DC to DC converter is then connected to power and ground of a USB C cable which is routed through a cable passthrough drilled in the robot to access the top. This is implemented as shown (left shows bottom, right shows top):

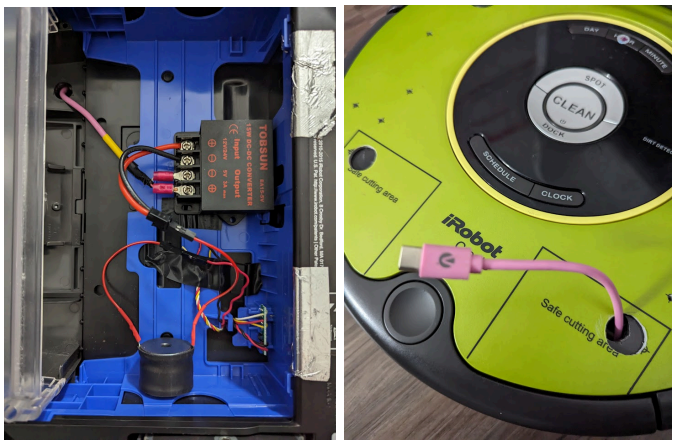


Figure 21: Hardware Implementation for the Main Brush Motor Driver power to USB C output

With power available on top of the robot, the chassis can be attached to the robot. The USB-C cable from the main motor brush driver is connected to the power bank input, and the power bank output is connected to the Raspberry Pi 4B. Both are placed on the middle level of the chassis. This is implemented as shown:

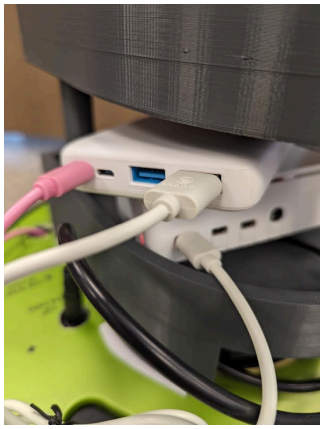


Figure 22: Hardware Implementation for powering power bank and Raspberry Pi

Two cables for controlling the LiDAR and iRobot Create 2 are then connected to the Pi. The serial-to-USB cable is connected to the serial port of the iRobot Create 2 and up through the bottom of the chassis to the Pi. The USB-to-micro USB cable for the LiDAR goes up through the hole in the chassis to the top where the LiDAR is attached. It is connected to the micro USB-to-Serial converter which is then connected to the LiDAR using the provided cable. The LiDAR is placed on top of the mailbox so it has a clear view of the surroundings. This is implemented as shown below (left shows Pi USB connections, right shows LiDAR):

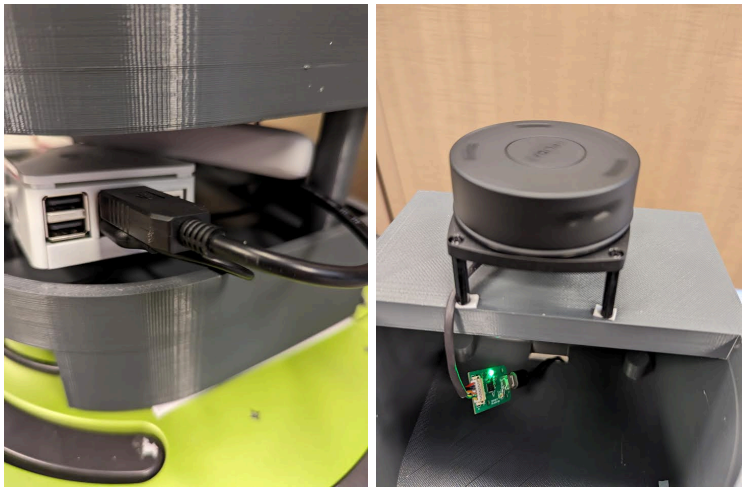


Figure 23: Hardware Implementation for connecting LiDAR and iRobot Create 2 to Pi

With the power input, LiDAR, and iRobot Create 2 connected to the Raspberry Pi 4B microcontroller, the robot hardware is fully implemented. For a full list of steps on implementing the hardware, refer to Appendix B: Hardware Setup. A photo of the full implementation of the robot follows:

Figure 24: Create 2 Hardware Implementation of the chassis and external components

8.9.2 Create 3 Hardware Implementation
By Matt Reid

The Create 3 implementation requires connecting the Raspberry Pi 4B microcontroller to the iRobot Create 3 USB-C port, and to the LiDAR for sending commands and receiving data. The Raspberry Pi 4B is mounted inside the cargo bay using a 3D printed mount provided by iRobot. The Create 3 does not require any external power hardware as the USB-C port of the Creat 3 robot provides a 5V, 3A power output that the Pi requires. The LiDAR is mounted on a 3D printed mount on top of the robot. Since the chassis only needs to hold mail and no external components, a letter holder was mounted to the robot using zip ties. A picture of the final hardware implementation of the robot without the chassis is shown below:

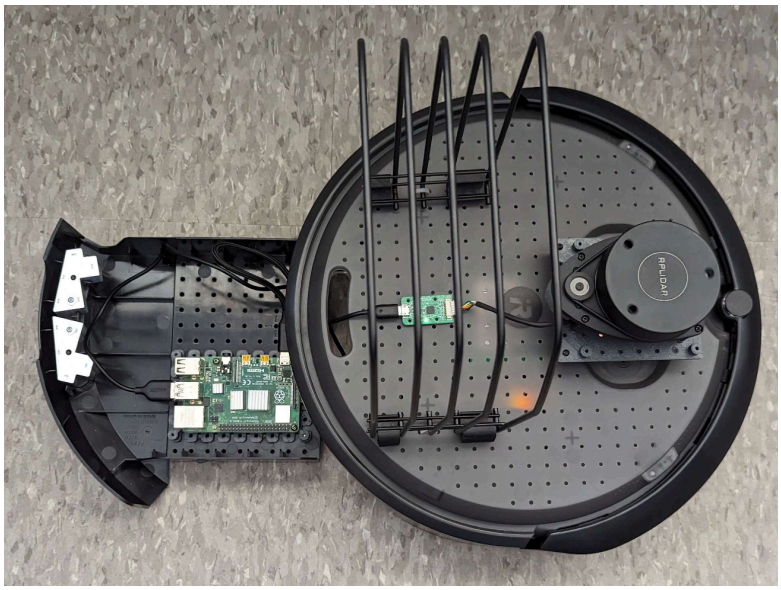


Figure 25: Create 3 Hardware Implementation

8.10 Web Application Implementation
By Cassidy Pacada and Max Curkovic

The web application is a core implementation in terms of being able to interact with the robot. It is how users interact with the robot - they are able to send a delivery to a robot. These user requests serve as a way for the robot to be controlled through an intuitive front-end design. The team's primary objective with the web application is to be able to control the robot using a very simple interface to send actions and navigation events, through the use of controllers, in communication with the existing ROS nodes.

The application is designed using the Spring Boot framework, making use of Object-Relational Mapping (ORM) and data persistence. It makes use of the Spring Model-View-Controller pattern, which processes all of the robot's navigation requests on the

back-end. Furthermore, the HTML pages use Thymeleaf to allow for the back-end to dynamically interact with the front-end, and each action is processed using a JavaScript AJAX call.

The web application is deployed using Microsoft Azure, using its free, resource limited web application hosting platform. When deployed, the web application can be accessed at the following link: https://cudelivery.azurewebsites.net/.

8.10.1 Web Application Front End - AppUser

By Cassidy Pacada and Max Curkovic

When a user first launches a web application, they are directed to the home page. A user that is not logged into their account will only be able to see the home page, as seen in Figure 26 below:

Figure 26: Home page for the Web Application (User not logged in)

The user will be required to register an account in order to access the website and be able to make a delivery. On the Azure server, the account is stored for future retrieval by the user. Cookies (implementation is discussed in section 8.10.2) are used by the browser to ensure that the user remains logged in for the duration of their session.

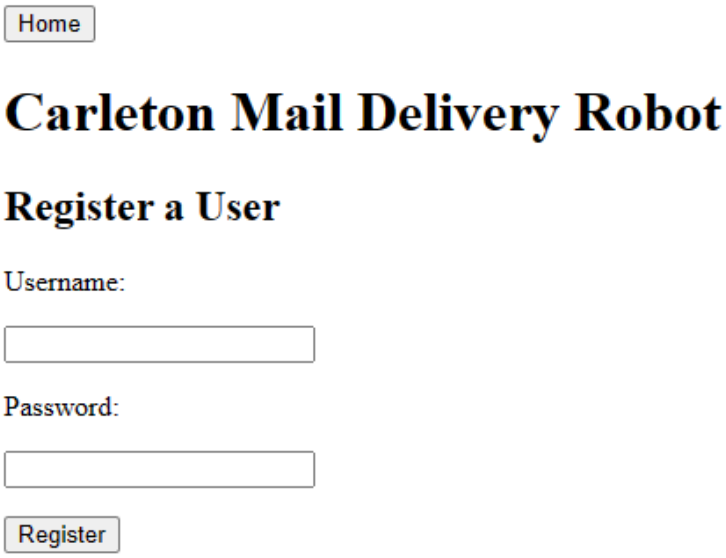


Figure 27: Register page for the Web Application

Once an account is registered, a user will be able to log into their account and access the dropdown menus to create a delivery request for the robot.

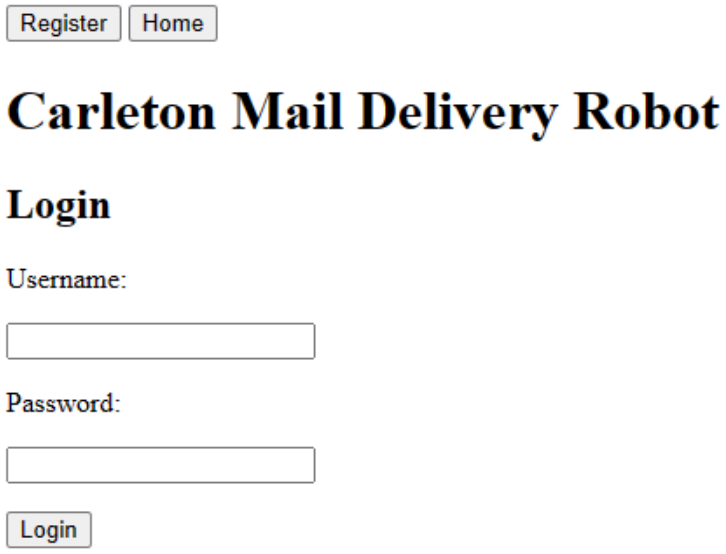


Figure 28: Login page for the Web Application

The home page is updated accordingly once a user is logged in. They will be able to select a list of buildings from a dropdown menu, and those values are sent to the robot in order to perform the delivery.

