Attendance Tracking using Bluetooth Low Energy-Enabled Smartphones

by

Conor Maguire

Supervisor: Dr. Hitesh Tewari

Dissertation

Presented to the University of Dublin,
in partial fulfilment of the requirements
for the degree of

Master in Computer Science

Submitted to the University of Dublin, Trinity College, May, 2018

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work, and has not been submitted as an exercise for a degree at this or any other university.

Conor Maguire

Date

Permission to lend

I agree that the Trinity College Library may lend or copy this dissertation upon request.

Conor Maguire

Date

Summary

Low attendance in lectures has become an issue of concern for many colleges and universities. It has been shown that attendance figures have a strong relationship with final grade, with students who have higher attendance generally performing better. Colleges have employed a variety of flawed methods of tracking attendance in an attempt to tackle this problem.

The aim of this dissertation is to develop a system that combats many of these flaws by using students' smartphones and Bluetooth Low Energy (BLE) technology. BLE is a short range communication protocol released by the Bluetooth Special Interest group as a subset of Bluetooth. Its low power consumption, flexible network topology support, and its high rate of adoption among smartphone brands means there is potential to incorporate this technology into such an attendance tracking system.

A prototype attendance tracking system using these elements is presented in this dissertation, which attempts to demonstrate its feasibility in a college setting. The implementation develops an Android application that broadcasts a student's presence using Bluetooth Low Energy. A small device in each lecture room communicates with each smartphone, reading each student's ID number multiple times during the lecture.

The implementation is evaluated in terms of its speed, power consumption, and accuracy. Its speed and power consumption are determined to be within acceptable ranges, however the accuracy of the scanning device is hampered by the hardware chosen for the implementation. The Raspberry Pi Zero W chosen for this purpose resulted in inconsistent detection of students. The initial work in this dissertation is promising and indicates that the chosen system is feasible in a college setting. However, hardware changes and large scale testing in a trial lecture would required before it can be conclusively determined.

Attendance Tracking using Bluetooth Low Energy-Enabled Smartphones

Conor Maguire

University of Dublin, Trinity College, 2018

Supervisor: Dr. Hitesh Tewari

Abstract

Low attendance in lectures has become an issue of concern for many colleges and universities. It has been shown that attendance figures have a strong relationship with final grade, with students who have higher attendance generally performing better. Colleges have employed a variety of flawed methods of tracking attendance in an attempt to tackle this problem.

The aim of this dissertation is to develop a system that combats many of these flaws by using students' smartphones and Bluetooth Low Energy (BLE) technology. BLE is a short range communication protocol released by the Bluetooth Special Interest group as a subset of Bluetooth. Its low power consumption, flexible network topology support, and its high rate of adoption among smartphone brands means there is potential to incorporate this technology into such an attendance tracking system.

A prototype attendance tracking system using these elements is presented in this dissertation, which attempts to demonstrate its feasibility in a college setting. The implementation develops an Android application that broadcasts a student's presence using Bluetooth Low Energy. A small device in each lecture room communicates with each smartphone, reading each student's ID number multiple times during the lecture.

The implementation is evaluated in terms of its speed, power consumption, and accuracy. Its speed and power consumption are determined to be within acceptable ranges, however the accuracy of the scanning device is hampered by the hardware chosen for the implementation. The Raspberry Pi Zero W chosen for this purpose resulted in inconsistent detection of students. The initial work in this dissertation is promising and indicates that the chosen system is feasible in a college setting. However, hardware changes and large scale testing in a trial lecture would required before it can be conclusively determined.

Acknowledgements

I would like to thank Dr. Hitesh Tewari for the initial idea for the topic and for his help and guidance throughout the entirety of the research.

I would also like to thank Dr. Jonathan Dukes for pointing me in the right direction towards the beginning of the project.

I would like to thank the lads who have gone through this college journey with me.

Thanks to my brother Ciaran for proof reading my first draft and offering helpful suggestions.

Finally I would like to express my gratitude to my parents, Nell and Tony, who have supported me throughout my life and encouraged me through these last five years of college.

