Group 10

System Design Project

Mentor: Vesko Stefanov

Client: Iordanis Chatzinikolaidis

18th April 2018

Technical Report

Alex Shand s1530411
Campbell Scott s1530451
Grzegorz Wilk s1504889
João Catarino s1512284
Katie Worton s1530238
Rosina Paige s1407973
Stephen Waddell s1346249

1. Introduction

We at *!spam* have set out to improve office efficiency. We built a robot that delivers mail to the employee's desks all by itself. Our system includes the robot, a web interface, and a conversational interface. We achieved everything we planned in the Project Plan in the time proposed, as well as produced the additional Conversational Agent.

This document aims to describe our system's implementation, explain our methodologies, reflect on the work produced and justify our design decisions.

2. Milestones

For our Project Plan we decided to align our objectives with the structure of the course, using client demos and presentations as key deadlines. We utilized the structure of this Project Plan throughout:

- **First demo** - build a robust base structure with integrated dispensing hardware with the ability for the robot to follow a line
- **Second demo** - incorporate sensors for location feedback and collision avoidance, navigate a system of lines with route planning as well as create the web interface with a manual letter input mode, a notification system in case of unexpected hardware faults and live status and settings information
- **Third demo** - add a classifying module for automatic recipient identification
- **Final demo** - polish the robot's functionality and focus on preparing both our reports and the final client demo

As an ambitious team, we pushed ourselves even further than our planned milestones to produce an all-in-one autonomous product. To achieve this for the final demo we added:

- A conversational user interface that utilizes Google Assistant for hands-free robot control
- A secondary email server to notify employees that their mail is out for delivery
- Robotic voice feedback and text on the EV3 displays to provide instructions for users
- Mail error checking to ensure we truly would not be spreading spam around the office

From our Project Plan, we did not pursue the proposed extension feature that "The robot would also be able to audibly call for help if it encounters a closed door or has to move between floors in a lift", which was not formally adopted as part of our milestones. We chose to discontinue this suggestion as we would wish to implement a better solution to the multi-floor problem than this in the future (section 7.4). As such, we implemented every one of our planned basic and extension features successfully in our milestones.

3. Team Methodology

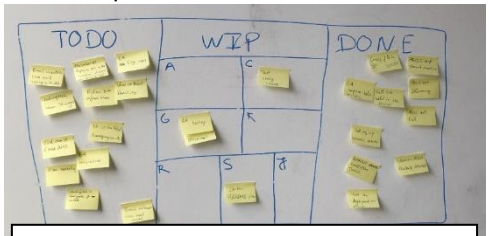

Fig 1. Example sprint to-do board

Our team adopted an agile approach to this project. We met weekly to define the goals for the week, update on our progress and to plan our long-term objectives. This proved to be very useful in discussing what the team members have been doing, generating ideas to solve the problems that emerged and ensuring everyone knew about each other's work. We also had some sprint meetings to speed up the development cycle and break important barriers that could be more easily fixed by the team as a whole.

Our communication strategy was always very open and transparent. In addition to meeting in the robotics lab, we also regularly communicated through Slack. We also used Trello for task management. Although it was not easy to adopt in the beginning, as the team had never used it; in later stages, it became the norm to organize and prioritize our tasks.

Our workflow revolved around GitHub and our shared Google Drive folder where everyone could see, review and improve everyone's work. We set out a plan from the beginning to have two main software branches: "master" and "dev". Our aim was to keep "master" as our last stable version codebase and "dev" (development branch) as our feature integration branch, a staging area for new features that hadn't been released yet. This was intended to maintain a high level of project integrity and enable parallel development. New branches off from "dev" would be made regularly for new independent features; with "merge sessions" with the whole team to merge and integrate our changes into a single branch.

With each merge into "master", the team chose to create a GitHub release. This was intended to both mark the history of our project and as a further effort to maintain code integrity with a 'last-working version'. The team coincided these releases with our client demos: providing a target for us to integrate all our changes and implement our proposed features. Each pull request was consistently checked off by another member of our team. This way we ensured that code quality was a team responsibility and were able to utilize the full software skillset of the team: developing a culture of collaborative ownership.

Our development process also utilized feature freezes 3 days before major deadlines with the focus switching to fixing bugs and not making any major additional changes to the codebase. By structuring ourselves around these releases we were able to maintain focus on the necessary features to be demonstrated and to ensure no recent developments would compromise the presentation demos.

4. Hardware

4.1 Overview

The following numbers refer to numbered elements on the robot:

