The University of Edinburgh
School of Informatics

System Design Project

Technical Report

STUDENT NAMES:
Chang Shu
Iordan Vlad
Yin Jason
Stilianos Nicoletti
Thomas Kozlowski
Akshay Chandiramani
Agnibho Chattarji

STUDENT NUMBERS:
s1783039
s1512322
s1540366
s1516821
s1530760
s1558717
s1552184

DATE: April 17th, 2018

Contents

1 Introduction

The requirement of this course (System Design Project) was to design and make an assistive robot. To this end, we reviewed several clinical studies to find and identify a specific problem that we could help with. For example, we came across a study that reported how playing “brain training” games on an app can reduce the risk of dementia [1]. Another study revealed that 1 in 14 people in the UK over the age of 65 will develop dementia and this number is expected to double in 20 years [2]. Together, these and other related studies highlighted the societal importance of tackling dementia, thus our robot Leo for SDP was designed around combating dementia.

Next, we researched the potential benefits of a robotics approach over an application approach to brain training games. The results of this research were quite encouraging. For example, there is research showing that tactile interactions with physical objects have greater cognitive benefits [3] [4]. In addition, there was a study [5] conducted by the University of Edinburgh that suggested how our target demographic(the elderly) would react. In the study, researchers split participants based on age to directly compare the effects of age on multi-tasking when the subjects were assessed using a computerized and a prop-based version of the Craik and Bialystok’s Breakfast task [6]. Notably, while age-related decline in multitasking performance was found using the computerized task, significant age differences were not found on a majority of measures when the prop-based version was administered. These results revealed a reduction in age-related deficits in multi-tasking when more contextualized, non-computer based tasks are used. Moreover, another study found that when it comes to delivering health care instructions, human subjects respond better to a robot compared to a computer [7]. Strikingly, this study reported that participants spoke and smiled more towards the robot than the tablet and were more likely to follow relaxation instructions given by the robot. These findings helped us to formulate the scientific foundation and rationale for our project.

Since our project was aimed at leveraging brain training games, our next step was to decide on what kind of games the robot would implement. We chose two types of games. The first was a memory and pattern repetition game, which was built based on simple memory games available on apps that that are utilized by hospitals to diagnose decline in memory. The second was a reaction time testing game which would quantify the motor reflexes of the user. We felt this was a useful game as an earlier study [8] reported that older people had a faster response time after 35 hours of practice compared to individuals who had no such practice.

2 Overview

2.1 External appearance of the robot

The robot was designed to look like a penguin, as can be seen in Figure 1.

The costume is primarily made from wool and linen with a couple of additional pieces of fabric for the beak and button on the robot. The costume was hand stitched and fitted to the underlying robotic framework and is designed as one large slip on piece that has a velcro seam at the back to hold the costume on. The shaped pieces of fabric in Figure 1 correspond directly to the position of the lego heart, diamond and triangle that can be seen in Figure 2, with the appropriate holes in the fabric to let the LED lights shine through. Figure 1 does not show the final camera placement but on the final model of the robot, a hole is positioned on the costumes neck to let the camera peak through.

Our justification for making the robot look like a penguin is explained below.

Firstly, we based our design on the idea that the robot should be approachable and not intimidating, because it is being used by the elderly. Taking inspiration from another robot (PARO the companion seal robot [9]), we decided that it would be easiest to model the robot on a cuddly animal.

Secondly, we considered what we could physically build for the robot. The problem with most animals is that they have a four-legged structure, which we decided was too difficult in terms of hardware design and would not allow us to play the type of games we were looking to play.

Finally, we wanted two robotic arms for our robot to move, which had attached to them buttons for people to press and this would act as the primary form of user input. Based on this and previous decisions, we ended up with a sketch not too dissimilar to Figure 2. Taking this humanoid shape in mind, we finally settled on the penguin design, as it is a well known animal, many people find it cute, it has two legs and it's movement patterns could be mimicked by the robot.

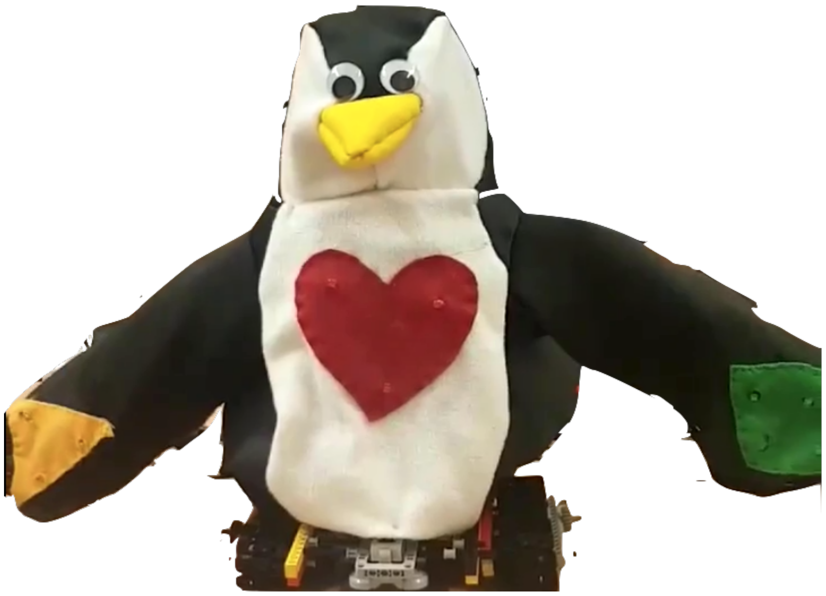


Figure 1: Leo Costume

2.2 Hardware Components

1. 2 X Lego EV3 Bricks
2. Raspberry Pi 3 Model B
3. 2 X Lego EV3 Ultrasonic Sensors
4. Lego EV3 Gyro Sensor
5. 3 X Lego EV3 Touch Sensors
6. 4 X Lego EV3 Large Servo Motors
7. 2 X Lego EV3 Medium Servo Motors
8. Set of LEDs of colour RED
9. Set of LEDs of colour GREEN
10. Set of LEDs of colour YELLOW
11. Amazon Echo Dot
12. Power Bank
13. Logitech HD WEBCAM C270