Contents

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Topic 2

1.3 Objectives 2

1.4 Dissertation Overview 3

2 Background **4**

2.1 Existing Attendance Tracking Approaches 4

2.1.1 Sign-in Sheets 4

2.1.2 Lecture PIN Codes 5

2.1.3 Card Tapping/Scanning 5

2.2 Related Work 6

2.3 Bluetooth Low Energy 7

2.3.1 BLE Protocol Stack 7

2.3.2 Advertising and Connections 10

2.3.3 GAP 10

2.3.4 GATT 11

3 Design **14**

3.1 System Overview 14

3.1.1 Communication 15

3.2 Attendance Tracker 16

3.3 Attendance & Scheduling API 17

3.3.1 Data Models 17

3.4 Mobile Application 18

4 Implementation **20**

4.1 Attendance Tracker 20

4.1.1 Hardware 20

4.1.2 Lecture Scheduling 21

4.1.3 Scanning Process 23

4.2 Attendance & Scheduling API 26

4.2.1 Data Storage 26

4.2.2 API Routes 27

4.2.3 Configuration and Deployment 31

4.3 Mobile Application 31

4.3.1 Main Activity 32

4.3.2 Background Service 33

5 Evaluation & Results **37**

5.1 Experimentation 37

5.1.1 Recording Speed 37

5.1.2 Full System Cycle 39

5.1.3 Advertisement Detection 41

5.1.4 Power Consumption 42

5.2 Security Considerations 43

5.2.1 Secure API Payloads 43

5.2.2 Client Authentication 43

5.2.3 API Denial-of-Service 44

5.2.4 Bluetooth Security 45

5.3 Summary 45

6 Conclusion **47**

6.1 Overview 47

6.2 Future Work 48

6.2.1 Hardware 48

6.2.2 College Integration and Trial 48

6.2.3 Bluetooth 5 49

A Annotated Dockerfile Example **50**

B Android BLE GATT Server Callback **51**

Bibliography **53**

List of Figures

2.1 BLE protocol stack 8
2.2 BLE channel arrangement 11
2.3 GATT Profile Hierarchy 13

3.1 High level overview of the system design 16
3.2 Data model relationships in scheduling API 18

4.1 A Raspberry Pi Zero W 21
4.2 Main Activity of the Android application 32
4.3 The attendance tracking notification displayed by the AdvertiserService 34
4.4 Android application state diagram 36

5.1 Student number read times 38
5.2 Attendance service log 40

List of Tables

4.1 Endpoints for the Scheduling API 29

5.1 Connection times 38

Chapter 1

Introduction

Bluetooth Low Energy (BLE) was introduced by the Bluetooth Special Interest Group (SIG) as a subset of the Bluetooth specification focused on low power consumption. It supports devices broadcasting to other nearby devices and is optimised for transmitting small bursts of data between the two [1]. For these reasons, it has seen particularly strong adoption in contexts such as home automation, fitness, and healthcare, where sensors, "wearables", or other small devices need to transmit data in bursts while maintaining low power consumption.

With the growing market share of BLE-enabled mobile phones, there exists a potential to incorporate this technology into a lecture attendance tracking solution using student's smartphones. This research aims to produce such a system capable of combating the flaws of traditional attendance tracking approaches.

1.1 Motivation

Low attendance figures in lectures are an issue for many colleges and universities. Studies have been conducted into how low attendance can affect academic performance and it has been found that there is a strong positive relationship between a student's attendance and their final grade [2, 3]. Even in the case of lecture materials being made available online, attendance still matters as there is no guarantee that the student will engage with the materials in their own time [4].

Colleges have employed many methods of tracking student attendance such as sign-in sheets, card scanning, or lecture PIN codes. However many of these methods share similar flaws of being slow, cumbersome, difficult to use, or distracting. Most methods

also fail at actually verifying a student's attendance due to the ease at which they can be faked. Furthermore, a smartphone is an item that a student is much less likely to forget, meaning a smartphone-based system could have a higher potential for ease of use and student engagement.

1.2 Research Topic

The topic of this dissertation is the design and implementation of a system for recording student attendance in college lectures using BLE. The goal of the dissertation is to create a prototype system which could be feasibly implemented in a university setting and which could also offer improvements over existing methods of taking attendance.

In the proposed system, students sign into a smart phone application using their college login credentials and open the app at the beginning of the lecture. The app advertises its presence using BLE Advertising. A Raspberry Pi (or similar BLE-enabled device) will be present in the lecture hall and will use BLE scanning to detect this advertisement, connect to the phone via Bluetooth, and read the student number before disconnecting. The Raspberry Pi will do this for each student it picks up while scanning, and will do it twice, for example: ten minutes after the beginning of the lecture, and ten minutes before the end of the lecture. This helps verify that a student was in close proximity of the Raspberry Pi (i.e. they were actually present in the lecture), and also that the student was present for the duration of the lecture and not just outside the lecture hall for the first scan of the tracker.

1.3 Objectives

The primary objective of this research is to create a prototype attendance tracking solution that could feasibly be employed in a college scenario. The proposed system should adhere to a number of properties:

- It should have minimal disruption to the lecture. It should seamlessly proceed in the background so that the lecturer and students may focus on the lecture materials.
- The system should be easy to use by the students. It should require minimal interaction so as not to distract them like passing around a sign-in sheet would.

