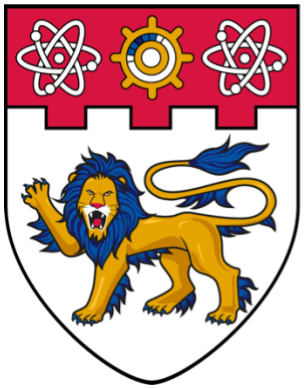


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

VIRTUAL EYE: HELPING THE VISUALLY IMPAIRED MAP THE REAL WORLD

SINGH AISHWARYA

School of Computer Science and Engineering

2023

NANYANG TECHNOLOGICAL UNIVERSITY

SCSE22-0364

VIRTUAL EYE: HELPING THE VISUALLY IMPAIRED MAP THE REAL WORLD

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering in Computer Engineering of the Nanyang Technological University

by

Singh Aishwarya

U1923952C

Supervisor: Dr. SMITHA Kavallur Pisharath Gopi

Examiner:

School of Computer Science and Engineering

2023

ABSTRACT

This research paper discusses the design and implementation of an enhanced indoor navigation system, aimed towards improving the existing technology by aiding the visually impaired. Loss of vision can drastically impair an individual's direction and mobility, especially in unfamiliar surroundings. Due to this, visually impaired individuals often find themselves needing further support and time to gain familiarity with new indoor settings. The objective of this paper is to present an enhanced indoor navigation system, customized to cater to the needs of the visually impaired. This has been achieved through the use of Bluetooth Low Energy (BLE) beacons, a BLE-supported Android device with in-built motion sensors and an Android Mobile Application.

The mobile application can be operated in three different modes – regular navigation, assisted navigation and free-roam mode. Regular Navigation has been designed for the visually abled

ACKNOWLEDGEMENTS

The author of this research paper would like to like to express her heartfelt gratitude to the following individuals and organizations for their support and contributions to this project:

Firstly, the author would like to thank her advisor and mentor Dr. SMITHA Kavallur Pisharath Gopi for her guidance and support throughout the entirety of the research. Her advice and constructive feedback were invaluable in shaping the direction and quality of the work presented in this paper.

The author would also like to like to express her appreciation for Fujitsu for the production of their Bluetooth Low Energy beacons and detailed hardware documentation. The clear setup instructions and hardware specifications significantly aided the implementation of the system.

Finally, the author would like to express her gratitude to her friends and family for their unwavering support throughout the year.

ACRONYMS

BLE	Bluetooth Low Energy
GPS	Global Positioning System
GLONASS	Global Navigation Satellite System
WLAN	Wireless Local Area Networks
RFID	Radio-Frequency Identification
VM	Virtual Machine
UI	User Interface
TTS	Text-to-Speech
LTS	Long-Term Support
SCSE	School of Computer Science and Engineering
CARA	Cybercrime Analysis & Research Alliance
IPS	Indoor Positioning System
RSSI	Received Signal Strength Indicator
IDE	Integrated Development Environment
MAC	Media Access Card
BFS	Breadth-First Search
DFS	Depth-First Search

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Background

Over the past decade, the popularity of navigation tools as resources for accurate route planning and wayfinding has grown immensely. The major tools in the market today have enabled accurate navigation and demonstrated advances in path planning and movement control, and most importantly, in localization. Localization accomplishes the complex goals of accurate navigation by necessitating the support of precise topographic representation of space [1].

Localization can be broadly classified into outdoor and indoor variants. Outdoor localization has experienced a surge in technological advancements, as traditional celestial navigation techniques have grown to depend on satellites, such as – Global Positioning System (GPS), Galileo and the Global Navigation Satellite System (GLONASS) [2]. The needs of these modern outdoor localization techniques have been met through efficient data collection using mobile mapping technologies and advancements in modelling and data storage. This has led to outdoor navigation becoming ubiquitous, with GPS receivers included in a variety of consumer electronic devices such as smartphones and tablet computers.

Innovations in outdoor navigation systems have encouraged researchers to explore indoor navigation techniques to assist pedestrians in finding their way through complex indoor environments such as airport terminals, subways, and retail malls. GPS signal blockages by building walls and ceilings and multi-path interferences have increased the complexity of indoor navigation systems. As a result of this, the plethora of such existing systems rely heavily on electromagnetic signals, which are customised on the basis of their range, accuracy, and their capacity to penetrate solid objects [3]. One of the fundamental techniques for indoor localization is the use of Wireless Local Area Networks (WLAN). Wi-Fi technology, in particular, is frequently used for this purpose, where an accuracy of several decimetres has been achieved. Radio-Frequency Identification (RFID) is another option for highly

accurate indoor localization, primarily up to several centimetres if smart floors are used [4].

Despite the fast-paced innovations in the field of indoor navigation, existing solutions do not completely cater to the needs of those with visual disabilities. According to the World Health Organization, 285 million people worldwide are visually impaired, with 39 million blind and 246 million with poor vision [5]. The loss of vision can severely dimmish an individual's orientation and movement – particularly in uncharted, environments. While guide dogs and canes are strong current resources to support their navigation, visually impaired people often encounter the need for further support and time to gain familiarity with new indoor settings. Existing indoor navigation applications are yet to address the greatest challenges of obstacle navigation and landmark identification faced by the visually impaired.

The aim of this study was to improve the existing solutions for indoor navigation, by customizing them to cater to the needs of the visually impaired. This was achieved by incorporating existing RFID beacon localization techniques with computer vision-based object detection to create a real-time voice-enabled assistant. The objective of this assistant is to aid the visually impaired in capturing vital information about their surroundings and navigating them through unknown indoor settings.

1.2 Objective

The fundamental objective of this study is to design and implement an indoor navigation system, consisting of a client-side Android Application and a cloud-based server. The application serves the purpose of providing both map-based visual indoor navigation as well as auditory and kinaesthetic navigation for the visually impaired.

The Android smartphone application, *VirtualEYE*, interacts with FUJITSU FWM8BLZ02 BLE Beacons to aid in indoor navigation. The application is also connected to a remote Flask Server running on an Azure Cloud Virtual Machine (VM). The server functions as the 'brain' of the entire system, enabling real-time calculation of paths, distances, and directions for the Android client. The system in its entirety is able to perform the following features –

- Receive Instructions/Commands from the user via:
    - User Interface (UI) input
    - Kinetic input
    - Voice input
- Interact with the BLE beacons placed around the test area to:
    - Retrieve the landmark in the closest proximity to the user
    - Calculate the distance between the user and the retrieved landmark
- Perform obstacle detection to:
    - Warn the user under assisted navigation of obstacles in the path
- Provide Tactile and Audio feedback for the visually impaired users:
    - Vibrate the phone to indicate the correct direction during navigation
    - Give audio warnings and instructions using Text-To-Speech (TTS) to provide easier interaction with the application
- Display an interactable map of the test area for visual navigation:
    - Present a list of available locations the user can select and navigate to
    - Update the map with markers of selected locations and the path calculated
    - Provide written directions for navigation

1.3 Scope

The central focus of this project is the implemented indoor navigation system, where the system consists of the client-side Android application *VirtualEYE*, the remote Flask Server, and the BLE beacons. The purpose of the remote Flask server is to perform heavy, time-sensitive computations and reduce the burden on

VirtualEYE to provide a more seamless and real-time experience to the user. It has been developed using Python version 3.8.5 and has been converted to a docker image. This docker image has been uploaded on a Linux VM (Ubuntu Server 16.04 – Long Term Support (LTS)) on Azure Cloud. This is done to facilitate time-critical computations and communication.

VirtualEYE has been developed in Kotlin **(enter version)** using Android Studio **(enter version)**. The tables below depict the permissions required by the application as well as the hardware features the application needs to function on the user's Android device.

Table 1-1: Permissions required by VirtualEYE, with stated reasons

Permission Required	Reason for Permission
BLUETOOTH BLUETOOTH_ADMIN BLUETOOTH_CONNECT BLUETOOTH_SCAN	A set of Bluetooth permissions is required to enable access to the smartphone's Bluetooth and to discover and interact with the BLE beacons.
ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION	The location data of the user is required to pair with the BLE beacon data to approximate the user's current location during indoor navigation
CAMERA	Required to perform Obstacle Detection for Assisted Navigation
INTERNET	Required to communicate with the remote Flask Server on Cloud – connect and stream data and response
RECORD_AUDIO	Required for enabling voice commands for the visually impaired users to set their destinations using speech
VIBRATE	Required to provide tactile feedback for the visually impaired user during Assisted Navigation

Table 1-2: Hardware features required by VirtualEYE, with stated reasons

Hardware Feature Required	Reason for requirement
bluetooth_le	Required to specifically interact with BLE-type beacons
camera	Required to perform obstacle detection for Assisted Navigation

The BLE beacons used in the indoor navigation system have been provided by FUJITSU. The beacons have been configured manually in *VirtualEYE*'s source code, and the communication between the two has been enabled via the Bluetooth Adapter present in the Android smartphone. The aforementioned test area has been set to Level 1 of the School of Computer Science and Engineering (SCSE) building. The beacons have been placed in the following important locations (venues with greater footfall) –

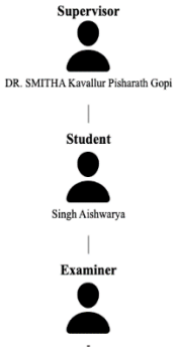
1. Cybercrime Analysis & Research Alliance @ NTU (CARA)
2. Student Lounge
3. Software Lab 1
4. Hardware Lab 1
5. Hardware Lab 2
6. Software Lab 2
7. Hardware Projects Lab

1.4 Project Plan

1.4.1 Project Organization

This project, and all of its components, were managed and accomplished completely by the author under the supervision and guidance of her supervisor, Dr. SMITHA Kavallur Pisharath Gopi. The project will be evaluated by the supervisor and an examiner from the faculty of the School of Computer Science and Engineering at the end of the Academic Year 2022/2023 Semester 2 examinations.

Figure 1-1: Project Organisation



1.4.2 Work Breakdown Structure

Table 1-3: Work Breakdown Structure

S/N	Phase	Task and Description	Effort Estimation
1	Project Initiation	• Arrange the first meeting with the supervisor • Post-meeting research and establishment of project outline and expectations	1 week
2	Research and Exploration	• Indoor Navigation - Path Finding Algorithms - Different indoor navigation techniques - Calculating orientation and user direction - Localization • BLE beacons • Obstacle Detection in Android - Trade-offs with accuracy and speed - Captured frame analysis • Client-Server Architecture • Map Rendering in Android	7 weeks
3	Software Development	• Develop and Implement System Architecture • Server Side: - WebSocket-based communication - HTTP communication - Flask Server setup - Develop path-finding algorithms - Create a communication protocol with the client • Client Side: - Bluetooth initialization and configuration - Sensor initialization and configuration - Create and customize the TTS manager	26 weeks

		- Initialize and prepare 'SpeechRecognizer' - Server (HTTP) communication - Camera configuration - Utilizing the Obstacle Detection model - Initialize a map of the test area - UI considerations • Develop, Test, and Improve	
4	Project Closure	• Final Report Preparation • Oral Presentation Preparation • Videos	TBA

1.4.3 Project Schedule

	Task Name	Duration	Start Date	End Date	Completion Status
1	PROJECT INITIATION	7 days	25/04/2022	01/05/2022	100%
a	Supervisor Meeting	1 day	25/04/2022	26/04/2022	100%
b	Outline of Expectations	1 day	26/04/2022	27/04/2022	100%
c	Post Meeting Research Draft Proposal	5 days	27/04/2022	01/05/2022	100%
2	RESEARCH AND EXPLORATION	44 days	02/05/2022	15/06/2022	100%
a	Indoor Navigation	3 days	02/05/2022	05/05/2022	100%
b	Indoor Positioning System	7 days	05/05/2022	12/05/2022	100%
c	Bluetooth Low Energy Beacons	7 days	12/05/2022	19/05/2022	100%
d	Obstacle Detection	3 days	19/05/2022	22/05/2022	100%
e	Android Speech Recognizer API + Text-to-Speech	3 days	22/05/2022	25/05/2022	100%
f	Android Indoor Mapping Tools	7 days	25/05/2022	01/06/2022	100%
g	Android Map Rendering APIs + Techniques	7 days	01/06/2022	08/06/2022	100%
h	Android Sensor Study	7 days	08/06/2022	15/06/2022	100%
3	PROPOSE SYSTEM ARCHITECTURE	3 days	16/06/2022	19/06/2022	100%
4	FINALIZE BLE BEACONS + SYSTEM ARCHITECTURE	7 days	20/06/2022	27/06/2022	100%
5	SOFTWARE DEVELOPMENT	184 days	28/06/2022	27/12/2022	100%
a	Requirements Elicitation and Analysis	14 days	28/06/2022	12/07/2022	100%
b	System and Object Design	21 days	12/07/2022	02/08/2022	100%
c	Implementation and Development	98 days	02/08/2022	08/11/2022	100%
d	Testing and Maintenance	49 days	08/11/2022	27/12/2022	100%
6	FYP REPORT	113 days	01/01/2023	20/03/2023	100%
a	Interim Report Submission	22 days	01/01/2023	23/01/2023	100%
b	Final Report Submission	56 days	23/02/2023	20/03/2023	100%
c	Amended Final Report Submission	35 days	20/03/2023	14/04/2023	100%
7	FYP ORAL PRESENTATION				

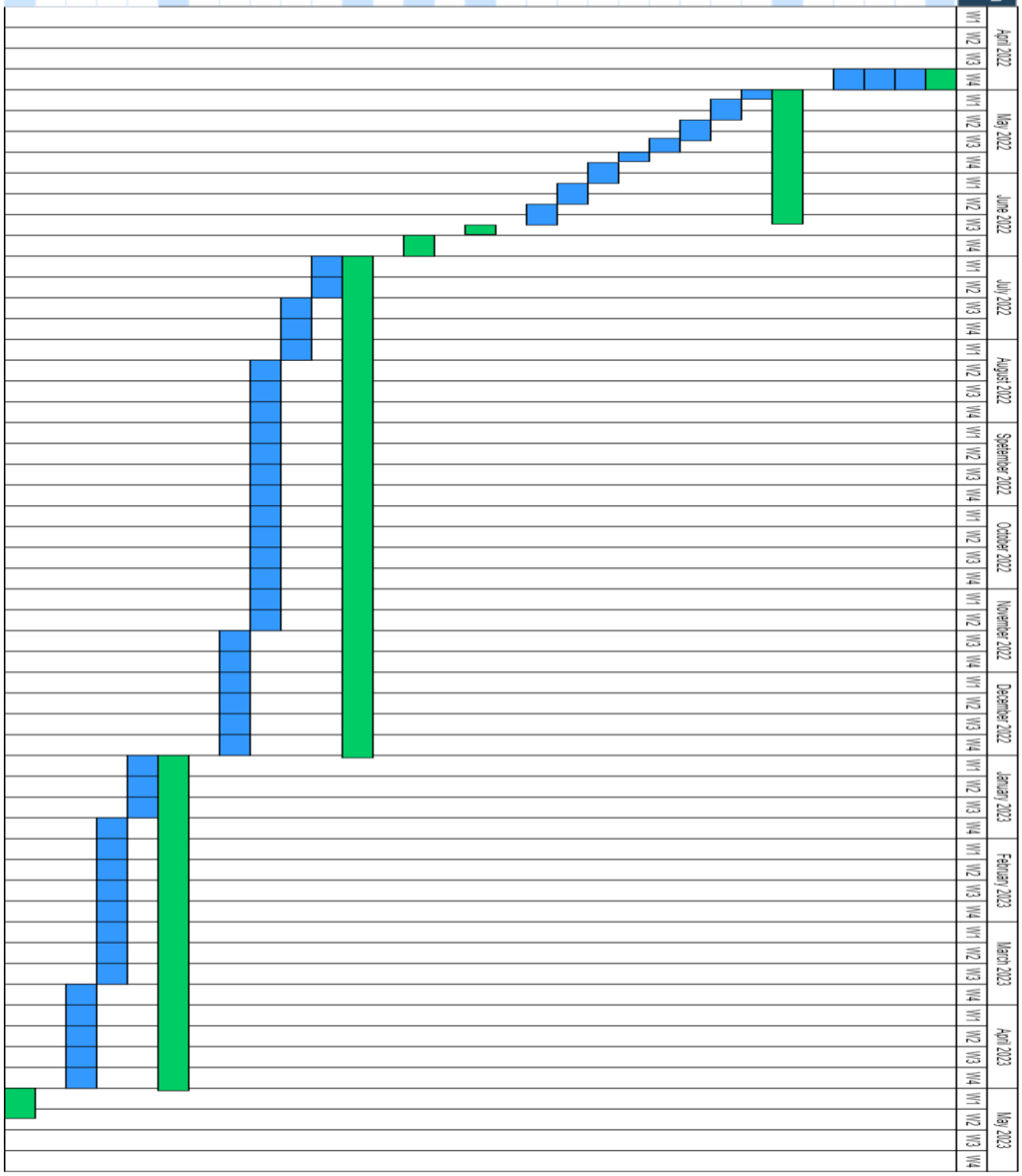


Figure 1-2: FYP Schedule

CHAPTER 2: LITERATURE REVIEW

Indoor mapping is the process of making interior locations and landmarks discoverable, using indoor positioning and navigation techniques. Each of these techniques has its own set of advantages, which have been explored in this chapter and have motivated the implementation of this project.

