Carleton University COMP 4905 Honours Project Report

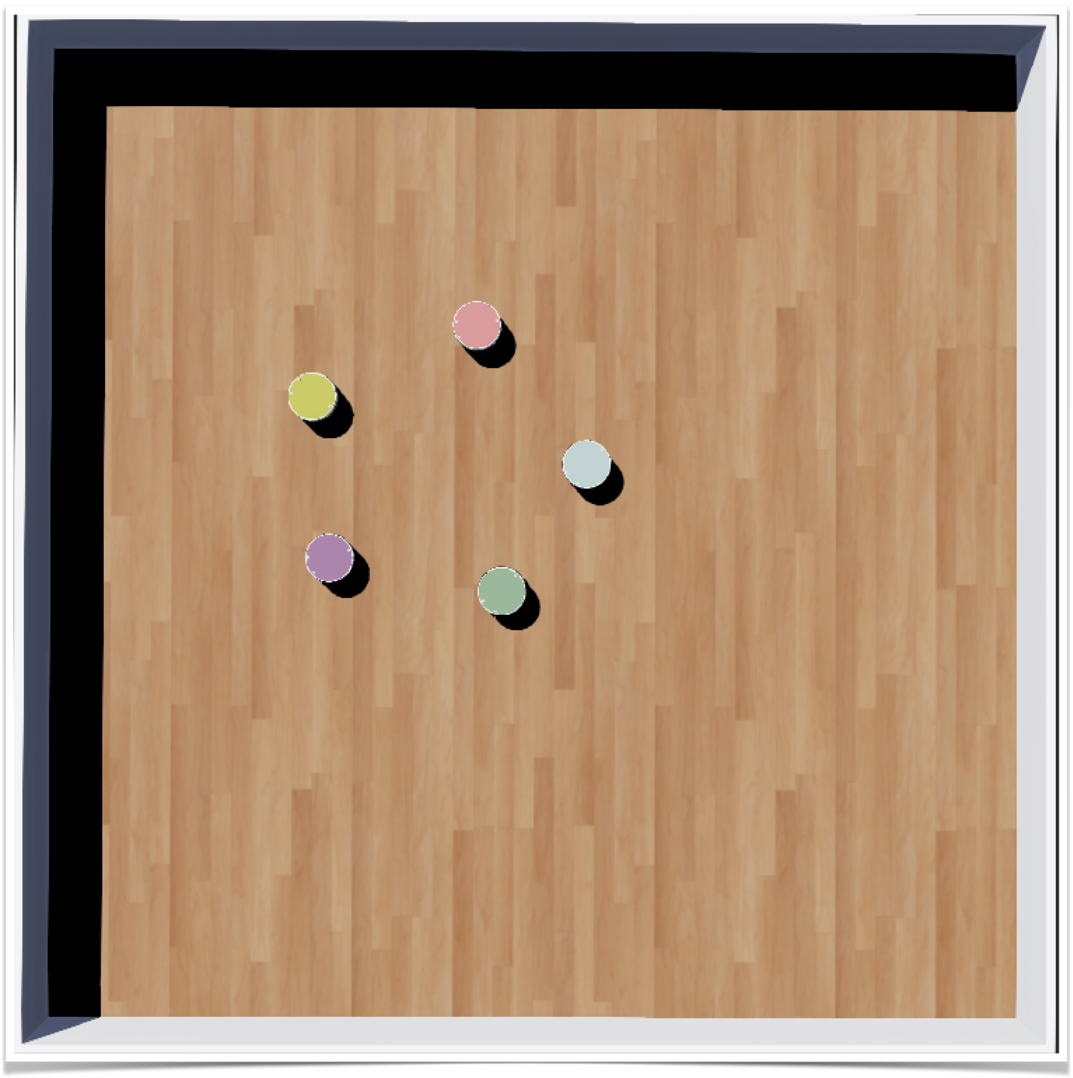
Student:

Yixiang Huang #101071528

Supervisor:

Mark Lanthier

2D Robot Light Show
Controller, Firmware, and Network



Idea

This project was inspired by drone light shows that are recently very popular during festivals, hundreds of commercial drones each carrying a LED light, forming shapes, moving around to present stunning animations at night. These light shows are made with proprietary software and therefore not accessible, therefore I have been wanting to build my own DIY solution (although simpler) to perform a robot light show.