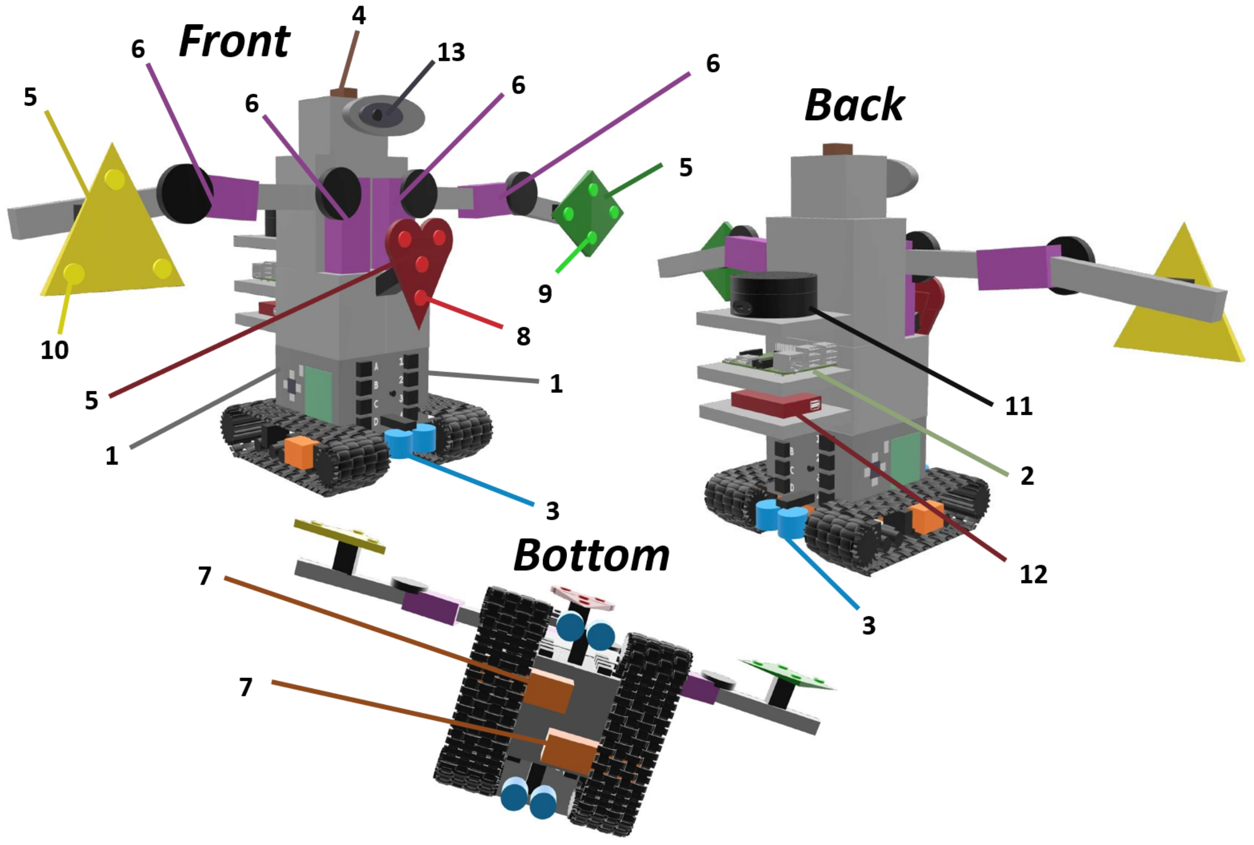


Figure 2: Leo Parts

Figure 2 offers three perspectives of how Leo looks from the front, back and the bottom. It also includes labels that show where Leo's hardware components can be found in relation to one another.

2.3 How the components are used

Figure 2 shows the various components installed on our robot, Leonidas. There are two motors (7) at the bottom which are responsible for rotating Leo's caterpillar tracks, allowing Leo to move in any direction. The two ultrasonic sensors (3) located at the front and the back core, together with the gyro sensor (4) on Leo's head, prevent Leo from falling off his playing environment (table). The two EV3 bricks (1) receive data from sensors and send commands to the motors. The two motors

(6) located behind the Red colored LEDs rotate Leo's left and right shoulders up or down during gaming. The motors (6) located behind the colored Green LEDs and colored Yellow LEDs on Leo's left and right arms rotate Leo's left and right elbows. There are three touch sensors (5) attached to Leo; the green on the left, the red in the center, and the yellow on the right, which the user pushes while playing a game. On the top is the camera (13), which is used to help Leo face the user from an appropriate distance. On the back is the Raspberry Pi (2) which is used for connecting all other components together. The Echo dot (11) is used for Alexa support and the power bank (12) provides a power supply to the Raspberry Pi and the Echo dot.

2.4 Software Architecture

Figure 3 explains the software architecture of Leo. Because we have 6 functional motors on our robot, we have used 2 EV3s, one for moving Leo's hands and playing user games, and the other for optimal positioning when the games are being played. Since both the EV3s can receive instructions from Alexa, the Android app, and data from the web camera, we needed a central control unit for data flow and connecting the various components in our architecture. We decided that a Raspberry Pi would be optimal for our needs, our budget, and also physically small enough to fit on Leo.

The app is the key controller which communicates with the script running on the Raspberry Pi and

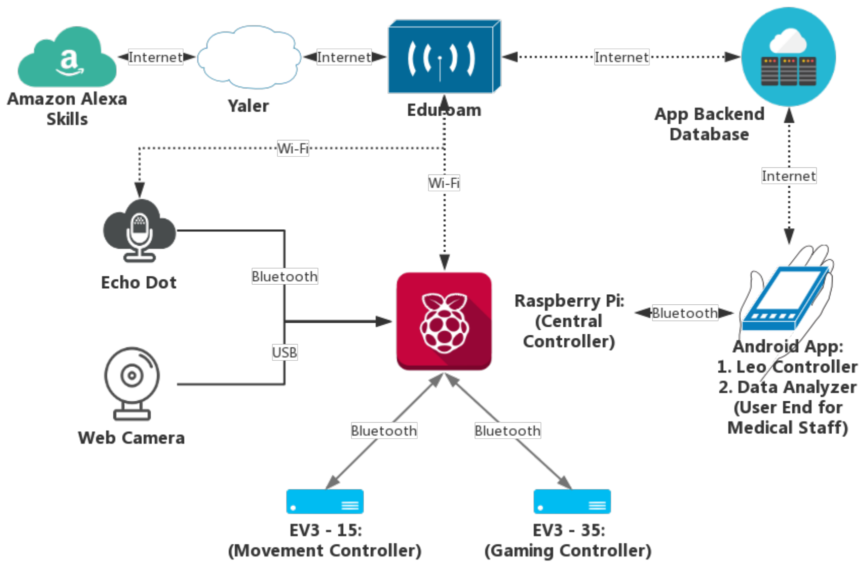


Figure 3: Leo's Architecture

sends instructions to execute specific commands/games. It also communicates with the backend to store data for all the users and retrieve data for analysis. The application's communication, features and libraries are described in detail in the next section.

We wanted to enable an easier and more pleasant gaming experience, and therefore we decided to add the Echo dot as the external speaker to play music during the games and the component that recognizes voice commands. If a voice command is detected, the Echo dot will send the recorded audio to the Alexa Skills Service run by Amazon. After the voice commands are processed they are translated to specific commands that can control Leo. Alexa Skills Service will send these commands to the Raspberry Pi to perform a task or execute a game accordingly.

3 Technical content

3.1 Robot Construction Design and Decisions

This Section 3.1 describes the robots construction design. The robot's structure is made up of 3 distinct areas each with their own design decisions - the body, the drive system and most importantly the arm system.

3.1.1 Arms

We wanted our robot to have two mechanical moving arms with buttons attached on their ends for the user to interact with. With that goal in mind, we came up with Figure 4 as a basic design. (the final version had a few modifications to the buttons but the principal design remained the same.)

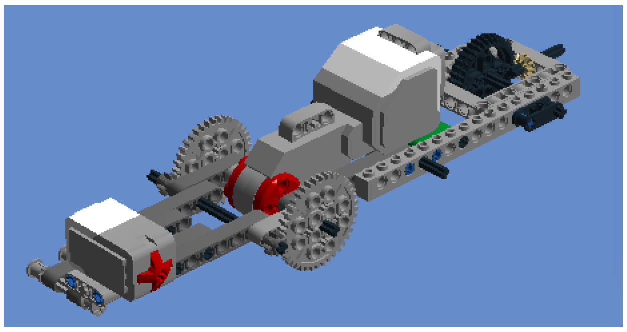
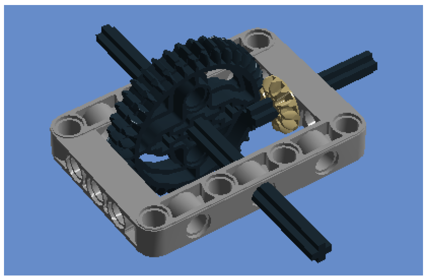


Figure 4: Basic Arm Model

As can be seen in Figure 5a and Figure 5b we designed the arm such that it can only move in a single plane, raising and lowering the arm with some movement at the elbow. We decided upon this design for a couple of reasons. First, we did not want to overcomplicate the design of the robot and have unnatural arm movements, thus a simple up and down motion was chosen. This was also easy to program. Secondly, it was not possible to extend beyond a single plane integrity wise, as we did try a couple of methods of allowing two planes of movement for the arms namely adding another motor onto the shoulder but it was an awkward fit and was structurally unstable.(the robot's lego was bending and the gears weren't moving smoothly)

Moving on to how the elbow (Figure 5b) and (Figure 5a) joints work. The shoulder joint works via a large motor inside the main body of the robot that turns the axle connected to the small bevel gear in Figure 5a. This then raises the entire arm rotating around the large bevel gear. We calculated the gear system from the large motor to have a gear ratio of 1:9 (There are gears linking the shoulder to the large motor (not pictured)).We found through testing that this gear ratio is needed to keep the arm locked otherwise the weight of the arm will turn the large motor and the arm will be insecure. The elbow joint on the other hand works via a simple small gear train with a gear ratio of 3:1; this gear ratio is required. If the motors aren't set to hold position and are wasting power, the elbow joint will move freely due to the weight of the button area.