2.1 Indoor Positioning System (IPS)

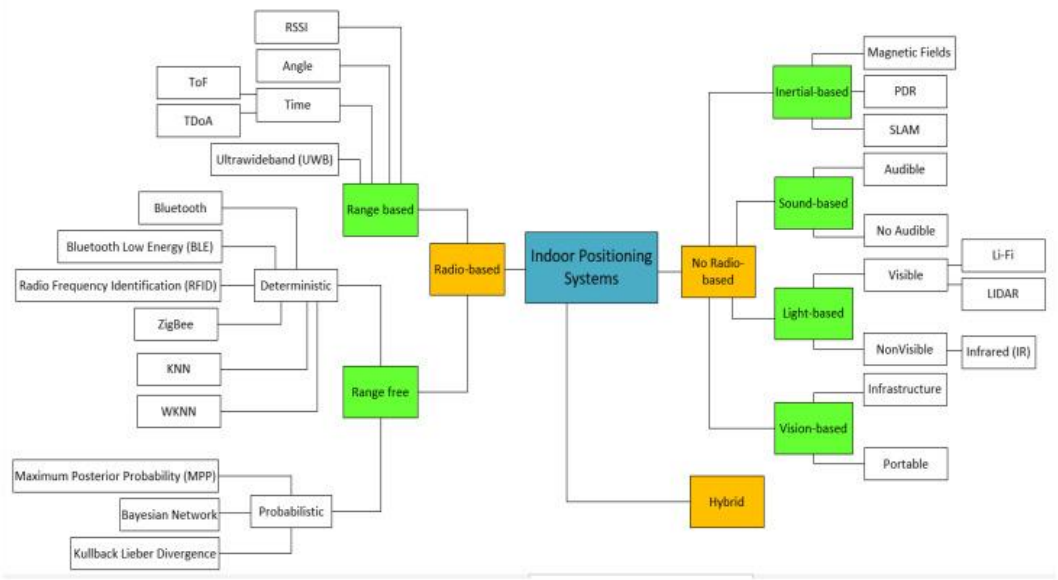
Positioning Systems, based on the tools employed, can be grouped into outdoor or indoor. Most navigation systems today are well-established in outdoor contexts, delivering accurate positioning and navigation assistance with negligible errors [6]. On the other hand, in indoor settings, satellite-reliant positioning systems such as GPS and GLONASS are ineffective. The accuracy of these systems suffers considerably as a result of signal loss due to collisions with indoor structures and can therefore not be used as reliably in indoor environments [7].

Indoor positioning systems (IPS) can be thought of as GPS, but for indoor locations. While GPS is utilized outside, IPS can detect real-time positions inside a facility to correctly establish the coordinates of people or assets. IPSs can be used for a variety of purposes, including identifying and tracking individuals, enabling medical monitoring, and supporting visually impaired individuals in their everyday routines. Indoor positioning approaches have gained a lot of traction, allowing the development of systems for a wide range of model situations. This has allowed IPSs to be classified in a variety of ways based on numerous parameters [8].

IPS can be grouped based on architectural design: self-positioning, infrastructure positioning, and supported by self-directed infrastructure. Devices choose their own placements in self-positioning architecture. The positions of devices released in the environment are used to estimate device placements in infrastructure positioning architecture. Lastly, in self-directed infrastructure-aided architecture, an external system calculates the position and delivers it to the monitored user in response to a request.

Additionally, most IPS employ radio frequency, vision-based, audible, and inaudible sound, inertial and light-based technologies. A complete classification of IPSs is shown below in Figure 2-1 [9].

Figure 2-1: Classification of IPS



To choose an appropriate IPS for the blind, it is critical to understand the efficiency and limits of different alternatives. Despite accuracy being a crucial performance parameter, other factors such as response time, availability, and scalability are equally significant and must be addressed. The following sections outline different systems, focusing on their most prominent characteristics and drawbacks.

2.1.1 Radio Frequency-based IPS

There are a number of diverse devices and wireless networks that can be deployed in indoor environments for indoor positioning. Some of these include Internet of Things (IoT) devices, RFID devices, Bluetooth beacons, and traditional Wi-Fi networks. The method of obtaining information in Radio Frequency–based IPS is the categorization parameter to classify the system into range-based and range-free Radio Frequency IPS.

2.1.1.1 Range–based Radio Frequency IPS

Methods based on range extract geometric information (distance or angle) from different nodes and then combine the geometric restrictions of each anchor to derive the user's location [10]. There are several methods for extracting geometric information from these signals. The most popular are approaches based on signal propagation time between the transmitter and the receiver, the angle of arrival (AoA), or received signal intensity (received signal strength indicator, RSSI). The focus of this study will be on RSSI.

RSSI-based location algorithms employ the intensity of the received signal to calculate the distance between the user and a node. These approaches assume that the attenuation of a signal as it travels from a transmitter to a receiver is proportional to the distance travelled [11]. To determine the distance between two places, a signal propagation model must be used to represent the environment with Bluetooth devices. The log-distance trajectory loss model is perhaps the most used model, with attenuation (in dB) equal to the logarithm of the distance traversed –

$$RSSI = P1\text{m} - 10\alpha \log_{10} d - \gamma$$

In the equation above, P_{1m} is the reference power value measured in dBm at a distance of one meter from the transmitter, α is the exponent of the loss of path γ, and d is the distance between the receiver and transmitter. P_{1m} and α are model parameters that must be determined via calibration. This is done by collecting the RSSI values at predefined positions, with known distances to the positioned nodes, which is usually done using regression methods. After this, the distance can be estimated according to the path-loss model using the Maximum Likelihood Estimator [12].

2.1.1.2 Range-free Radio Frequency IPS

Range-free approaches rely on data collected from connections. The resulting values may be utilized to estimate the position without having to calculate any

measure of reach at a node. This can be done in two ways: proximity algorithms and digital printing methods.

Proximity algorithms employ connection information to determine the user's position based on the number of nodes in the vicinity. Proximity algorithms are based on a basic concept: if a user receives a signal from a node, the user's position must be close to the position of the said node [13]. The procedure is carried out in the following steps: The user initially scans the channel for radio signals from the placed nodes. The user's position is calculated to the position of the node when it is detected. When many nodes are available, the model will choose the one with the strongest signal.

Figure 2-2: Diagrammatic representation of the proximity algorithm

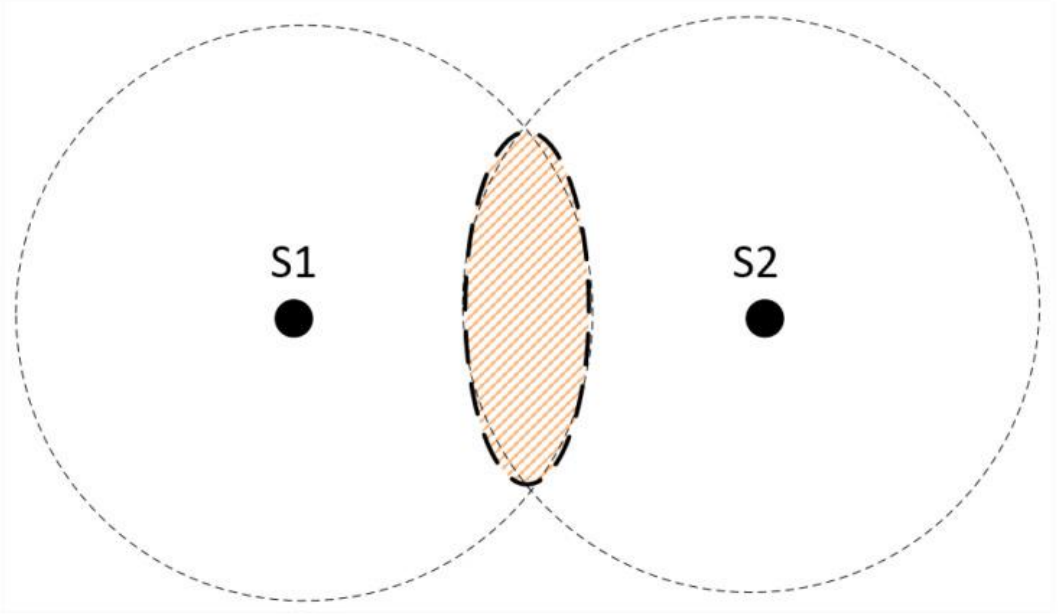


Figure 2-2 represents the working of the proximity algorithm. The circles reflect the coverage area of the placed nodes, and an intersection between them is required. When a user moves around the circle maintained by node S1, their location is estimated using placed node S1 as a reference. When a user enters the circle of node S2, the position of placed node S2 is calculated. The placed node will be chosen based on the RSSI at the junction of the two circles. Thus, the amount of inaccuracy offered by proximity algorithm approaches is proportional to the coverage area size. In other words, if the coverage area is wide, the error increases, and if the coverage

area is small, the number of anchor nodes required for total internal area coverage grows.

The fingerprinting location system on the other hand makes use of information sensed in certain areas, which must first be determined. As a result, the approach necessitates the observation of two phases: offline and online. The region is divided into small cells in the first phase to specialize in the identification of each point of interest, allowing data to be collected to establish a database. The core concept of digital printing is to record a signal pattern in a manner akin to a relational database. This pattern is used as a guideline when looking for a value via equivalence or approximation. The position is approximated in the second phase by comparing the collected data to the records stored in the database. The biggest downside of the approach, regardless of the origin of the fingerprints, is the work necessary to gather samples to build the database, which raises the cost and restricts scalability.

2.1.2 Inertial Sensors-based IPS

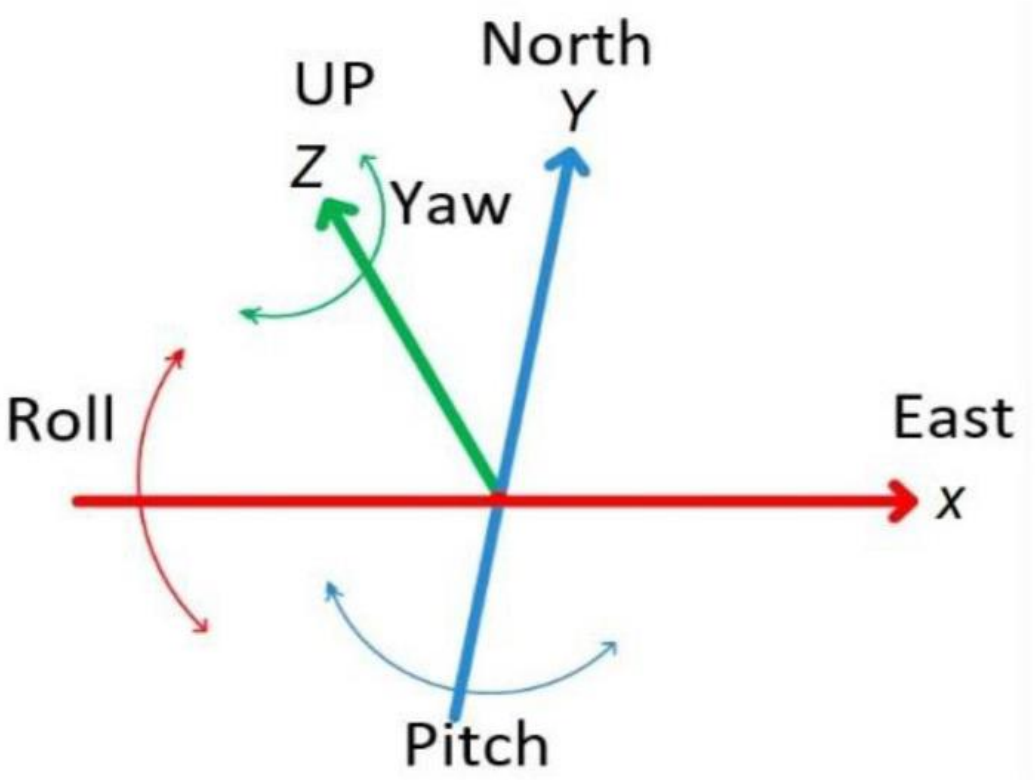
Network-based systems determine user locations by analysing the resources of wireless network signals. This paradigm necessitates intervention in the environment in the form of physical infrastructure. Inertial information-based systems determine their locations without the need for external resources.

Accurate information about a target's location is required, which is why these models use the geographic coordinate model, which includes latitude, longitude, and altitude, to determine the absolute (inertial) position. The sensors enable the absolute (inertial) location to be experienced instantaneously in the roll, pitch, yaw (RPY) system utilizing the x-, y-, and z-axes (Figure 2-3) [14]. The x-axis represents the nominal orientation (front), the y-axis is orthogonal to the x-axis and points to the left side, and the z-axis (yaw) points up.

Inertial sensors are typically built into units of inertial measurement (IMUs), which include an accelerometer, a gyroscope, and a magnetometer with three axes each. The magnetometer is not an inertial sensor; but, because it is part of the IMU, it will be considered in combination with the other sensors in this study. To create an

address, the magnetometer sensor employs the earth's natural magnetic field or artificial magnetic fields of alternating current. When acceleration in each direction is sensed, accelerometers may be used to calculate changes in the user's location. This is a very approximate estimate that may be improved by observing variations in a course with a gyroscope.

Figure 2-3: Roll-Pitch-Yaw Reference



Since there is no requirement to put equipment in the environment, inertial systems are completely scalable in terms of size and number of users. Furthermore, smartphones now include inbuilt IMUs. As a result, the system has a low cost. However, it is critical to consider the lower accuracy that is caused by inertial drift.

2.1.3 Computer Vision-based IPS

IPS that employ computer vision recognize, monitor, and steer persons in indoor spaces using information gathered by cameras and a set of image processing algorithms [15]. The camera on a smartphone is used by the user to observe and

take photographs or videos from that user's point of view. The collected data may be compared to previously saved files or taught models – decision trees, neural networks, etc.