Outcome

The outcome of this project is an entire toolset for 2D robot light shows, including the software for designing animations, the choice of the middleware for communication, and the firmware that runs on robots.

Table of content:

1. Communication

1.1 Message

A message contains two parts, the payload and the topic.

The payload (an array of bytes) is the body of the message. Like texting, payload can be just encode/ decode to String for better readability. For example, a message from the controller to robots might be instructions for robots to perform some action. On the other hand, a message from robot to controller can be a signal of its current location or status.

The topic of a message is set by the publisher when a message is sent, it decides who receive the message.

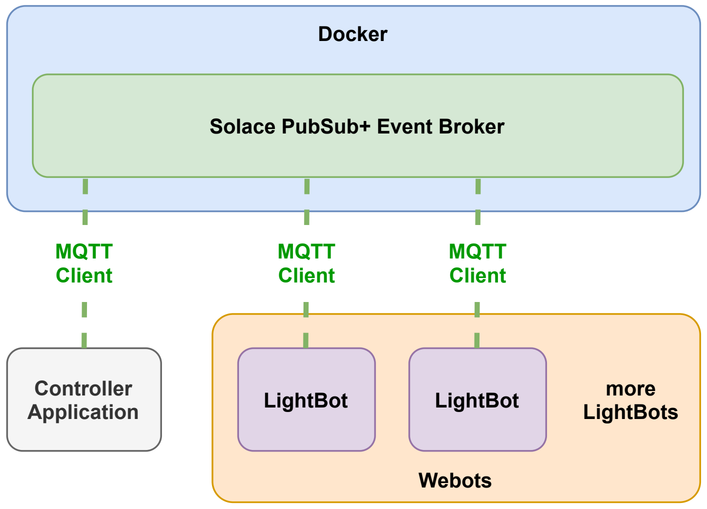
1.2 Publisher and Subscriber

A subscriber register topics to subscribe with the event broker through its MQTT client, when a message with corresponding topic published to the network, a callback function in subscriber's program will be executed automatically to process the message.

A publisher can publish message to the network containing any payload and topic, and don't need to worry about the delivery of the message. The message broker guarantees everyone who subscribed to the topic receives the message. If no one in the network subscribed to the topic at the time of publishing, the message will be destroyed. In this implementation, both the controller and robots are both publisher and subscriber, forming a bi-directional communication channel.

1.3 Message Broker

Similar to a post office, the message broker is a server that distributes messages according to the topic attached to the message. In this configuration, all robots and the controller only connects to the broker via their own Java MQTT client. Implementation-wise, the event broker is a Solace PubSub+ event broker that runs in a Docker container, no configuration is required.



CONNECTIONS BETWEEN CONTROLLER AND ROBOTS

1.4 Topics

Messages are routed by the broker according to topics, a topic is a String that indicates which category the message belongs to. One message belongs to exactly one topic.

Both the controller and robots subscribes to a certain set of topics at startup, the communication begins within these topics.