In our design, we made a button area at the end of each arm that had a lego plate shaped either as a diamond or as a triangle that sat on top of the Lego Mindstorm button. This extended the surface area of the button and secured the LED lights to the corners of the shapes on the outer costume. We altered this design later as we found the penguin fabric would stick to the button, thus we added a small spring to help keep the fabric off. Unfortunately this does not solve the issue entirely, but allows the robot to be functional.


(a) Shoulder joint

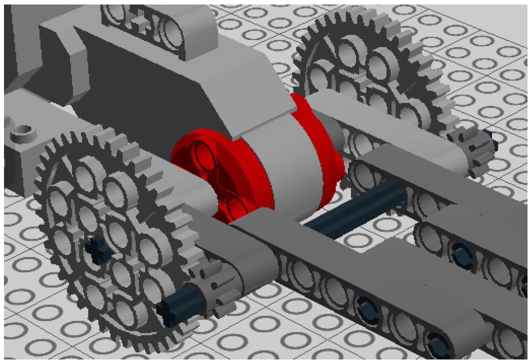

(b) Elbow Gear Train

Figure 5: Operate EV3s

3.1.2 Drive System

This section is on the drive system of the robot.

Figure 2 provides a simple visual on how the drive system works.

The drive system has a simple frame that connects to the 12 connection points on the bottom of the EV3's. Suspended on the frame are the two LEGO EV3 medium motors which drive one set of tracks each. It's a simple system that can be easily controlled.

In terms of a track system vs a wheel system, we tried a wheel system but construction of a stable base to hold the weight of the robot did not turn out well. Controlling it proved difficult. The track system on the other hand still had problems supporting the weight of the robot, however it was fully functional and was good at negotiating gaps on a table.

In terms of turning, the medium motors have an axle to a gear train that directly turns the tracks.

3.2 Sensing and controlling algorithms

This section describes the ways in which the robot interacts with the motors and the sensors. We chose to use the EV3s which were given to us at the beginning of the course to move the robot. We programmed the various sensors and motors through the ***python3 ev3dev*** library. We chose to write the scripts in Python because the documentation provided was sufficient for the tasks we needed, and also because it was a language familiar to all team members, so it would have been easy for any of us to edit the code if required. We decided to group all the movement functions into two key python modules. The first module (present on the EV3 which is concerned with gaming) contained functions needed in the reaction games, such as waiting for the player to hit buttons, or setting the hands in different positions. In order to make the module as general as possible, the functions have parameters such as speed of movement and a list with all the motors to be moved synchronously (so if you want to move only one arm, you just give as arguments the motors linked to that arm).

The second module (present on the EV3 which is concerned with movement) contained functions needed for proper positioning of the robot. This would include functions used for both precise (rotation at set angles, movement backward and forward for a definite time with a modifiable speed) and continuous movements (infinite rotation, forward/backward movement, with a stop function). Whenever any movement function is run, the function for edge detection(also contained in the second module) is run in a different thread. This function checks continuously for the output given by the ultrasonic sensors (which point towards the ground) and is able to stop the motors if the distance to the ground is higher than usual.

Regarding integration, both modules are imported by the scripts which receive information from the Raspberry Pi (Figure 6). Those scripts use the information given by the Raspberry Pi to decide which module functions to call.

3.3 Communication

Since our robot involves a lot of devices including a Raspberry Pi, two EV3 Mindstorms bricks and multiple Android devices, we needed to come up with a way to make all these communicate with each other. At first, we tried connecting the devices using ***python PyBluez*** library. This involved communication using Bluetooth sockets and connection on ***RFCOMM protocol***. It was not very stable, causing our application to crash with messaging delay times over 10 seconds. Then, we tried to use ***Mosquitto MQTT (Message Queue Telemetry Transport)*** [10] as a communication protocol over ***TCP/IP protocol***, which also uses Bluetooth. This protocol involves publishers, responsible for sending messages to the network, subscribers for listening to incoming messages with a particular topic and a broker which is responsible for the coordination. This was a much more solid protocol and faster, achieving delay times of around 1.5 seconds. So, we decided to continue using ***MQTT*** as our robot's communication protocol. We have used the Raspberry Pi as the robot messaging coordinator. This was very important since we were able to make our robot ignore messages sent from the phone application if the current game was still running.

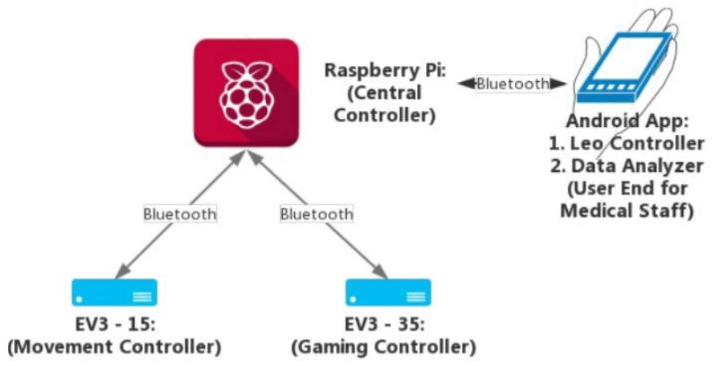
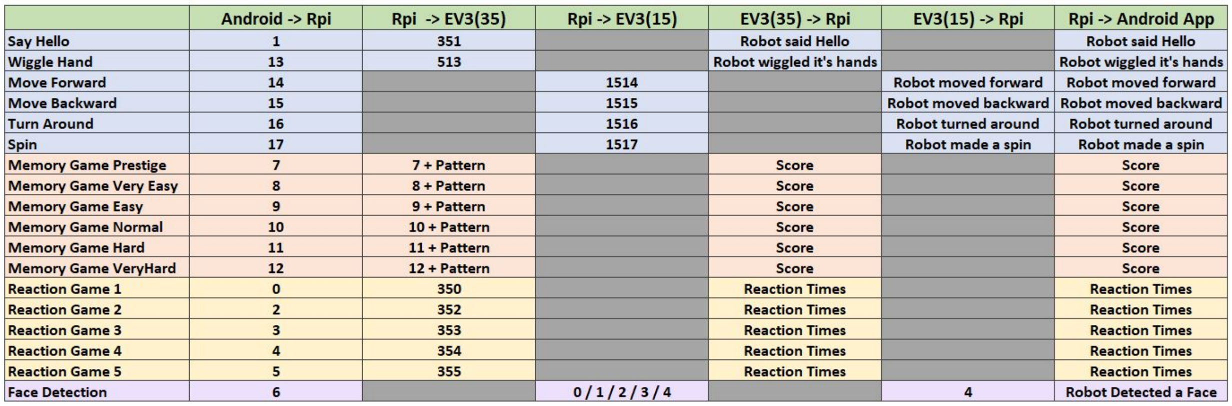


Figure 6: Communication Diagram

As you can see in Figure 6, there is a bidirectional Bluetooth communication between the Raspberry Pi, the two EV3 bricks and the Android application. In order for that to work we needed to install ***Mosquitto*** package and ***python MQTT*** library to the Raspberry Pi and the two EV3 bricks. In addition to that, we needed a Python script for the Raspberry Pi so that it can receive messages from the Android application, sending messages to the EV3 bricks accordingly and vice versa. Moreover, we needed to edit part of the Android application, so that our application would be able to send game messages to the Raspberry Pi and of course receive and display messages from Raspberry Pi regarding scores and reaction times for the current game played. Apart from that, we needed two Python scripts, one for each EV3 brick, so that they can receive messages from the Raspberry Pi on which game to play on robot and send back feedback messages to Raspberry Pi after the game ends, and from there to the Android application.