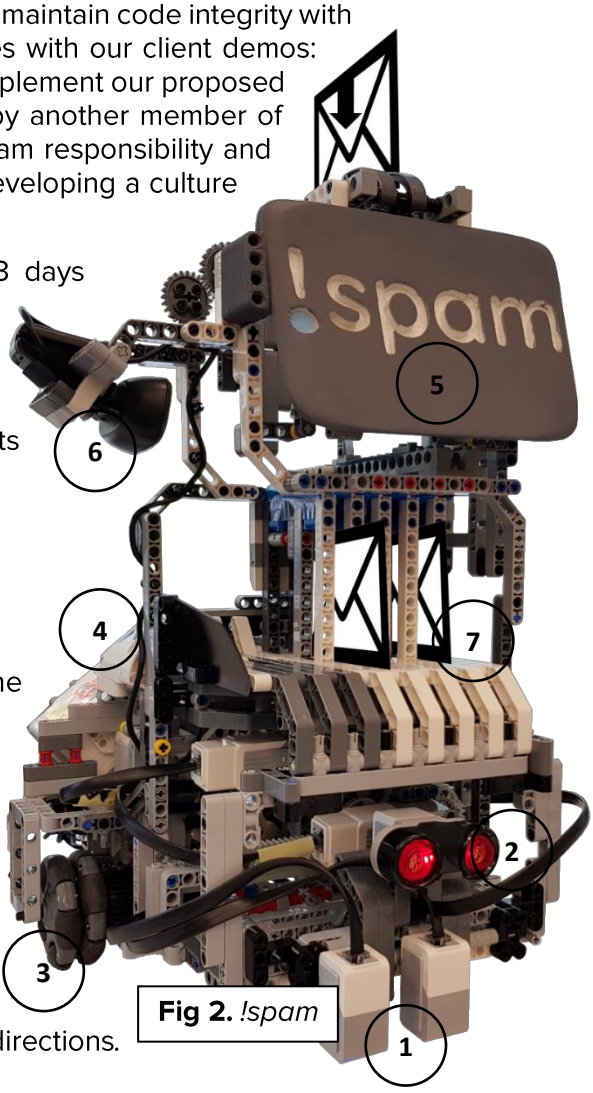

Fig 2. *!spam*

1. The **two color sensors** are the primary way in which *!spam* finds its way around the line-mapped office. The right sensor is used to PID-track the line, while the left is used to detect junctions and desks.
2. The **ultrasonic distance sensor** detects obstacles that are in *!spam*'s way so that it stops instead of crashing into them.
3. *!spam*'s 4 **holonomic wheels** allow it to move in all directions.
4. The 2 **EV3 bricks** behind are the brain of *!spam*.
5. The **classifying module** is where letters are inserted into the robot.
6. The **camera** captures pictures of the letters to determine who their recipients are.
7. The **dispensing module** is the cargo area where the letters get dropped after they are classified. It also includes a separate parcel slot (on the left).

4.2 Construction

The base chassis was built in the first weeks with structural integrity in mind. It was cross-pillar reinforced to achieve this. We chose a 4-wheeled system with individually-powered holonomic wheels, with their axles crossing and equidistant, incident on the circle of rotation, allowing for turning in-place and providing even weight distribution to meet cargo weight requirements. Additionally, the base was constructed with the dispenser assembly already integrated into it to follow the principle of early integration. This decision allowed us to exploit the structural integrity of the dispenser railing to aid the base, lower the center of mass as well as keep it compact.

The robot needed to support the "lifecycle" of letters within it: scanning the letters as they get loaded in, storing them during delivery and dispensing them at the appropriate desks. The chosen scheme was a gravity fed system which stores the letters in vertical slots and dispenses them by lifting the floor of each slot up – allowing the letters to slide forward and out. To limit the number of motors in the system, a prismatic joint going vertically (doing the slot lifting) was mounted on a sideways-moving dispenser assembly following the mentioned dispenser rail within the robot. Belt and gear power-transmission helped in keeping this assembly compact.

Following that, we built the scanner module focusing on reliability. From a fixed insertion slit the letters fall through a transparent plastic chute which guides them to the stopping points (space between slots) for camera capture. We didn't need to introduce more motors on this module because we used the sideways movement of the dispenser assembly to guide the letters. However, this introduced problems of precision, caused by friction. To solve this problem, sliding the upward extension along smooth surfaces proved more accurate than gearing. Finally, through an iterative construction process of the scanner module and the vertical slots, the elements where the letters got caught during their fall got amended.

Aesthetically, we improved the robot by modeling a 3D printed branded hood.

5. Codebase

5.1 Backend

The interactions between all *!spam*'s parts are described:

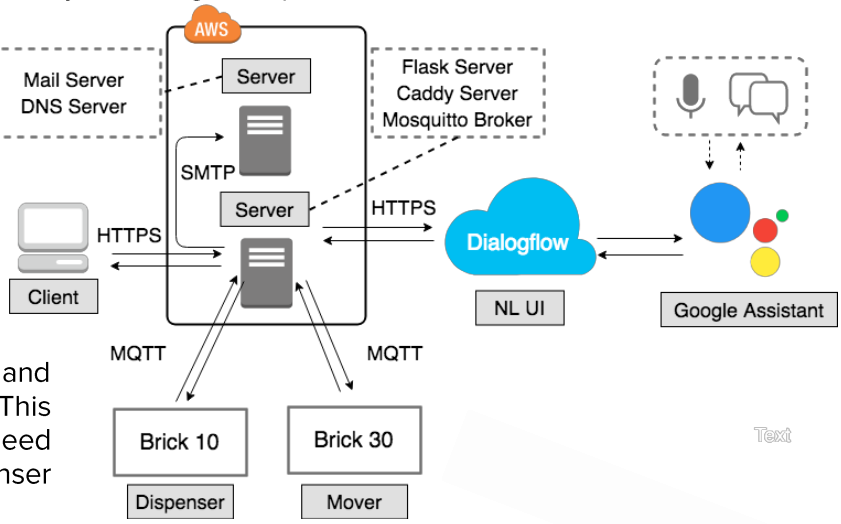

Fig 3. Diagram of system interactions

The robot itself consists of 2 EV3 bricks, one used for movement in the office (Mover) and the other one for dispensing and classifying letters (Dispenser). This decision was made because we need to power 6 motors (2 for the dispenser assembly and 4 for wheels).

Our main server hosts a Mosquitto broker, a Caddy server and a Flask Application. The first is what manages all communications server-to-brick and brick-to-brick. With the publish-subscribe based MQTT protocol, messages are handled in topics and clients subscribe and publish on the relevant topics. We initially experimented with RPyC and a wired connection between the bricks, but it proved too unstable during setup to reliably work, with MQTT thus providing better results (see section testing 6.1). We use the default port for the MQTT protocol (1883). The Caddy server is what handles the requests to ports 80 (HTTP) and 443 (HTTPS). It also manages the SSL

certificate of our domain spamrobot.ml using Let's Encrypt. Lastly, Flask is the framework we used to develop our application, as it provides simplicity, flexibility, control, and is based on Python: a language that the group is very familiar with. Our Flask Application is the main component of our backend. Flask is the web-framework which serves web-pages to the users, communicates through MQTT messages with the EV3 bricks and constructs responses and actions for the Conversational Agent.

All the forms in the web application (login page, report page, automatic and manual mode) are submitted via a `POST` request that sends the inputs as key-value arguments. Also, to solve a notification in the notifications page (clicking a notification), it propagates that message to the server by issuing a `GET` request to the same page, with a query value corresponding to the id of the problem in the database.

In choosing a database, we searched for extensive documentation, ease of use and support with Flask. That led us to SQLite. Although there are disadvantages to SQLite: the nonexistence of multi-user support and the write throughput limitation (only one write operation can happen at any given time), these weren't needed features for our project, therefore, SQLite was an appropriate choice for our system.

To aid the manipulation of the database, we used SQLAlchemy, an Object Relational Mapper that integrates as a Flask extension very easily and facilitates the creation, migration and manipulation of databases.