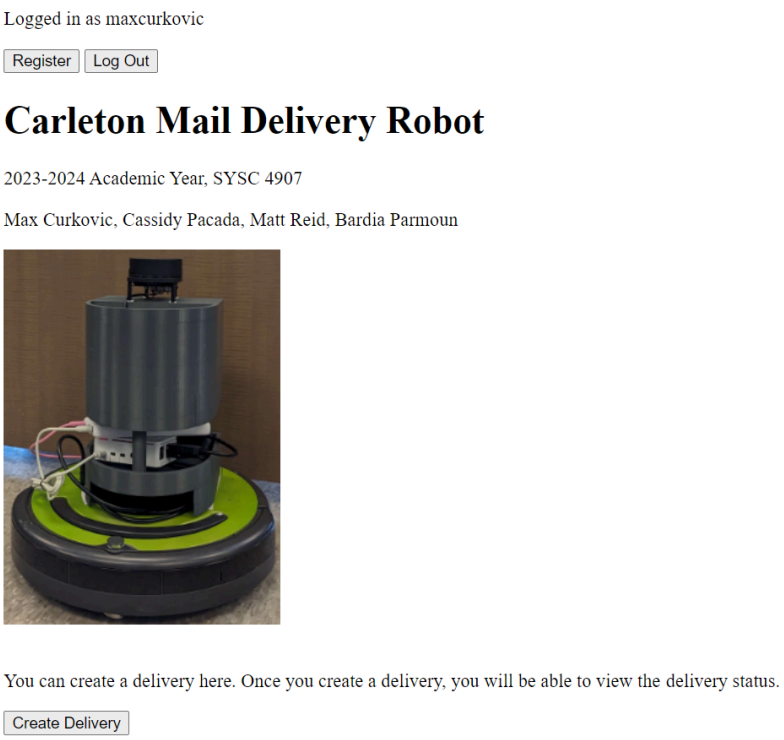


Figure 29: Home page for the Web Application (User is logged in)

A regular user will only be able to access the Create Delivery section, as well as a button to view the current delivery on the homepage once the delivery is created. When "Create Delivery" is clicked, the user will be prompted to select a source and a final destination for the robot's path.

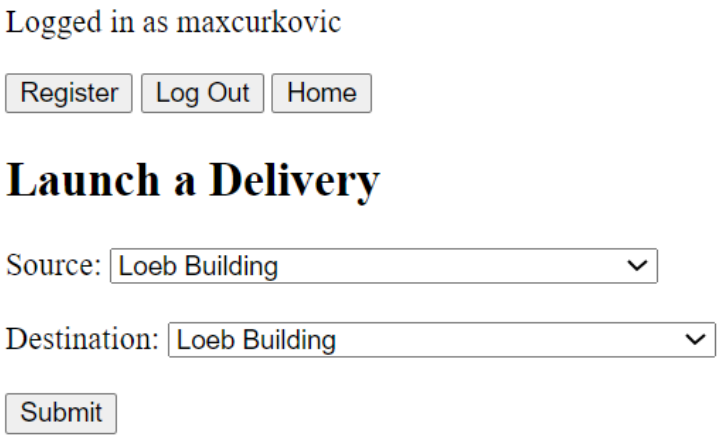


Figure 30: Launch Delivery page for the Web Application

Once the delivery has been sent, the user will be redirected to the status page and be able to see the status of their live delivery. New updates are added every 2 seconds in the textbox.

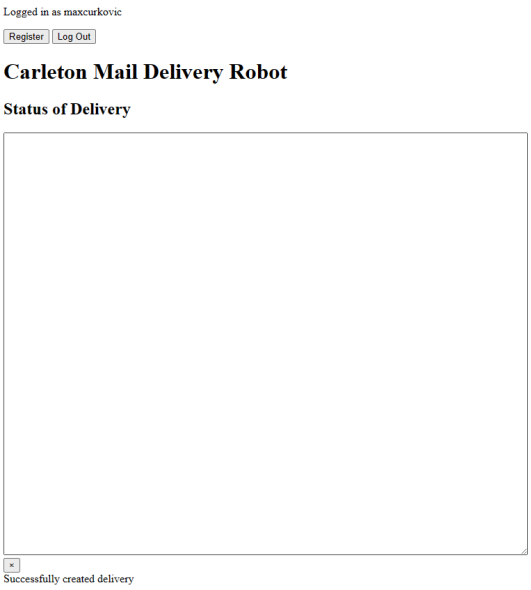


Figure 31: Status of Delivery page for the Web Application

8.10.2 Web Application Front End - SuperUser
By Cassidy Pacada and Max Curkovic

Admins (superusers) have far more access to the back-end. When a user is logged into an admin account, the home page will have a button that directs them to an admin page. All of the admin-only privileges are endpoint-protected on the back end, and a regular user will not be able to access them through the path in the URL.

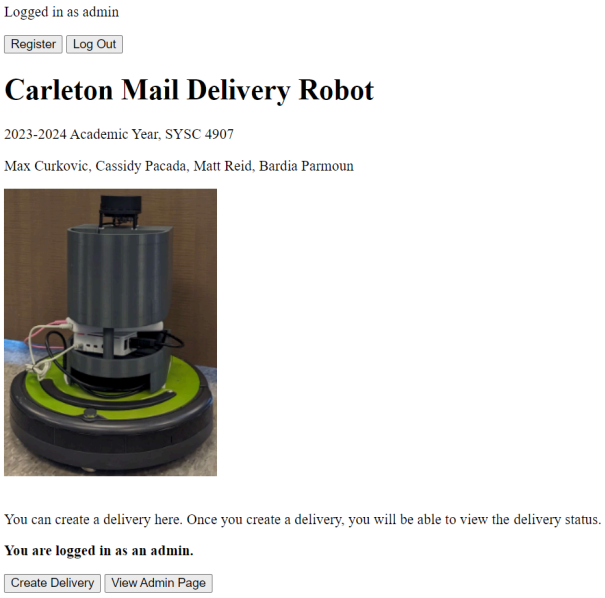


Figure 32: Home page for the Web Application (Superuser is logged in)

On the admin page, there are numerous options for an admin to use, including removing a user, adding another admin account, and managing the current list of robots.

Figure 33: Admin page for the Web Application

Registering an admin works very similarly to registering a regular user, as the Superuser is an extension of the regular user in the back-end. Removing a user is also straightforward, only prompting the requirement of a current username to remove a user from the database.

Figure 34: Remove User page for the Web Application

One of the major differences between a superuser and a regular user is the superuser's ability to manage the robots and view/manage their current deliveries. The team utilizes both the Create 2 and the Create 3; as such, the ability to utilize multiple robots was necessary to manage different delivery requests. As an example - the below figure shows the Create 2 robot in the list of currently active robots.

Figure 35: Manage Robots page for the Web Application

The "View Robot Page" brings the superuser to view the list of deliveries for a particular robot. In the figure below, the list of deliveries for the Create 2 example is shown. Each delivery entry consists of a button to redirect the user to that respective delivery's status page.

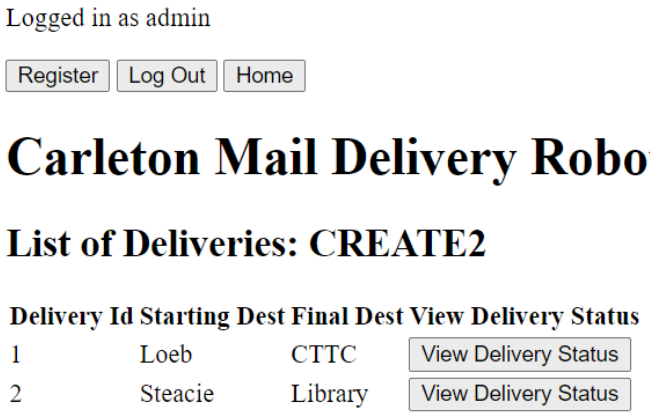


Figure 36: Robot Deliveries page for the Web Application

8.10.3 Web Application Back End - Spring Boot Implementation

By Max Curkovic

As indicated previously, the web application was developed using Spring Boot, with support for Python. The back-end utilizes Object Relational Mapping (ORM) and data persistence for User, Delivery, Robot and Superuser entities. This allows for User, Delivery, Robot and Superuser objects to be stored in a database that can be used for future use, as well as for the Robot in order to start a delivery. The below figure depicts the UML class diagram for the models:

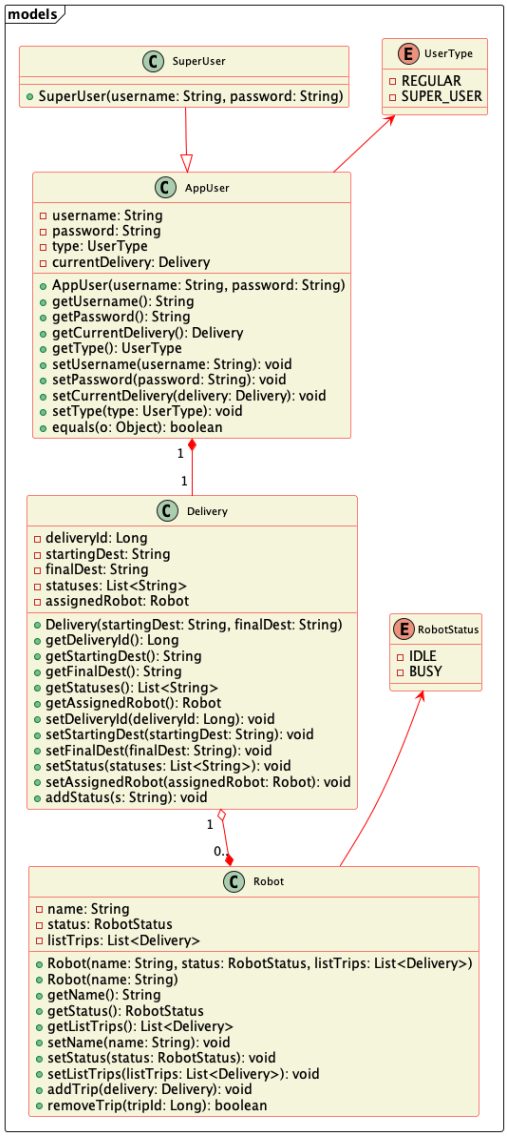


Figure 37: UML Diagram for the Web Application

The web application utilizes an API RestController to control the core functionality, including login, logout, register user, register superuser, remove user, create robot, create delivery, update delivery status, kill robot, and get robot deliveries. The back-end utilizes Rest Controllers in order to return JSON-formatted strings, which in turn, hold the data for each entity. An example would be when a user/superuser is registered, and the username and password are held in JSON as follows: {"username":"maxcurkovic","password":"carleton"}. The JSON string is mainly used to display text on the HTML pages using Thymeleaf, and to allow for the specific data to be retrieved by the web application at any point. A Delivery entity works very similarly, with a starting destination and a final destination as inputs, and a Robot entity also

works similarly, with a name and a list of corresponding deliveries. The APIController is integral for communication with the Client ROS node from the robot codebase.