	Android -> Rpi	Rpi -> EV3(35)	Rpi -> EV3(15)	EV3(35) -> Rpi	EV3(15) -> Rpi	Rpi -> Android App
Say Hello	1	351		Robot said Hello		Robot said Hello
Wiggle Hand	13	513		Robot wiggled it's hands		Robot wiggled it's hands
Move Forward	14		1514		Robot moved forward	Robot moved forward
Move Backward	15		1515		Robot moved backward	Robot moved backward
Turn Around	16		1516		Robot turned around	Robot turned around
Spin	17		1517		Robot made a spin	Robot made a spin
Memory Game Prestige	7	7 + Pattern		Score		Score
Memory Game Very Easy	8	8 + Pattern		Score		Score
Memory Game Easy	9	9 + Pattern		Score		Score
Memory Game Normal	10	10 + Pattern		Score		Score
Memory Game Hard	11	11 + Pattern		Score		Score
Memory Game VeryHard	12	12 + Pattern		Score		Score
Reaction Game 1	0	350		Reaction Times		Reaction Times
Reaction Game 2	2	352		Reaction Times		Reaction Times
Reaction Game 3	3	353		Reaction Times		Reaction Times
Reaction Game 4	4	354		Reaction Times		Reaction Times
Reaction Game 5	5	355		Reaction Times		Reaction Times
Face Detection	6		0 / 1 / 2 / 3 / 4		4	Robot Detected a Face

Figure 7: Communication Messages

In Figure 7 you can see all types of messages sent by each device according to what the robot needs to do. The overall flow of the communication is initialized from the Android application, from there to the Raspberry Pi, and then to the EV3 bricks and back. For the basic movements such as"Say Hello","Spin" etc, the messages are clear to understand. For the memory games, a random pattern related to difficulty was generated every time by the Raspberry Pi and added to the message to be sent to the EV3(35) brick. When the memory game ends, the total score achieved is sent back to the Android application. For the reaction games, a different message was sent to the Raspberry Pi and from there to the EV3(35) depending on the game. Then after the game finishes, the reaction times of the user are sent back to the the Android application. It easy to see in Figure 7, cells filled in gray showing that not all devices were receiving and sending messages every time. The reason for that is different EV3 bricks are responsible for different tasks. For example, if we wanted to play a memory game, then we won't send any sort of message to EV3(15) which is responsible for the wheels, instead we will only communicate with the EV3(35). This was easily done by using different topics supported by ***Mosquitto*** for each device, making the communication more organized and straight forward.

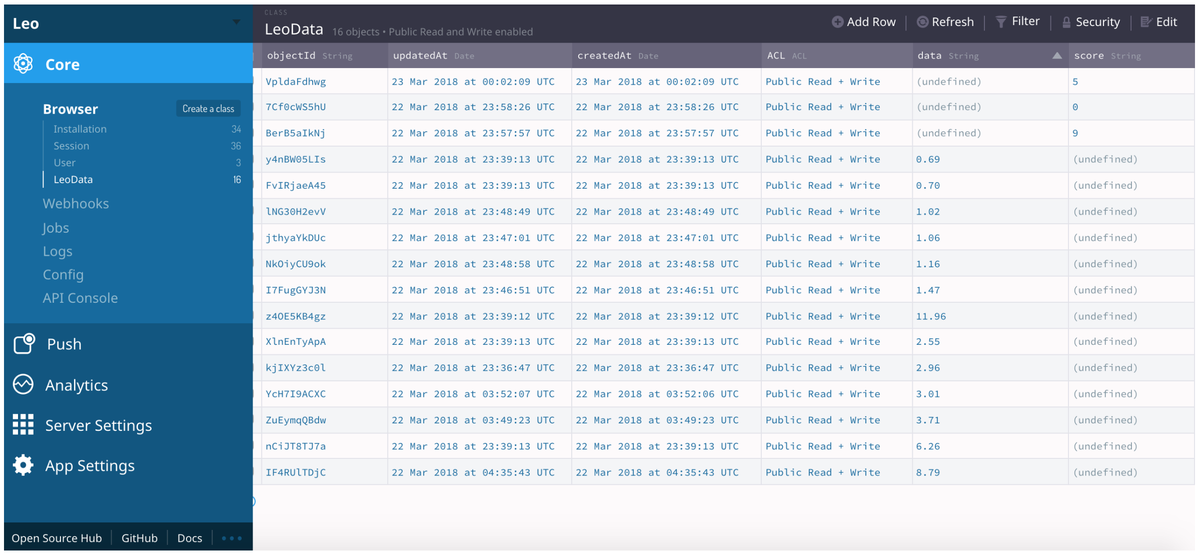
In Appendix A, you can see a simple example of our communication code when you are commanding the robot to "Say Hello". The Android application publishes message "1" to "rpi" topic, which is the message that identifies this move on the Raspberry Pi. The Raspberry Pi receives the message, since it subscribes on "rpi" topic. Then the Raspberry Pi sends message "351" to topic "ev3", which is the message that identifies the move on the EV3(35). The EV3(35) receives the message since it subscribes on topic "ev3" and makes robot to move and say "Hello". When robot finishes its job the EV3(35) publishes message "Robot said Hello Message" to topic "rpi". Then the Raspberry Pi receives the messages and publishes the same message to "android" topic. The Android application receives the message and displays it on the screen.

3.4 Android application and backend

The android application has the following implementation features

- The app uses the ***Paho Android service (Android MQTT)*** to communicate with the Raspberry Pi (see Section 3.3 for explanation of **MQTT**). It subscribes to the broker (which in this case is the Raspberry Pi) and sends a byte as a message ranging from 0-17 depending on the command the robot has to execute (Figure 7).

- The app connects to the backend through WiFi. We decided to build a backend using ***Parse Back4App*** [11] because all of our data is in the form of numbers (scores and reaction times for the users). **Back4app** allows us to provide a user friendly representation of those numbers and enables us to export all the data stored in the form of a JSON file. The **scores** column provides all the user scores from the different memory games and the **data** column provides all the user reaction times from the various reaction games (Figure 8), which are added to the backend using the ***ParseObject*** and ***ParseInstallation*** Java classes methods.



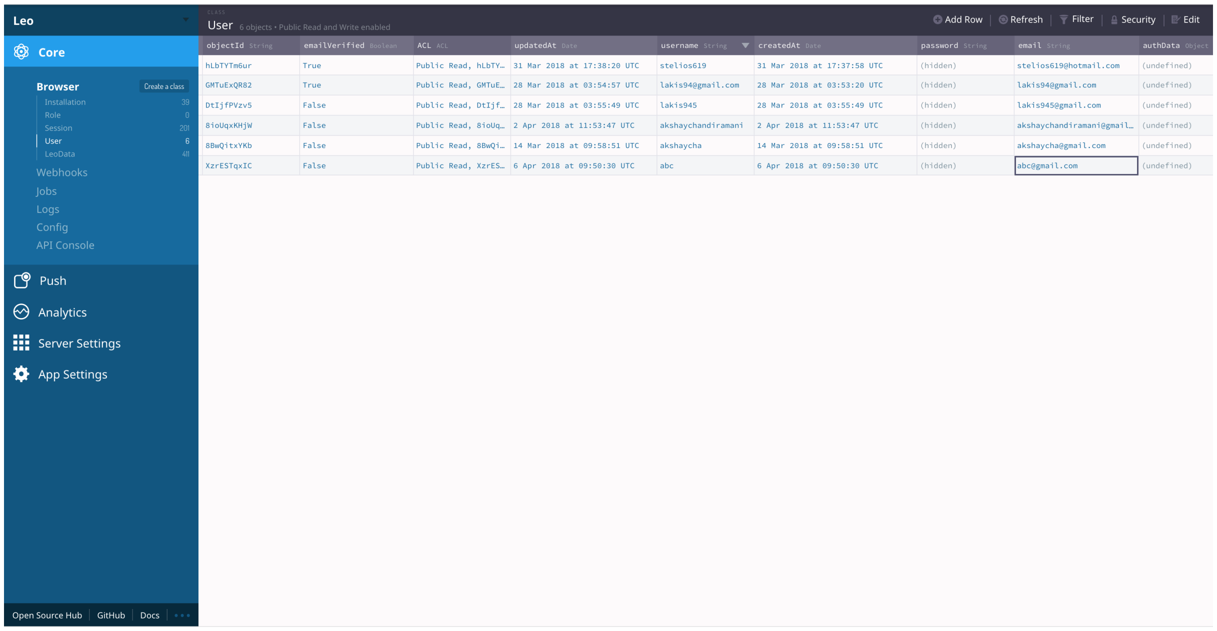
objectId String	updatedAt Date	createdAt Date	ACL ACL	data String	score String
VpldaFdhwg	23 Mar 2018 at 00:02:09 UTC	23 Mar 2018 at 00:02:09 UTC	Public Read + Write	(undefined)	5
7Cf0cWS5hU	22 Mar 2018 at 23:58:26 UTC	22 Mar 2018 at 23:58:26 UTC	Public Read + Write	(undefined)	0
BerB5aIkNj	22 Mar 2018 at 23:57:57 UTC	22 Mar 2018 at 23:57:57 UTC	Public Read + Write	(undefined)	9
y4nBW05LIs	22 Mar 2018 at 23:39:13 UTC	22 Mar 2018 at 23:39:13 UTC	Public Read + Write	0.69	(undefined)
FvIRjaeA45	22 Mar 2018 at 23:39:13 UTC	22 Mar 2018 at 23:39:13 UTC	Public Read + Write	0.70	(undefined)
lNG30H2evV	22 Mar 2018 at 23:48:49 UTC	22 Mar 2018 at 23:48:49 UTC	Public Read + Write	1.02	(undefined)
jthyaYkDUc	22 Mar 2018 at 23:47:01 UTC	22 Mar 2018 at 23:47:01 UTC	Public Read + Write	1.06	(undefined)
NkOiyCU9ok	22 Mar 2018 at 23:48:58 UTC	22 Mar 2018 at 23:48:58 UTC	Public Read + Write	1.16	(undefined)
I7FugGYJ3N	22 Mar 2018 at 23:46:51 UTC	22 Mar 2018 at 23:46:51 UTC	Public Read + Write	1.47	(undefined)
z4OE5KB4gz	22 Mar 2018 at 23:39:12 UTC	22 Mar 2018 at 23:39:12 UTC	Public Read + Write	11.96	(undefined)
XlnEnTyApA	22 Mar 2018 at 23:39:13 UTC	22 Mar 2018 at 23:39:13 UTC	Public Read + Write	2.55	(undefined)
kjIXYz3c0l	22 Mar 2018 at 23:36:47 UTC	22 Mar 2018 at 23:36:47 UTC	Public Read + Write	2.96	(undefined)
YcH7I9ACXC	22 Mar 2018 at 03:52:07 UTC	22 Mar 2018 at 03:52:06 UTC	Public Read + Write	3.01	(undefined)
ZuEymqQBdw	22 Mar 2018 at 03:49:23 UTC	22 Mar 2018 at 03:49:23 UTC	Public Read + Write	3.71	(undefined)
nCiJT8TJ7a	22 Mar 2018 at 23:39:13 UTC	22 Mar 2018 at 23:39:13 UTC	Public Read + Write	6.26	(undefined)
IF4RUlTDjC	22 Mar 2018 at 04:35:43 UTC	22 Mar 2018 at 04:35:43 UTC	Public Read + Write	8.79	(undefined)