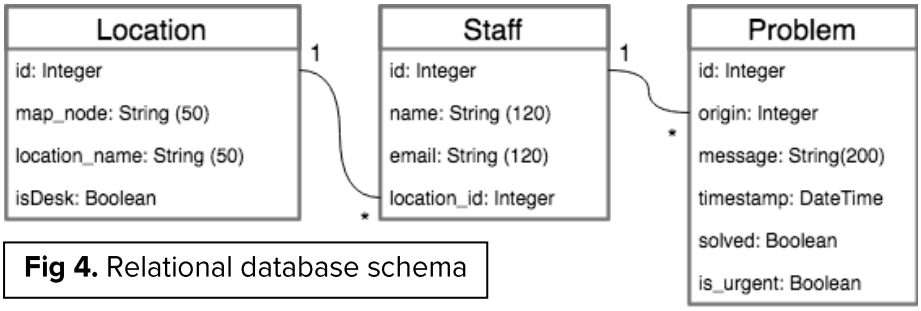

Fig 4. Relational database schema

We used the following packages to help us develop our application:

- **flask**: Flask package.
- **Jinja2**: Templating language used in Flask.
- **sqlalchemy**: Object Relational Mapper (ORM) for relational databases.
- **Pillow**: Python Imaging Library for Image processing.
- **pylibdmtx**: Datamatrix barcode decoder in Python.
- **flask-WTF**: Flask extension for producing forms.
- **flask-SQLalchemy**: Flask extension that integrates the ORM into the application.
- **flask-admin**: Flask extension that produces the settings page for the database.
- **flask-mqtt**: Flask extension that integrates an MQTT client into the application.
- **flask-migrate**: Flask extension that handles database migrations using Alembic.
- **flask-assistant**: Framework for building the Assistant with Dialogflow.
- **flask-mail**: Simple interface to send emails via SMTP.
- **flask-testing**: Extension for enabling automated unit tests.
- **flask-socketio**: Flask extension that allows for low latency and bi-directional communication between browser and server. This is used to show live classification in Automatic Mode.
- **Catch2**: Test Framework for C++. Used to test the path planning library.

Our infrastructure is deployed in Amazon Web Services (AWS) for several reasons. Firstly, it is a free service (for the first 12 months) we can easily access from anywhere. It allows for remote management and therefore we can release software updates to our customers much easier and quicker. Secondly, it provides the customer a way to control their robot without disruption. Lastly, to implement a multi-robot central management system, a feature we intend to develop in the

future, we need to have a central entity that can communicate to all *!spam* robots, and the cloud is the best option for that.

We also use a secondary server for e-mail, to showcase the notification functionality. To set it up, we used mail-in-a-box. We also set custom DNS on that server to route the emails and web traffic appropriately.

5.2 User Interface/ Frontend

5.2.1 Website Interface

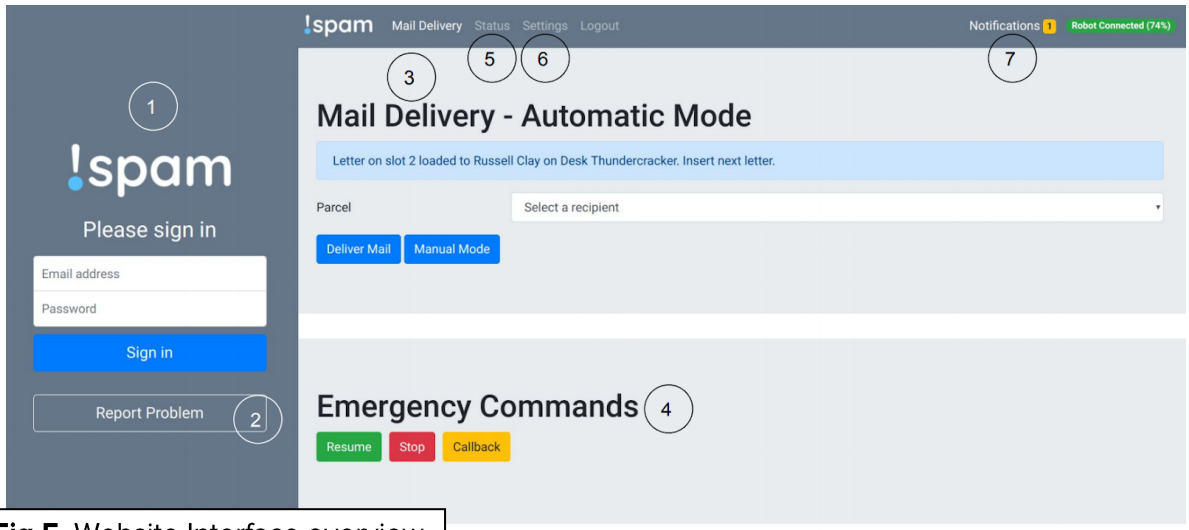

Fig 5. Website Interface overview

!spam's user interface is a website (https://spamrobot.ml) hosted remotely, accessible from anywhere - phones (left) and computers (right) alike (the numbers correspond to the picture's labels):

1. The first screen is the **Login page**. It allows the receptionist to log into the robot control section of the site.
2. The **Report Problem** button is for employees in the office to report problems about their mail or with the robot. It directs them to a simple form without having to log in.
3. The **Mail Delivery** section is the primary way that the receptionist interacts with the robot. The snapshot above shows the default **Automatic Mode** of interacting with *!spam*. There is also a **Manual Mode** where the desk locations for each slot can be set using dropdowns.
4. The **Emergency Commands** section allows the receptionist to issue commands after sending the robot out for delivery.
5. The **Status** page displays connection status, battery life and the last known location of the robot, along with the number of mail pieces delivered. Information about connectivity and battery levels is also visible at a glance in the upper right corner.
6. The **Settings** page allows users to edit who is working at which desks. It is useful if more staff are hired, or if someone changes their desks.
7. The **Notifications** tab is how *!spam* tells the receptionist if something has gone wrong at any point during delivery. This is also where problems reported by office workers appear.

5.2.2 Web Implementation

We developed our web interface using Bootstrap 4, an open source toolkit for HTML, CSS and Javascript, and Jinja2, a templating language for Python. Bootstrap makes it easy to prototype fast, customize and build applications that are responsive by default, allowing our interface to be used in smartphones, tablets and desktops without any visualization problems.