Indoor locating techniques that rely on a single sort of sensory perception are more prone to failure under specific physical and logical situations and dispositions [16]. One method for reducing the errors or limits of these systems is to combine distinct sensory experiences, resulting in a hybrid arrangement that, while more expensive to install, gives more accurate and dependable findings. To test the degrees of accuracy and constraints, the scientists created an indoor positioning system that integrated camera sensors, inertial sensors, and the Wi-Fi sensor found on smartphones.

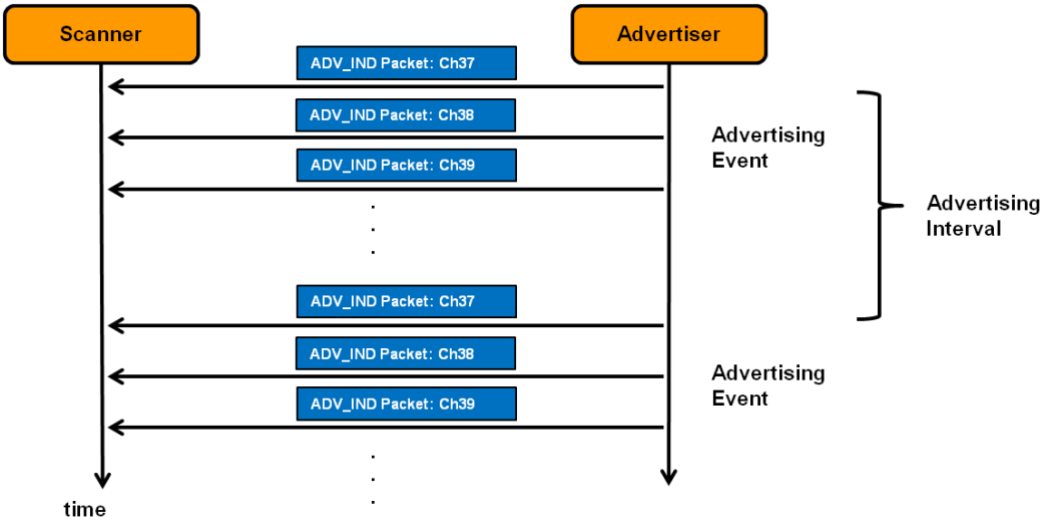
2.2 Bluetooth Low Energy (BLE) Beacons

A Bluetooth beacon is a tiny wireless device that uses Bluetooth Low Energy (BLE) technology to communicate. It sends a radio signal composed of letters and numbers that are broadcasted at short, regular intervals. Once in range, a Bluetooth-enabled device such as a smartphone, gateway, or access point may identify the beacon. BLE technology is a new wireless personal area network technology standard that is part of the Bluetooth 4.0 standard. Both Bluetooth and BLE technologies operate at 2.4GHz and use the same modulation mechanism, Gaussian Frequency Shift Keying. In theory, BLE application throughput is roughly 300kbps, which is up to seven times less than traditional Bluetooth [17]. It is therefore more power efficient than classic Bluetooth – allowing beacons to run for years on small coin-cell batteries.

BLE beacons are programmed to broadcast ‘IDs’ at set frequencies (configurable by the user). This ID serves as a unique identifier, which is used by a Bluetooth-enabled device to perform an action function based on the ID received. With this, the entire BLE architecture can be explained in terms of three processes – advertising, scanning and connection. Advertising and Scanning make up the discovery process in the BLE architecture. In this, devices use advertising channels

to identify each other, where one device performs advertisement and the other performs scanning. The discovery process pertinent to this project is Passive Scanning – where the advertiser is not informed if the advertised packet has been received by the scanner or not. This has been illustrated in *Figure 2-4* [18].

Figure 2-4: Illustration of Passive Scanning



Based on the advertising data (device name, service UUID, RSSI, and so on), the Scanner chooses an appropriate Advertiser to connect with and start an exchange. Once a Scanner has gathered enough information to determine which Advertiser to connect to (including its MAC address), it becomes an Initiator, commencing a BLE Link Layer connection procedure. The devices are then linked, and data packets may be transferred. The Initiator then becomes the Link Layer Master, while the Advertiser becomes the Link Layer Slave.

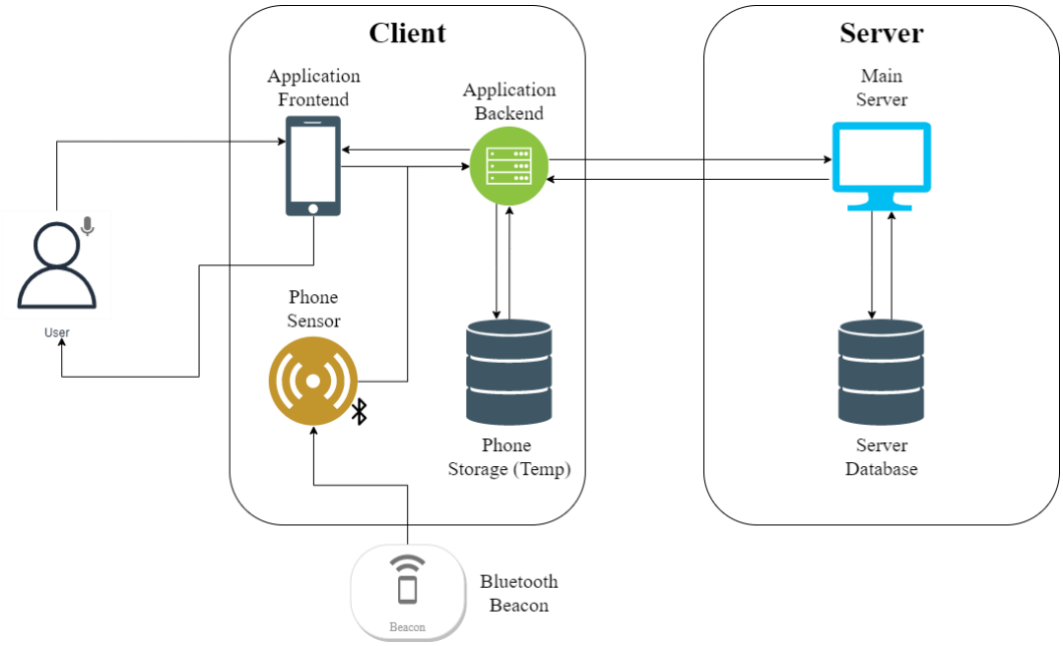
CHAPTER 3: SYSTEM ARCHITECTURE

This chapter discusses the architecture of the proposed indoor navigation system. The discussion begins with details on the initial plans and considerations for the system and continues towards the description of the final architecture of the system, consisting of the path planning server and the client-side Android application, *VirtualEYE*.

3.1 Initial Considerations

The figure below illustrates the first proposed architecture for the indoor navigation system –

Figure 3-1: Initial proposed System Architecture



The system architecture in *Figure 3-1* illustrates how all the components are connected to each other. The user interacts solely with the client-side Android application *VirtualEYE*, via audio or UI input. In the above architecture, the client will be connected to the server via a WebSocket connection. The backend of *VirtualEYE* will initiate a connection with the Python WebSocket and start communication once the user has given instructions to the application. For the

detection of BLE beacons, the Bluetooth manager of the smartphone will initialize the scanner and start scanning for nearby beacons. This information will be sent to the server, where the server will check its database against the received BLE beacon 'fingerprint' and send back an appropriate response. In addition to this, obstacle detection will also be performed by the server, where images are streamed from *VirtualEYE* to the server via the same WebSocket. The image frames from the camera stream will be encoded in Base64 format and will be then sent to the server for obstacle detection. The smartphone's temporary storage is used to hold these encoded frames temporarily before sending them for obstacle detection. The server, after receiving the encoded images, performs obstacle detection using the ImageAI API and YoloV4 model. The obstacles detected are then sent back to the application's backend, where the TTS engine is used to alert the user of the same.

This architecture solely relies on BLE beacons for indoor navigation. Based on the BLE beacon detected, along with its fingerprint data, the server will calculate the shortest path to the chosen destination in real time. It relies on basic RSSI-distance calculations to estimate the distance to the next beacon/node. Upon implementing this architecture, there were several pitfalls identified -

- User Interaction:
    - Using only audio input for the visually impaired proved to be unreliable as the input captured was not accurate for most trials.
    - The use of only BLE beacons did not provide enough information about the visually impaired user's environment – it was also not able to accurately localize and determine where the user was heading.
    - Audio feedback being the only medium of feedback for the visually impaired user was not very efficient in terms of providing timely instructions and alerts of their surroundings – there was an overwhelming about of feedback being given, at a noticeable delay as well.
- Client – Server Communication:
    - There was a significant delay in obstacle detection with the initial system architecture. Despite adjusting the frame rate on the client

side and tuning the model parameters on the server side, there was a noticeable lag in the results received. This had a major impact on the application's performance.

    - Using the same WebSocket connection also leads to an added delay in real-time path calculation for indoor navigation. The user would be alerted of the location much after passing it – another damper on seamless user experience.

- BLE Beacons:
    - The RSSI value received from the BLE beacons was not precise – there were a lot of fluctuations in the value received by the Android application.
    - The values of RSSI varied significantly depending on the location it was placed, despite being the same physical distance from the phone.
    - The distance calculated from the RSSI value received was not enough to accurately localize the user in the indoor setting.
    - There were no checks/algorithms to determine turns taken by the user – the beacons were not enough to determine this information.

Based on the observations from the first round of testing of different functions from the initially proposed system architecture, the conclusion was to modify the system architecture to address the pitfalls observed during testing.

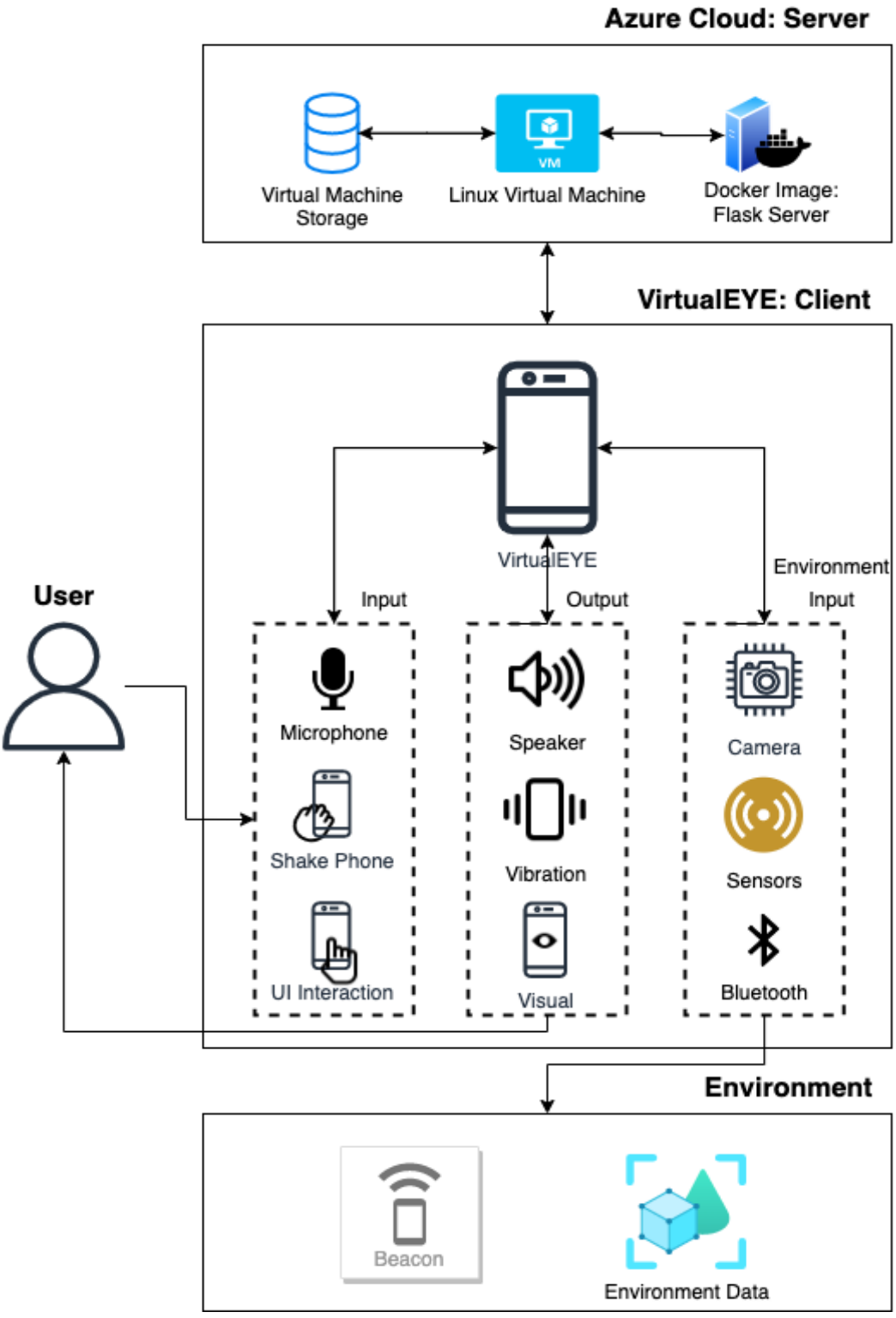
3.2 Final Architecture

This section provides a detailed outline of the final, implemented system architecture. The changes made to the architecture proposed in Section 3.1 are listed in this section, along with the reasons for the same. The final system architecture, along with its individual components has been discussed in detail in the upcoming sub-sections.

3.2.1 Final System Architecture

Figure 3-2 illustrates the final system architecture that has been implemented.

Figure 3-2: Final implemented System Architecture



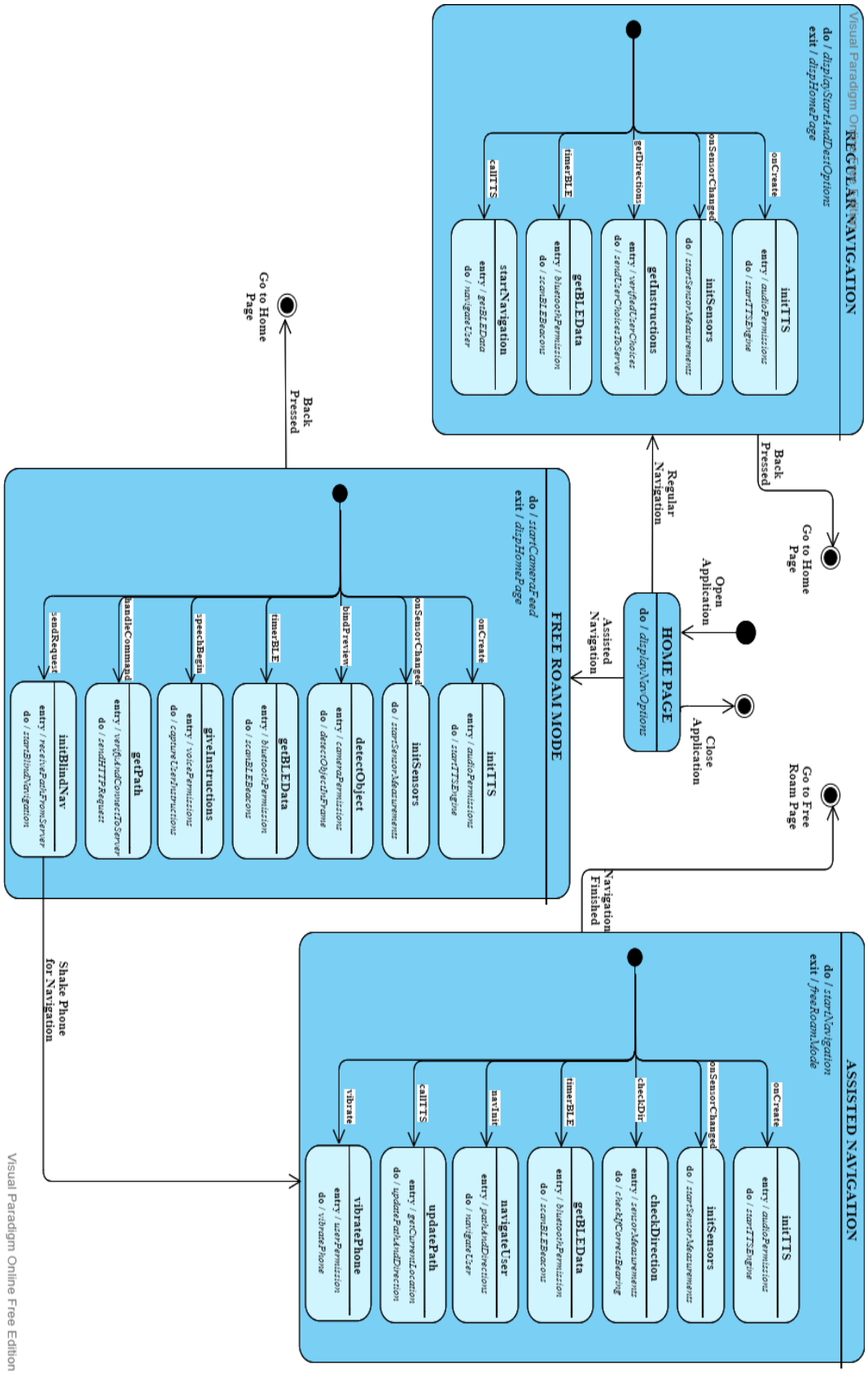
In this final architecture, new input sources and feedback mediums have been introduced to address the issues of the initial architecture. The user can interact with VirtualEYE via three input sources: UI Interaction (interacting with buttons, dropdown menus, etc for the visually abled), audio input (via microphone to set destination for navigation) and shake input to navigate to other parts of the application or enable microphone for audio detection. The feedback given by the application is also done in three ways: visual feedback (output presented on screen for the visually abled), audio feedback via phone speakers to provide instructions and alerts during navigation, and tactile feedback via phone vibration to navigate the user in the correct direction.