Figure 8: View of all reaction times and scores stored in the backend

- The app has user login or registration as the startup screen. Details for all the users and their sessions are stored in the **User** class of the backend using methods from the ***ParseUser*** Java class. This also checks if the email that the user has registered is a valid email or not (Figure 9).

- Data visualisation is in the form of numbers for the scores and graphs for the different reaction times. The message sent back from the Raspberry Pi after the game has been completed is filtered (using different classes in Java) depending on the content of the message, and is accordingly saved to the backend. Another screen pops up, with either the score, single reaction time, graph or response after a basic command (like 'wave hand') has been executed. We used the ***Android Graph View plotting library*** [12] for our graphs in the app.

- Data analysis in the app is basically a recommendation system based on the comparison of the users mean score and mean reaction time to the general mean score and mean reaction time for all the users and data. The app scrapes all the data saved in the backend using methods



User 6 objects • Public Read and Write enabled

objectId String	emailVerified Boolean	ACL ACL	updatedAt Date	username String	createdAt Date	password String	email String	authData Object
hLbTYTm6ur	True	Public Read, hLbTY…	31 Mar 2018 at 17:38:20 UTC	stelios619	31 Mar 2018 at 17:37:58 UTC	(hidden)	stelios619@hotmail.com	(undefined)
GMTuExQR82	True	Public Read, GMTuE…	28 Mar 2018 at 03:54:57 UTC	lakis94@gmail.com	28 Mar 2018 at 03:53:20 UTC	(hidden)	lakis94@gmail.com	(undefined)
DtIjfPVzv5	False	Public Read, DtIjf…	28 Mar 2018 at 03:55:49 UTC	lakis945	28 Mar 2018 at 03:55:49 UTC	(hidden)	lakis945@gmail.com	(undefined)
8ioUqxKHjW	False	Public Read, 8ioUq…	2 Apr 2018 at 11:53:47 UTC	akshaychandiramani	2 Apr 2018 at 11:53:47 UTC	(hidden)	akshaychandiramani@gmail…	(undefined)
8BwQitxYKb	False	Public Read, 8BwQi…	14 Mar 2018 at 09:58:51 UTC	akshaycha	14 Mar 2018 at 09:58:51 UTC	(hidden)	akshaycha@gmail.com	(undefined)
XzrESTqxIC	False	Public Read, XzrES…	6 Apr 2018 at 09:50:30 UTC	abc	6 Apr 2018 at 09:50:30 UTC	(hidden)	abc@gmail.com	(undefined)

Figure 9: View of all user details and email verification statuses stored in the backend

from the ***ParseQuery*** Java class and suggests an easier or harder memory/reaction game based on the users average performance in these games.

- We format the user data from the users current session using a **StringBuilder()** and allow them to send it to their email (using whatever provider they prefer) using an **email intent** by pressing the red button in the corner of any screen. The settings screen also allows the user to view their registered username and email and their aggregate score based on their scores and reaction times from the current session. It also allows the user to change the email for which they would like their game data to be sent to.
- The application UI uses the ***Android ViewPager, PagerAdapter, DrawerLayout and NavigationView*** classes to create a sliding menu for selecting the screen the user wants to see and interact with. The application can be downloaded and installed from the link provided in the references. [13]
- We did consider creating a separate server for the analysis of data, but decided against it because this would have unnecessarily added another software component for the user to setup, and since the app could access all the data in the backend, we decided to use the app itself for data analysis. The app was quite simple to integrate into our system, because the only things we had to ensure were that it was sending and receiving correct information to and from the Raspberry Pi. The two way communication between the app and the backend was also an important feature to integrate into our system but it did not involve any other components.

3.5 Voice command

Since a certain percentage of the elderly suffer from either presbyopia or other problems with their eyesight and would not be comfortable with using the mobile app, adding voice commands is a reasonable alternative for elderly people to interact with Leo. According to a previous study, using voice commands has a great potential to ease everyday life for the elderly in terms of security and usability [14]. In order to develop a robust and efficient voice command service under the restriction of time and computational resources, we chose Alexa Skills Kit [15] as the solution. We picked the Echo dot as the user-end device for transcribing voice commands and sending them back to Alexa Skills service for processing. The script on the Raspberry Pi can communicate with the Alexa Skills service through Flask [16], a micro-framework for python, and Yaler [17], a relay infrastructure for secure access to embedded systems. (Figure 10)

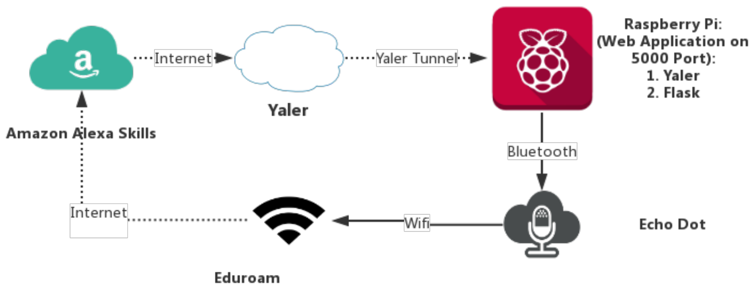


Figure 10: Architecture of Voice Command System

3.5.1 Alexa Skills Kit

Alexa is voice service built by Amazon and the brain behind tens of millions of devices which are the Echo family of devices. By using the Alexa Skills Kit (ASK) which is a collection of self-service APIs, tools, documentation, and code samples, we can independently develop custom skills for controlling Leo. We aimed to use Alexa to enable Leo to take some handy voice commands for setting itself up, starting a game and detecting the user. Generally, the model of Alexa Skill can be divided into 4 essential components:

- Invocation: Invocation is a calling name to begin an interaction with a particular custom skill. For our Alexa Skill, the invocation is *'Leo'*.

- Intents: Intents are sets of utterances that have similar intentions. For instance, the intent for triggering a memory game includes two utterances (the {*mode*} will be introduced later):

  1. *Leo to play the memory game* {*mode*}
  2. *Leo to start the memory game* {*mode*}

  Both of these two utterances have the same intentions. Therefore, either utterance from the user will get classified into the MemoryGame intent. Properly choosing utterances for each intent will greatly improve the accuracy of classification.