The PageController simply holds all of the get mappings for the web application, which is how a page is rendered in a user's browser. For example, a get mapping to "/status/1" would simply allow the user to go to the delivery status page for the delivery with an ID of 1. There are certain pages that require a user to be logged in, which is implemented through the use of an aspect annotation @NeedsLogin.

To store the entity objects themselves, CrudRepositories were created, which essentially act as tables in an SQL database, but in an ORM form. Whenever a new user is registered, a new delivery is created or a new robot is registered, their respective objects are added to their respective CrudRepositories (UserRepository, etc.) and are available for retrieval by the web application at any point. It is important to note that a User-Delivery relationship is one to one, and a Delivery-Robot relationship is many to one. These relationships, modeled using Object-Relational Mapping, are captured in the below ER diagram:

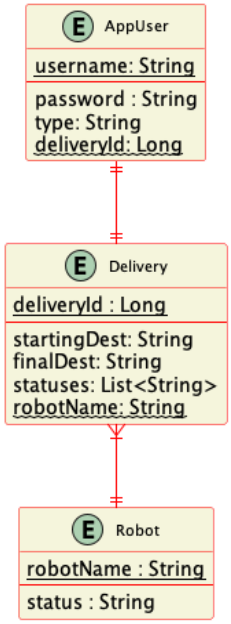


Figure 38: ER Diagram for the Web Application

In order to implement the login and logout features, cookies were implemented as part of a CookieController. Cookies are used to store user data and utilize it throughout a session on a web application. When a user logs into their account, their data is stored in a cookie that is used by the web application overall to maintain their access to the website. Once a user logs out, the user's data is removed and the cookie is deleted.

Finally, JavaScript AJAX was used to control the actual calls that allow for each mapping in the APIController to execute. These calls are mainly used to redirect users to the correct page after each action is performed, and controls any “bad input”, especially for the login and register pages. These JavaScript scripts are called within the HTML pages.

8.10.4 Web Application Back End - ROS Implementation

By Max Curkovic and Bardia Parmoun

The connection between ROS and the Spring Boot application is fairly straightforward. To allow for this, a Client node was created within the project, with a publisher for publishing trips to the web application and a subscriber for the web application to subscribe to updates. Essentially, the Client node acts as the communication link between the user input from the front end and the robot.

In order to receive GET requests and send POST requests from the ROS node to the web client, the HTTP library “requests” is utilized within the node. To make a call to the client, there are two primary types of request calls that are used:

- requests.get("https://cudelivery.azurewebsites.net/api/v1/getRobotDeliveries/{id}")
- requests.post("http://cudelivery.azurewebsites.net/ap1/v1/updateStatus/{id}” , data={‘deliveryId’: 1, ‘startingDest’: ‘cttc’. ‘finalDest’: ‘loeb’, status: ‘status’})

The GET request is primarily used to get the delivery information, parse the starting and final destination, publish it, and perform the necessary navigation tasks using this delivery information. The POST request is primarily used to post the new delivery to the web client, and append the current status to the textbox for the user to view.

The Client node consists of two important functions: one to send a request to the Captain, and one to handle an update from the Captain. The sendRequest function uses the getRobotDeliveries API call to retrieve all the deliveries for the given robot (using GET). From this returned JSON string, it determines the next trip to take and sends the request to be published. The handleUpdate function uses the updateStatus API call to publish a new update sent from the Captain to the web application to be displayed to the user (using POST). With these two functions, the robot is able to interact with the web application by taking a delivery route sent from the user, and by sending updates to the web application for the user’s convenience.

It is important to note that a timer constant, called self._request_timer, is created in the Client node as well, and sends the status update to the user every ten seconds. This ensures that the textbox is not overloaded with information, and the user can properly see the progress of the robot while it is on its current delivery.

8.11 Create 3 Transition

By Bardia Parmoun

With iRobot Create 3, providing native ROS support there is no longer a need to use the Create driver. As such, the launch file for the robot was updated to avoid loading unnecessary nodes. To make the transition as smooth as possible the same launch file is used for the all 3 robot versions. This allows the source code to remain fairly consistent.

There are two major differences between the Create 2 and Create 3. Firstly, the Create 3 added a new 'hazard_detection' which includes reading from every single hazard sensor that the robot has. This means that the generic /bumper topic is no longer the right approach for detecting the bumper sensor. To overcome this, the code for the 'bumper_sensor' node was updated to subscribe to the proper topic based on the robot's model. This topic generates a list of hazards for the robot. The team uses hazards of type 1 and 2 which are related to the bumper and the cliff sensors.

Secondly, the Create 3 introduced a new method of docking. Unlike Create 2, the robot no longer subscribes to the docking topics. Instead both dock and undock are ROS actions. This resulted in the action translator also being updated to implement the appropriate docking behaviour based on the robot's model.

Additionally, due to the new size of the mailbox, the viewing range of the LiDAR sensor was updated to avoid false positives with the mailbox.

Finally, it should be noted that with the new topics introduced in the Create 3, the user can get detailed status about the robot's docking state and its proximity to the dock station. These new topics allow the robot to be more accurate and reliable with docking. In addition, it fully eliminates the documented issue that is described in section 10.

8.12 Project Repository

By Bardia Parmoun

All the information related to the project along with its codebase can be found at: https://github.com/bardia-p/carleton-mail-delivery-robot

9 Testing

By Cassidy Pacada, Max Curkovic and Bardia Parmoun

This section details the testing strategies that have been implemented for testing the robot, including both unit and integration testing, as well as a specification of the Github workflow for the repository.

9.1 Testing Strategy and Justification

By Cassidy Pacada

The team's testing strategy involves implementing both unit testing and integration testing to ensure that all of the robot's components and their interactions are working as intended. Integration testing was implemented through simulations to determine both that the data is being sent correctly throughout the system. The majority of the unit testing efforts went towards the state machine as it is the heart of the robot's behaviours. Additionally, it is not a node and thus, the overhead required to add and run more tests is much lower than for the robot's other components.

After all of the software tests and simulations have passed, only then is the physical robot tested. It is important to test the hardware as it can act in different ways from the software environment. For example, the actual robot is limited by its motor and may react to inputs slower than its software would. However, physical testing takes a much longer time as it requires more setup and the team must be present on campus every time a test is run. This creates a large amount of overhead which can slow down the development process. As such, the team aims to catch and fix as many faults as possible prior to running the robot.

9.2 Unit Testing

By Cassidy Pacada

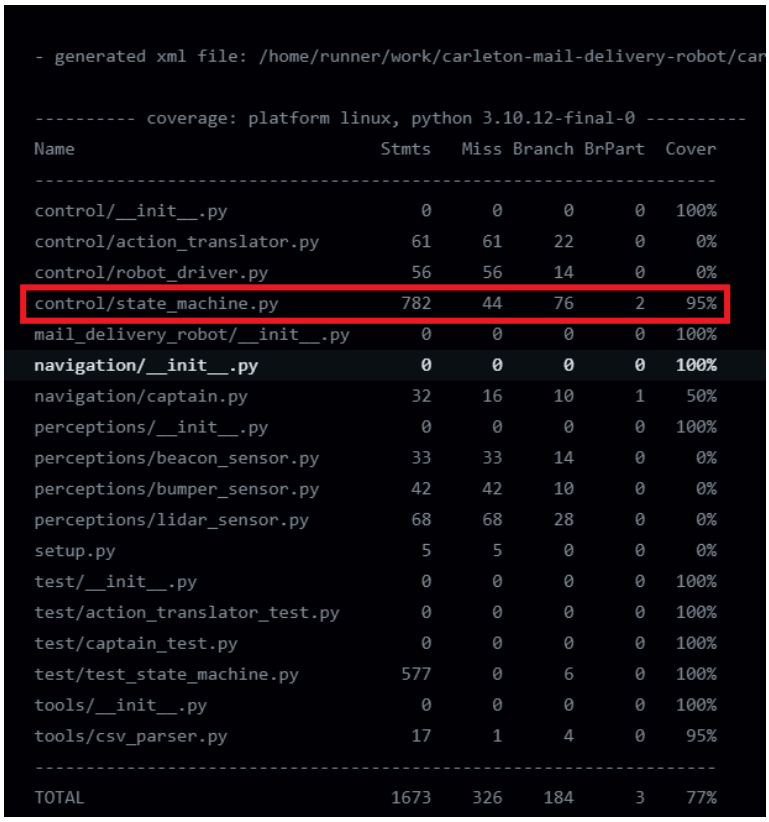
The unit testing for the state machine aims to ensure that the robot will always be in its intended state and that it will perform the intended action upon reaching said state. Initially, the team implemented tests with the goal of meeting the all-states criterion but it became clear that certain faults were not being caught. As such, the team implemented tests that satisfied the all-transitions criterion which involved traversing every possible edge between the state nodes. One example of a test path is shown below:

```
def test_path_one():
    '''
    Test Path 1: Asserts that state transitions are good for right turns and that the completed actions
    are as expected
    '''

    actionPublisher = ActionPublisherStub()
    state = state_machine.No_Dest(actionPublisher)

    state = send_update(state, T2)S
    assert state.stateType.value == "NO_DEST"
    state = send_update(state, T11)
    assert state.stateType.value == "COLLISION_NO_DEST"
    state = send_update(state, T13)
    assert state.stateType.value == "COLLISION_TURN_RIGHT"
    state = send_update(state, T6)
    assert state.stateType.value == "SHOULD_TURN_RIGHT"
    state = send_update(state, T1)
    assert state.stateType.value == "HANDLE_INTERSECTION"

    actionPublisher.extract_data()
    expected_actions = ["WALL_FOLLOW", "L_TURN", "L_TURN", "R_TURN", "R_TURN"]
    assert actionPublisher.action_data == expected_actions
```

Figure 39: Unit test path example for the state machine.