- The proposed solution, were it to be implemented by a college, should provide the college with accurate attendance data that is reasonably resistant to falsification. It should ideally require more effort to fake one's attendance than to go the lecture.

- It is also important that the proposed solution would be scalable, so that it functions for large lectures and so that a college may roll out the system to all the rooms on their campus.

The desired outcome is to a have a demonstrable implementation of a system that meets these requirements and would improve upon existing sign-in methods.

1.4 Dissertation Overview

Chapter 2 of this dissertation presents discussion of background information and state-of-the-art research relevant to the problem domain. With this underlying context in mind, Chapter 3 details the high-level design of the attendance tracking system, along with its individual components. Chapter 4 discusses the implementation of the proof of concept, describing hardware and software implementations of the various components of the system. An evaluation of the system is provided in Chapter 5, assessing the system in its current implementation, as well as presenting some security considerations. Finally, Chapter 6 summarizes the research and identifies potential areas for future work.

Chapter 2

Background

This chapter reviews background information and state of the art relevant to the chosen topic. Firstly, several existing solutions for attendance tracking are examined and their shortcomings are outlined. Then, a discussion of related research of Bluetooth LE attendance systems is presented. Finally, some technical details of the BLE specification are explained so as to provide relevant context for the rest of the dissertation.

2.1 Existing Attendance Tracking Approaches

Colleges and universities have attempted to tackle the issue of low attendance using a variety of methods. Usually this involves a combination of some grade-based incentive for attending lectures, as well as a method of recording the attendance. None of these existing approaches to attendance tracking are ideal solutions and each suffer from a variety of shortcomings.

2.1.1 Sign-in Sheets

Perhaps the most popular traditional method of taking attendance is a sign-in sheet that is passed around from student to student for them to sign their name on. It is a simple solution, however its accuracy is not verifiable as it can be easily faked by students signing in their fellow classmates. Furthermore, it can be slow and cumbersome in a large class and produces a minor distraction for students and the lecturer.

2.1.2 Lecture PIN Codes

An alternative solution which the author has experienced during their studies is a lecture PIN-based sign-in system. This involves the lecturer creating some web page with a HTML form which the students fill out at the beginning of the lecture. The form contains fields for the students' name and student number, as well as some randomised PIN that the lecture invents and writes on the blackboard at the beginning of the lecture. In practice, this random PIN does little to prove a student was actually present in a lecture as the PIN could be shared around between classmates through methods such as social media. It also disrupts the beginning of the lecture as the students are all focused on signing themselves in and the lecturer must remember to display the URL and PIN for the students.

2.1.3 Card Tapping/Scanning

In the case of Trinity College, student identification cards are equipped with bar codes as well as a MIFARE Classic 4K contactless chip[1]. For this reason, some consideration was initially given to developing an attendance system based on either bar code scanning or Near Field Communication (NFC). This solution would have a device near the entrance of each lecture room which would allow students to tap (with NFC) or scan (with the bar code) their student ID card to record their attendance. The device could be the lecturer's NFC-enabled mobile phone or a small device installed in the lecture room. The device would record the student number of each tapped card and send the results to the college. An advantage of such an approach is that the NFC chips are already integrated into Trinity College's "TCards". The student would also be required to have the student card in their possession. However, it is clear that this solution would not feasibly scale in the case of large sized lectures as it would result in students queueing to tap their card and blocking the lecture hall entrance. This reason, as well as the risks of a student potentially forgetting their card or giving their student card to a fellow student to tap in also means that this approach has few benefits over a traditional sign-in sheet solution.

[1] `https://www.nxp.com/products/identification-and-security/mifare-ics/mifare-classic:MC_41863`

2.2 Related Work

There has been some research into applying Bluetooth Low Energy-based approaches to the issue of student attendance, with varying degrees of success:

A system using BLE beacons was designed and implemented by Dr. Dwight Deugo of Carleton University, Ottawa in order to combat the limitations of their existing card swiping and QR code scanning system [5]. Bluetooth SIG succinctly describes beacons as "small devices, strategically placed throughout a location, that transmit a continuous signal to any mobile device in range" [6]. These Bluetooth-compatible devices allow actions to be triggered automatically or performed by a nearby mobile device. An important distinction regarding beacon-based BLE communication is that the communication is one-way (from the beacon to the mobile device); that is to say it is not bi-directional.