- Slots: Slots can be regarded as the variables in utterances. For the intent example mentioned above, {mode} is the slot of those utterances of the MemoryGame intent. For this intent, {mode} has different possible values like easy, medium and hard. It will be assigned one of these preset values when Alexa Skills processes the recorded utterance as an input. Values for these slots will get stored in a JSON file and get sent to the Raspberry Pi. Hence, the Raspberry Pi can start the memory game in the requested game mode according to the JSON object in the file received.

- Endpoint: By setting the HTTPS address of the Raspberry Pi as the endpoint, the Raspberry Pi will receive POST requests when a user interacts with our built Alexa Skill. The request body contains parameters that the Raspberry Pi can use to execute corresponding instructions and generate a JSON-formatted response.

3.5.2 Yaler

Yaler is a reliable solution for access to embedded systems located behind a firewall, NAT or mobile network router which enables the Raspberry Pi to be located through the internet even if it is masked behind Eduroam. Compared to Ngrok which is a similar solution, Yaler will assign a permanent DNS domain to the Raspberry Pi instead of assigning a different DNS domain after each startup, which effectively avoids the inconvenience of changing the endpoint configuration for Alexa Skills. The following 3 steps are the most crucial parts for setting up Yaler:

- Change the configuration in the yalertunnel.service script (the 's' prefix is crucial since HTTPS access is enabled only by adding this prefix):

```
WorkingDirectory=/home/pi/yalertunnel
ExecStart=/home/pi/yalertunnel/yalertunnel server s:127.0.0.1:80
try.yaler.io sparta -min-listeners 1
```

- Enable and start yalertunnel.service as a daemon process at startup:

```
  $ sudo systemctl daemon-reload
  $ sudo systemctl enable yalertunnel.service
```

3.5.3 Flask

Flask is a relatively simple and handy web framework that runs on the Raspberry Pi. Compared to other more popular web frameworks like Angular or React, Flask aims to keep the core simple but extensible. The role of the Raspberry Pi in the voice command system is basically acting as a server responding to POST requests (which have the JSON objects) sent from Alexa Skills, and sending proper control signals to each EV3 in order to perform the task as requested by the user. Flask is the most convenient framework to establish a tiny web application to meet these demands. The steps for building web application with Flask can be found in Appendix B.

3.6 Face Detection

The reason face detection is a part of the project is because we wanted the robot to always face the user to make it easy for the user to interact with the robot. With a simple voice command through the Echo Dot as "Alexa, ask Leo to find me" or a click of a button in the app, the robot will look for the closest person and start adjusting the distance between itself and the user, and move to a distance that is close enough for the user to press the buttons on the robot. We have used a webcam to capture video. The data is sent to the Raspberry Pi via a USB cable.

Face detection is built based on the XML training file created by Rainer Lienhart [18] and the Haar cascade classifier from OpenCV library. OpenCV supports two types of classifiers, the Haar cascade classifier and the Local Binary Pattern classifier (LBP for short). The reason that we prefer the Haar cascade classifier over LBP is because of its low false positive rate. Although LBP has a shorter training time, it is also less accurate than the Haar Cascade when detecting faces due to its simplicity.

We converted the video stream to grey scale drawing rectangles around the detected faces. From this, we found out that the classification was still not very accurate. To deal with the numerous false positives, we adjusted the parameters such as scale factors. We also made sure the face detection only detects the closest face; in this way, only the closest user is being detected. This also eliminated any faces in the background.

At an earlier stage, we planned to connect the webcam to the EV3 directly. This would have allowed the EV3 to detect faces and command the motors at the same time, which would have made the movement instant. However because of the slow computation power on the EV3, it was not possible to run the script on the EV3 at all. We then decided to run the script on the Raspberry Pi instead and allow the Raspberry Pi to communicate with the EV3 by using MQTT protocol, as shown in Figure 11.

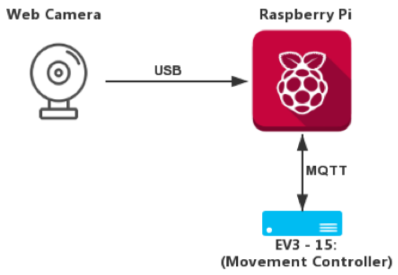


Figure 11: Face Detection System

There are currently two ways to start face detection: the first way is through the app by pressing the “face detection” button, and the second way is by commanding the Echo Dot: “Alexa, ask Leo to find me”. Both the options will run the script on the Raspberry Pi and activate the webcam, which is indicated by a green light on it. The webcam will examine the closest face by checking the size of the user’s face; if the face appears too small/big, the Raspberry Pi will send a command to the EV3 telling it to move forward/backward until the robot is in a reachable position for the user. There are also two borders located by the algorithm; if the user moves out of the left border, the Raspberry Pi will tell the EV3 to rotate left until the user’s face is at the centre and vice versa for the right border. The decision process for the movement is shown in the diagram below (Figure 12).

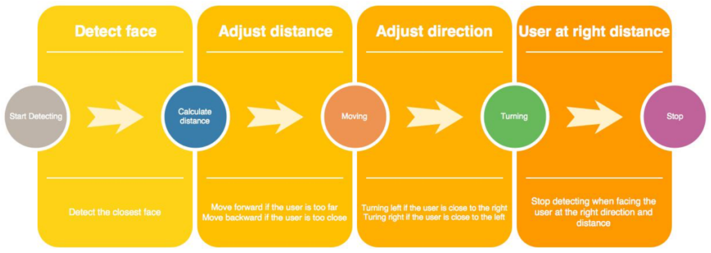

Detect face
Adjust distance
Adjust direction
User at right distance
Start Detecting
Calculate distance
Moving
Turning
Stop
Detect the closest face
Move forward if the user is too far
Move backward if the user is too close
Turning left if the user is close to the right
Turing right if the user is close to the left
Stop detecting when facing the user at the right direction and distance

Figure 12: Decision made by the Raspberry Pi

4 Testing and Evaluation

4.1 Testing

4.1.1 Unit Testing

- **Edge Detection**
  For unit testing of edge detection, we made the robot move forward at different angles with respect to the edge. A test was successful if the robot did not fall off the table. We repeated every test 5 times. At the first iteration of tests (see Figure 13), we discovered that the robot had a weak spot while approaching the edge going backwards. Specifically, when the angle lied in the area which is colored red, 11 out of 20 total tests passed. This was caused by the robots unequal weight distribution (more on the back). We proceeded with adjusting the placements of the ultrasonic sensors on the robot, we redid the tests, and we got a 100% success rate.

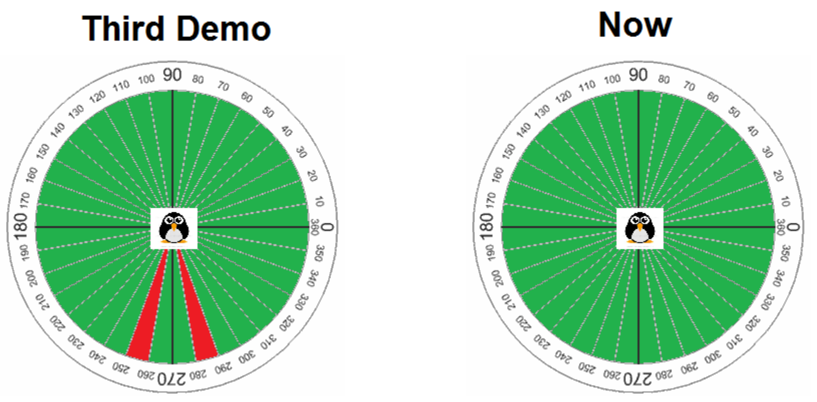


Figure 13: High level results for edge detection unit testing

- **Face detection**
  For unit testing of face detection, we had two main tests:

  1. Place the robot on the table. Start face detection. Ask the user to come close to the robot. The robot is expected to move backwards, then stop detecting, and signal that the face has been found.
  2. Place the robot far from the user. Start face detection. The robot is expected to move forwards, then stop, and signal that the face has been found.