Each page has its own HTML file, however, the main structure is defined in `spam/spam/templates/index.html` which the other files "extend". This helps maintainability and consistency throughout the interface and makes it easy to create new pages on the website.

5.2.3 Conversational Interface

To implement the voice and chat assistant, we used Dialogflow, a tool that allows creating natural language conversation agents. It allows to develop one agent and deploy it in several platforms including Google Assistant, Skype, Facebook Messenger, Alexa and Cortana.

Dialogflow works with Intents and Entities. Intents are the possible actions that can be triggered by the conversational interface. We defined 14 intents for our agent such as "Stop", "Battery Check" and "Deliver Mail". In each of these intents, we defined some training phrases as Dialogflow needs examples of how the same Intent can be asked in different ways to be trained. Entities are the pieces of information needed to perform an action or answer a request. We defined two Entities: @user and @desk. For example, entities are used like this: "Where does @user work?" and @user would resolve to the name spoken. This will then be sent to our flask application via the webhook defined at https://spamrobot.ml/fulfillment, via `POST` requests, which will in turn query the database accordingly, using the functions implemented in the application.

5.3 Office Map

The office map is a grid of 50mm white lines for the robot to follow, and asymmetrical green markings denoting junctions and desks. This is for the robot to center itself on the junctions by overrunning them - this is possible as it has a color sensor on one side only. Along with the information stored in the databases, the server also holds the map. The scheme is encoded as follows (trailing commas required on all lines):

```
<Initial Node>: {<End Node> : (<Distance in cm>, <initial orientation>, <final orientation>),
                 <End Node> : (<Distance in cm>, <initial orientation>, <final orientation>),
                 ...},
```

Specifying each edge in one direction is enough as we assume an undirected graph. The distance is measured from the center of one junction to the center of another. Orientation is the way in which the robot will be facing relative to the 0° starting point orientation - assuming it is east, 270° would be north. Desks are encoded as 0 distance from a junction and only include the orientations for distinguishing which side of the line they are, eg. 90° for a desk on the south of a line going east. The desk's final orientation is always the opposite of initial orientation, eg. 270° in this case.

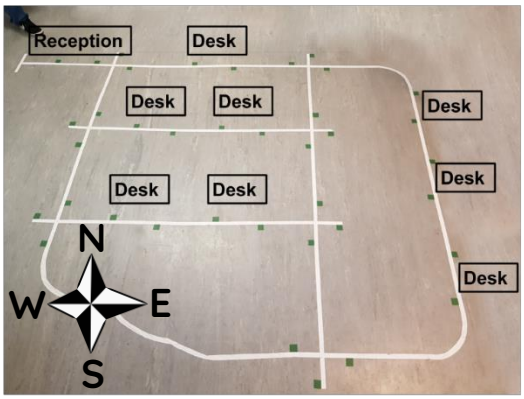

Fig 6. Office map

5.4 Robot/EV3 Software

Because of the limited computational power of the EV3 bricks, we tried to offload as much computation as possible to the server. Therefore, the routing code and data-matrix image recognition for recipient classification reside on the server. Consequently, the robot has limited awareness of where it is or its cargo. As such, both bricks act as state-machines, executing the server's requests and most of their code is responsible for immediate sensing and actuating. To ensure that the server only tries to command the robot when it is connected, both bricks have a dedicated thread that sends an MQTT message every 5 seconds. This message contains battery

voltage information, also used in the Status section of the webpage for displaying remaining battery percentage.

5.5 Dispenser brick

This brick is almost entirely reactive. It listens for MQTT commands from the other brick (to dispense appropriate letter slots while delivering) or the server (to take pictures while loading). The brick's control holds 2 Boolean states, `isLoading` and `isAutomatic`, to determine whether to keep sending pictures or not. It uses the fswebcam library to take them.

5.5.1 Recipient Recognition Protocol

The protocol for image sending limits concurrency through MQTT image-request chaining.

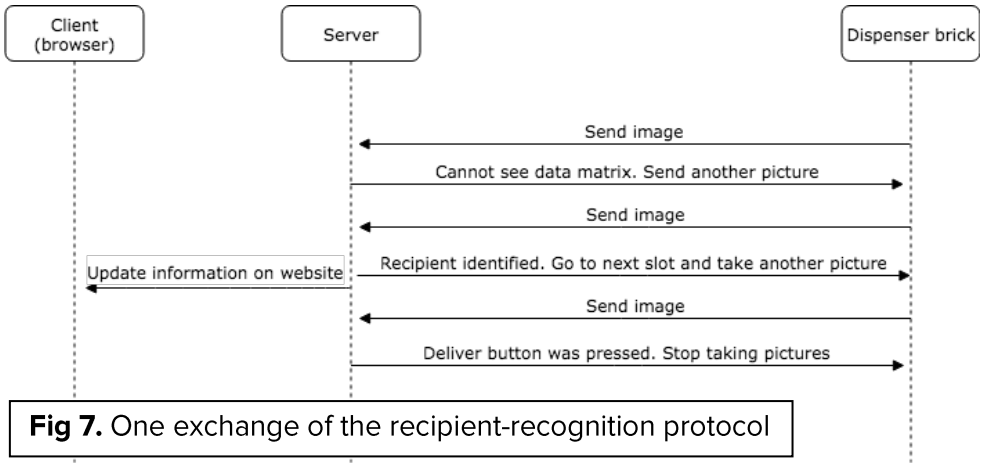

Fig 7. One exchange of the recipient-recognition protocol

While developing this protocol, we witnessed random misclassifications. It was due to the spawning multiple concurrent chains like these by switching between Manual and Automatic modes on the website while the robot was still processing an image to be sent. This is because Automatic mode causes the brick to take the first photo. We solved it by ensuring that the old chain of communication dies out prior to initiating a new one.

5.5.2 Recipient Recognition from the server's perspective

The server automatically detects and decodes the data-matrix image sent by the robot using pylibdmtx. It queries the database for the desk locations of the identified recipient. After each scan, it sends a message to the client browser through SocketIO to notify about whether the scan was successful or not (e.g. person found or not). During this process, the server collects the locations of each inserted letter to perform route planning (section 5.6.3) when it is instructed to do so.