This attendance tracking system consists of a mobile application reading a Universally Unique Identifier (UUID) from the nearby beacon and sending it to a server to record their attendance. However, this process involves some effort on the part of the student. The student must input their personal details and optionally configure the server's IP upon first time use. Then they are required to scan for all beacons in the area and select the correct beacon corresponding to the lecture they are attending. One of this dissertation's desired objectives outlined in Section 1.3 is for the attendance tracking solution to be easy to use and seamless for the students so as to eliminate distraction. This beacon system has too many manual steps of user interaction/configuration to satisfy this goal. Furthermore, it requires the student to have WiFi connectivity as well as BLE in order to sign in.

There also exists some research into a collaborative attendance tracking protocol using BLE [7]. This research designed a protocol which used a mesh network topology between BLE devices to take attendance. A mesh network is a decentralized network where all nodes communicate directly to each other rather than with a central node. This design used micro-controllers which would be embedded in an ID card. Some promising results were shown in this mesh network approach, with a best recorded node discovery rate of 1.34 nodes (ID cards) recorded per second. However, the research viewed attendance tracking through lens of creating collaborative attendance tracking protocols, rather than a complete system design that could be employed by a college.

Furthermore, students are more likely to forget an ID card than they are to forget their smartphone.

2.3 Bluetooth Low Energy

Bluetooth Low Energy was introduced alongside the Bluetooth 4.0 specification, as a subset specification with a focus on low power communication. BLE, which started out as a Nokia research project named "Wibree", is aimed at ultra-low power devices which are designed to run for years on a coin cell battery. With the emergence of the Internet of Things, these devices are being embedded and integrated into everyday contexts such as health, fitness, home automation, and smart grids. The goal of BLE was to create the lowest-power short-range wireless technology possible, with each layer of its stack being optimised to reduce the power consumption needed to perform its task [8].

Unlike "Bluetooth Classic" (also known as Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR)), Bluetooth Low Energy is optimised for short burst transmission of data (e.g. temperature readings) instead of continuous streaming of data (e.g. audio streaming via Bluetooth headphones). While Bluetooth Classic only supports a one-to-one network topology known as "Point-to-Point", BLE supports Point-to-Point, Broadcast (one-to-many), and mesh network (many-to-many) topologies. This makes it a very flexible technology for use in applications involving power-constrained devices.

2.3.1 BLE Protocol Stack

The BLE protocol stack consists of three primary building blocks ("layers"): *Application, Host*, and *Controller*. These layers, along with an intermediary layer named the "Host Controller Interface", can be seen in Figure 2.1.

1. Application Layer

As the topmost layer of the stack, the Application layer houses the logic, data handling, and UI of the user application. It is highly use-case dependant.

2. Host layer

The Host layer is responsible for implementing and exposing APIs that allow applications to make use of the physical Bluetooth radio. It consists of several layers:

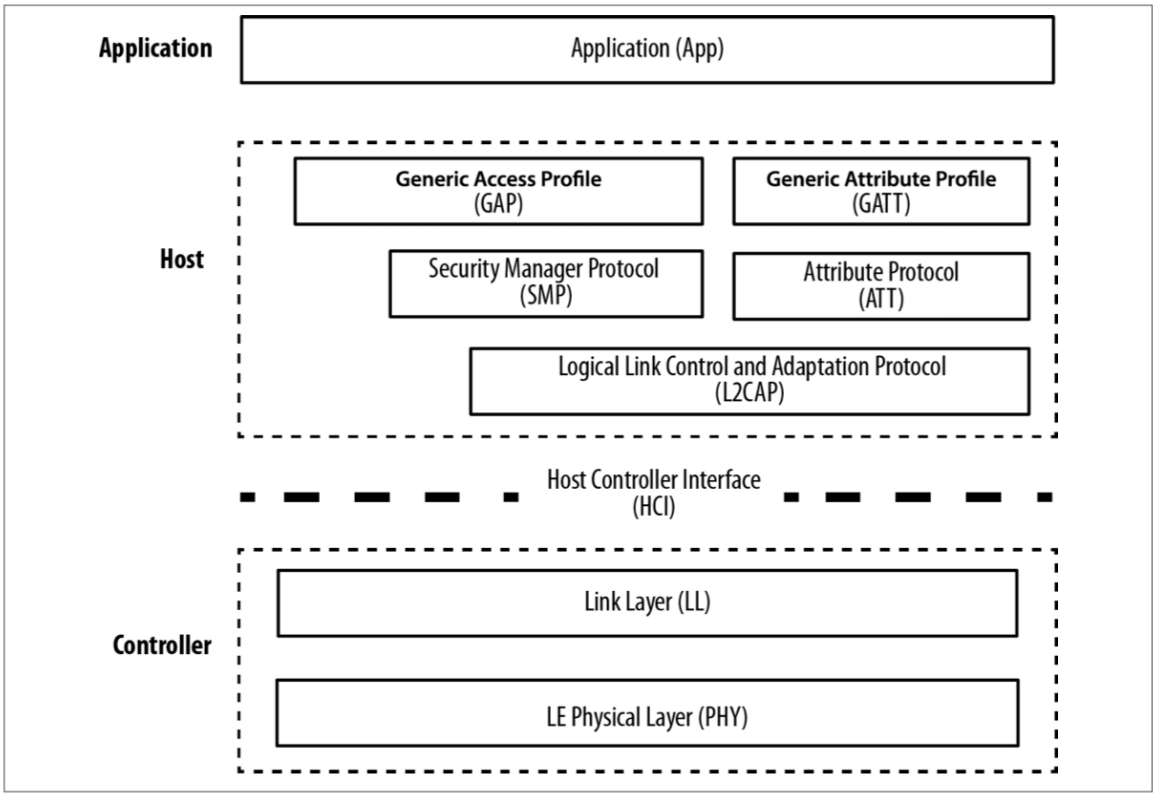


Figure 2.1: The layers of the BLE protocol stack[9]

- **Generic Access Profile (GAP)**: GAP defines roles and procedures for device discovery and connection establishment in order to ensure interoperability between BLE-enabled devices. Section 2.3.3 examines this in further detail as it is relevant to the project implementation.

- **Generic Attribute Profile (GATT)**: GATT builds upon the ATT and defines a hierarchical data model that is used to structure the data transferred between devices. As with GAP, this is discussed in more depth in Section 2.3.4.

- **Logical Link Control and Adaptation Protocol (L2CAP)**: The L2CAP multiplexes the data from several of the upper layers and encapsulates them into the standard BLE packet format. It is also responsible for fragmentation and recombination. It breaks up large packets from the upper layers into 27 byte payloads and passes them to the lower layers, and vice versa from lower layers to upper layers.

- **Attribute Protocol (ATT)**: The ATT protocol employs a simple client/server architecture that stores and serves data in the form of "Attributes". It also handles responding to client requests for this data. Attributes are defined as

having a 16-bit handle (used by clients to address the attribute), a UUID denoting its type, permissions, and a value.

- **Security Manager (SM)**: The Security Manager is responsible for implementing the security procedures and cryptographic algorithms that allow devices to communicate with each other over an encrypted link. It defines two roles in the context of a connection: the *Initiator* of the connection (GAP *Central* role), and the *Responder* (GAP *Peripheral role*). The SM provides bonding, pairing, and connection re-establishment needed for these two roles to communicate securely.

Host Controller Interface (HCI)

As indicated by its name, the HCI is an interface between the Host and Controller that sends commands and events between the two layers. It is an intermediary building block between these layers that allows a BLE device to be controlled over a serial interface such as UART or USB.

3. Controller layer

- **Link Layer (LL)**: The Link layer is the layer of the protocol that directly interacts with the PHY and manages the physical link. It is responsible for advertising, scanning, and creating/maintaining connections. It is also responsible for complying with the timing requirements of the specification. The link layer defines four roles:
  - Advertiser: A device emitting advertising packets.
  - Scanner: A device scanning for advertising packets.
  - Master: A device that establishes a BLE connection controls and manages its timing.
  - Slave: A device that accepts a BLE connection initiated by the Master and follows its timing.
- **Physical Layer (PHY)**: The Physical layer contains the analogue communications circuitry that modulates and demodulates analogue signals and turns them into digital symbols [9]. This modulation uses Gaussian Frequency Shift Keying (GFSK). As explained in Section 2.3.2, this radio operates on the 2.4 GHz band, hopping between one of 40 communication channels.

2.3.2 Advertising and Connections

Bluetooth Low Energy communicates on the 2.4 GHz ISM band over one of 40 channels, each 2 MHz apart. As shown in Figure 2.2, 37 of these channels are data channels and 3 are advertising channels. The protocol uses the Frequency-Hopping Spread Spectrum (FHSS) technique to "hop" between these channels in a pre-agreed sequence in order to facilitate connections during interference.

One of the aspects of the BLE protocol that sets it apart from Bluetooth Classic is its use of Advertisements. Advertising packets are one of the two types of packets used by the protocol; the other being data packets. Advertising packets allow masters to identify slaves that are ready to accept a connection. Alternatively, they can be used to broadcast, in a one-way fashion, small amounts data that don't require a full connection. Advertising packets generally contain the broadcaster's MAC address, an optional payload (31 bytes), and an indicator stating whether or not they're connectable.