  The two tests were repeated on different members of the team, and the following conclusions were drawn:

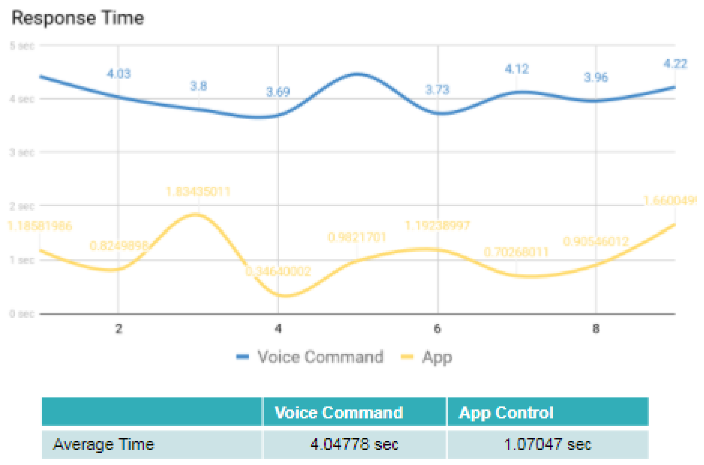
  1. Face detection works better if the user does not wear glasses.
  2. Face detection works better if the user does not have facial hair.
  3. Face detection works better if the user is in a room with good lighting.
  4. Face detection works better if the background is white.

4.1.2 System Testing

- **Playing with Leo using the app**
  We tested the system by playing games with Leo. The player starts a game from the app. The user plays a game with the robot. A test is considered successful if the results of the game show up on the app and on the backend. Another datapoint extracted from each test is the time interval between the command being given and the robot starting the game.

- **Playing with Leo using Alexa**
  We tested the system by playing games with Leo again. The player starts a game by voice command. The user plays a game with the robot. A test is considered successful if the results of the game show up on the backend. Another datapoint extracted from each test is the time interval between the command being given and the robot starting the game.

The results of the tests showed (Figure 14) that all of the games can be played successfully and data collection is successful as well. The average response time for voice commands was roughly 4 seconds, while the average response time for app commands was roughly 1 second. We decided to continue to use Alexa, since voice commands represent a feature which added to the robots user-friendliness.



	Voice Command	App Control
Average Time	4.04778 sec	1.07047 sec

Figure 14: Android App vs Alexa

4.2 Strengths and weaknesses of the built system

4.2.1 Strengths

- User Approachable:
  One of our robot's advantage over any other robot is its appearance. A penguin like outfit makes our robot user friendly, and increases the user's level of comfort without making them scared when they play with it.

- Multiple Interfaces:
  As mentioned above, we have implemented two ways in which the user can control the robot and run games on it. This includes running games by pressing the game button on our android application or executing a game with a voice command. The use of multiple interfaces makes controlling our robot more interesting and also make it further accessible by more users, including users with some sort of disability.

- Portability:
  Although our robot has a lot of hardware (including two EV3 bricks, a Raspberry Pi, an echo dot, a power bank and a good number of sensors and motors), it is still very compact and can easily be carried around. Everything was closely and neatly packed together into one parcel able piece. In addition to that, we did not need any sort of external devices into order for our robot to operate, as all complex processing is done by the Raspberry Pi.

4.2.2 Weaknesses

- Internet connection required:
  Since the voice command system is based on Alexa Skill (a service powered by Amazon) and the app backend is based on Back4App, disconnecting any components in either the app subsystem or voice command subsystem will cause a failure. Both solutions for controlling Leo can only work in an Internet environment, which strongly limits its usage scenarios.

- Response Time of Voice Command:
  The voice subsystem use the HTTPS protocol via the Internet. Compared to the local communication of the app based MQTT Bluetooth protocol, the response time for voice commands is noticeably longer.

- Lack of Flexibility and security:
  The voice command system can hardly be fully custom since the main speech recognition model is encapsulated by Amazon. Although we can possibly develop some suitable intent model for our Leo Control skills, we cannot further adjust or tune to make it more usable for Leo. Furthermore, the security configuration and implementation of Alexa Skills is not that transparent as well, which may cause some potential security risk for our voice command system. Similarly, our backend system for the app also is based on a third-party service, which makes it difficult to ensure we can fully control the security risk.

5 Reflection

5.1 Abandoned components and approaches

1. **PyBluez:** We did not use the PyBluez communication library since it was not stable and was very slow compared to our MQTT approach. Our testing showed that the PyBluez approach made our Android application crash, and was sometimes not sending messages at all. Figure 15 also explains how the reaction times of MQTT for communication were much faster than with PyBluez.

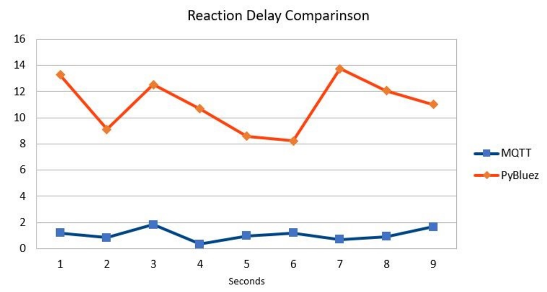


Figure 15: MQTT vs PyBluez

2. **Emotion Detection:** The reason we did not implement emotion detection is because we feel the feature does not add much value to our project. Since we had limited time and the processing power of the Raspberry Pi was low, we decided it would be better to allocate our resources to improve existing features.

3. **Music Synchronization:** We did not implement music and movement synchronization since it was very complex to implement. We tried using some python libraries that could make our robot play music but we did not have enough resources to make our robot move in sync with the music played. Apart from that, capturing the beat of music played in real time required large processing power, which something we did not have with the components provided. Acquiring additional components would have proved too expensive.

5.2 Reflection on how the system performed

The system had been tested in the last two days of SDP in two official instances, the final client demo and the investor demo, and unofficially at our team's stand during the final SDP day. We will discuss in this section the two issues which appeared during the final days, and what we could have done to prevent them. Then, we will mention how we would have better organized the project if we were to start over.

1. **Robot fell off the table during face detection**
   This error baffled us at the beginning, since we had thoroughly tested edge detection and face detection, both individually yielding satisfying results (section 4.1). The error appeared because we did not test how these units of the system would interact with each other. The failure was caused by the fact that the processing speed decreased so much (edge detection and face detection had to be done in parallel) that the robot detected the edge too late.

2. **Robot's audio cues were not heard**
   On one side, it can be argued that our robot was never supposed to perform inside a crowded and noisy room. This was clearly assumed when we decided to implement voice commands, a feature which would tremendously suffer in a noisy environment. However, we considered using an alternative solution (such as lighting all the LEDs, or having the app display a message) to signal to the user that it is time to start inputting the remembered sequence.

Apart from the stated errors, it can be argued that the system was not optimal for the memory game. This is because the pattern to memorize was shown via LEDs lit through the GPIO pins of the Raspberry Pi, whereas the input from the user was processed by the EV3. This meant there was constant communication between the EV3 and the Raspberry Pi, which exposed the system to failure (a message not sent or received would have ended the game) and did not give users a fluent experience (the user has to wait after the pattern has been shown, then the user has to wait again after the signal is inputted). In simpler terms, this issue is caused by the fact that the robot is not controlled by one central unit, but by multiple units.

A possible fix to this problem would have been using the Raspberry Pi GPIO pins to also control the motors and the sensors (maybe using BrickPi). The arguments against doing this were that it would nullify all our work done upto that point (the ev3dev library would not be usable anymore). Since the EV3s were given to us by the university with lots of documentation, it would have been a high risk to invest a big part of our budget on something that might not work. In hindsight, we should have invested our resources into having only one unit managing all the robot movements. This would have reduced the complexity of the system, lowered the probability of system failure, and would have led to a more secure system at the end.

5.3 Original and strong work

1. **Aesthetic appearance**
   The penguin suit for the robot was stitched by hand and was an original design. We came up with the idea ourselves and felt based on the feedback that it was well received. It was a major addition to the project; as mentioned earlier aesthetic value was crucial to the success of the project.

2. **Games**
   We did extensive research to come up with the ideas for our games. They were scientifically backed as discussed in the introductory section. Once the scientific rationale was set, we came up with creative ideas for our pattern recognition games. We created a simple system to code these games and had a lot of versatility which allowed us to create difficulty levels. The games were unique and entirely our creation.