For interacting with the environment, additional sensors – magnetometer and accelerometer – are used in conjunction with the Bluetooth scanner and camera. The sensors provide additional information about the user's current bearing and direction, as well as the speed and movement of the user that aids in localization during indoor navigation. The Bluetooth scanner remains from the previous architecture to interact with the BLE Beacons placed in the test environment and the camera is used to perform obstacle detection.

To address the client–server communication issues in Section 3.1, the WebSocket has been replaced with a Flask server in the final architecture. The communication speeds with the Flask requests are much more efficient and provide a better user experience by reducing calculation delay. To further improve the efficiency of the communication, the server has been shifted from a local IP to a Linux Virtual Machine on Azure Cloud. This was done by creating a docker image of the Flask server and then hosting it on the cloud. This allows any device with the application to connect to the server via the Internet and eliminated the need to be connected to the same network to perform localization.

The aforementioned details can be visualized in the overall dialog map of the indoor navigation system in *Figure 3-3*. The sections following *Figure 3-3* continue the discussion of the final architecture in detail by providing relevant illustrations to explain the program flow of the application.

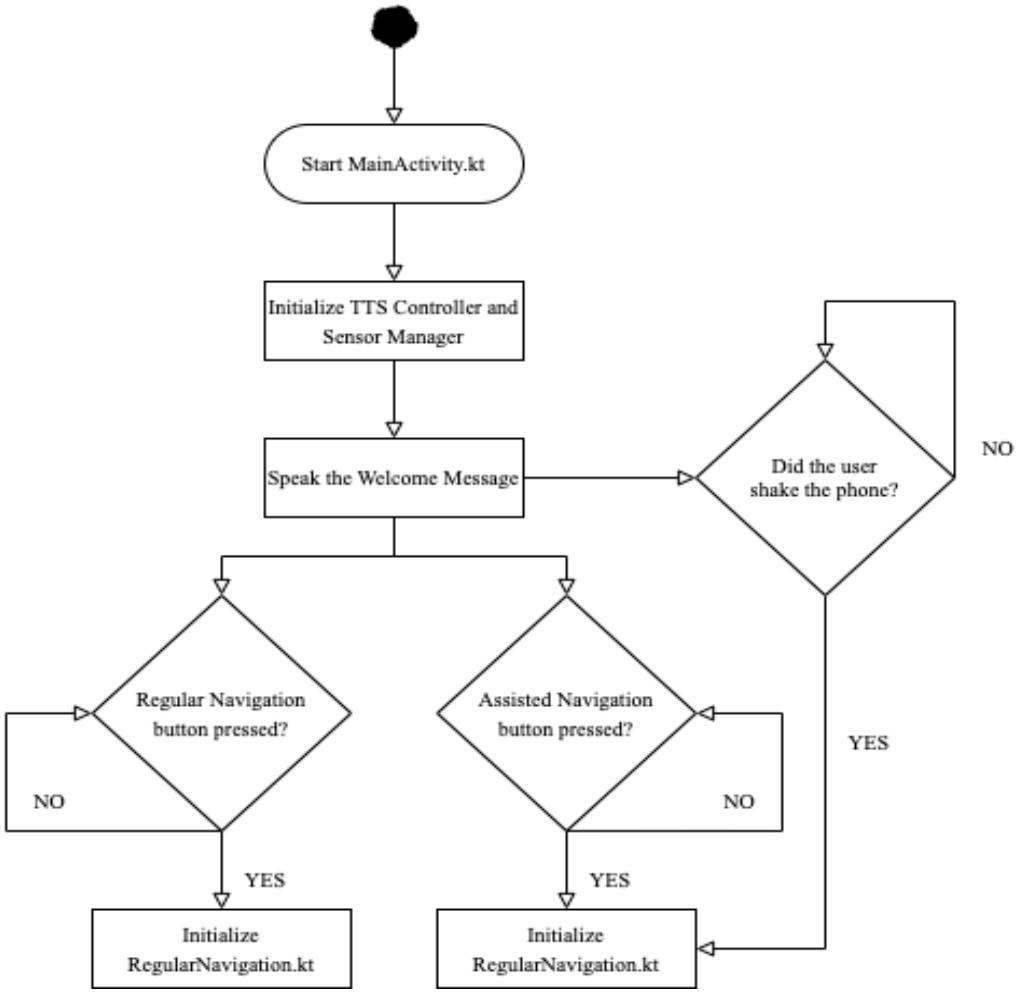
Figure 3-3: Complete System Dialog Map



3.2.2 Home Screen: MainActivity.kt

The application opens up to the Home Screen, or MainActivity.kt. *Figure 3–4* illustrates the dialog map for the same. Upon the creation of the activity, the TTS controller and sensor manager are initialized in the application. After this, the welcome message is spoken (providing instructions and further context for the visually impaired users). Next, the application awaits user input. The application can accept two different types of inputs – kinetic (shaking the phone) or UI interaction (buttons). Kinetic input is provided for the visually impaired – once the user shakes the phone, the user is redirected to Free Roam Mode (AssistedNavigation.kt). Apart from this, there are two on-screen buttons – one that redirects the user to Regular Navigation Mode (RegularNavigation.kt) and one that redirects the user to Free Roam Mode.

Figure 3-4: MainActivity.kt: Dialog Map



3.2.3 Regular Navigation (Vision-Based): RegularNavigation.kt

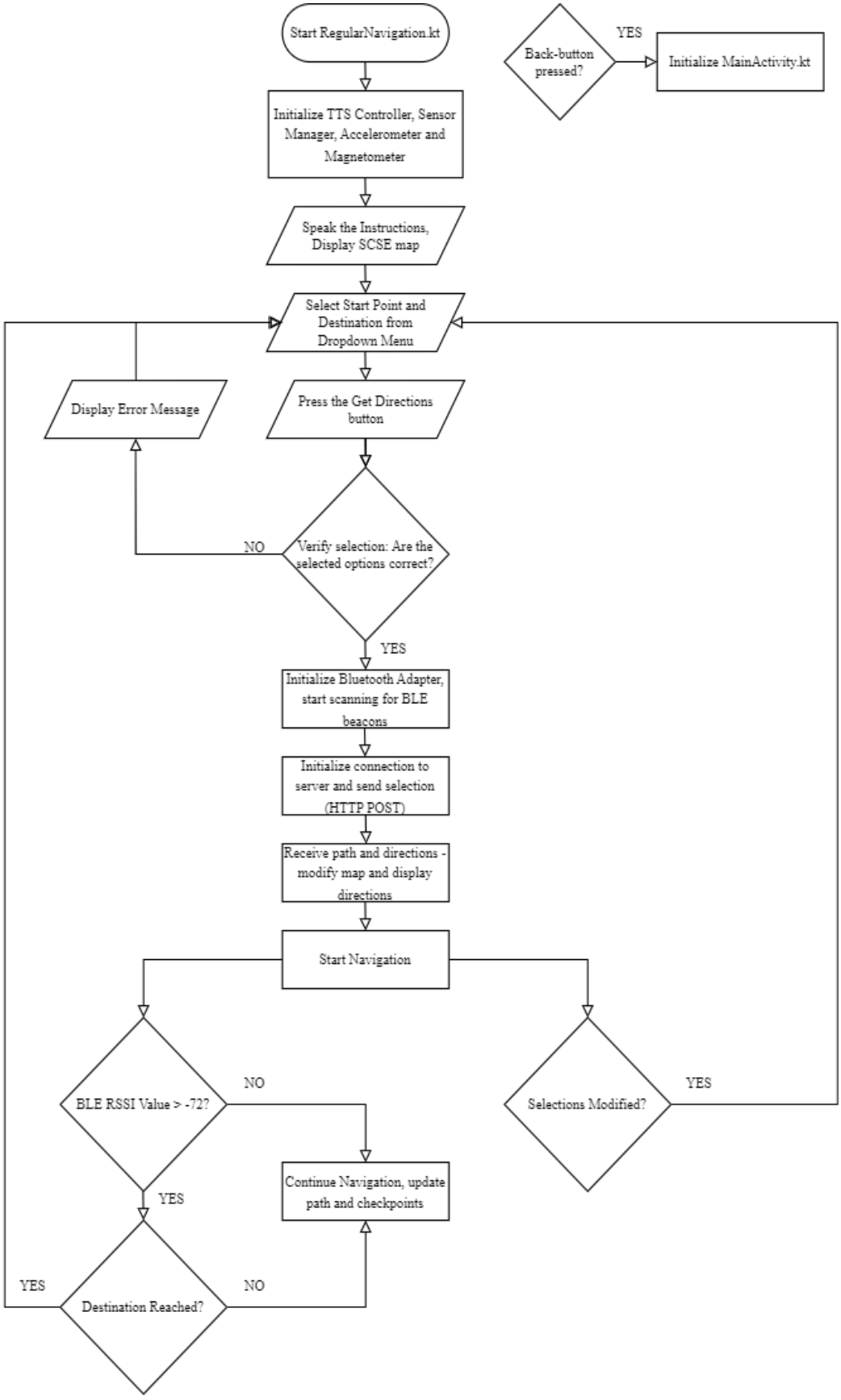
Regular Navigation mode supports vision-based indoor navigation. The screen consists of an indoor map of the test environment – SCSE Level 1 – as well as drop-down lists for users to select the start location and destination. *Figure 3-5* illustrates the flow of the program in RegularNavigation.kt.

Upon initialization of the activity, the TTS controller, sensor manager, accelerometer, and magnetometer are loaded, and the welcome message is spoken as well. At this point, the application awaits user input. Back press on the smartphone redirects the user back to the Home Screen. The start location and destination can be selected from the drop-down menu, and upon pressing the Get Directions button, an HTTP request is made to the server, sending the verified selected options. The server responds with the directions and the path, and the UI updates the map with the same. At the same time, the Bluetooth Manager is initialized and BLE beacon scanning begins.

In this activity, BLE beacons are used with the other aforementioned sensors to perform localization. The sensors aid in calculating the direction and bearing, as well as the distance based on the steps taken by the user. This is used in conjunction with the data received from the BLE beacons that are scanned by the application. The BLE beacons are used for proximity ranging and the sensors are used to re-affirm and validate the information calculated from the BLE beacon data.

Based on the testing methods described in Chapter 5, if the BLE RSSI value exceeds -72, the beacon is considered in range and its information is used to calculate the distance to the beacon. This process continues until the destination node or beacon has been detected. Every time a beacon is detected, the application sends the corresponding node/location to the server and the navigation takes place seamlessly, in real-time for the user.

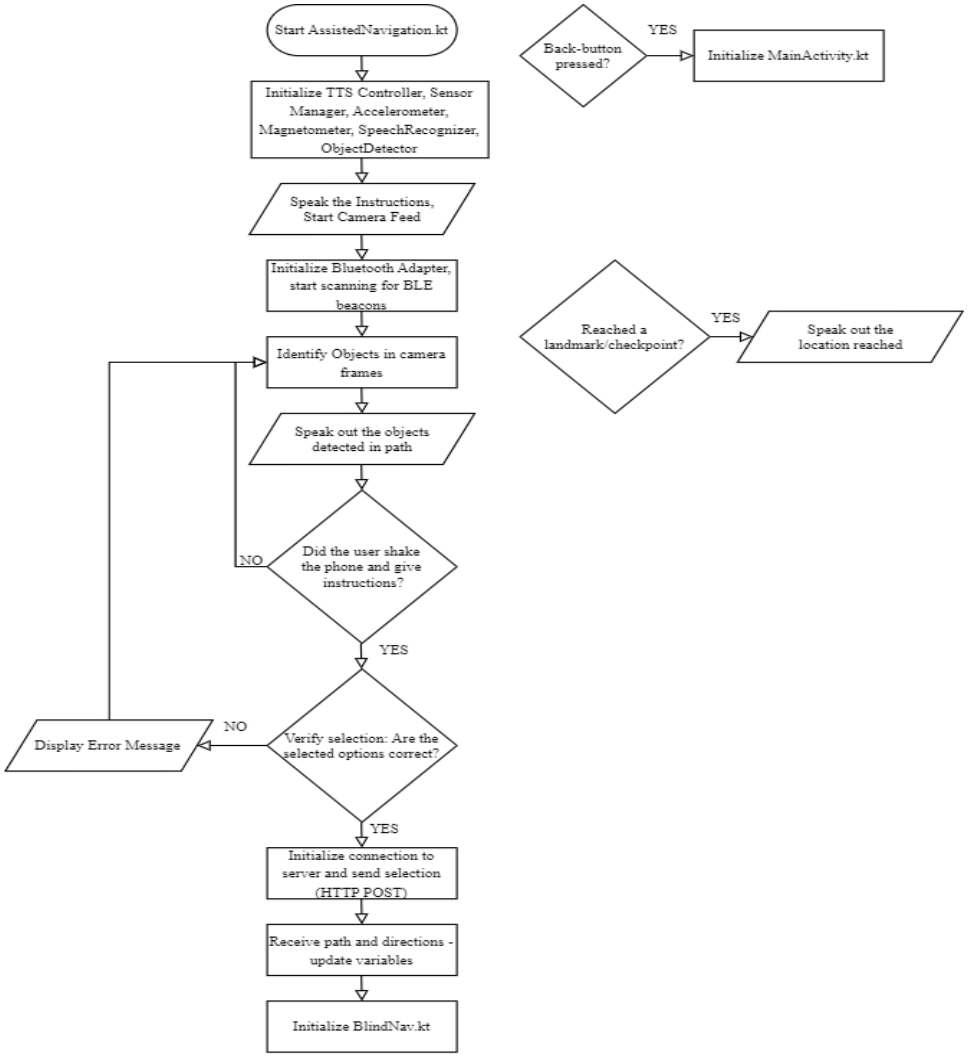
Figure 3-5: RegularNavigation.kt: Dialog Map



3.2.4 Free Roam Mode: AssistedNavigation.kt

Free Roam Mode can be accessed from the Home Screen as described in Section 3.2.2. *Figure 3-6* illustrates the logic flow in the activity. Most initializations remain the same from the previous activities, however, in addition to them, the camera stream is booted up in this activity. Obstacle detection takes place in this activity, where frames are analysed from the camera stream, and the obstacle detected is spoken out to alert the user. Additionally, the activity awaits user input (kinetic – phone shake) to begin Assisted Navigation (BlindNav.kt) for visually impaired users. The verification and request processes are similar to that in Section 3.2.3, and the results from the same are used to update the global variables before the next activity is initialised.