5.5.3 Dispenser Library

The dispenser library is responsible for controlling the dispenser assembly. It exposes 2 coroutines, `stop` and `dump`. Both coroutines position the assembly underneath (or between) the appropriate slots and execute some `in_between_action`. In the case of `dump`, it is shaking the letter out from the slot. In case of `stop` on a wall between slots, it is waiting for a letter to be captured by the camera and then dropped in an appropriate slot. It was implemented as a coroutine because the time for which the letters need to be stopped could vary. But since the two functions only differ in the `in_between_action`, both were implemented as a partial function.

To improve reliability, the dispenser assembly is calibrated on an inner wall, otherwise the sideways-movement imprecisions accumulated into big errors. Unfortunately, the ev3dev library function `run_to_abs_pos` would sometimes cut power to the motors incorrectly, so it was reimplemented using `run_forever` whilst checking odometry values. This allowed us to speed up the initial motion, and slow it later for precision. Additionally, a shake while dispensing was implemented to reduce the number of stuck letters.

5.6 Mover brick

5.6.1 Threaded control

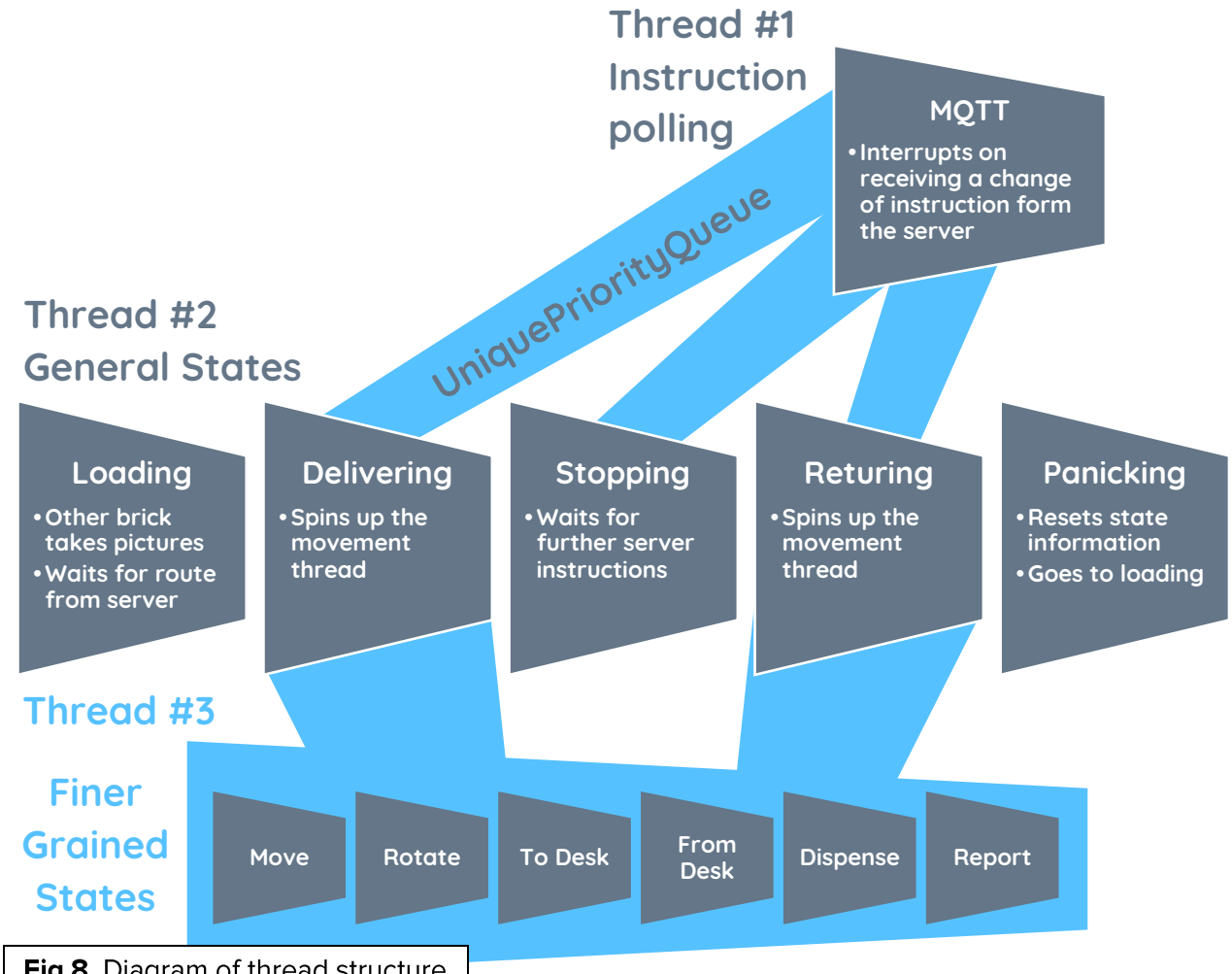

Fig 8. Diagram of thread structure

The Mover's state-machine control interacts asynchronously with our server through MQTT and with a finer-grained state-machine. On a high level, the main thread (#2) holds general state information and spins up child move threads (#3), while another thread (#1) waits for any server instructions. The move child thread executes sequences of movements. Let's say we are in delivering state: If a "Stop" command is issued from the server, it will be passed to the main thread via a thread-safe queue (its unique and priority, to handle multiple requests appropriately). At this point, the main thread (#2) will throw an exception across thread boundaries, asking the move thread (#3) to die. The move thread will then stop the motors, save the route covered so far with centimeter precision (so that the same path may be "Resumed" by the server) and exit. The sharing of variables between the general and finer state machines is done via the use of global variables with appropriate locks.