References

[1] George Savulich, Thomas Piercy, Chris Fox, John Suckling, James Rowe, John O'Brien, Barbara Sahakian. *Cognitive training using a novel memory game on an iPad in patients with amnestic mild cognitive impairment.*
https://academic.oup.com/ijnp/article/20/8/624/3868827

[2] Alzheimer's Society. *Facts for the media.*
https://www.alzheimers.org.uk/info/20027/news_and_media/541/facts_for_the_media

[3] Hunter EM, Phillips LH, MacPherson SE. *Effects of age on cross-modal emotion perception.*
https://www.ncbi.nlm.nih.gov/pubmed/21186914

[4] C B. Hall, R B. Lipton, M Sliwinski, M J. Katz, A. Derbyand, J Verghese *Cognitive activities delay onset of memory decline in persons who develop dementia.*
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2725932/

[5] Marla Kosowicz, Sarah E.MacPherson *Improving multitasking assessment in healthy older adults using a prop-based version of the Breakfast task.*
https://www.tandfonline.com/doi/full/10.1080/23279095.2015.1136310

[6] Fergus I. M. Craik, Ellen Bialystok *Planning and task management in older adults: Cooking breakfast.*
https://link.springer.com/article/10.3758/BF03193268

[7] Jordan A.Mann, Bruce A.MacDonald, I.-Han Kuo, Xingyan Li, Elizabeth Broadbent *People respond better to robots than computer tablets delivering healthcare instructions.*
https://www.sciencedirect.com/science/article/pii/S0747563214005524

[8] Elizabeth M. Zelinski, Ricardo Reyes *Cognitive benefits of computer games for older adults.*
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4130645/

[9] http://www.parorobots.com

[10] MQTT. *Communication Protocol: MQTT is a machine-to-machine connectivity protocol.*
http://mqtt.org/

[11] https://www.back4app.com/

[12] http://www.android-graphview.org/

[13] https://github.com/akshayCha/LeoApp

[14] Portet F, Vacher M, Golanski C, Roux C, Meillon B. *Design and evaluation of a smart home voice interface for the elderly: acceptability and objection aspects.*
Personal and Ubiquitous Computing. 2013 Jan 1;17(1):127-44.

[15] Amazon Corp. *Alexa Skills Kit.*
https://developer.amazon.com/alexa-skills-kit

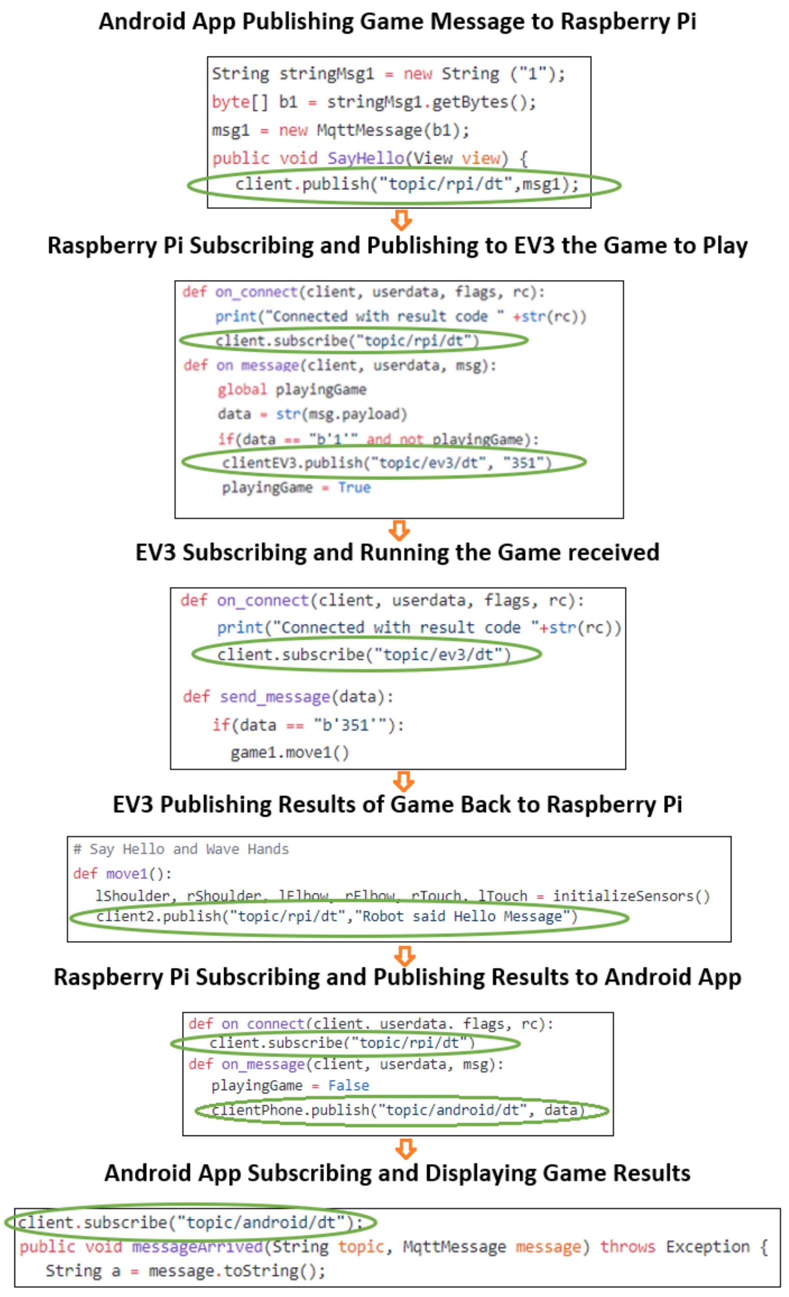
[16] Armin Ronacher. *Welcome to Flask.*
http://flask.pocoo.org/docs/0.12/

[17] Yaler GmbH. *Access devices from the Web: Yaler is a relay infrastructure for secure access to embedded systems.*
https://www.yaler.net

[18] Rainer Lienhart. *trained classifiers for detecting objects of a particular type.* `https://github.com/opencv/opencv/blob/master/data/haarcascades/`

Appendices

Appendix A: Code for MQTT Communication


Android App Publishing Game Message to Raspberry Pi
Raspberry Pi Subscribing and Publishing to EV3 the Game to Play
EV3 Subscribing and Running the Game received
EV3 Publishing Results of Game Back to Raspberry Pi
Raspberry Pi Subscribing and Publishing Results to Android App
Android App Subscribing and Displaying Game Results

```
String stringMsg1 = new String ("1");
byte[] b1 = stringMsg1.getBytes();
msg1 = new MqttMessage(b1);
public void SayHello(View view) {
   client.publish("topic/rpi/dt",msg1);
```

```
def on_connect(client, userdata, flags, rc):
    print("Connected with result code " +str(rc))
    client.subscribe("topic/rpi/dt")
def on_message(client, userdata, msg):
    global playingGame
    data = str(msg.payload)
    if(data == "b'1'" and not playingGame):
     clientEV3.publish("topic/ev3/dt", "351")
     playingGame = True
```

```
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("topic/ev3/dt")

def send_message(data):
   if(data == "b'351'"):
     game1.move1()
```

```
# Say Hello and Wave Hands
def move1():
   lShoulder, rShoulder, lElbow, rElbow, rTouch, lTouch = initializeSensors()
   client2.publish("topic/rpi/dt","Robot said Hello Message")
```

```
def on_connect(client, userdata, flags, rc):
   client.subscribe("topic/rpi/dt")
def on_message(client, userdata, msg):
   playingGame = False
   clientPhone.publish("topic/android/dt", data)
```

```
client.subscribe("topic/android/dt");
public void messageArrived(String topic, MqttMessage message) throws Exception {
   String a = message.toString();
```

Figure 16: Simple example of communication code

Appendix B: Code for Flask

- Import Flask and the Ask module as a python external library:

```
from flask import Flask
from flask_ask import Ask, statement, convert_errors
```

- Initialize Flask and Ask objects:

```
app = Flask(__name__)
ask = Ask(app, '/')
```

- Set the corresponding functions for different intents' POST requests and then map the values of JSON objects to the related variables in python:

```
@ask.intent('memoryGame', mapping={'mode':'mode'})
def memory_game(mode):
        print ("playing memory game:", mode)
```

- Start the web app based on Flask:

```
port = 5000
app.run(host='127.0.0.1', port = port)
```