Additionally, it is crucial that the actions performed while transitioning to each state are performed correctly. The state machine includes an ActionPublisher which sends the action to the ActionTranslator which turns said actions into commands that the physical robot can understand. However, in a testing context, there is no ActionTranslator node that the ActionPublisher can send commands to. As such, a stub was created that simply stored all the state machine actions. For each path, a list of completed actions was compiled and compared to a list of expected actions. Additionally, the tests needed to be able to account for the newly implemented long actions. Long actions are essentially when one action is repeated several times during the course of one transition. To reduce the number of actions that needed to be captured, the tests simply told the state machine to perform the last action in the sequence and ignore the previous ones.

Through unit testing, the team was able to achieve 95% coverage of the state machine with exceptions for the functions that interact with other nodes. As nodes cannot be tested using traditional unit testing, the robot_driver, the action_translator, and the captain are lacking coverage. This means that they must be tested through other methods which is where integration testing comes into play.



```
- generated xml file: /home/runner/work/carleton-mail-delivery-robot/car

---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name                              Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------
control/__init__.py                   0      0      0      0   100%
control/action_translator.py         61     61     22      0     0%
control/robot_driver.py              56     56     14      0     0%
control/state_machine.py            782     44     76      2    95%
mail_delivery_robot/__init__.py       0      0      0      0   100%
navigation/__init__.py                0      0      0      0   100%
navigation/captain.py                32     16     10      1    50%
perceptions/__init__.py               0      0      0      0   100%
perceptions/beacon_sensor.py         33     33     14      0     0%
perceptions/bumper_sensor.py         42     42     10      0     0%
perceptions/lidar_sensor.py          68     68     28      0     0%
setup.py                              5      5      0      0     0%
test/__init__.py                      0      0      0      0   100%
test/action_translator_test.py        0      0      0      0   100%
test/captain_test.py                  0      0      0      0   100%
test/test_state_machine.py          577      0      6      0   100%
tools/__init__.py                     0      0      0      0   100%
tools/csv_parser.py                  17      1      4      0    95%
-------------------------------------------------------------------
TOTAL                              1673    326    184      3    77%
```

Figure 40: Testing report outlining how much of the code has been covered.

9.3 Integration Testing

By Cassidy Pacada

The team's goal for integration testing is to simulate the robot's behaviours in a variety of situations without the need to physically manipulate the robot. This cannot be done through unit testing as running a node requires the use of launch files which cannot be used with the colcon test package.

9.3.1 What the Integration Tests Do

By Cassidy Pacada

The integration tests are implemented through a simulator that allows the user to select various parameters that will control the simulation. The user can control how often the simulated robot will encounter collisions and inconsistent wall data, its starting location, its destination, what locations it encounters along the way, as well as the overall length of the simulation. This allows the user to test the robot's behaviour more precisely. For example, if the navigation needed to be tested, the user would manipulate the simulation's starting position, final

destination, and the beacons that it will detect on its journey. Similar adjustments can be made to test behaviours such as wall following and collision handling. Simulated environment data is constantly provided to the simulated robot which responds to it and changes its state (as seen in Figure 42) just as the real robot does.

By default, the simulation runs for 45 seconds and will start at University Center with a distance of 0.2m from the wall and an angle of 0.1rad. Its destination is Nicol and along the way it should not encounter any collisions or unexpected wall data.

```
cassidypacada@ubuntu:~/cmds_ws$ ros2 launch mail_delivery_robot test.launch.py "path:=Canal:Nicol"
[INFO] [launch]: All log files can be found below /home/cassidypacada/.ros/log/2024-03-04-19-57-39-191644-ubuntu-3037
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [action_translator-1]: process started with pid [3040]
[INFO] [robot_driver-2]: process started with pid [3042]
[INFO] [captain-3]: process started with pid [3044]
[INFO] [stub_sensor-4]: process started with pid [3046]
[robot_driver-2] [INFO] [1709611059.856693287] [control.robot_driver]: Robot Starting STATE IS: NO_DEST
[robot_driver-2] [INFO] [1709611060.193091277] [control.robot_driver]: 0.4:0.1:0.4:4:3
[action_translator-1] [INFO] [1709611060.222598322] [control.action_translator]: WALL_FOLLOW
[action_translator-1] [INFO] [1709611060.223504858] [control.action_translator]: WALL_FOLLOW
[action_translator-1] [INFO] [1709611060.224100652] [control.action_translator]: WALL_FOLLOW
[action_translator-1] [INFO] [1709611060.239139792] [control.action_translator]: WALL_FOLLOW
[action_translator-1] [INFO] [1709611060.249853734] [control.action_translator]: WALL_FOLLOW
```

Figure 41: Simulating the robot encountering the nodes at Canal and Nicol

```
[robot_driver-2] [INFO] [1709608885.464970935] [control.robot_driver]: GOT COLLISION
[action_translator-1] [INFO] [1709608885.471898448] [control.action_translator]: WALL_FOLLOW
[action_translator-1] [INFO] [1709608885.491954616] [control.action_translator]: L_TURN
[action_translator-1] [INFO] [1709608885.502471206] [control.action_translator]: L_TURN
[action_translator-1] [INFO] [1709608885.511759003] [control.action_translator]: L_TURN
[action_translator-1] [INFO] [1709608885.522205901] [control.action_translator]: L_TURN
[action_translator-1] [INFO] [1709608885.531622261] [control.action_translator]: L_TURN
[robot_driver-2] [INFO] [1709608885.541468281] [control.robot_driver]: Changed State STATE IS: COLLISION_NO_DEST
[action_translator-1] [INFO] [1709608885.553046328] [control.action_translator]: L_TURN
[action_translator-1] [INFO] [1709608885.562533557] [control.action_translator]: L_TURN
```

Figure 42: Robot changing state based on the given simulated data

An example of a simple integration test can be seen above in figures 33 and 34. The robot will travel from University Center to Nicol and will encounter the Canal intersection along the way. The simulated robot's actions are logged in the terminal so that the user can determine if the expected behaviours are being performed.

9.3.2 Running Integration Tests

By Cassidy Pacada

The integration tests can be run with the command *ros 2 launch mail_delivery_robot test.launch.py.* There are six parameters that this command takes which will customize the simulation to handle a variety of scenarios.

The parameters are (quotations included):

- *"init_pos:=[distance]:[angle]"*: allows the user to choose the simulated robot's initial position from the wall by providing a distance in meters and an angle in radians.
- *"path:=a:c…"*: allows the user to provide the simulated robot with a list of beacons that it will "encounter" while it is running. As such, the user can test how the robot handles being placed in intersections that it is not necessarily expecting. There is no default path, instead the simulation's path will be based on the given *delivery* parameter.
- *"duration:=[time]"*: allows the user to specify how long the simulation will run for by providing a length of time in seconds.
- *"collision_freq:=[time]"*: allows the user to specify how often the robot should encounter a collision by providing a frequency in seconds. (ex. if the value is 15, the robot will have a collision every 15 seconds).
- *"wall_diff:=[percent]":* allows the user to choose the wall difficulty that the robot should encounter. The percentage represents the fraction of random wall data that the robot will encounter as opposed to expected wall data.
- *"delivery:=[source]:[destination]"*: allows the user to provide the source location at which the robot will start and the destination that the robot should end up at.

9.3.3 Integration Test Implementation

By Cassidy Pacada

The team implemented the simulation by creating one stub that would simulate and send data for all of the robot's sensors. Each simulated sensor has a timer and will send data based on its timer's interval. For example, the simulation is set to receive LiDAR data every 0.2 seconds but it will only receive a new beacon every 15 seconds and only if there are beacons left in its path list. Each sensor is stubbed to have their callback functions implemented within the stub_sensor file. This allows the stub file to have publishers and subscribers that interact with other nodes in the project, namely, the captain, the robot_driver, and the action_translator. These nodes remain unchanged thus allowing the team to determine whether they are communicating properly and whether the data being passed through the system is correct.

```python
def bumper_callback(self):
    '''
    The callback for the bumper timer.
    Sends a bumper event.
    '''
    calc = String()

    should_collide = random.random()
    if should_collide <= COLLISION_PERCENTAGE:
        calc.data = "PRESSED"
    else:
        calc.data = "UNPRESSED"

    self.bumper_publisher.publish(calc)
```

Figure 43: Stubbed callback function for the bumper sensor

```python
def readBump(self, data):
    '''
    The callback for /bumper.
    Reads the bump data and acts accordingly.

    @param data: The new bumper data received.
    '''
    # Checks to see if the bumper was pressed at all.
    is_pressed = False
    is_pressed = is_pressed or data.is_left_pressed
    is_pressed = is_pressed or data.is_right_pressed
    is_pressed = is_pressed or data.is_light_left
    is_pressed = is_pressed or data.is_light_front_left
    is_pressed = is_pressed or data.is_light_center_left
    is_pressed = is_pressed or data.is_light_center_right
    is_pressed = is_pressed or data.is_light_front_right
    is_pressed = is_pressed or data.is_light_right

    # Updates the bumper state.
    bumpEvent = String()
    if (is_pressed):
        bumpEvent.data = Bump_Event.PRESSED.value
    else:
        bumpEvent.data = Bump_Event.UNPRESSED.value

    # Slows the publishing of the messages to ensure the detection is smooth.
    if (self.lastState != bumpEvent.data or self.counter > self.config["MAX_BUMP_COUNT"]):
        self.lastState = bumpEvent.data
        #self.get_logger().info(bumpEvent.data)
        self.publisher_.publish(bumpEvent)
        self.counter = 0
    self.counter += 1
```

Figure 44: Actual callback function for the bumper sensor

As seen in figures 43 and 44, the stubbed callback function is much simpler than the actual callback function since the data is simulated. However, both functions are able to interact

with the robot_driver's subscribers because they both publish bumpEvents. By ensuring that the way the simulated sensors interface with the other nodes remains the same, there is no need to modify any existing nodes. By reducing the number of test nodes that need to be created, the amount of effort required to create new tests has been reduced greatly.