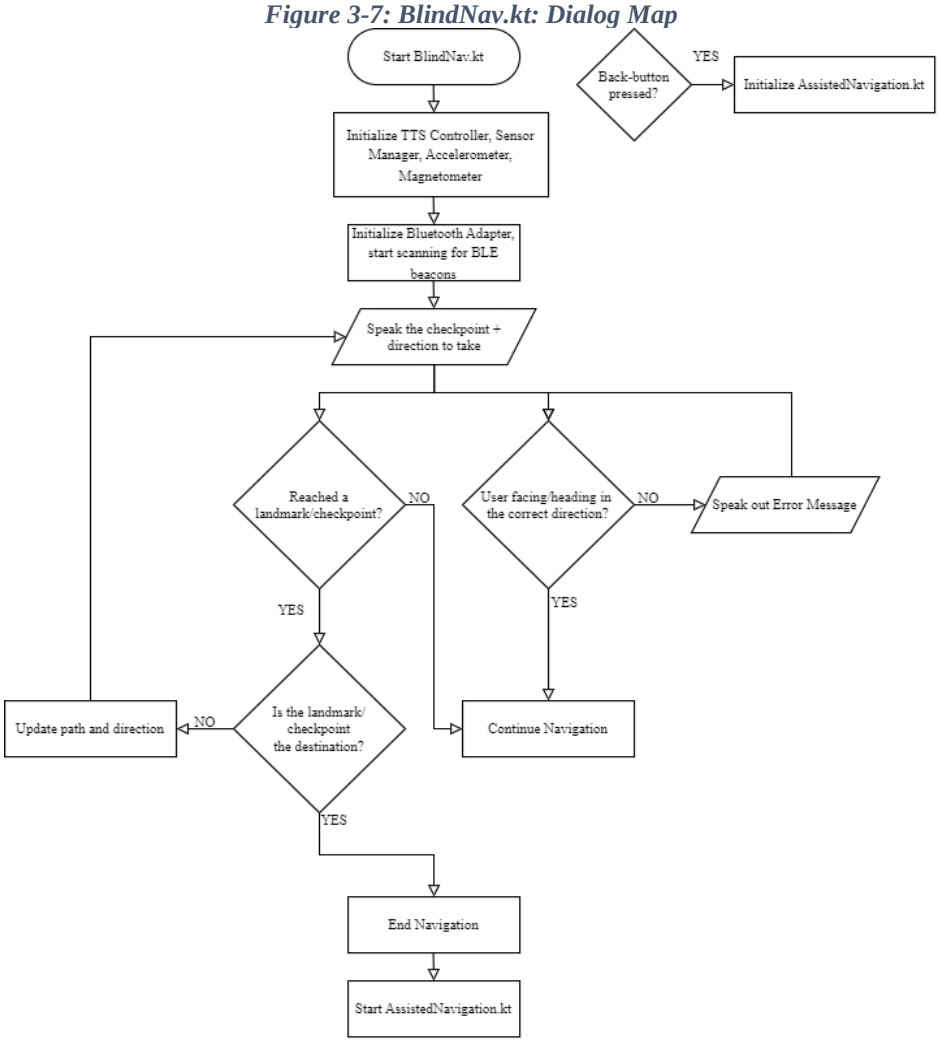
Figure 3-6: AssistedNavigation.kt: Dialog Map



3.2.5 Assisted Navigation (Visionless Navigation): BlindNav.kt

This activity is similar to the Regular Navigation mode in terms of navigation algorithms and logic. Figure 3-7 illustrates the algorithmic flow of this activity. Sensor and TTS initializations remain the same as previous activities. The significant difference in this activity is the feedback given from the application. Visually impaired users struggle to navigate in uncharted, indoor settings, and thus the purpose of this activity is to provide as much information about the user's surroundings. This activity tells the user how far they are from the next checkpoint, provides tactile (vibration) feedback to indicate if the user is heading in the right direction, and continues to give information about the user's immediate surroundings to safely guide the user to their destination.

Figure 3-7: BlindNav.kt: Dialog Map



CHAPTER 4: SYSTEM IMPLEMENTATION

This chapter discusses the implementation of the final architecture discussed in Chapter 3. It will cover the tools and how they were utilized in implementing the architecture.

4.1 System Implementation Tools

Table 4-1 illustrates the tools and APIs that were used to implement the final architecture in Section 3.2 –

Table 4-1: System Implementation Tools

Tool/API	Description
Android Studio	• Integrated Development Environment (IDE) used to develop *VirtualEYE* Android application • Programming Language: Kotlin + XML
SketchFab	• Modelling software used to construct indoor maps for the test environment
Tensorflow Lite	• Open-source smartphone library to deploy models on devices (smartphones)
Firebase ML Kit	• Ready to use smartphone APIs to perform machine learning tasks on the device
Visual Studio	• IDE used to develop the Flask Server • Programming Language: Python 3.8.5 • Dependencies used: Flask Framework,
Docker	• Tool to allow the server, with its dependencies, to run in packages called containers without the need to set up the environment.
Azure Cloud	• Cloud platform that provides Linux VM to host the server on the Internet

4.2 Indoor Mapping

The chosen test environment, SCSE Level 1, was mapped using the aforementioned tool SketchFab. The building and indoor map reference was taken from NTU Maps [19] as seen in *Figure 4-1* –

Figure 4-1: Screenshot of SCSE Level 1 from NTU Maps

This reference was used in the SketchFab software to create indoor environments to an appropriate scale with labels to identify important landmarks and checkpoints. To add this map to the Android application, an external library - Subsampling Scale Image View [20] – was used. This is a library created by David Morrissey to view images in a custom view. These views are designed to display large images (such as maps and building data) without affecting the application's performance and image quality. The library also allows the zooming and panning of images, as well as the addition of custom markers – features that are used in *VirtualEYE* for indoor mapping and navigation. This is illustrated in *Figure 4-2* –

Figure 4-2: Screenshot of the map from VirtualEYE, displayed using the library

4.3 Beacon Configuration and Placement

The beacons used in this study fulfilled the bare minimum requirements of a BLE beacon – it only advertises the device name, MAC (Media Access Card) Address, and its RSSI value. The manufacturing company only provides a technical specification document [21] to understand the hardware capabilities of the beacons. The user of the beacons must write their own code, tailored for their use case, in order to effectively use the beacons (as no SDK is provided). Hence, these beacons

had to be configured and tested multiple times to choose the best parameters for indoor navigation and localisation.

4.3.1 Beacon Detection

To communicate with the beacons, the Android's BluetoothAdapter has been utilised. *Figure 4-3* is the code snippet used to initialize and define the aforementioned adapter –

Figure 4-3: Screenshot from Android Studio: Initialising Bluetooth Adapter

```
// BLE Scanner Init
val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
if (bluetoothAdapter == null) {
    finish()
}

mBluetoothLeScanner = bluetoothAdapter?.bluetoothLeScanner
mScanCallback = initCallbacks()

val settings = ScanSettings.Builder()
    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
    .build()

val filter1 = ScanFilter.Builder()
    .setDeviceName("FCL Beacon1")
    .build()

val filter2 = ScanFilter.Builder()
    .setDeviceName("FWM8BLZ02")
    .build()

val filters = mutableListOf(filter1, filter2)

mBluetoothLeScanner?.startScan(filters, settings, mScanCallback)
```

If the adapter is null, the scanning does not start up. If not null, it launches a BluetoothLeScanner, which searches for BLE devices. The code also includes a callback method (*mScanCallback*) that is invoked when a BLE device is discovered.

The code then uses ScanSettings.Builder to configure the scan settings. The scan mode is set to SCAN MODE LOW LATENCY for this use case, which implies that the scan will be performed at a high pace but will require more power (which ensures a more real-time scanning algorithm).

The Builder then constructs two ScanFilter objects, filter1, and filter2, which are used to filter BLE devices based on their names. Filter1 is seeking for a device called "FCL Beacon1", whereas filter2 is looking for a device called "FWM8BLZ02" – to filter out the required beacons from other BLE devices in the vicinity. The filters, together with the settings and the callback function, are added to a list and sent to the *startScan()* method. This initiates the search for BLE devices that meet the provided filter conditions.

Figure 4-4: Screenshot from Android Studio: Scan Method

```
private fun initCallbacks(): ScanCallback {

    return object : ScanCallback() {
        @SuppressLint("MissingPermission", "NewApi")
        override fun onScanResult(
            callbackType: Int,
            result: ScanResult
        ) {
            super.onScanResult(callbackType, result)

            if (result.device != null) {
                if(result.device.name != null) {
                    //addDevice(result.getDevice(), result.getRssi());
                    Log.i( tag: "BLE NAME: ", result.device.name)
                    //println(result.device.name)
                    Log.i( tag: "BLE MAC: ", result.device.toString())
                    bleMAC = result.device.toString()
                    //println(result.device)
                    Log.i( tag: "BLE RSSI: ", result.rssi.toString())
                    rssiVal = result.rssi
                    //println(result.rssi)
                }
            }
```

The code snippet in *Figure 4-4* defines the method "initCallbacks()" that returns a ScanCallback object. The ScanCallback class is used to receive callbacks when a BLE device is discovered [22]. The code overrides the ScanCallback class's onScanResult method. When a BLE device is discovered, this function is invoked. The callbackType and result arguments are required by the method. The callbackType parameter is an integer representing the kind of callback (e.g., SCAN FAILED ALREADY STARTED), and the result parameter is an instance of the ScanResult class containing information about the discovered device. The if-condition then checks if a device has been detected, and Log statements are used to view results for debugging and performing tests.

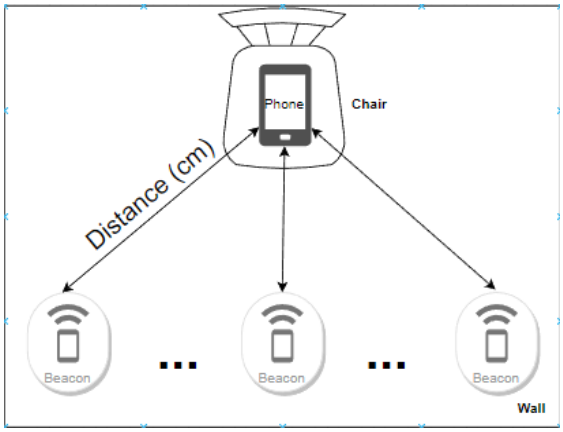
4.3.2 Beacon Configuration and Distance Estimation

Using the boilerplate scanner code in Section 4.3.1, multiple experiments were conducted to come up with a suitable calculation for distance estimation in the proximity ranging function for indoor localisation. Before using the literature method described in Section 2.1.1.1, the initial approach was to derive a custom relationship between RSSI value and Distance from the beacon (in centimetres).

4.3.2.1 Experiment 1: Base Case

Figure 4-5 illustrates the setup for the base case –

Figure 4-5: Experiment 1: Setup



To calculate the relationship between distance and RSSI value, keeping distance constant, the RSSI value was noted at constant time intervals. This process was repeated for 5 trials. The angle of the beacons placed from the phone was also changed for the same distance to observe any changes in the RSSI values scanned. The results have been summarised in the tables below –

Figure 4-6: Experiment 1: Scan Results

Distance	600cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-91	-95	-92	-96	-96
30	-91	-95	-94	-98	-95
60	-92	-92	-93	-98	-98
90	-95	-93	-95	-98	-96
120	-92	-93	-95	-104	-98
150	-98	-95	-95	-106	-104
180	-95	-95	-95	-98	-110
Average	-93.42857143	-94	-94.14285714	-99.71428571	-99.57142857

Final Avg	-96.17142857

Distance	550cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-91	-90	-91	-91	-91
30	-91	-90	-90	-91	-92
60	-90	-90	-90	-91	-91
90	-91	-90	-92	-91	-91
120	-90	-91	-90	-90	-91
150	-90	-91	-93	-91	-90
180	-91	-91	-93	-90	-90
Average	-90.57142857	-90.42857143	-91.28571429	-90.71428571	-90.85714286

Final Avg	-90.77142857

Distance	500cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-89	-92	-91	-90	-90
30	-87	-89	-89	-90	-87
60	-87	-90	-85	-90	-90
90	-87	-89	-87	-90	-90
120	-89	-87	-89	-90	-90
150	-89	-85	-91	-89	-90
180	-87	-89	-85	-89	-90
Average	-87.85714286	-88.71428571	-88.14285714	-89.71428571	-89.57142857

Final Avg	-88.8

Distance	450cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-83	-82	-83	-83	-84
30	-88	-87	-82	-83	-87
60	-82	-82	-83	-82	-89
90	-87	-82	-87	-82	-84
120	-87	-87	-88	-88	-83
150	-82	-85	-82	-83	-88
180	-88	-82	-82	-82	-90
Average	-85.28571429	-83.85714286	-83.85714286	-83.28571429	-86.42857143

Final Avg	-84.54285714

Distance	400cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-81	-81	-81	-81	-87
30	-85	-84	-81	-79	-85
60	-84	-84	-85	-79	-82
90	-81	-81	-85	-84	-82
120	-81	-82	-83	-80	-81
150	-82	-82	-79	-80	-81
180	-81	-82	-79	-80	-81
Average	-82.14285714	-82.28571429	-81.85714286	-80.42857143	-82.71428571

Final Avg	-81.88571429

Distance	350cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-83	-78	-76	-83	-81
30	-85	-78	-76	-78	-81
60	-85	-78	-76	-79	-78
90	-84	-80	-81	-76	-78
120	-84	-82	-77	-77	-78
150	-80	-82	-77	-78	-78
180	-82	-78	-77	-78	-78
Average	-83.28571429	-79.42857143	-77.14285714	-78.42857143	-78.85714286

Final Avg	-79.42857143

Distance	300cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-78	-78	-81	-83	-77
30	-80	-78	-77	-77	-77
60	-77	-78	-76	-77	-78
90	-75	-79	-81	-82	-78
120	-80	-77	-77	-77	-78
150	-80	-77	-88	-84	-78
180	-79	-78	-77	-78	-84
Average	-78.42857143	-77.85714286	-79.57142857	-79.71428571	-78.57142857

Final Avg	-78.82857143

Distance	250cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-78	-78	-79	-78	-77
30	-80	-78	-79	-77	-77
60	-78	-78	-76	-77	-78
90	-78	-82	-79	-77	-78
120	-78	-82	-78	-77	-78
150	-78	-82	-78	-78	-78
180	-78	-82	-77	-78	-77
Average	-78.28571429	-80.28571429	-78	-77.42857143	-77.57142857

Final Avg	-78.31428571

Distance	200cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-78	-78	-75	-84	-77
30	-78	-78	-79	-77	-77
60	-78	-78	-76	-77	-79
90	-76	-79	-79	-77	-78
120	-76	-76	-78	-77	-78
150	-76	-76	-78	-81	-78
180	-79	-76	-77	-82	-77
Average	-77.28571429	-77.28571429	-77.42857143	-79.28571429	-77.71428571