Advertisements are broadcasted on one or more of the 3 advertising channels: *37, 38,* and *39*. They are sent by the advertiser at a fixed interval known as the *advertising interval*. Similarly, a device scanning for these packets will scan at a fixed *scan interval* for a specified length of time known as the *scan window*, meaning the advertisement will be detected when these two randomly overlap.

The scanner may passively scan these packets and not establish a connection. Alternatively, if these advertisements indicate that the sender is connectable, the scanner may assume the role of master and establish a Bluetooth Low Energy connection with this device. The two devices determine timings of the hops required in the connection, and may negotiate an encryption scheme and key sharing protocol to communicate securely. During this connection, the two devices can communicate on a one-to-one basis over one of the 37 data channels.

2.3.3 GAP

The Generic Access Profile (GAP) is an interface that defines device roles, modes and procedures for the discovery of devices and services, the management of connection establishment and security [11]. It provides an agreed-upon framework for BLE-enabled devices to implement that determines how they identify and communicate with each other. The four roles it defines are *Broadcaster, Observer, Peripheral,* and *Central*.

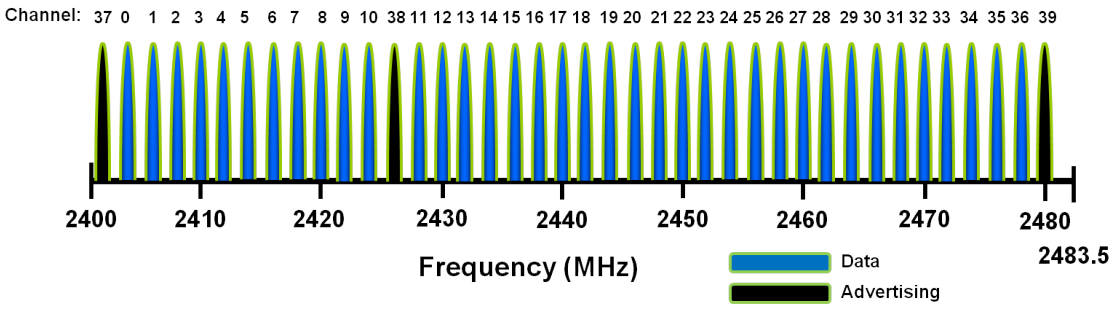


Figure 2.2: BLE channel arrangement on the 2.4 GHz ISM band [10]

- The **Broadcaster** transmits data through the advertisement process and does not support connections. It broadcasts its data periodically to any other devices in advertising packets instead of data packets.

- An **Observer** can be viewed as the complement to the Broadcaster as it monitors for data transmitted by Broadcasters in its general proximity. It does so via BLE scanning, which scans for advertising packets emitted by the Broadcaster. It only passively collects the advertised data and does not create connections.

- The **Peripheral** is similar to the Broadcaster role in that it sends out advertising packets. Unlike the Broadcaster, the Peripheral is connectable and emits these advertisements for the purpose of scanning devices to identify it and establish a connection with it. Its equivalent role in the Link Layer is the "Slave" role. These are usually power-constrained devices like sensors or heart rate monitors which communicate their data back to a Central device.

- A device in the **Central** role scans for advertisements from Peripheral devices in order to identify devices which are available for connection. The Central device creates the connection in this scenario and is equivalent to the Link Layer "Master". Devices in the Central role are generally more powerful devices such as computers or mobile phones which collect data from multiple less powerful Peripheral devices.

2.3.4 GATT

The Generic Attribute Profile (GATT) provides a way to encapsulate the use case of transferring data between a *central* and *peripheral*. It can be considered "the backbone of BLE data transfer because it defines how data is organized and exchanged between

applications" [9]. GATT uses the concepts of **Services** and **Characteristics** for structuring the data that is exchanged once a BLE connection has been established between two devices.

GATT Services

A GATT Service is a logical collection of related characteristics which are grouped together for the purposes of describing a certain use case. For example, a temperature sensor may define a **Thermometer Service**, which may contain relevant Characteristics such as **Current Temperature, Sensor Battery Level, etc**. Services are identified by a Universally Unique Identifier (UUID), which other connected devices can use to query for the service. For official Bluetooth SIG services, this is a 16-bit UUID, whereas custom developer-created services must use 128-bit UUIDs.