To stop the tests, an os system call was made to kill all of the running nodes. The tests will stop after the given duration value has elapsed.

9.3.4 Future Implementation
By Cassidy Pacada

Future groups could improve the tests by finding a way to capture the logs in a file so that they can be examined more easily. As of now, the logs are simply being displayed in the command line which makes it difficult to analyze and compare the robot's behaviours to its behaviours in the past. As well, future teams could move away from testing with logs and towards integration testing through visual simulation. With the Create 3, it is possible to use simulation software such as Gazebo which would provide a more accurate and robust picture of the system. In the tests' current state, the robot's state is communicated through logs. While this is effective for the team's current purposes, it does take some effort to sort through the logs to ensure that the robot is behaving as expected. Gazebo offers a 3D simulation so that the robot's actions can be demonstrated visually just as it would appear in real life.

9.3.5 Impact
By Cassidy Pacada

The integration tests have proven to be useful as they allowed the team to test the navigation implementation without manually running the robot to and from various destinations across campus. This has allowed the team to discover a couple of errors in the map implementation which have since been fixed. As well, they allow the team to determine whether the robot is taking the correct action in the appropriate places since the tests can simulate events such as collisions, intersections, and unexpected beacons. All these parameters can be controlled at will when running the command making testing much easier than before.

9.4 Workflow Specification
By Cassidy Pacada

For integration testing, the team will make launch files for each of the test nodes so that it can be opened with the *ros 2 launch* command. A bash script containing all the test commands will be created for ease of testing. A Github workflow was created to automatically run the ROS unit tests every time a pull request or a commit to main occurs. This will ensure that the program is functional at all times and is meant to keep the project aligned with the practice of continuous

integration. The team is researching to determine if the integration tests can also be added to the workflow.

As the web application was created, two additional Github workflows were added. These are the Maven workflow and the Azure workflow. These ensure that both the local and the deployed versions of the web application are functional at every stage by automatically running the web application related tests. The separation of workflows between the ROS system and the web application means that, should an error occur, it is much easier to pinpoint the location.

9.5 Web Application Testing

By Max Curkovic

To ensure that the web application's controllers and endpoints work as expected, a simple but effective testing framework is implemented using Java's JUnit framework, which works seamlessly with Spring Boot. For each test class, the @Test annotation symbolizes a test method, and ensures that all actions invoked (such as adding a model entity to a CRUD repository) does not transfer over to the live application.. There are two types of tests created for the web application: model tests and controller tests.

The model tests ensure that the methods defined within the individual models work as expected. This includes testing of the User, Superuser, Delivery, and Robot model. An example of a model test would be to test whether a delivery can be added properly to a Robot's list of trips. For each model, a test is also used to check that a model entity can persist on the web application's back-end, checking to ensure that a model entity was properly added to the CRUD repository.

The controller tests ensure that the GET and POST mappings within the APIController and the PageController are properly tested. For both test classes, a mock MVC is created to ensure that the mappings can be properly evoked. All GET mapping tests within the PageController check that the title of the HTML page matches the one retrieved by the mock MVC. The POST mapping tests within the APIController utilize the CRUD repositories and the model methods to ensure that the actions taken with the posted JSON data work as expected.

Whenever a change is made in the Github repository, the Maven workflow (see section 9.6.2) will always build and compile the web application and all tests are required to pass prior to merging a pull request.

9.6 Workflow Execution

By Max Curkovic and Bardia Parmoun

To facilitate the testing of all aspects of the project, there are several workflows that have been created that are run whenever a pull request is merged into the master branch.

9.6.1 ROS Colcon Test Workflow

By Max Curkovic

The ROS Github workflow is specified as the ros_colcon_test.yaml file within the repository. Whenever a pull request is created to merge a particular branch to the main branch, the commits will always traverse the pipeline and follow a series of listed steps specified within the workflow.

Firstly, the branch will checkout to the master branch, as to prepare the project for the incoming changes. Secondly, the latest version of ROS is set up within the workflow. This version of ROS is Humble, as Foxy does not work properly with building the tests. The workflow will execute the commands to set up ROS. Finally, the workflow runs colcon build and colcon tests on all the unit and integration tests within the project and ensures that they build, and pass. The workflow also provides a coverage report to determine whether the existing tests are adequate. Once this has been validated, the pull request will be able to merge onto the main branch, as long as the changes are approved by the assigned reviewers. This workflow is only triggered when there is a change in the “mail_delivery_robot” package.

9.6.2 Maven Workflow

By Max Curkovic

The Maven workflow is specified as the maven.yml file within the repository. It sets up its own environment of JDK 17, and is used to package the Spring Boot web application in Maven and run its corresponding build and tests to ensure it can build correctly. Once this has been validated, the pull request will be able to merge onto the main branch, as long as the changes are approved by the assigned reviewers. This workflow is only triggered when there is a change in the “webapp” package.

9.6.3 Microsoft Azure Web Deployment Workflow

By Max Curkovic

The Microsoft Azure web deployment workflow is specified as the master_cudelivery.yml file within the repository. Very similarly to the Maven workflow, it sets up its own environment of JDK 17, and is used to package the Spring Boot web application in Maven and run its corresponding build and tests to ensure it can build correctly. Assuming that this build succeeds, the artifact that is created from the Maven package is downloaded, and is deployed to Microsoft Azure under the name “cudelivery”. Once this has been validated, the pull request will be able to merge onto the main branch, as long as the changes are approved by the assigned reviewers.

9.6.4 Diagrams Workflow

By Bardia Parmoun

The diagrams workflow is specified as the diagrams.yml file within the repository. The purpose of this workflow is to compile and generate all the plantuml diagrams in the repository. The workflow will use the plantuml.jar tool located in the tools package and run it on every plantuml script. This ensures that the scripts are always working as expected. This workflow is only triggered whenever there is a change to any of the .plantuml files.

10 Documented Issues

By Matt Reid

This section outlines any issues with the system that have been identified and need to be addressed. For each issue, a temporary solution that will be used so that development is not blocked, a permanent solution that will be implemented by the end of the year, and any other potential solution(s) for if the permanent solution does not work are provided. There is currently one issue documented in Section 10.1 which is an issue with automatic undocking where the robot cannot be activated by the serial port.

10.1 Automatic Undocking in Create 2

By Matt Reid

Since the robot must dock to recharge between trips, it is desirable to be able to dock and undock the robot on command. An issue was identified where it becomes impossible to communicate with the Create 2 robot using the serial port when it is on the docking station. It was found that this was due to the robot sending charge information over the serial port, constantly filling the data lines while it charges.

Temporary Solution:

A temporary solution to this issue is to manually undock the robot when it is done charging and is ready to go on another trip. This solution is not ideal as the robot is no longer fully autonomous, requiring manual intervention to move it off of the docking station.

Permanent Solution:

A permanent solution that was provided by iRobot when a similar question was asked about the robot failing to receive the undock command, was to contact them by email to get an updated firmware for the robot processor [13]. This update should fix the sleep/wakeup functions so that they can be used at any time as expected.

Other Solutions:

If an update cannot be obtained from iRobot, or the firmware update is unsuccessful, the team can connect a solenoid to the Raspberry Pi which pushes the “Clean” button on the robot twice to undock it. By pushing the “Clean” button, the robot will undock and start moving so that it can clean. Then by pushing it again, it will stop trying to clean and will be off the dock and ready to receive commands again.

Link to the GitHub issue:

This issue is currently tracked on the github repository for the project at: https://github.com/bardia-p/carleton-mail-delivery-robot/issues/23

10.2 Stopping the Create 2

By Bardia Parmoun and Matt Reid

In order to get the Create 2 to run, the driver puts the robot in unsafe mode. An issue was identified where it becomes impossible to stop the robot through its CLEAN button. In other words, the only way to stop the robot is by sending a direct reset command through its serial port. It was found that this was due to the fact that the robot is put to unsafe mode, meaning it is no longer listening to the buttons.

Temporary Solution:

A temporary solution to this issue is to add a remote command to the web app that allows the user to kill the robot. The "client" node for the robot will repeatedly query the web app to check to see if the robot needs to be stopped. If so, the node will run the serial reset command and stop the robot.

Permanent Solution:

The temporary solution described above is dependent on the robot's internet connection. This could be problematic in certain parts of the tunnels. A permanent solution for this would be to add a button that connects to the Raspberry Pi through the GPIO pins. This button would send the serial reset command upon being pressed, forcing the robot to stop.

Link to the GitHub issue:

This issue is currently tracked on the github repository for the project at: https://github.com/bardia-p/carleton-mail-delivery-robot/issues/64

11 List of Required Components/Facilities

By Cassidy Pacada

This section lists the required components, facilities, and justification of purchases that allow for the robot's operation.

Table 24: List of required components/facilities, their objectives of the project, and cost

Required Facility / Equipment	Purpose / Rationale	Estimated Cost
Tunnel Access	Necessary to test the robot's obstacle-avoidance capabilities in its intended environment	$0
LiDAR Sensor	Will be used to improve the robot's navigational capabilities	$167
Power Bank	Will be used to power the LiDar Sensor	$35
Create 3 Mailbox	Will be used to hold mail on the Create 3	$20

11.1 Justification of Purchases

By Cassidy Pacada

Sensors

Table 25: Comparison of sensors based on power usage, effectiveness, and price

	Meng Jie TF-Luna	Existing IR Sensors	Slamtec RPLIDAR AM18
Power Usage	< 0.35W	Around 0.01W	0.5W
Range of Effectiveness (Distance)	Effective at a range of up to 8 meters.	Effective at a range of approximately 20 cm.	Effective at a range of up to 12 meters.
Directional Range of Effectiveness	Effective in only one direction.	Effective in only one direction per sensor.	Effective range of 360°.
Price	$56.17	Free (re-used)	$167

Multiple LiDAR options were considered such as the Meng Jie TF-Luna which is effective up to 8 meters and is advertised to have low power consumption. [14] This was an asset as the LiDAR draws power directly from the robot and the team did not want it to drain the battery too quickly between home bases. Additionally, the MengJie LiDAR was on the cheaper end of LiDAR options which was valuable as the team wanted to ensure the given budget was used effectively.