Final Avg	-77.8

Distance	150cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-74	-79	-79	-74	-76
30	-76	-74	-74	-75	-76
60	-79	-78	-74	-75	-76
90	-79	-79	-77	-76	-83
120	-75	-77	-76	-76	-83
150	-73	-73	-74	-74	-76
180	-78	-75	-74	-74	-76
Average	-76.28571429	-76.42857143	-75.42857143	-74.85714286	-78

Final Avg	-76.2

Distance	100cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-77	-77	-78	-74	-76
30	-77	-77	-74	-75	-76
60	-75	-78	-74	-75	-75
90	-77	-77	-77	-75	-79
120	-76	-76	-78	-76	-77
150	-76	-73	-74	-74	-75
180	-76	-78	-74	-74	-76
Average	-76.28571429	-76.57142857	-75.57142857	-74.71428571	-76.28571429

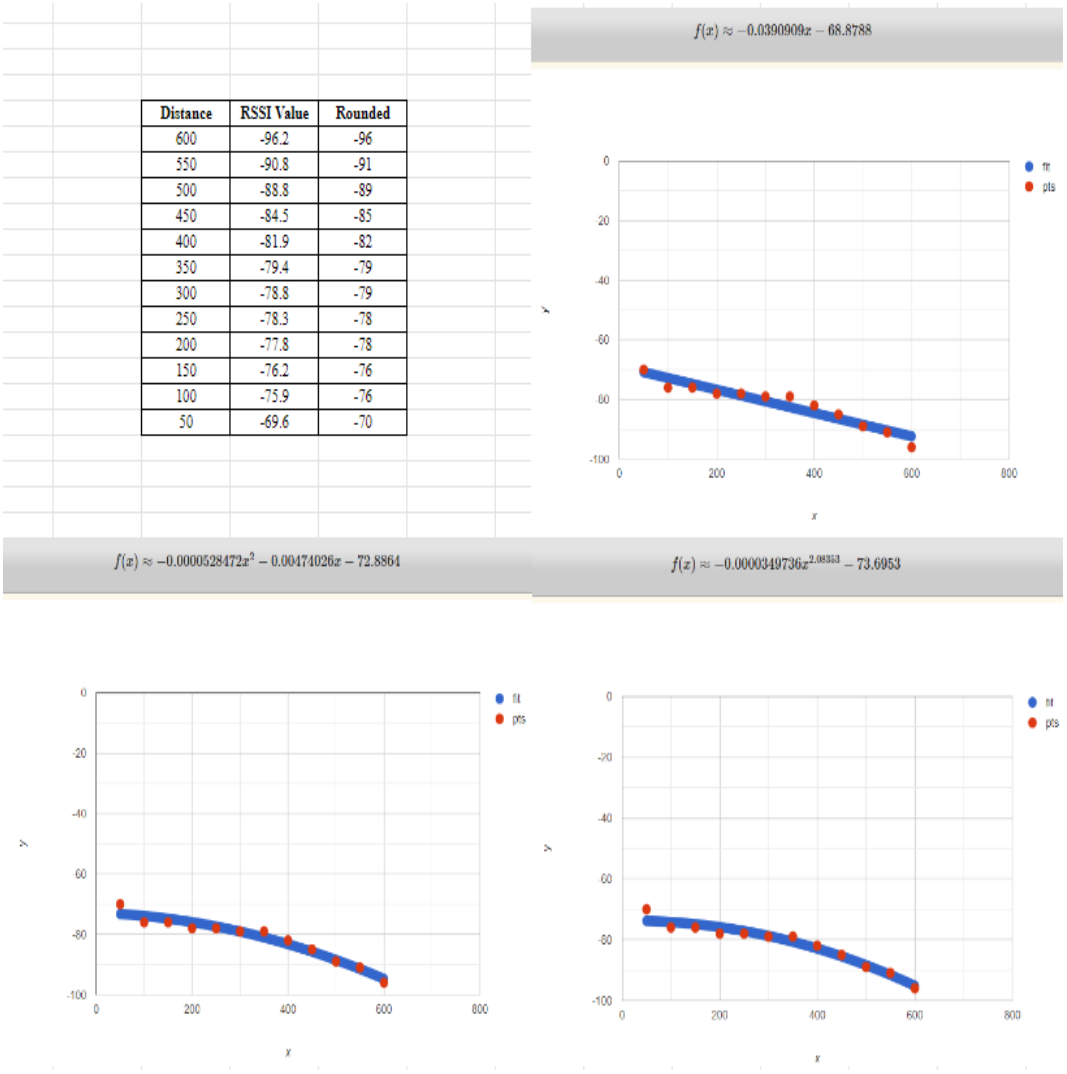
Final Avg	-75.88571429

Distance	50cm				
Time (Seconds)	RSSI Values				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-75	-73	-71	-67	-69
30	-73	-71	-71	-69	-68
60	-65	-65	-70	-69	-68
90	-65	-73	-70	-69	-69
120	-74	-71	-67	-68	-68
150	-72	-78	-67	-68	-69
180	-65	-76	-67	-68	-69
Average	-69.85714286	-72.42857143	-69	-68.28571429	-68.57142857

Final Avg	-69.62857143

The final average values from this experiment were then taken (rounded up to the nearest whole number) and multiple equations were derived to estimate the relationship between distance and RSSI values (Linear, Quadratic, Root) [23]. The results from this have been illustrated below –

Figure 4-7: Experiment 1: Final Results



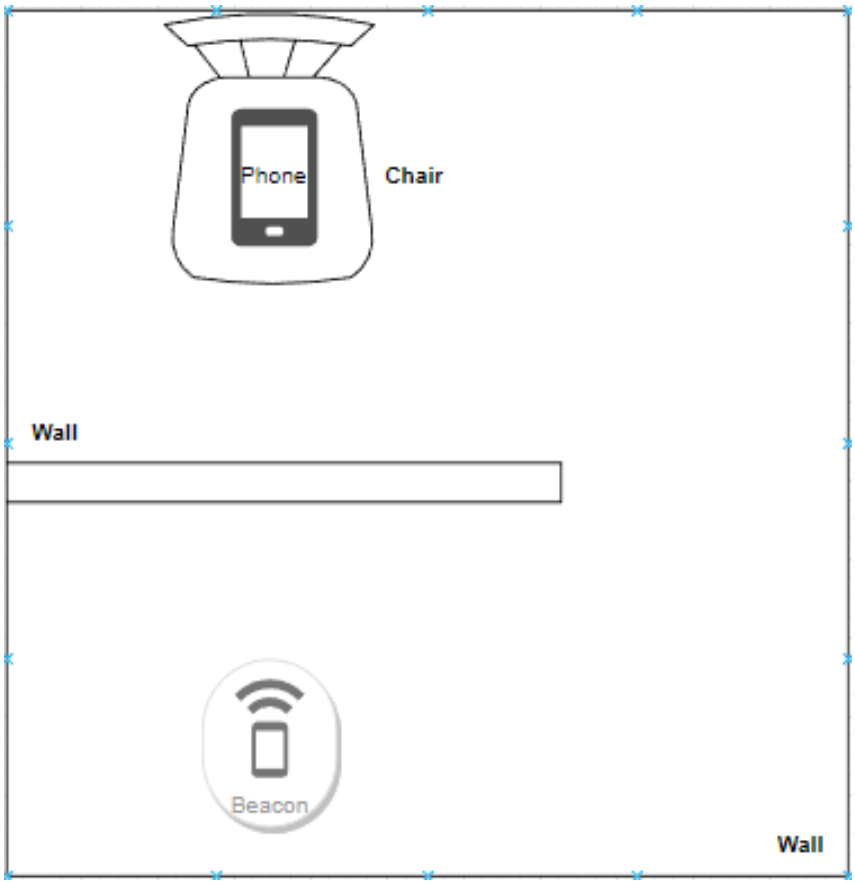
Distance	RSSI Value	Rounded
600	-96.2	-96
550	-90.8	-91
500	-88.8	-89
450	-84.5	-85
400	-81.9	-82
350	-79.4	-79
300	-78.8	-79
250	-78.3	-78
200	-77.8	-78
150	-76.2	-76
100	-75.9	-76
50	-69.6	-70

Despite doing multiple trials and averaging across multiple observed values, there was still a significant difference in the accuracy of the distance estimated using this method (for all three derived curves). The distance estimated was only accurate for a specific range of values (for distances less than 200cm). The next step was to conduct another experiment to further analyse the results and try to improve the derived equations.

4.3.2.2 Experiment 2: Base Case + Obstacle (Wall)

The setup in this experiment was similar to that in Experiment 1, however, an additional obstacle (a wall) was introduced in the experiment area. This was to cater to a more realistic scenario, where the beacons and scanner would be interacting with each other with multiple obstacles, the main one being walls. *Figure 4-8* illustrates the setup –

Figure 4-8: Experiment 2: Setup



The scan results and final calculations were conducted in the same manner as Experiment 1. The introduction of the obstacle did bring variations in the derived equations, however, upon testing, the same problem as in Experiment 1 was encountered. For this experiment, the estimated distances were only accurate up to 165cm – still not ideal for the study's use case.

4.3.2.3 Experiment 3: Literature Method

The setup for this experiment was the same as that of Experiment 1. However, the RSSI values received from the BLE beacon at set distances were measured to test the literature method for distance calculation. These RSSI values were substituted into the equation for the literature method and the calculated values were compared to the true physical distance.

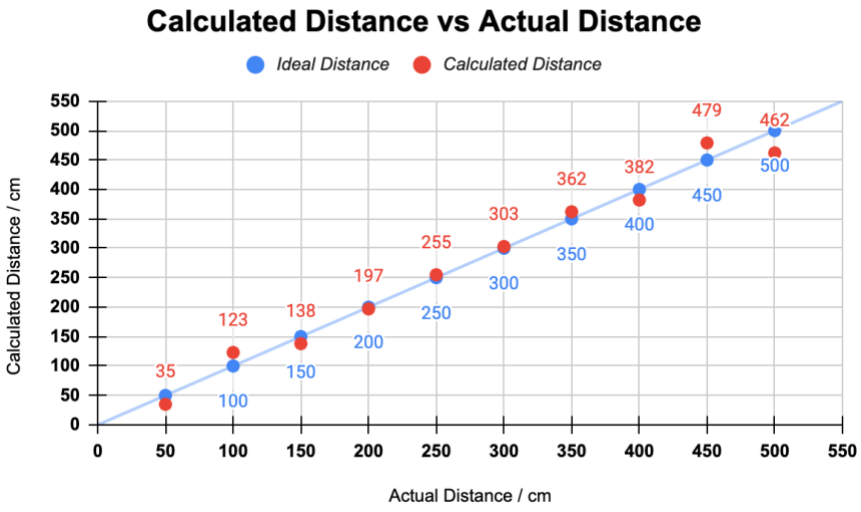
The equation below illustrates the relationship between distance and RSSI as described by the literature method –

$$d = 10^{\frac{Tx-RSSI}{10n}}$$

Here, *d* is the distance calculated in meters, T_x is the power of the transmitted signal in dBm (decibel-milliwatts), and *n* is the path loss exponent, which is a constant that depends on the environment and typically ranges between 2 and 4. This approximation assumes that the RSSI drops exponentially with distance – the higher the value of RSSI, the smaller the distance between the transmitter (BLE beacon) and the receiver (*VirtualEYE*).

The value of T_x was obtained from the provided hardware specification document. After several trials, an optimum value of 2.7 was determined for *n* as well. *Figure 4-9* below illustrates the results of this approach –

Figure 4-9: Results of the Literature Method



4.4 Server Implementation

The server for indoor navigation has been implemented using Flask for Python 3.8.5. The implementation has been divided into client-server communication, shortest path calculation, and cloud migration. This section provides details for the same.

4.4.1 Setting up communication

Figure 4-10 illustrates the communication on the server side – how it interprets the data received from the client, process it, and returns the output back to the client –

Figure 4-10: Server – Communicating with the client

```
@app.route('/sendLoc', methods=['GET', 'POST'])
def process1():

    '''
    /sendLoc?startLoc=xxxx&destLoc=xxxx - to be sent from client side
    '''

    # Process the POST data - retrieve the user's instructions
    start_loc = request.args.get('startLoc')
    dest_loc = request.args.get('destLoc')
    print(start_loc)
    print(dest_loc)
    directions = []
    bearings = []

    shortest_path = pathCalc.shortest_path(graph, start_loc, dest_loc)

    for i in range(len(shortest_path)-1):
        directions.append(graph[shortest_path[i]][shortest_path[i+1]])

    bearings = [item.split('/')[1] for item in directions]
    directions = [direction.split('/')[0] for direction in directions]
    shortest_path.pop(0)

    print(shortest_path)
    print(directions)
    print(bearings)

    return jsonify({"path": shortest_path, "directions": directions, "bearings": bearings})
```

This is a Flask application function that handles the */sendLoc* route. It accepts two query parameters, *startLoc*, and *destLoc*, which are supplied as query parameters in a POST request. The code initially uses the *request.args.get()* method to collect the start and destination locations from the request. The *pathCalc.shortest_path*() function is then used to calculate the shortest route between the two locations,

which is supposed to return a list of nodes indicating the shortest path. The function obtains and saves the relevant direction information held in the graph object for each subsequent pair of nodes in the shortest path list. Then it divides each direction information string into two parts: the direction and the bearing, which are then stored in separate lists of directions and bearings. Finally, as a response to the client, the method delivers a JSON object comprising the shortest path, directions, and bearings.

4.4.2 Shortest Path Calculation

Several shortest-path algorithms were tested to calculate the most optimal algorithm. This section goes into the details of the algorithms tested and the final algorithm chosen for shortest-path calculation.