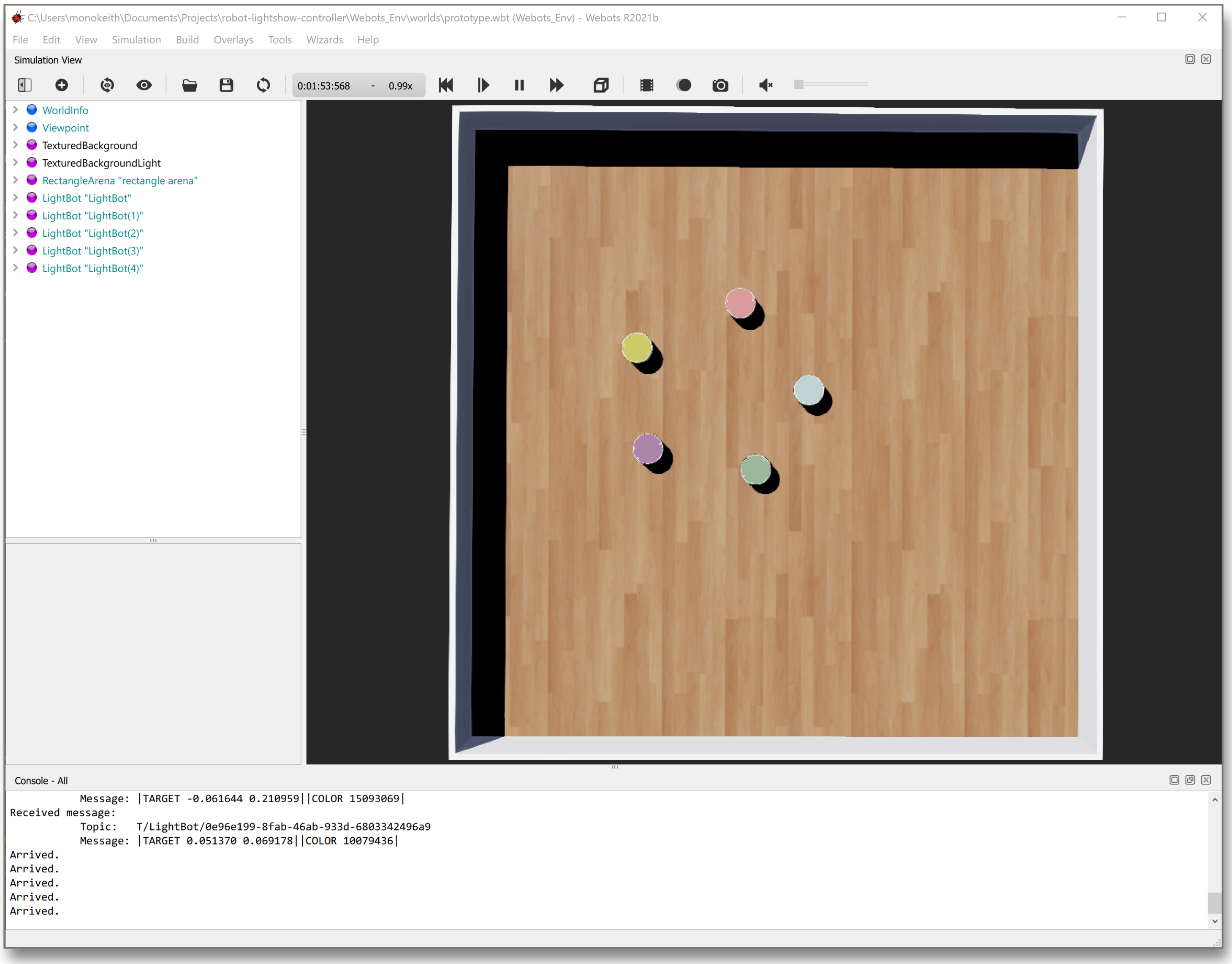
More specifically, the controller initially subscribe to the topic **T/LightBot/UUID** and wait for robots to report their UUID. The UUID of a robot was randomly generated as a part of the startup process, the robot also subscribes to the topic **T/LightBot/<UUID>**, which allows others (in this case it's the controller) in the same network to send a message containing one or many commands that will only receive by this robot. The controller maintains a set of connected robots, once the controller receive a message from this **T/LightBot/UUID** topic, it will trigger a registration of a BotPixel to this specific UUID.

There are two more topics for other purposes, **T/LightBot/Broadcast** and **T/LightBot/Arrival**. The Broadcast topic (subscribed by all robots) was useful if the controller starts after robots, or just need to check if any robot dies, it was used to trigger the robots to report their UUID, so the controller can refresh the connection status. The Arrival topic (subscribed by the controller) was used for robots to report its status when a requested action is complete.

2. Robots and Environment

2.1 Environment

The environment that robots lives in is a simulator called Webots, it emulates an environment with real-world physics and offers support to multiple programming language. It is very convenient and efficient to test robots in this environment, most designs that worked in this simulator should offers similar effect in a real-world environment.



```
            Message: |TARGET -0.061644 0.210959||COLOR 15093069|
Received message:
            Topic:   T/LightBot/0e96e199-8fab-46ab-933d-6803342496a9
            Message: |TARGET 0.051370 0.069178||COLOR 10079436|
Arrived.
Arrived.
Arrived.
Arrived.
Arrived.
```

TEST ENVIRONMENT WITH 5 LIGHTBOTS IN WEBOTS

2.2 Hardware

The robot was modified from a Gctronic Elisa-3 robot [https://cyberbotics.com/doc/guide/elisa3], with added compass and GPS module, and re-packaged into a PROTO file. This robot have a rounded shape, and a big RGB LED on top, which makes it one of the best choices for a light show.