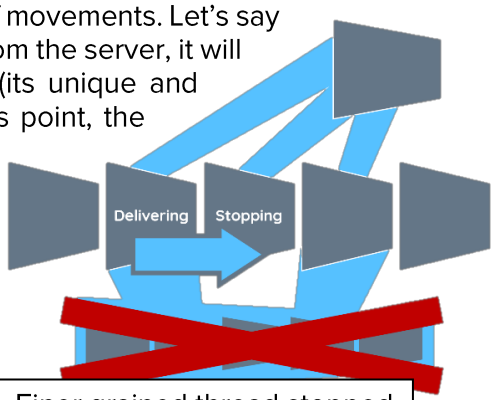

Fig 9. Finer-grained thread stopped

5.6.2 Movement Library

The aforementioned fine-grained move thread calls into the movement library for actions such as move and rotate. We created this library to abstract away motor control. Common functionality is shared via partial functions and hard-coded values are factored out into a configuration file. Upon calling `move.rotate(50, 30, right)`, the robot will rotate 50 degrees to the right, expecting a line to be there. 30 specifies the tolerance in percentage of where the robot will search for the line; if it doesn't find it, it will panic. This external verification of the onboard odometry was done for redundancy. Similarly, `forward(50, 30)` will make the robot move 50cm forward while following the white line, with 30% tolerance, expecting to see a green marker with its color sensor. Line-following was initially done with 2 color sensors, trying to keep the line between the two. This was replaced with a modified PID controller based on github.com/Klabbedi/ev3. It changes the difference in wheel velocity based on the range of reflectivity values – aiming for the middle of some two extremes, in our case set to follow an edge of the line. This way, we only use one color sensor for line following and the other for external verification, while improving reliability and smoothness of movement at the same time.

To make the PID control feasible for our holonomic setup, we had to calculate by how much the front and back wheels should move ($\dot{V}_x$). Fortunately, we could operate the PID control in velocities ($\dot{V}_1$ *and* $\dot{V}_2$) rather than torque, allowing us to do so. The key here was calculating the immediate center of rotation and measuring the change in the projection of the vector pointing from it to the front wheel onto the one pointing to the center of the robot. This change is measured in relation to the angular velocity ($\dot{\theta}$). This quantity gives precisely the speed of the front and back wheels.

Furthermore, thanks to our 3-degree of freedom holonomic wheels, we could make the robot follow a straight line while rotating through 90°. This is used in approaching desks (To and From Desk) and calculated by adding the rotatory ($\dot{V}_3$) and diagonal ($\dot{V}_1$ *and* $\dot{V}_2$) speeds. This requires updating the diagonal speeds each cycle in relation to the angle rotated through so far (Ω).

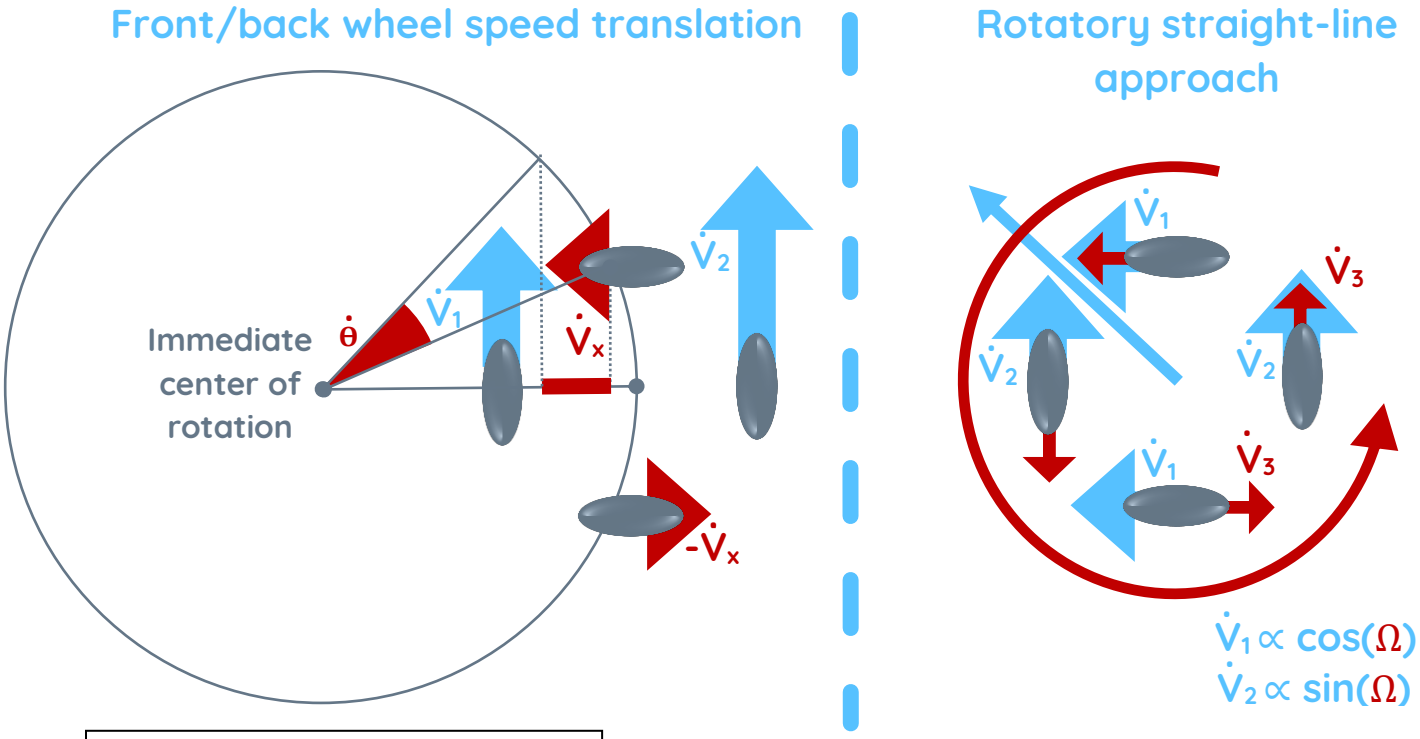

Fig 10. Holonomic property diagrams

5.6.3 Route Planning from the server's perspective

The path planning algorithm uses Dijkstra's algorithm to find the optimal path between the recipients desks, using the stored office map configuration and distance measurements. To avoid an exponential blowup, the order of desks is chosen greedily based on shortest distance.

Although we aimed to run this code on the EV3 brick (that is why it is written in C++), we had to offload this to the server as it took too much time and power on the robot itself. Upon finding the route, the server generates the sequence of commands necessary for the robot to get there/dispense and interleaves them with report commands. This is so that while the robot is delivering, it will keep the server updated on its location (necessary for callback). After this computation, this sequence of commands is sent to the robot as a json via MQTT, emails are sent to the respective recipients and the robot switches from `Loading` state to `Delivering` state.