GATT Characteristics

GATT Characteristics are the lowest level concept in GATT transactions and are used to encapsulate user data [12]. Like a Service, a Characteristic is identified by UUID which denotes its type. It also contains a *value* field which holds the actual data of the characteristic and *properties* which specify the read/write permissions allowed on the characteristic. The value may be accessed by querying a Service for the UUID of the relevant Characteristic. Characteristics may also have optional **descriptors** associated with them. These are metadata or configuration flags relating to the characteristic [13].

GATT Client & Server

- A **GATT Server** is a device that stores data about one or more services and their associated characteristics. It responds to read or write requests for this data made by the GATT Client, as well as optionally sending notifications to the client about this data.
- The **GATT Client** is the device that sends requests to the server regarding GATT data. It initiates connections and requests to this server and queries for services by UUID lookup. It may also optionally receive updates, in the form of *notifications* or *indications*, from the server when values are updated.

Generally, a peripheral will act as the GATT Server and a central will take the role of

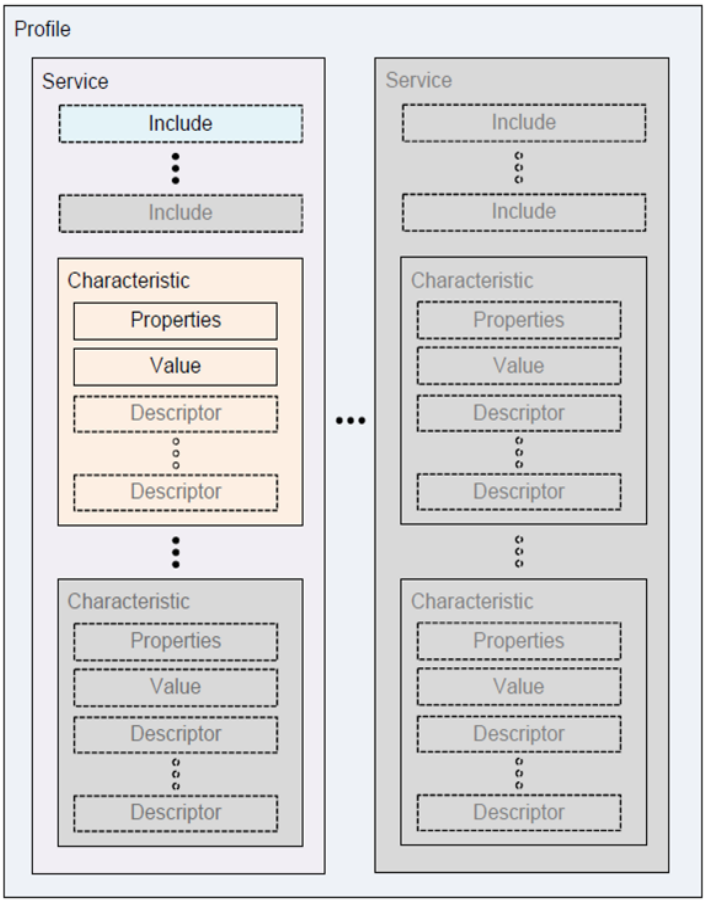


Figure 2.3: An illustration of the nested structure of a GATT Profile [13]

GATT Client. However, this is not always the case and they may reverse roles.

Chapter 3

Design

This chapter presents a high-level overview of the designed attendance tracking system. The functionality of each of each of the three primary system components is outlined. A brief description of the communication that occurs between the components is given to provide a synopsis of how the system works.

3.1 System Overview

The system is comprised of three core components: the Attendance Tracker, the Scheduling API, and the Mobile Application.

- The **attendance tracker/scanner** is the program running on the device installed in each lecture room and is responsible for recording the presence of each student in a lecture. Lecture schedules are pulled from the scheduling API and attendance tracking sessions are scheduled accordingly. It scans for each student using BLE scanning and then uploads the results to the API for storage.

- The **scheduling API** is a web service that stores schedules for each room and enables each Raspberry Pi to update its schedules automatically, without manual configuration. It also acts as a recipient for the attendance data recorded by the attendance tracker, all of which is stored in a database.

- Each student will use the **mobile application** in order to sign in to a lecture. The application signs the student into the lecture by transmitting their student number to the attendance tracker twice - near the beginning of the lecture and towards the end of the lecture. The rationale for this is that it proves that a

student was in the lecture for entire duration, rather than using the application from outside the lecture at the beginning.

A network diagram showing a high-level overview of the designed attendance system may be seen in Figure 3.1.

3.1.1 Communication

Initially, there were two variations of the design: a *connectionless* approach, and a *connection-based* approach.