2.3 Limitations with current design

A drawback of the current firmware implementation is that it is heavily relying on the GPS location, which is very accurate in the simulator. The robot relies completely on the accurate location given by the simulator to constantly adjust orientation when moving to destination location. However, in a real-world environment, GPS won't be able to offer this level of accuracy, some position estimation algorithm based on accelerometer or wheel spin is needed to provide this data. Since the main focus of this project is the controller software rather than the accuracy of movements, I did not spend time figuring this out.

2.4 Firmware

The firmware of the robot receives instructions from controller and perform corresponding actions. It was implemented in Java with a Paho MQTT library as the only dependency. The firmware and its dependencies were compiled and output as JAR files, which can be conveniently add to Webots's controller directory.

2.4.1 Threads

The firmware runs in two threads, a message processing thread and an action thread.

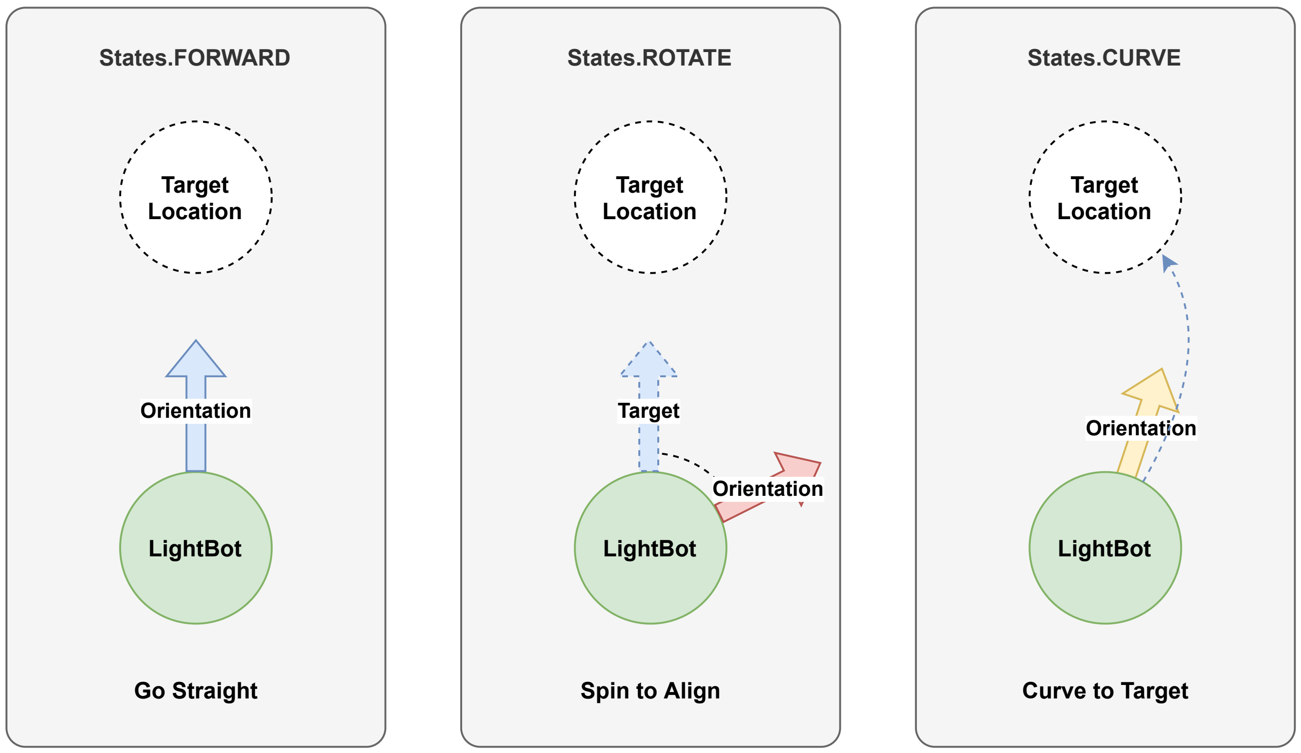
The action thread is the main thread started by Webots, and have a loop associated with time step in the Webots environment, the loop run once per time step. It is used for constantly adjusting wheel spin according to its current orientation and location.