Another option that was considered was to retain the existing IR sensors but to add more of them around the robot for better coverage. The team already had a number of IR sensors so the added cost would be minimal. Additionally, as the previous team had been working with the sensors, this option meant that it would be possible to retain much more of the existing code.

In the end, the team chose to use the more expensive LiDAR option as its benefits outweighed the benefits of the other options. One of the major problems with the IR sensors was that they were only effective from a distance of approximately 20cm. Even if the team had more directional coverage, the robot would need to remain extremely close to the wall in order to navigate properly. This may have been an issue in the middle of wide intersections where the robot could lose the wall and become disoriented.

The MengJie LiDAR was not selected as it can only range in a single direction. Even with its 8 meter range, this would give us the same issue that the robot has with the single IR sensor where the robot would have several blind spots on its left, front, and back sides. The LiDAR that was selected has 360° capabilities, meaning that multiple purchases would not have to be purchased to obtain full coverage. As well, it has a range of 12 meters which ensures that it should be able to sense walls from the middle of intersections with no difficulty. These advantages made it the clear option for our project and justified the additional cost.

Battery Bank

Table 26: Comparison of battery pack candidates based on supplied power, size, and price

	Anker Portable Charger [15]	**Anker PowerCore 10000 Portable Charger [16]**
Supplied Power	15W	12W
Size	6.2in x 2.9in x 0.8in	3.6in x 2.3in x 0.9in
Price	$56.50	$38.42

The main candidates considered for the battery bank that would be used to power the Raspberry Pi and the LiDAR were the Anker Portable Charger and the Anker PowerCore 10000

Portable Charger. The biggest determining factor for our decision to choose the Anker PowerCore was that it was smaller than the Anker Portable Charger. The robot has a small surface area and a lot of the space is already being taken up by the Raspberry Pi, the LiDAR, and various wires. Although a chassis has been designed to hold everything in place, the team prefers to preserve as much space as possible so the robot will have space for the mail holder.

Even though the Anker Portable Charger supplies more power than the Anker PowerCore, the team calculated the amount of power drawn by the Raspberry Pi and the LiDAR and determined that a supply of 12W was sufficient. A secondary benefit to the Anker PowerCore is that it is the cheaper option which allows the team to save the budget for any other supplies that may be needed.

Create 3 Mailbox

Table 27: Comparison of mailbox options based on size and price

	Staples Metal Mesh Tabletop File Organizer [17]	Staples Wire Desk File Sorter [18]	Merangue Magnetic Mesh Magazine Holder [19]
Size	10.83in x 13.19in x 5.51in	8.25in x 7.375in x 5.875in	12.05in x 9.33in x 2.48in
Price	$19.99	$15.99	$26.99

Many different types of mailbox, letter tray, and file sorters were considered for the Create 3 chassis that would be used to hold mail on the robot. Due to time constraints and 3D printing size restrictions, it was determined that a store bought file holder would be ideal. The top candidates of file holders were identified as the Staples Metal Mesh Tabletop File Organizer, the Staples Wire Desk File Sorter, and the Merangue Magnetic Mesh Magazine Holder. The main factor in deciding which option to use is the width as it is ideal to have as much space between the LiDAR and mailbox as possible to avoid interference. Due to the Staples Metal Mesh Tabletop File Organizer having a width of 5.51in, it was determined that it was not feasible as an option. Although the Staples Wire Desk File Sorter has a width of 7.375in, it can easily be cut to a smaller size as it has compartments that are connected by 3 metal wires.

Between the Staples Wire Desk File Sorter and the Merangue Magnetic Mesh Magazine holder, it was determined that the Staples Wire Desk File Sorter was the best option as it is $11 cheaper and has space underneath for the LiDAR wire to travel below the mail without any modifications.

12 References

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics vol. 7, May 2022. (accessed Sep. 22, 2023).

[2] “Create 2 Robot: What We Offer” edu.irobot.com. https://edu.irobot.com/what-we-offer/Create-robot (accessed Sep. 22, 2023).

[3] ”Overview - Create 3 Docs” edu.irobot.com. https://iroboteducation.github.io/Create3_docs/hw/overview/ (accessed Sep. 22, 2023).

[4] “What is LiDAR?” Synopsys. https://www.synopsys.com/glossary/what-is-lidar.html (accessed Sep. 30, 2023).

[5] “Slamtec RPLIDAR A1 Low Cost 360 Degree Laser Range Scanner. Introduction and Data Sheet” Slamtec. https://bucket-download.slamtec.com/d1e428e7efbdcd65a8ea111061794fb8d4ccd3a0/LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf (accessed Oct. 8, 2023).

[6] “Spring Framework” spring.io. https://spring.io/projects/spring-framework/ (accessed Feb. 18, 2024).

[7] “Spring Python’s documentation” docs.spring.io. https://docs.spring.io/spring-python/1.2.x/sphinx/html/index.html (accessed Feb. 18, 2024).

[8] “Microsoft Azure Web Apps” azure.microsoft.com. https://azure.microsoft.com/en-ca/products/app-service/web (accessed Mar. 4, 2024).

[9] “Thymeleaf 3.1” thymeleaf.com. https://www.thymeleaf.org/documentation.html (accessed Mar. 4, 2024).

[10] “JUnit User Guide” junit.org.https://junit.org/junit5/docs/current/user-guide/#writing-tests (accessed Mar. 18, 2024).

[11] “Raspberry Pi 4 Tech Specs” Raspberry Pi. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/ (accessed Oct. 15, 2023)

[11] "Battery Power from Create 2" iRobot Education. https://edu.irobot.com/learning-library/battery-power-from-Create-2. (accessed Oct. 15, 2023)

[12] "Create 2 losing serial communication after toggling full to passive while charging" Robotics Stack Exchange. https://robotics.stackexchange.com/questions/15433/Create-2-losing-serial-communication-after-toggling-full-to-passive-while-chargi (accessed Oct. 27, 2023)

[13] "Meng Jie TF-Luna small size and Low Power, lidar range finder sensor module, single-point micro ranging module, ranging from 0.2-8m, Resolution 1 cm, compatible with UART / I2C communication interface," Amazon.ca: Electronics, https://www.amazon.ca/Single-Point-Resolution-Compatible-Communication-Interface/dp/B09DCLPYV1/ref=sr_1_18?keywords=lidar&qid=1696789560&s=electronics&sr=1-18 (accessed Oct. 8, 2023).

[14] "Anker Portable Charger, Power Bank, 20K" Amazon.ca: Electronics, https://www.amazon.ca/Anker-PowerCore-Technology-High-Capacity-Compatible/dp/B07S829LBX/ref=sr_1_6?crid=1Z7F36BX965BO&keywords=anker%2Bpower%2Bbank%2B5v%2B3a&qid=1697405645&sprefix=anker%2Bpower%2Bbank%2B5v%2B3a%2Caps%2C85&sr=8-6&th=1 (accessed Oct. 15, 2023).

[15] "Anker powercore 10000 Portable Charger," Amazon.ca: Electronics, https://www.amazon.ca/Anker-PowerCore-Ultra-Compact-High-speed-Technology/dp/B0194WDVHI/ref=sr_1_29?crid=AKV3WJBMCKR8&keywords=anker%2Bpower%2Bbank%2B5V%2F2.4A&qid=1697245268&sprefix=anker%2Bpower%2Bbank%2B5v%2F2%2B4a%2Caps%2C73&sr=8-29&th=1 (accessed Oct. 15, 2023).

[16] "Staples Mesh Tabletop File Organizer - Letter Size - Silver", Staples.ca, https://www.staples.ca/products/3006171-en-staples-metal-mesh-tabletop-file-organizer-letter-size-silver (accessed Feb, 29, 2024).

[17] "Staples Wire Desk File Sorter - 7-Slot - Black", Staples.ca, https://www.staples.ca/products/482705-en-staples-wire-desk-file-sorter-7-slot-black (accessed Feb, 29, 2024).

[18] "Merangue Magnetic Mesh Magazine Holder - Single File - Letter Size - Black", Staples.ca,https://www.staples.ca/products/568839-en-merangue-magnetic-mesh-magazine-holder-single-file-letter-size-black (accessed Feb, 29, 2024).