5.6.4 Navigating

!spam's progress through the server-precomputed instruction sequence is described below. The finer grained state thread handles each instruction appropriately.

`Move`, `Rotate`, `FromDesk` and `ToDesk` are executed by the movement library as explained in the movement library section (5.6.2). Each of these motions keeps track of the odometry progress so far and can save it in case of an interrupt. So, if a Stop command is issued, the robot stops and can be later resumed (cycling through `Delivering` ➔ `Stopping` ➔ `Delivering` general states). Similarly, if the robot detects an obstacle on its path, it will stop and wait to resume when it's clear. If it waits for a long time, it will panic and send a notification to the receptionist with its last location. Panic will also occur in case of losing the line (missing the expected green marker within the tolerance).

`Dispense` causes the mover brick to instruct the dispenser brick to call the dispenser library (section 5.5.3). The bricks wait on one another.

`Report` keeps the server aware of *!spam*'s current location. This is so that it may be monitored in the status page of the website. It also enables issuing `Callback`, which causes the robot to ask for a route back to reception from the next junction on its path (switching `Delivering` ➔ `Returning`). It doesn't turn around right away as it doesn't know whether it's on a straight or curved line. Otherwise, the robot will ask for a route back after completing all deliveries (finishing the sequence of instructions).

6. Testing

To ensure the code works as intended and to prove reliability, each element of our system went through rigorous testing, both quantitatively and qualitatively.

6.1 Hardware

Throughout the semester we have created many extensive testing suites, which lead to many improvements and verifications. Testing the precision of odometry in driving and turning revealed up to 5° deviation, which was precise enough for us to implement panic mode in case the robot gets lost. Full testing of the battery consumption, by running the robot from full charge to 0%, enabled to translate voltage to percentage remaining accurately.

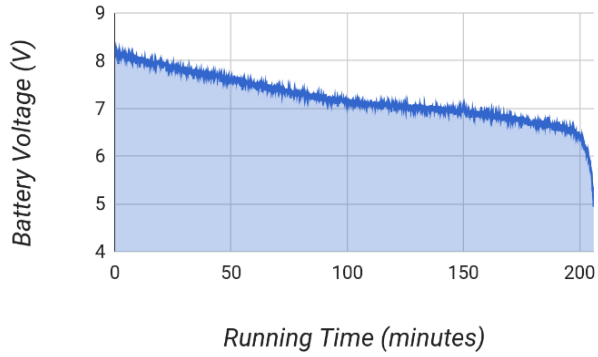

Fig 11. Battery voltage over time graph

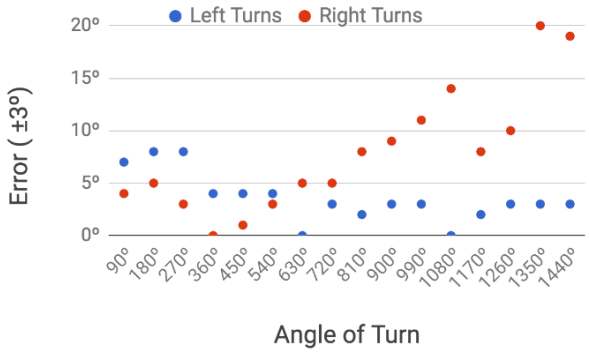

Fig 12. Precision of turning graph

Verifying an average 0.55 seconds MQTT communication latency allowed us to use it in communicating between bricks. We have also documented 96% accuracy of data-matrix decoding in 100 observations with a maximum of 5.5 seconds between decodings.

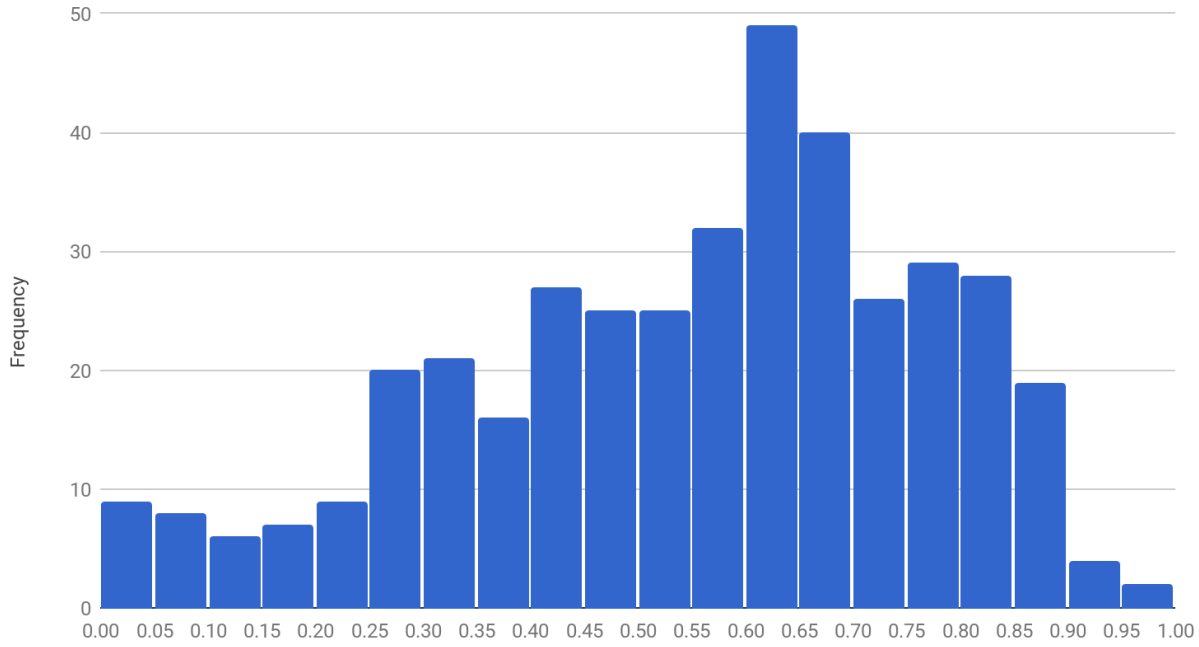

Fig 13. Latency of MQTT communication

Most notably, however, an initial 89% success rate of the dispenser (on 160 observations) encouraged us to improve it for the Third Client demo by over 10%. These improvements included the shaking coroutine, increasing the gaps above slots and a firmer slot wall attachment.