The message processing thread waits for and process messages received from the MQTT client, and update the state of the robot (for example, target location) accordingly.

2.4.2 Target Location

The only way to set or update target location is through messaging. This updates the target location recorded in the Location class.

The Location class is responsible for determining the orientation and distance to the goal location, it keeps track of the delta between the current location and the target location, and computes the shortest path every time step. These information will then be use by the action thread to adjust the movement of the robot. To ensure a smooth movement, when the orientation offset is small, the robot curves towards the goal, instead of stop and spin to re-align. This mechanism can ensure the robot only spins to align at the beginning of a movement, which is not noticeable because of its rounded shape.

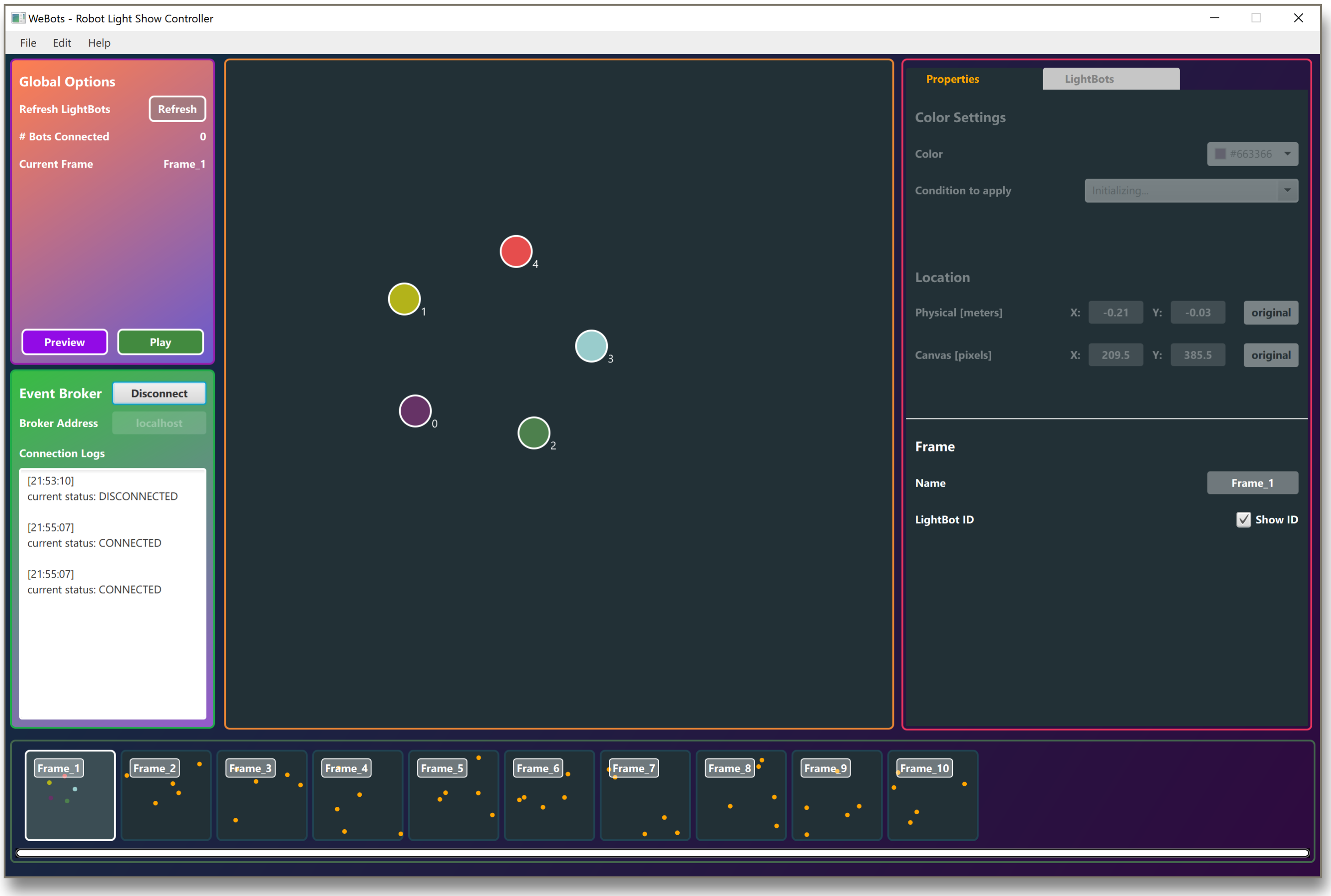


2.4.3 RGB LED

The LED on top of the robot is also directly controlled by messaging, Webots takes a single integer representing the 8-bit per color RGB value. This value can be calculated by the following formula:

$$RGB = (R \cdot 65536) + (G \cdot 256) + B$$

3. Controller application



Refresh LightBots	Refresh
# Bots Connected	0
Current Frame	Frame_1
Broker Address	localhost

Connection Logs

```
[21:53:10]
current status: DISCONNECTED

[21:55:07]
current status: CONNECTED

[21:55:07]
current status: CONNECTED
```

Color	#663366
Condition to apply	Initializing...
Physical [meters]	X: -0.21 Y: -0.03
Canvas [pixels]	X: 209.5 Y: 385.5
Name	Frame_1
LightBot ID	✓ Show ID

SCREENSHOT OF CONTROLLER APPLICATION

3.1 Functionality

The controller application is a software build with JavaFX for purposes of both designing the animations and controlling the robots to play the animations.

Initially there are a fixed amount of circles on the screen, each circle represent a robot which represents a BotPixel, the canvas represent the floor in Webots. User can select one or more pixels by clicking on the circle, and de-select by clicking on empty space. When one or more pixels are selected, they can be drag-and-drop to relocate. When a single pixel is selected, user have an additional choice to type in the desired location's coordinates to be more accurate. Animations are formed by multiple frames, all frames contains the same amount of BotPixels, representing the color and target location of each robot at a given point of time. When playing the animations, the controller will send locations of pixels of the current frame to the corresponding robots. Before continuing to the next frame, it waits until all robots have arrived their destinations.

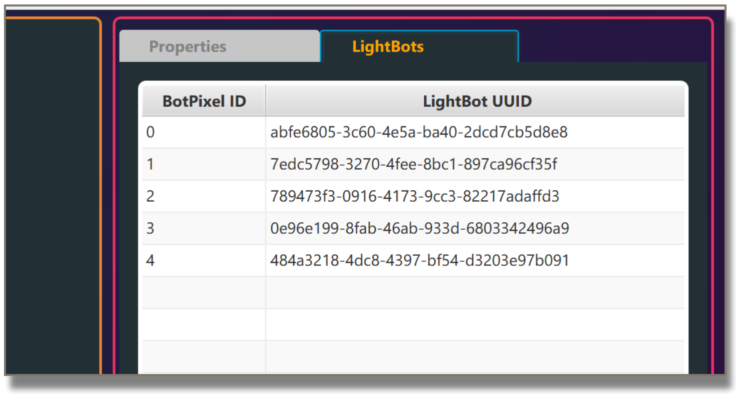
3.2 Design pattern

The graphical user interface of this application was built with JavaFX with help from fxml and SceneBuilder. It follows the MVC design pattern in order to organize code. Object oriented programming is used wherever possible.

3.3 Connection

As indicated in section 1.3, the controller does not directly communicate with any robots, it only connect to the message broker with a single MQTT client. In this way, robots and controller can start with any order, rebooting robots or controller will not result in requiring to reboot the other. Because they all connect to the message broker, the controller also benefits from not having to manage all the connections and distribution of messages.

In the situation of controller starts after all robots have started, which means the controller missed all the UUIDs of robots, it simply sends a request of reporting UUID by sending the request message to the **T/LightBot/Broadcast** topic. In result, all robots receives this request and report their UUID as usual.



BotPixel ID	LightBot UUID
0	abfe6805-3c60-4e5a-ba40-2dcd7cb5d8e8
1	7edc5798-3270-4fee-8bc1-897ca96cf35f
2	789473f3-0916-4173-9cc3-82217adaffd3
3	0e96e199-8fab-46ab-933d-6803342496a9
4	484a3218-4dc8-4397-bf54-d3203e97b091

A map which correlates a BotPixel to a connected LightBot's UUID is automatically maintained. This is useful when distributing target locations to robots. Instead of broadcasting to all robots, this ensures that only the correct robot receives the instructions, in order to save bandwidth and processing power.