[19] "iRobot® Create® 3 Printable Compute Board Parts" iRobot Education Create 3 Docs, https://iroboteducation.github.io/Create3_docs/hw/print_compute/ (accessed Feb 6, 2024)

[20] “iRobot® Create® 3 Printable Sensor Mount Parts” iRobot Education Create 3 Docs, https://iroboteducation.github.io/Create3_docs/hw/print_sensor_mounts/ (accessed Feb 6, 2024)

Appendix A: Chassis Design

Create 2 chassis design

The Create 2 chassis was designed to hold the external components on top of the robot as well as the mail to be delivered. It was designed using CAD software and 3D printed. Photos of the design as well as technical drawings can be seen below:

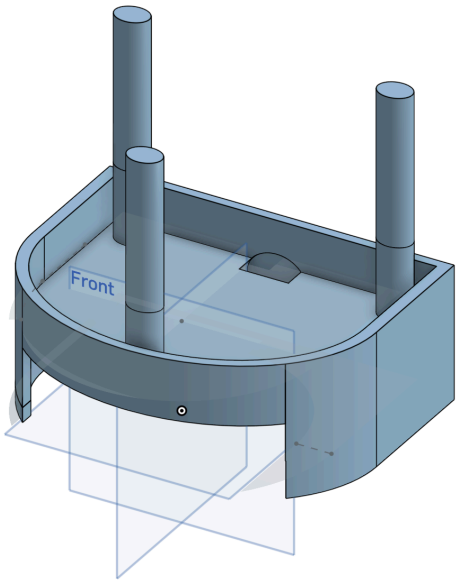


Figure 45: The chassis design for the robot holding the circuits for the robot

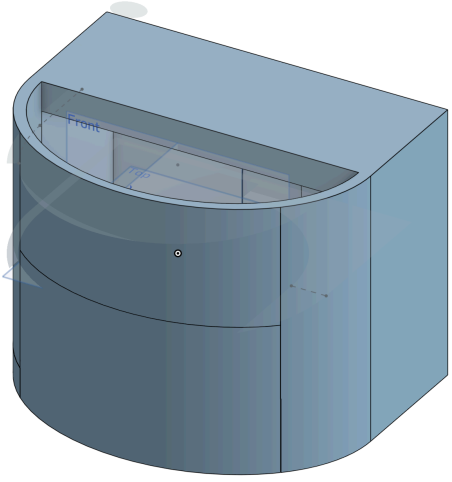


Figure 46: The design of the mailbox for the robot

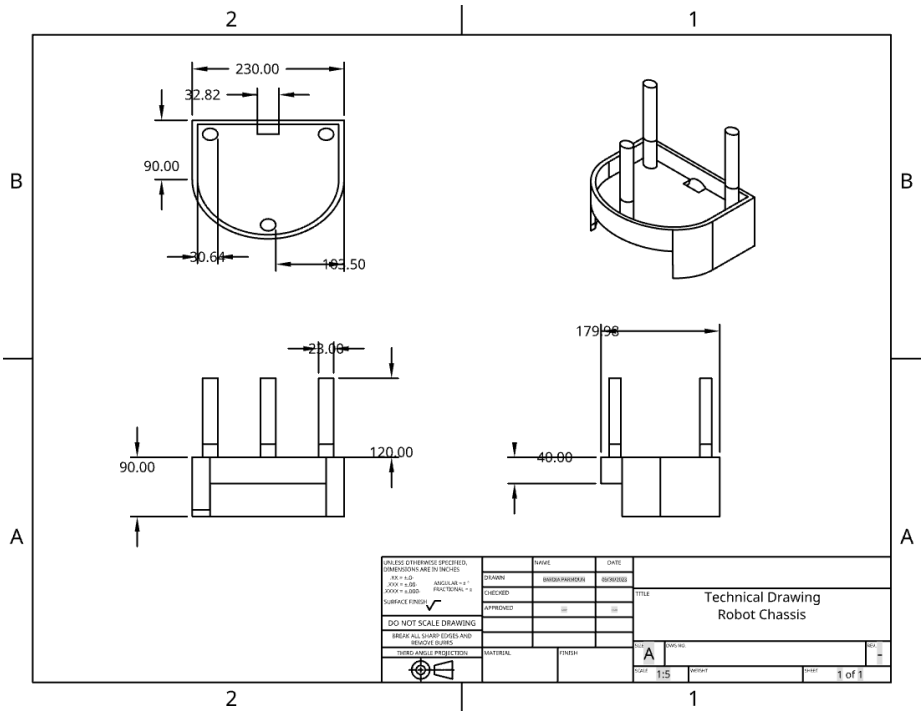


Figure 47: The technical drawing for the robot chassis with measurements

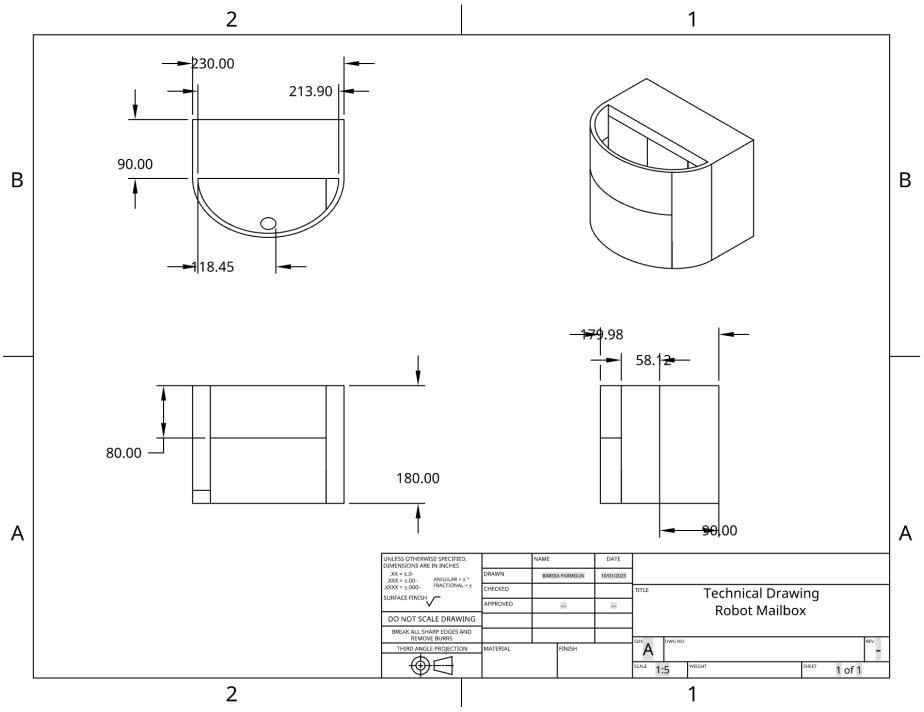


Figure 48: The technical drawing for the robot mailbox with measurements

Create 3 chassis design

For the Create 3, the chassis design involves a stand alone mailbox as well as mounts for the LiDAR and Raspberry Pi. The Create 3 mailbox will be designed to fit an interdepartmental mail envelope which is 10in x 13in. This is too big for 3D printing so a store bought wire desk file sorter was chosen as discussed in Section 11. The design of the mounts for the Raspberry Pi and LiDAR which were provided by iRobot are shown below:

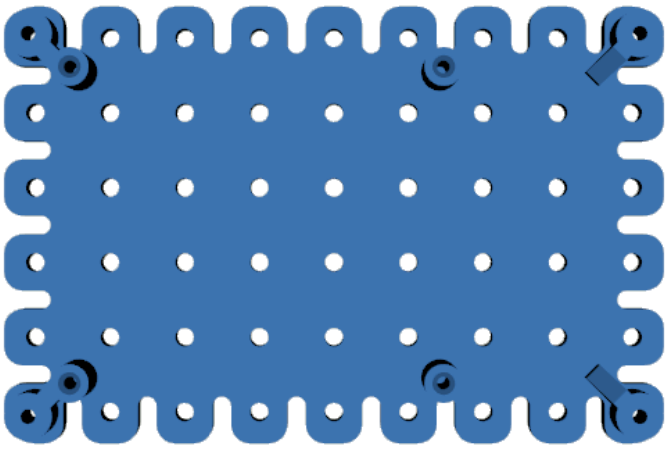


Figure 49: Mount design for attaching the Raspberry Pi to the cargo bay [20]

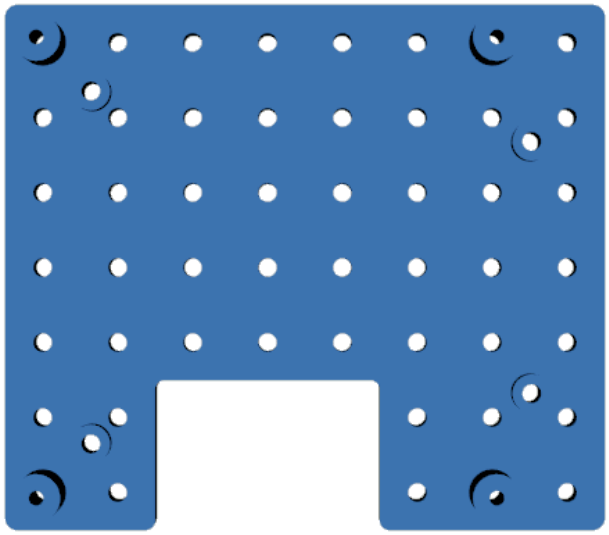


Figure 50: Mount design for attaching the RPLIDAR A1 to the faceplate [21]

To fit the robot without blocking the LiDAR, the file sorter was modified by cutting off 3 compartments. This modification can easily be done using wire cutters and cutting the 2 horizontal wires after the 3 compartments. The file sorter before modification, and then on the robot is shown in the figures below:

Figure 51: Staples Wire Desk File Sorter Used for Create 3 Mailbox [17]

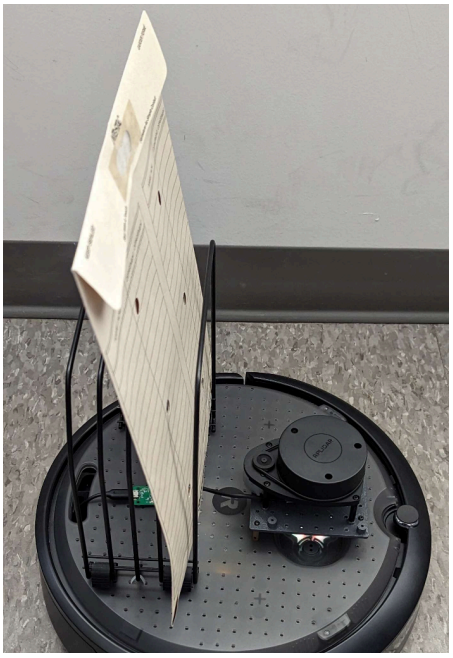


Figure 52: Modified mailbox attached to the Create 3

Appendix B: Hardware Setup

Setting up the hardware for Create 2

Refer to Figure 9 of Section 7.6 for a schematic of the hardware connections and Figures 15 and 16 of Section 8.8 for images of the implementation.

Connecting the main brush motor driver to a usb c cable

1. Remove the vacuum brush and housing from the bottom of the Create 2.
2. Cut the red and black wires going to the motor and remove the motor
3. Solder the red motor wire to a 2.2mH, 1.4A inductor
4. Solder the other pin of the inductor to the positive input of a 15W DC-DC converter
5. Connect the black motor wire to the ground input of the 15W DC-DC converter
6. Cut one end off of a USB C cable and connect the red wire to the positive output terminal and a black wire to the negative output terminal of the 15W DC-DC converter
7. Drill a hole into the cargo compartment and route the USB C cable to the top of the robot

The USB C cable will now provide power from the main brush motor driver to the external components.