4.4.2.1 Algorithm 1: Breadth-First Search (BFS)

BFS is a graph traversal approach that explores all vertices at the current depth level before continuing further. BFS starts at a root node and explores all neighbouring nodes before moving on to the next level. The figures below illustrate the graph in the server's context, the code implementation, and the results –

Figure 4-11: Server – BFS Algorithm Visualization

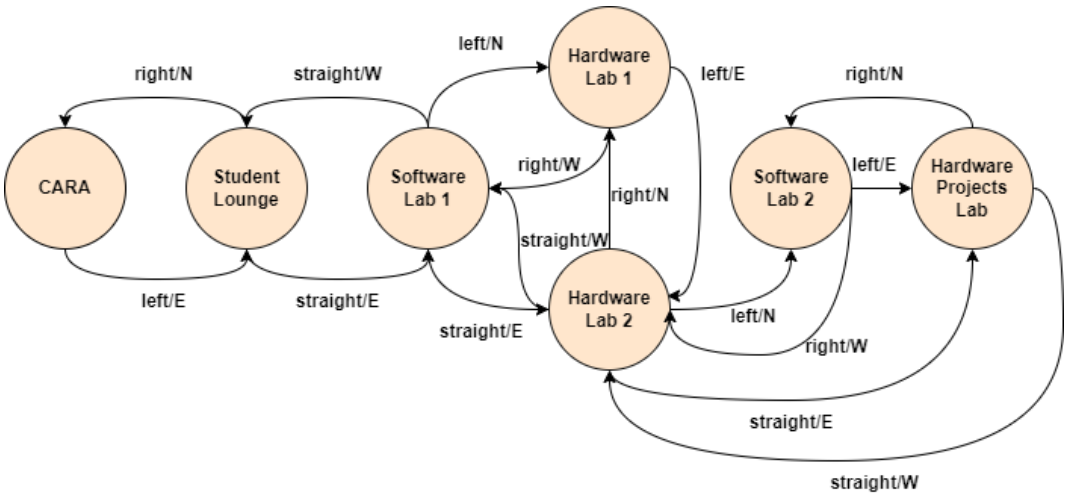


Figure 4-12: Server – BFS Algorithm code implementation

```
def shortest_path(graph, start, end):
  # Keep track of visited nodes
  visited = []
  # Keep track of nodes to be checked using a queue
  queue = [[start]]
  # Loop through the queue
  while queue:
    # Get the first path in the queue
    path = queue.pop(0)
    # Get the last node in the path
    node = path[-1]
    # If the node has not been visited
    if node not in visited:
      # Mark it as visited
      visited.append(node)
      # Add all the neighboring nodes to the queue
      for neighbor in graph[node]:
        new_path = list(path)
        new_path.append(neighbor)
        queue.append(new_path)
        # If the neighbor is the end node, return the path
        if neighbor == end:
          return new_path
  # If no path was found, return None
  return None

for start in graph:
  for end in graph:
    if start != end:

      start_time = time.perf_counter()
      path = shortest_path(graph, start, end)
      end_time = time.perf_counter()
      time_taken = end_time - start_time

      print(f"Shortest path from {start} to {end}: {path}")
      print(f"Time taken to find the shortest path: {time_taken} seconds")
      print(" ")
```

Figure 4-13: BFS Results

Start Location to Destination	Shortest Path Output	Time Taken (Seconds)
cara to lounge	[cara, lounge]	0.000008
cara to sw1	[cara, lounge, sw1]	0.000004
cara to hw1	[cara, lounge, sw1, hw1]	0.000008
cara to hw2	[cara, lounge, sw1, hw2]	0.000007
cara to sw2	[cara, lounge, sw1, hw2, sw2]	0.000012
cara to hw_proj	[cara, lounge, sw1, hw2, hw_proj]	0.000012
lounge to cara	[lounge, cara]	0.000002
lounge to sw1	[lounge, sw1]	0.000002
lounge to hw1	[lounge, sw1, hw1]	0.000004
lounge to hw2	[lounge, sw1, hw2]	0.000005
lounge to sw2	[lounge, sw1, hw2, sw2]	0.000010
lounge to hw_proj	[lounge, sw1, hw2, hw_proj]	0.000008
sw1 to cara	[sw1, lounge, cara]	0.000003
sw1 to lounge	[sw1, lounge]	0.000402
sw1 to hw1	[sw1, hw1]	0.000007
sw1 to hw2	[sw1, hw2]	0.000003
sw1 to sw2	[sw1, hw2, sw2]	0.000008
sw1 to hw_proj	[sw1, hw2, hw_proj]	0.000007
hw1 to cara	[hw1, sw1, lounge, cara]	0.000020
hw1 to lounge	[hw1, sw1, lounge]	0.000003
hw1 to sw1	[hw1, sw1]	0.000001
hw1 to hw2	[hw1, hw2]	0.000002
hw1 to sw2	[hw1, hw2, sw2]	0.000006
hw1 to hw_proj	[hw1, hw2, hw_proj]	0.000006
hw2 to cara	[hw2, sw1, lounge, cara]	0.000011
hw2 to lounge	[hw2, sw1, lounge]	0.000008
hw2 to sw1	[hw2, sw1]	0.000003
hw2 to hw1	[hw2, hw1]	0.000002
hw2 to sw2	[hw2, sw2]	0.000002
hw2 to hw_proj	[hw2, hw_proj]	0.000003
sw2 to cara	[sw2, hw2, sw1, lounge, cara]	0.000063
sw2 to lounge	[sw2, hw2, sw1, lounge]	0.000008
sw2 to sw1	[sw2, hw2, sw1]	0.000005
sw2 to hw1	[sw2, hw2, hw1]	0.000003
sw2 to hw2	[sw2, hw2]	0.000002
sw2 to hw_proj	[sw2, hw_proj]	0.000002
hw_proj to cara	[hw_proj, hw2, sw1, lounge, cara]	0.000011
hw_proj to lounge	[hw_proj, hw2, sw1, lounge]	0.000009
hw_proj to sw1	[hw_proj, hw2, sw1]	0.000005
hw_proj to hw1	[hw_proj, hw2, hw1]	0.000004
hw_proj to hw2	[hw_proj, hw2]	0.000002
hw_proj to sw2	[hw_proj, sw2]	0.000002

4.4.2.2 Algorithm 2: Depth-First Search (DFS)

In DFS, the algorithm starts at the root node and travels as far as feasible down each branch before returning to the root node. This process is repeated until all nodes have been visited or the required node has been located. The figures below illustrate the graph in the server's context, the code implementation, and the results –

Figure 4-14: Server – DFS Algorithm Visualization

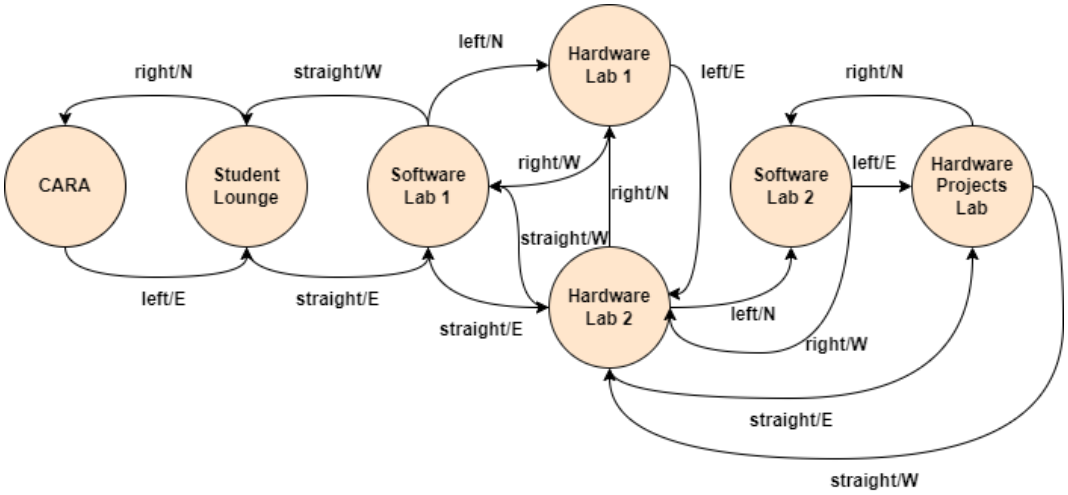


Figure 4-15: Server – DFS Algorithm code implementation

```
def shortest_path(graph, start, end):
  # Keep track of visited nodes
  visited = []
  # Keep track of nodes to be checked using a stack
  stack = [[start]]
  # Loop through the stack
  while stack:
    # Get the last path in the stack
    path = stack.pop()
    # Get the last node in the path
    node = path[-1]
    # If the node has not been visited
    if node not in visited:
      # Mark it as visited
      visited.append(node)
      # Add all the neighboring nodes to the stack
      for neighbor in reversed(graph[node]):
        new_path = list(path)
        new_path.append(neighbor)
        stack.append(new_path)
        # If the neighbor is the end node, return the path
        if neighbor == end:
          return new_path
  # If no path was found, return None
  return None

for start in graph:
  for end in graph:
    if start != end:

      start_time = time.perf_counter()
      path = shortest_path(graph, start, end)
      end_time = time.perf_counter()
      time_taken = end_time - start_time

      print(f"Shortest path from {start} to {end}: {path}")
      print(f"Time taken to find the shortest path: {time_taken} seconds")
      print(" ")
```

Figure 4-16: Server – DFS Results

Start Location to Destination	Shortest Path Output	Time Taken (Seconds)
cara to lounge	[cara, lounge]	0.000010
cara to sw1	[cara, lounge, sw1]	0.000005
cara to hw1	[cara, lounge, sw1, hw1]	0.000010
cara to hw2	[cara, lounge, sw1, hw2]	0.000007
cara to sw2	[cara, lounge, sw1, hw1, hw2, sw2]	0.000014
cara to hw_proj	[cara, lounge, sw1, hw1, hw2, hw_proj]	0.000011
lounge to cara	[lounge, cara]	0.000003
lounge to sw1	[lounge, sw1]	0.000002
lounge to hw1	[lounge, sw1, hw1]	0.000007
lounge to hw2	[lounge, sw1, hw2]	0.000006
lounge to sw2	[lounge, sw1, hw1, hw2, sw2]	0.000011
lounge to hw_proj	[lounge, sw1, hw1, hw2, hw_proj]	0.000009
sw1 to cara	[sw1, lounge, cara]	0.000005
sw1 to lounge	[sw1, lounge]	0.000002
sw1 to hw1	[sw1, hw1]	0.000003
sw1 to hw2	[sw1, hw2]	0.000002
sw1 to sw2	[sw1, hw1, hw2, sw2]	0.000011
sw1 to hw_proj	[sw1, hw1, hw2, hw_proj]	0.000011
hw1 to cara	[hw1, sw1, lounge, cara]	0.000008
hw1 to lounge	[hw1, sw1, lounge]	0.000005
hw1 to sw1	[hw1, sw1]	0.000002
hw1 to hw2	[hw1, hw2]	0.000002
hw1 to sw2	[hw1, sw1, hw2, sw2]	0.000013
hw1 to hw_proj	[hw1, sw1, hw2, hw_proj]	0.000011
hw2 to cara	[hw2, hw1, sw1, lounge, cara]	0.000011
hw2 to lounge	[hw2, hw1, sw1, lounge]	0.000008
hw2 to sw1	[hw2, sw1]	0.000003
hw2 to hw1	[hw2, hw1]	0.000004
hw2 to sw2	[hw2, sw2]	0.000003
hw2 to hw_proj	[hw2, hw_proj]	0.000002
sw2 to cara	[sw2, hw2, hw1, sw1, lounge, cara]	0.000014
sw2 to lounge	[sw2, hw2, hw1, sw1, lounge]	0.000010
sw2 to sw1	[sw2, hw2, sw1]	0.000005
sw2 to hw1	[sw2, hw2, hw1]	0.000006
sw2 to hw2	[sw2, hw2]	0.000003
sw2 to hw_proj	[sw2, hw_proj]	0.000002
hw_proj to cara	[hw_proj, sw2, hw2, hw1, sw1, lounge, cara]	0.000015
hw_proj to lounge	[hw_proj, sw2, hw2, hw1, sw1, lounge]	0.000013
hw_proj to sw1	[hw_proj, sw2, hw2, sw1]	0.000005
hw_proj to hw1	[hw_proj, sw2, hw2, hw1]	0.000006
hw_proj to hw2	[hw_proj, hw2]	0.000002
hw_proj to sw2	[hw_proj, sw2]	0.000002

4.4.2.3 Algorithm 3: Dijkstra's Algorithm

Dijkstra's method is a weighted graph search technique that discovers the shortest path between two nodes. It keeps track of unvisited nodes and their distances from the source node using a priority queue. It picks the node with the shortest distance and updates the distances of its neighbours at each step. The method is repeated until all nodes have been visited or the target node has been reached.

Figure 4-17: Server – Dijkstra's Algorithm Visualization

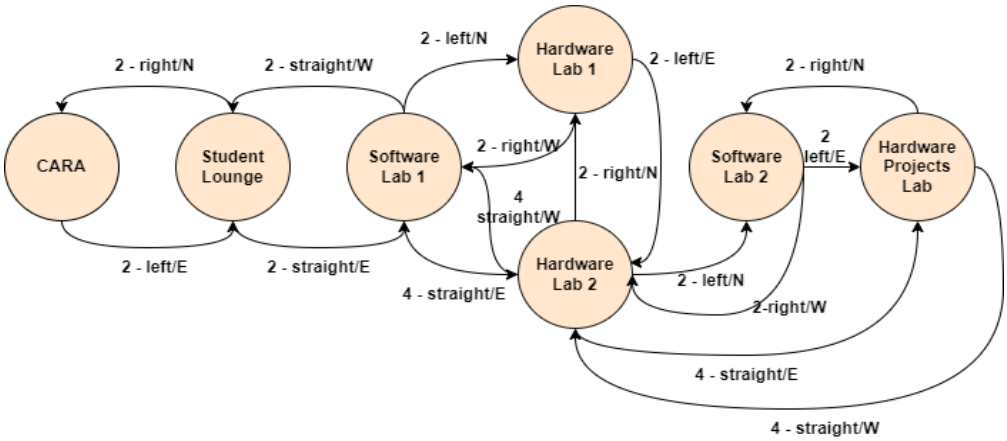


Figure 4-18: Server – Dijkstra's Algorithm code implementation

```python
def shortest_path(graph, start, end):
  # Keep track of the distances from the start node
  distances = {node: float("inf") for node in graph}
  distances[start] = 0
  # Keep track of the previous node for each node
  previous = {node: None for node in graph}
  # Create a priority queue to keep track of the nodes to visit
  queue = [(0, start)]
  while queue:
    # Get the node with the shortest distance from the start
    (distance, node) = heapq.heappop(queue)
    # If the node is the end node, return the path
    if node == end:
      path = []
      while node is not None:
        path.append(node)
        node = previous[node]
      return path[::-1]
    # Update the distances for all the neighbors of the node
    for neighbor in graph[node]:
      new_distance = distance + graph[node][neighbor]["weight"]
      if new_distance < distances[neighbor]:
        distances[neighbor] = new_distance
        previous[neighbor] = node
        heapq.heappush(queue, (new_distance, neighbor))

for start in graph:
  for end in graph:
    if start != end:

      start_time = time.perf_counter()
      path = shortest_path(graph, start, end)
      end_time = time.perf_counter()
      time_taken = end_time - start_time

      print(f"Shortest path from {start} to {end}: {path}")
      print(f"Time taken to find the shortest path: {time_taken} seconds")
      print(" ")
```

Figure 4-19: Dijkstra's Algorithm Results

Start Location to Destination	Shortest Path Output	Time Taken (Seconds)
cara to student lounge	[cara, student lounge]	0.000016
cara to software lab 1	[cara, student lounge, software lab 1]	0.000008
cara to hardware lab 1	[cara, student lounge, software lab 1, hardware lab 1]	0.000010
cara to hardware lab 2	[cara, student lounge, software lab 1, hardware lab 2]	0.000009
cara to software lab 2	[cara, student lounge, software lab 1, hardware lab 2, software lab 2]	0.000012
cara to hardware projects lab	[cara, student lounge, software lab 1, hardware lab 2, hardware projects lab]	0.000012
student lounge to cara	[student lounge, cara]	0.000007
student lounge to software lab 1	[student lounge, software lab 1]	0.000005
student lounge to hardware lab 1	[student lounge, software lab 1, hardware lab 1]	0.000007
student lounge to hardware lab 2	[student lounge, software lab 1, hardware lab 2]	0.000008
student lounge to software lab 2	[student lounge, software lab 1, hardware lab 2, software lab 2]	0.000010
student lounge to hardware projects lab	[student lounge, software lab 1, hardware lab 2, hardware projects lab]	0.000010
software lab 1 to cara	[software lab 1, student lounge, cara]	0.000008
software lab 1 to student lounge	[software lab 1, student lounge]	0.000006
software lab 1 to hardware lab 1	[software lab 1, hardware lab 1]	0.000006
software lab 1 to hardware lab 2	[software lab 1, hardware lab 2]	0.000008
software lab 1 to software lab 2	[software lab 1, hardware lab 2, software lab 2]	0.000010
software lab 1 to hardware projects lab	[software lab 1, hardware lab 2, hardware projects lab]	0.000010
hardware lab 1 to cara	[hardware lab 1, software lab 1, student lounge, cara]	0.000011
hardware lab 1 to student lounge	[hardware lab 1, software lab 1, student lounge]	0.000009
hardware lab 1 to software lab 1	[hardware lab 1, software lab 1]	0.000007
hardware lab 1 to hardware lab 2	[hardware lab 1, hardware lab 2]	0.000005
hardware lab 1 to software lab 2	[hardware lab 1, hardware lab 2, software lab 2]	0.000007
hardware lab 1 to hardware projects lab	[hardware lab 1, hardware lab 2, hardware projects lab]	0.000026
hardware lab 2 to cara	[hardware lab 2, software lab 1, student lounge, cara]	0.000011
hardware lab 2 to student lounge	[hardware lab 2, software lab 1, student lounge]	0.000009
hardware lab 2 to software lab 1	[hardware lab 2, software lab 1]	0.000008
hardware lab 2 to hardware lab 1	[hardware lab 2, hardware lab 1]	0.000006
hardware lab 2 to software lab 2	[hardware lab 2, software lab 2]	0.000007
hardware lab 2 to hardware projects lab	[hardware lab 2, hardware projects lab]	0.000007
software lab 2 to cara	[software lab 2, hardware lab 2, software lab 1, student lounge, cara]	0.000011
software lab 2 to student lounge	[software lab 2, hardware lab 2, software lab 1, student lounge]	0.000010
software lab 2 to software lab 1	[software lab 2, hardware lab 2, software lab 1]	0.000008
software lab 2 to hardware lab 1	[software lab 2, hardware lab 2, hardware lab 1]	0.000007
software lab 2 to hardware lab 2	[software lab 2, hardware lab 2]	0.000005
software lab 2 to hardware projects lab	[software lab 2, hardware projects lab]	0.000005
hardware projects lab to cara	[hardware projects lab, hardware lab 2, software lab 1, student lounge, cara]	0.000010
hardware projects lab to student lounge	[hardware projects lab, hardware lab 2, software lab 1, student lounge]	0.000010
hardware projects lab to software lab 1	[hardware projects lab, hardware lab 2, software lab 1]	0.000008
hardware projects lab to hardware lab 1	[hardware projects lab, hardware lab 2, hardware lab 1]	0.000007
hardware projects lab to hardware lab 2	[hardware projects lab, hardware lab 2]	0.000005
hardware projects lab to software lab 2	[hardware projects lab, software lab 2]	0.000005

From the results above, the BFS algorithm was the most optimal for the use case.