3.4 Threads

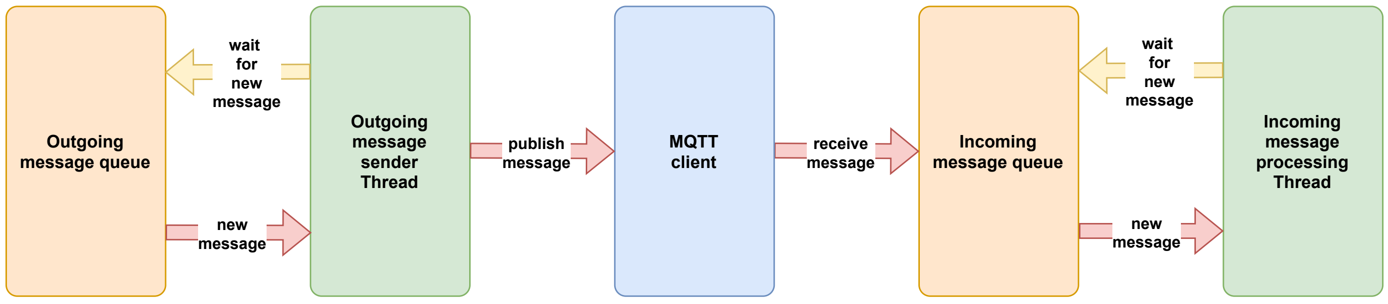
Multiple threads and synchronized functions were implemented in the controller. Two of these threads were responsible for sending and receiving messages.

3.4.1 Incoming message queue

Synchronizing incoming messages was critical. When a message is received by the MQTT client, a callback function executes to put it in a thread-safe message queue, which in this case it's just a LinkedList in a synchronized class. This is because when multiple messages arrives at roughly the same time, there are chances that multiple callback functions run at the same time. If these messages are being processed in parallel, there are chances of causing race condition, because the message processing function sometimes update the state machine. It is dangerous for this to happen, the program may get stuck in some weird state and then causes a freeze. Adding a thread-safe message queue solves this by processing incoming messages synchronously.

3.4.2 Outgoing message queue

On the other hand, the process of sending messages also involves an outgoing message queue, but the purpose of this design was not to prevent race condition, it's only for achieving better performance. Sending a message requires communication to the event broker, network lag may cause congestions. In this case, since the delivery of a message is guaranteed and there's no need to wait for a response, it's faster to offload the process of sending messages to another thread. And this was achieved by setting up a thread-safe outgoing message queue, similar to the incoming message queue.



SIMPLIFIED CHART OF MESSAGE RELATED THREADS

3.4.3 Play animation thread

There is also another thread being generated and terminated during life-cycles of the program, the thread responsible for playing animation. When user click the ‘play’ button, a new thread is generated to run the procedures, alongside with the two threads mentioned above.

3.5 Other design choices (draft)

need to implement before going in this report:

1. duplicate a frame / rearrange a frame
2. save frames as a file / able to open saved files
3. maybe add other features?

4. Conclusion

This is a valuable experience, I've never had any chance like this to implement such an 'huge' project all on my own. It again reminds me the importance of following design patterns as well as planning before coding.

During the project, I've learned building graphical user interface in JavaFx with SceneBuilder and fxml. This tool made it so much easier to design a UI comparing to the old way of coding in the dark with plain code. I learned that the UI in JavaFx could be very similar to HTML and CSS, as they are very similar in terms of syntax and properties.

The project also had given me the chance to work with messaging and event-driven architecture. I've heard the usage and examples a lot during the COOP terms at Solace, but I haven't had the chance to actually use it to build something until this project. Messaging made connection between one controller and multiple robots so much easier. At the end of the term, I recognized the potential of this technology, and really think it could play an important role in the field of robotics. It is especially useful when a large number of data needs to be routed to multiple independent systems, in real time.

I was also able to transfer knowledges from SYSC 3303 in to the project. When the system consists of multiple subsystems, the most elegant way to implement is to let each subsystem have its own thread. This vastly lower the complexity of the system. Since multiple subsystems exists in both the controller and the LightBotFirmware, they both were benefit from adopting such design.

I really appreciate this opportunity for me to improve and apply what I have learned to build something fun.