Connecting the chassis to the robot

1. 3D print the chassis design shown in Appendix A and attach the parts together
2. Using glue or double sided tape, attach the chassis to the top of the robot so that the back of the chassis is aligned with the safe cutting area lines and the serial port is not blocked
3. Using glue or double sided tape, attach the LiDAR to the top of the chassis so that the motor is on the back of the robot.
4. Place the Raspberry Pi 4B a power bank capable of providing a 15W (5V, 3A) output on the middle level of the chassis
5. (Optional) Using double sided tape, attach the Pi and power bank to the chassis

Connecting the external components

1. Connect the USB C cable from the main brush motor driver to the power bank input
2. Connect the power bank output the the USB C input of the Raspberry Pi 4B
3. Connect the serial-to-USB cable to the serial port on the Create 2 and route it up from underneath the chassis (using the area underneath to conceal the cable)
4. Connect a USB-to-micro USB cable to the Pi and a micro USB-to-serial converter.
5. Using the LiDAR serial cable, connect the LiDAR pins to the USB-to-serial converter.

Setting up the hardware for Create 3

Refer to Figure 9 of Section 7.6 for a schematic of the hardware connections and Figures 15 and 16 of Section 8.8 for images of the implementation. For the Create 3, power is provided when connecting the external components.

Connecting external components

1. Unscrew the faceplate of the Create 3 and ensure that the USB/BLE toggle on the adapter board is in the USB position. Screw the faceplate back into position.
2. Screw the 3D printed Raspberry Pi 4B mount in the cargo bay and attach the Pi.
3. Connect a USB-C cable between the USB-C port in the robot's cargo bay and the Raspberry Pi 4B's USB power input.
4. Connect a USB-to-micro USB cable to the Pi and a micro USB-to-serial converter. Route the cable through the cable passthrough hole to the top of the robot.
5. Screw the 3D printed LiDAR mount and standoffs into the faceplate and attach the LiDAR.
6. Using the LiDAR serial cable, connect the LiDAR pins to the USB-to-serial converter.
7. Use zip ties to attach the wire desk file sorter into the faceplate of the robot to use as the mailbox.

Appendix C: Software Setup

Setting up the Project for Create 2

Installing the Ubuntu Image

1. Using the Raspberry Pi Imager, install Ubuntu Server 20.04.x
2. SSH is already enabled, login with user:ubuntu, password:ubuntu and change the password.

Creating an account on the image

1. ***sudo su - root***
2. ***hostnamectl set-hostname***
3. ***adduser robot***
4. ***su - robot***

Setting up Wifi

1. Install wpasupplicant using: ***sudo apt install wpasupplicant***
2. Initialize the wpasupplicant file using: ***wpa_passphrase your-ESSID your-wifi-passphrase | sudo tee /etc/wpa_supplicant.conf***
3. Edit the wpa_supplicant file using: ***sudo vim /etc/wpa_supplicant.conf***

```
network={
        ssid="eduroam"
        key_mgmt=WPA-EAP
        indentity="<school wifi login username>"
        password="<wifi password>"
}
```

Figure 53: The contents of the wpa_supplicant file

Configuring ROS

1. Install **ROS Foxy** by following this link: https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html
2. Run ***sudo rosdep init***
3. Run ***source /opt/ros/foxy/setup.bash*** (must be run with every new terminal)

Creating a local codespace and clone all the necessary code

1. Run ***mkdir -p ~/cmds_ws/src***
2. Run ***cd ~/cmds_ws/src***
3. Run ***git clone git@github.com:bardia-p/carleton-mail-delivery-robot.git***
4. Follow this link to the get Create driver: https://github.com/AutonomyLab/create_robot/tree/foxy

5. Get the LiDAR package by running: ***git clone git@github.com:Slamtec/sllidar_ros2.git***
6. Install bluepy by running: ***sudo pip install bluepy***
7. Install requests by running: ***sudo pip install requests***
8. Run ***cd ~/cmds_ws***
9. Run ***colcon build***

Running the code

1. Run ***sudo -s***
2. Run ***cd ~/cmds_ws***
3. Run ***source /opt/ros/foxy/setup.bash***
4. Run ***source /home/ubuntu/cmds_ws/install/setup.bash***
5. Run ***ros2 launch mail_delivery_robot robot.launch.py 'robot_model:=Create_2'***

Running the unit tests

1. Run ***cd ~/cmds_ws***
2. Run ***colcon test --packages-select mail_delivery_robot --event-handlers console_cohesion+***

Running the integration tests

1. Run ***sudo -s***
2. Run ***cd ~/cmds_ws***
3. Run ***source /opt/ros/foxy/setup.bash***
4. Run ***source /home/ubuntu/cmds_ws/install/setup.bash***
6. Run ***ros2 launch mail_delivery_robot [testfile].launch.py***

Setting up the Project for Create 3

For troubleshooting and additional help with the setup process, refer to the Create 3 Docs at https://iroboteducation.github.io/Create3_docs/. The steps below will summarize the most efficient software setup process for running the project the Create 3.

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

1. Download the latest version of the Create 3 firmware: edu.irobot.com/Create3-latest-fw.
2. Power on the robot by plugging it into the charging dock.
3. Press and hold both side buttons until the light ring turns 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_IMPLEMENTATION dropdown is set to rmw_cyclonedds_cpp.
9. If you changed the RMW implementation, restart the application.
10. Navigate to the 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 SD card slot)

1. Using the Raspberry Pi Imager, select Ubuntu Server 20.04.x LTS (64-bit).
2. Select the gear icon and ensure that SSH is enabled, and that a username and password are set. The wireless LAN setup included will not work with the Create 3 and Carleton Wi-Fi so disregard the option.
3. Write the image to the Raspberry Pi 4B's microSD card.
4. Eject and reinsert the microSD card to access the system-boot partition.
5. Edit the "config.txt" file to add "dtoverlay=dwc2,dr_mode=peripheral" at the end.
6. Edit the "cmdline.txt" file to add "modules-load=dwc2,g_ether" after "rootwait".
7. Edit the "network-config" file to add Wi-Fi information as shown in Figure 36 below.

```
wifis:
    wlan0: # Connection to Carleton Wi-Fi
      dhcp4: true
      access-points:
         "eduroam":
             auth:
               key-management: eap
               method: peap
               identity: <school wifi login username>
               password: <wifi passowrd>
    usb0: # Connection to Create 3 Robot
      dhcp4: false
      optional: true
      addresses: [192.168.186.3/24]
```

Figure 54: The contents of the network-config file

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

Creating an account on the image

1. ***sudo su - root***
2. ***hostnamectl set-hostname***
3. ***adduser robot***
4. ***su - robot***

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 already 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/systemd/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***

Configuring ROS

1. Install **ROS Humble** by following this link: https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.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 placing it on the charging dock.
8. Run ***ros2 topic list*** and ensure the Create 3 API topics are displayed (If not, ensure the Pi is connected to the robot when the robot boots and ping the robot at 192.168.186.2.)

Creating a local codespace and clone all the necessary code

1. Run ***mkdir -p ~/cmds_ws/src***
2. Run ***cd ~/cmds_ws/src***
3. Run ***git clone git@github.com:bardia-p/carleton-mail-delivery-robot.git***
4. Get the LiDAR package by running: ***git clone git@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 ~/cmds_ws***
8. Run ***colcon build***

Running the code

1. Run ***sudo -s***
2. Run ***cd ~/cmds_ws***
3. Run ***source /opt/ros/humble/setup.bash***
4. Run ***source /home/ubuntu/cmds_ws/install/setup.bash***
5. Run ***ros2 launch mail_delivery_robot robot.launch.py 'robot_model:=Create_3'***

Running the unit tests

1. Run ***cd ~/cmds_ws***
2. Run ***colcon test --packages-select mail_delivery_robot --event-handlers console_cohesion+***

Running the integration tests

1. Run ***sudo -s***
2. Run ***cd ~/cmds_ws***
3. Run ***source /opt/ros/humble/setup.bash***
4. Run ***source /home/ubuntu/cmds_ws/install/setup.bash***
5. Run ***ros2 launch mail_delivery_robot [testfile].launch.py***

Setting up Firebase for IP address access

The CU Wireless network allocates a new IP to the Pi when it is started, so in order to get the current IP address for ssh access, a Firebase database is 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 "<PROJECT_NAME>.firebaseapp.com", the database URL value should be "https://<PROJECT_NAME>-default-rtdb.firebaseio.com/", and storageBucket should be "<PROJECT_NAME>.appspot.com". Replace <PROJECT_NAME> 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 mutliple 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: Project Bring Up Process

These guides assume that the hardware and software setup is completed and the program is ready to be started.

Create 2 Bring Up Process

1. Connect the power bank to the USB C port of the Raspberry Pi 4B.
2. Connect the USB-to-Serial cable to the top USB 3.0 port of the Pi and the Serial port of the Create 3.
3. Press the clean button.
4. Connect the LiDAR to the bottom USB 3.0 port of the Pi.
5. SSH to the Raspberry Pi 4B.
6. Run ***sudo -s***
7. Run ***cd ~/cmds_ws***
8. Run ***source /opt/ros/foxy/setup.bash***
9. Run ***source /home/ubuntu/cmds_ws/install/setup.bash***
10. Run ***ros2 launch mail_delivery_robot robot.launch.py 'robot_model:=Create_2'***

Create 3 Bring Up Process

1. Ensure that the Raspberry Pi 4B is plugged into the Create 3 by the USB-C cable.
2. Ensure that the LiDAR is plugged into the bottom USB 3.0 port of the Raspberry Pi 4B.
3. Ensure that the robot is off of the charging dock and powered off. To power off the robot, hold the power button for 10 seconds.
4. Turn on the robot by placing it on the charging dock.
5. Wait for the chime and green light from the robot.
6. SSH to the Raspberry Pi 4B.
7. Run ***ros2 topic list*** and ensure that the Create 3 topics are listed.
8. Run ***sudo -s***
9. Run ***cd ~/cmds_ws***
10. Run ***source /opt/ros/humble/setup.bash***
11. Run ***source /home/ubuntu/cmds_ws/install/setup.bash***
12. Run ***ros2 launch mail_delivery_robot robot.launch.py 'robot_model:=Create_3'***