We tested our robot's hardware integration with a focus on navigation accuracy by performing exhaustive testing. Our team ran the robot through 119 paths on the map (every possible combination of inputs), verifying on each occasion that path planning and line following works. The result of 97% proved that our robot can reliably navigate an inbuilt map. Of the 3 hardware errors we encountered, we identified that these were a result of an occasional re-centering fault and could fix this as a result. In its runs the robot also demonstrated the functional integration of the obstacle detection, dispensing movement and the speech and voice feedback for each delivery state in every run.

6.2 Software

6.2.1 Static

Throughout development, we have implemented continuous static code reviews using Codacy for both server and robot code. It pointed out unused variables, unused code, bad style, security flaws and error-prone lines. Notably, it helped us reduce duplicated code from 15% to 5%.

6.2.2 Dynamic

TravisCI provided us continuous integration by running unit tests whenever we merged branches, issued pull requests or pushed new commits in GitHub. Those verified our backend by testing route planning as well as the important event-driven triggers on Flask. Testing was managed by a makefile which TravisCI was configured to run. It rebuilds all the compiled code and runs the repository's test suite. The test suite is dynamically generated by searching the test directory for python files to run. Testing of the robot's control code was purely qualitative and conducted in pair with hardware testing.

6.2.3 Profiling

The code running on the EV3 bricks was profiled to find inefficiencies. Unfortunately, the call graphs we have generated have shown that only 5.5% of running time was spent executing our code. The majority was spent waiting for IO, as ev3dev is an IO-driven OS and all motor/sensor access is done through accessing files. Nonetheless, we verified our good use of resources.

7. Reflections

7.1 Final Client Demo

This was a more challenging demo than the others: being both longer and involving more technical detail and analysis. The robot performed as expected and we were able to demonstrate every feature of the robot. The clients appeared to be impressed with our progress as we had more than met the outlined goals for this demo (see section 2).

7.2 Final Investor Demo

The robot performed very well on the final day. The presentation itself went exceptionally well, with the robot performing exactly as planned. Additionally, we were one of the few groups which kept their robot running during the fair and delivered over 75 letters in our stand. Even so, the hardware stayed reliable throughout. We were very pleased with the announcement we had been the runners-up in the project.

7.3 Strengths, Weaknesses and Alternatives

Design Decision	Strengths	Weaknesses	Alternatives considered
Integrating the dispenser hardware from the get-go into the base.	Being able to build the base with the necessary weight capacity with the ability to test dispensing.	Increased complexity of build and committing to design choices early.	Building the base with line following and dispensing separately. *Difficulties integrating both sections without a complete rebuild required.*

Having the dispenser push the letters up and forward.	Improved compactness, required only 2 motors.	Requires shaking the letters out for reliable operation.	Have the letters pushed out from behind. *It would necessitate having the dispenser assembly behind slots, increasing the footprint of the robot.*
PID line following.	Smooth velocity changes resulting in fluid motion with only one sensor used.	Required hours of calibration of finding P, I and D constants.	Sandwiching the line between two sensors and bouncing from edge to edge. *Trembly movement, hence losing the line.*
Having the dispenser assembly power the classifier.	Limiting the amount of motors.	Careful friction management in upward power transfer.	Have an extra motor power the classifier. *It would have increased weight.*
MQTT brick-to-brick Communication.	Functional communication between the bricks.	Communicating through a server in Ireland.	Rpyc wired connection support. *Would have been faster but it was too unreliable at startup.*
Using a second EV3 for dispensing.	Increase the number of ports, one more battery to power the motors, which means more battery life.	More weight on the robot and the EV3 bricks have to communicate with each other.	The use of an Arduino: *Experimenting with Arduino motor control revealed imprecise odometry.* The use a motor splitter: *50 £ and reduced battery.*
Image Processing occurring on the server.	Improved throughput of image processing.	Hangover requests resulting in faulty classifications (section 5.5.1).	The use of an Arduino: *Limited space after the necessary adoption of an ev3 brick.* The image processing occurring on the 2nd ev3: *Limited computational power on the ev3 brick.*
Flask as our web framework.	Flask provides simplicity, flexibility and it is Python based.	It does not come with lots of functionality built-in. Flask extensions are needed.	Use of the Django web framework *For smaller projects it has too much overhead with unneeded functionality for our project. It is also a monolithic system.*
Selecting letter recipients by employee name on the website.	Readable and easy method for the receptionist to select the letter recipient.	When someone gets relocated to another desk, that information must be updated in settings.	A map of the desks, the desks names, a code for each desk or an ID for each employee. *Each of these was less readable in practice than our design choice.*

7.4 Problem Re-approach and Future improvements

If we were to approach the problem again, one of our primary focuses would be to improve our robot-to-server security. A current example is that our MQTT broker does not have TLS implemented nor username and password verification. This allows for attacks using an unauthorized MQTT client. This is a key area for improvement.

Some of the feedback we received on our Final Demo Day also involved a focus on the mail industry shifting from letters towards packages: with an emphasis that our design should handle these better. Were we to re-approach the problem we would place a greater importance on this. One way we could handle packages is by adding an expanded parcel handler system. This is something that we would have to model and prototype extensively (particularly handling variable parcel sizes) in developing *!spam*. Another aim would also be to allow internal mail delivery between colleagues in the workplace, particularly across multiple floor environments. The goal of this would be to allow employees to easily pass letters and packages between one another through the whole office building. From this basis, we would be to cater for multiple offices by adding unique administrator accounts to the UI on our server. This would allow our company to concurrently handle multiple robots at different locations. Therefore, we would and will adapt our system to the needs of the modern office, with the success of every prototype building on our experiences.

8. Conclusion

Our System Design Project was a marathon over a series of intrepid but fulfilling hurdles. As a team we were brought together over the idea of a mail-bot that we could bring into the future. It was a team-building experience; a learning experience and, thankfully, an enjoyable experience. As this project theme was new, we didn't have any examples of code or implementations we could use to base our own work on. All of the work described above, unless indicated otherwise, is original and was produced by us. All of which was a culmination of all our skills to produce a robot we can be proud of.