4.4.3 Cloud Migration

Cloud migration was performed for enhanced performance and easier connectivity to the server via the Internet. This was done using Docker, wherein a Dockerfile was written to convert the Flask application to a Docker Image, which was compressed, uploaded to GitHub, and extracted on the Azure Linux VM using Docker. With this, the client and server do not need to be connected to the same network. *Figure 4-20* illustrates the Dockerfile -

Figure 4-20: Dockerfile for Flask Application

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

RUN mkdir -p /server
RUN chown root /server
USER root
WORKDIR /server

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

4.5 Client Implementation

The client-side mobile application, *VirtualEYE*, has been implemented in Kotlin using Android Studio. The following sections provide details of how different Kotlin methods have been implemented.

4.5.1 TTS Controller and 'Shake Input'

The TTS controller and Sensor Manager from *Figure 3-4* are initialized as illustrated below –

Figure 4-21: TTS Controller and Sensor Manager Initialization

```
tts = TextToSpeech( context: this, TextToSpeech.OnInitListener {  it: Int
    if (it == TextToSpeech.SUCCESS) {
        tts.setSpeechRate(0.85f)
        tts.speak( text: "Hello, welcome to Virtual Eye! Please shake the phone to
    }
})

// Init Sensor Manager
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
```

The *SensorManager* on the Android smartphone is used to control the accelerometer sensor (for shake input), and TTS is utilized to provide voice output for the user. When the accelerometer sensor changes, the *onSensorChanged* method is invoked. It calculates the acceleration using the sensor's x, y, and z readings and checks to see if the acceleration is greater than 2. This has been depicted in *Figure 4-22*.

Figure 4-22: Shake Detection

```
override fun onSensorChanged(event: SensorEvent?) {
    if (event?.sensor?.type == Sensor.TYPE_ACCELEROMETER) {
        val x = event.values[0]
        val y = event.values[1]
        val z = event.values[2]

        val acceleration = (x * x + y * y + z * z) /
                (SensorManager.GRAVITY_EARTH * SensorManager.GRAVITY_EARTH)

        //  If the acceleration is greater than 2, it indicates that the phone has been shaken
        if (acceleration > 2) {
            // Move to the assisted navigation screen here
            val intent = Intent( packageContext: this, AssistedNavigation::class.java)
            tts.shutdown()
            startActivity(intent)
        }
    }
}
```

4.5.2 HTTP Requests

Figure 4-23 is a Kotlin coroutine that sends an HTTP POST request to the server's IP. It has two parameters: *startLoc* and *destLoc*, which are passed as query parameters in the request. The request is performed asynchronously in the IO Dispatcher, which is optimized for offloading blocking I/O operations to a different thread. The response body is read as a JSONObject and returned as the result of the coroutine. This JSONObject is then parsed to retrieve the path, directions, bearings and distances.

Figure 4-23: HTTP Requests

```
private suspend fun sendRequest(startLoc: String, destLoc: String): JSONObject {
    return withContext(Dispatchers.IO) {
        val request = URL("http://192.168.1.68:5000/sendLoc?startLoc=$startLoc&destLoc=$destLoc").openConnection() as HttpURLConnection
        request.requestMethod = "POST"
        JSONObject(request.inputStream.use { it.reader().use { reader -> reader.readText() } } )
    }
}
```

```
GlobalScope.launch {  this: CoroutineScope
    Log.i( tag: "Check",  msg: "HERE")
    val respJSON = sendRequest(startPoint, destLoc)
    // Update the UI or do something with the response here
    Log.i( tag: "RESP", respJSON.toString())

    val pathTemp = respJSON.getJSONArray( name: "path")
    for (i in 0 ≤ until  < pathTemp.length()) {
        Globals.path.add(pathTemp.get(i) as String)
    }

    val directionsTemp = respJSON.getJSONArray( name: "directions")
    for (i in 0 ≤ until  < directionsTemp.length()) {
        Globals.directions.add(directionsTemp.get(i) as String)
    }
```

4.5.3 Voice Input

The code snippet in *Figure 4-24* creates a speech recognizer. This is setup using the *SpeechRecognizer* class and a *RecognitionListener* is also set to handle the various callbacks for the recognition process – done by *createSpeechRecognizer*(). The *createIntent*() function creates an Intent object for speech recognition with language

parameters. The *handleSpeechBegin*() function starts an audio session by calling *startListening*() on the speech recognizer, passing it the intent created in the *createIntent*() function. The *handleSpeechEnd*() function ends the audio session by calling *cancel*() on the speech recognizer, and sets the listening status to false. Lastly, the *handleCommand*() function is called to handle the command provided by the user's speech.

Figure 4-24: Voice Input

4.5.4 Compass

The code snippet in Figure 4-25 uses Android sensors for bearing calculation. The function *onSensorChanged*() is called every time the data from a sensor changes. The function is checking for changes in either the accelerometer or magnetometer sensors and processing the data to determine the device's orientation. The first part of the function calls the *lowpass()* function on the sensor data to filter out any noise.

The second part of the function uses the filtered data from both sensors to calculate the device's orientation using the *SensorManager.getRotationMatrix()* and *SensorManager.getOrientation()* functions. The calculated orientation is converted to degrees to determine the device's orientation in relation to the four cardinal directions.

Figure 4-25: Compass Directions Calculation

4.5.5 Object Detection

The code snippet in *Figure 4-26* sets up camera preview and image analysis for object detection. The *bindPreview* function takes a *ProcessCameraProvider* as an input and sets up the camera preview using the Preview builder. The *CameraSelector* is used to specify the back-facing camera. The *setSurfaceProvider* method is called on the Preview instance to display the camera feed on the *previewView* widget in the user interface.

An *ImageAnalysis* instance is created to analyse each frame of the camera feed, target resolution set to *Size*(1280,720). An object detector is then used to detect objects in each frame. The *addOnSuccessListener* method is called on the detector's result, and a loop is used to draw a rectangle around each detected object and its label is passed to the TTS engine for speech output. The TTS output is only triggered if the object label is different from the previous frame.

Figure 4-26: Object Detection

CHAPTER 5: TESTING AND EVALUATION

5.1 Controlled Variables

The table below depicts the variables kept constant across all the test cases –

Table 5-1: Controlled Variables

Variable	Details
Location	SCSE Level 1
Participant	Visually-abled, SCSE student – The participant was blindfolded (with consent) for test cases 3 to 6, and was provided with a walking cane for the same
Obstacles	-
Path for Navigation	CARA to Hardware Lab 1

5.2 Test Case 1: Visual Navigation without the system

5.2.1 Setup

Figure 5-1: Test Case 1: Setup

5.2.2 Results and Participant Feedback

5.3 Test Case 2: Visual Navigation with the system

5.3.1 Setup

Figure 5-2: Test Case 2: Setup

5.3.2 Results and Participant Feedback

5.4 Test Case 3: Non-visual Free Roaming without the system

5.4.1 Setup

Figure 5-3: Test Case 3: Setup

5.4.2 Results and Participant Feedback

5.5 Test Case 4: Non-visual Free Roaming with the system

5.5.1 Setup

Figure 5-4: Test Case 4: Setup

5.5.2 Results and Participant Feedback

5.6 Test Case 5: Non-visual Navigation without the system

5.6.1 Setup

Figure 5-5: Test Case 5: Setup

5.6.2 Results and Participant Feedback

5.7 Test Case 6: Non-visual Navigation with the system

5.7.1 Setup

Figure 5-6: Test Case 6: Setup

5.7.2 Results and Participant Feedback

CHAPTER 6: CONCLUSION

CHAPTER 7: FUTURE WORK

This chapter discusses the possible future pursuits for the project.

7.1 Wearable Technology Integration

Employing wearable technology (smartwatch) can reduce the load from the mobile application and provide more feedback to the visually impaired user. This can enable the navigation system to use additional sensor data from the wearable technology to help localise and provide useful information about the user's surroundings. It will also reduce the need to hold the phone during navigation.

7.2 Computer Vision Based Navigation

Visual Positioning System (VPS) is a rapidly evolving technology in the field of computer vision and artificial intelligence. This method of localisation can be employed with BLE beacons for more accurate results for the user. Training models on environment images can be used to create a co-ordinate system for path planning algorithms and hence navigation.

7.3 Multi-floor Navigation

The current version of this system only accounts for single floor navigation. Employing the use of barometer sensors available in smartphones can enable the system to identify which floor the user is on, and hence navigate the user across different floors in a building.

7.4 Augmented Reality (AR) Navigation

This feature is applicable for vision based navigation in *VirtualEYE*. Incorporating Augmented Reality can technology provide a more seamless navigational

experience. Waypoints and important information can be provided to the user in a dynamic manner.

7.5 Proximity Estimation in Obstacle Detection

VirtualEYE's obstacle recognition capabilities are limited to detection. This can be extended to estimate distances to the obstacle, and provide the distance and feedback to the visually impaired user. The simplest method to achieve this could be to employ additional ultrasonic sensors on walking canes to estimate distances to the obstacles.

REFERENCES

[1] T. Becker, C. Nagel, and T. H. Kolbe, "Supporting contexts for indoor navigation using a multilayered space model," 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware, 2009.

[2] A. Satan and Z. Toth, "Development of Bluetooth based indoor positioning application," 2018 IEEE International Conference on Future IoT Technologies (Future IoT), 2018.

[3] F. Zafari, I. Papapanagiotou, and K. Christidis, "Micro-location for internet of things equipped smart buildings," IEEE Internet of Things Journal, vol. 3, no. 1, pp. 96–112, 2016.

[4] J. Hvizdos, J. Vascak, and A. Brezina, "Object identification and localization by smart floors," 2015 IEEE 19th International Conference on Intelligent Engineering Systems (INES), 2015.

[5] D. Pascolini and S. P. Mariotti, "Global Estimates of Visual Impairment: 2010," British Journal of Ophthalmology, vol. 96, no. 5, pp. 614–618, 2011.

[6] Dardari D., Closas P., Djuric P.M. Indoor tracking: Theory, methods, and technologies. IEEE Trans. Veh. Technol. 2015;64:1263–1278. doi: 10.1109/TVT.2015.2403868.

[7] Navarro A., Sospedra J., Montoliu R., Conesa J., Berkvens R., Caso G., Costa C., Dorigatti N., Hernández N., Knauth S., et al. Geographical and Fingerprinting Data to Create Systems for Indoor Positioning and Indoor/Outdoor Navigation. Challenges, Experiences and Technology Roadmap Intelligent. Data Centric Syst. 2019;2019:1–20. doi: 10.1016/B978-0-12-813189-3.00001-0.

[8] Alarifi A., Al-Salman A., Alsaleh M., Alnafessah A., Al-Hadhrami S., Al-Ammar M.A., Al-Khalifa H.S. Ultra-Wideband Indoor Positioning Technologies: Analysis and Recent Advances. Sensors. 2016;2016:707. doi: 10.3390/s16050707.

[9] Simões WCSS, Machado GS, Sales AMA, de Lucena MM, Jazdi N, de Lucena VF Jr. A Review of Technologies and Techniques for Indoor Navigation

Systems for the Visually Impaired. Sensors (Basel). 2020 Jul 15;20(14):3935. doi: 10.3390/s20143935. PMID: 32679720; PMCID: PMC7411868.

[10] Wang Y., Yang Q., Zhang G., Zhang P. Indoor positioning system using Euclidean distance correction algorithm with Bluetooth low energy beacon; Proceedings of the International Conference on Internet of Things and Applications (IOTA); Las Vegas, NV, USA. 22–24 January 2016; pp. 243–247.

[11] Palumbo F., Barsocchi P., Chessa S., Augusto J.C. A stigmergic approach to indoor localization using Bluetooth Low Energy beacons; Proceedings of the 12th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS); Karlsruhe, Germany. 25–28 August 2019; pp. 1–6

[12] Zanella A. Best Practice in RSS Measurements and Ranging. IEEE Commun. Surv. Tutor. 2016;18:2662–2686. doi: 10.1109/COMST.2016.2553452.

[13] Zhao Y., Fritsche C., Yin F., Gunnarsson F., Gustafsson F. Sequential Monte Carlo Methods and Theoretical Bounds for Proximity Report Based Indoor Positioning. IEEE Trans. Veh. Technol. 2018;67:5372–5386. doi: 10.1109/TVT.2018.2799174.

[14] Munoz Diaz E. Inertial pocket navigation system: Unaided 3D positioning. Sensors. 2015;15:9156–9178. doi: 10.3390/s150409156.

[15] Fraundorfer F., Scaramuzza D. Visual odometry: Part II: Matching, robustness, optimization, and applications. IEEE Robotics Autom. Mag. 2012;19:78–90. doi: 10.1109/MRA.2012.2182810.

[16] Zhao Y., Xu J., Wu J., Hao J., Qian H. Enhancing Camera-Based Multimodal Indoor Localization With Device-Free Movement Measurement Using WiFi. IEEE Internet Things J. 2020;7:1024–1038. doi: 10.1109/JIOT.2019.2948605.

[17] Ji, M., Kim, J., Jeon, J., & Cho, Y. (2015). Analysis of positioning accuracy corresponding to the number of BLE beacons in indoor positioning system. 2015 17th International Conference on Advanced Communication Technology (ICACT). doi:10.1109/icact.2015.7224764

[18] https://microchipdeveloper.com/wireless:ble-link-layer-discovery

[19] https://maps.ntu.edu.sg/#/ntu/d386ffa80e4e46f286d17f08/poi/search

[20] http://davemorrissey.github.io/subsampling-scale-image-view/javadoc/

[21] Enter link to tech spec

[22] https://developer.android.com/reference/android/bluetooth/le/ScanCallback

[23] https://www.dcode.fr/function-equation-finder