With a connectionless design, the student number is transmitted in the advertising packet itself as advertising data. When the tracker is scanning for advertising packets, it simply records the student number in the packet and does not initiate a BLE connection with the device. This approach has the advantage of being simple and quick as the tracker can sign students in as fast as it can scan for packets. However the broadcaster (the student) has no way of verifying the successful receipt of their student number and thus no way of knowing if they've been signed in successfully. Due to this lack of feedback, the application cannot know when to stop broadcasting and would need to be stopped manually or after a period of time.

The chosen design is a connection-based design, which takes a more structured approach by communicating the student number in a short-lived BLE connection with the scanner. The student number is defined as a characteristic of a BLE GATT service named **"Student Number Service"**. The student broadcasts their presence as before, but without their student number in the advertising packet's data. The attendance tracker then creates a connection with the phone and retrieves the student number before terminating the connection.

While slightly more complex than the connectionless approach, this design gives the mobile application feedback that the student number has been sent, meaning that the app may then stop advertising and sleep until the second scanning pass that will occur at the end of the lecture. This has the added benefit of not consuming excess battery by indefinitely sending out BLE advertisements. BLE-enabled devices generally have a simultaneous connection limit, meaning the attendance tracker will not be able to record all the students at once and the advertisers will have to wait. However, as the phones cease broadcasting and sleep one by one, the network congestion will decrease

gradually and they will be more likely to successfully connect. The confirmation of their successful attendance and reduced power consumption provided by this approach made these trade-offs worthwhile. For these reasons, the connection-based design was chosen and can be seen in Figure 3.1.

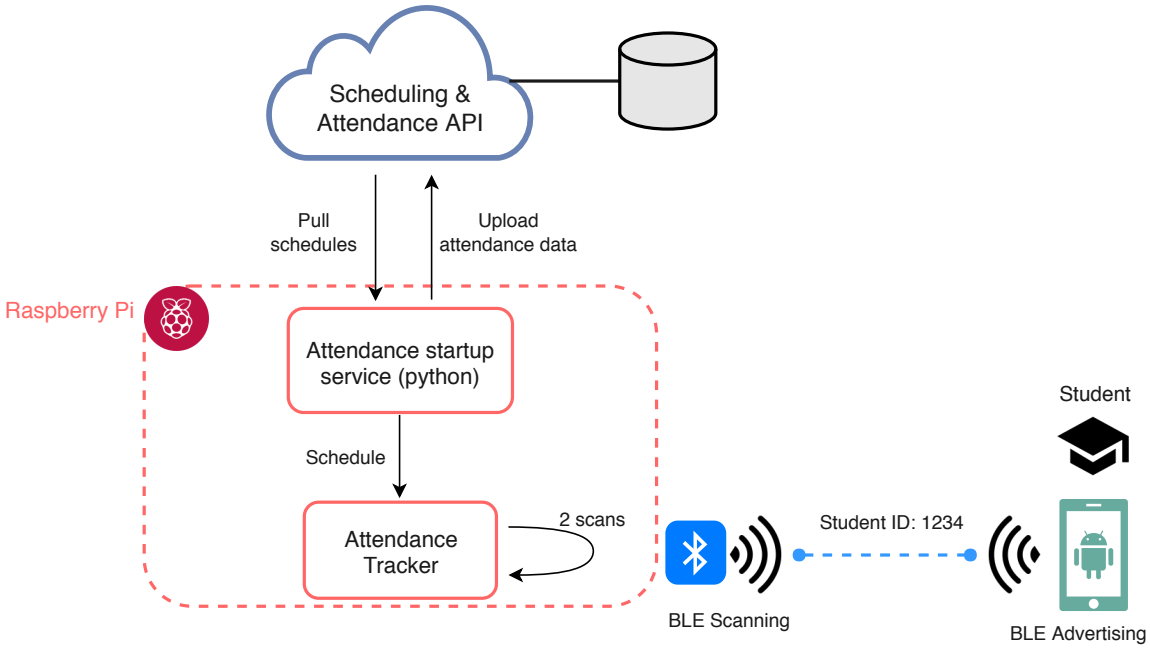


Figure 3.1: High level overview of the system design

3.2 Attendance Tracker

The attendance tracker component is made up of the physical device installed in the lecture room as well as a program for tracking attendance. This program acts as an infinitely running daemon, which is automatically started in the background at boot time. It schedules attendance tracking sessions based on the lecture schedules it requests from the scheduling API via HTTP, meaning the device can be configured remotely and doesn't require manual interaction from the lecturer. During one of these sessions, the attendance tracker will scan for BLE advertising packets emitted by the mobile application. A BLE connection is initiated by the attendance tracker after which it queries the mobile phone for the GATT service containing the student number. This GATT service is outlined in further detail in Section 3.4. Issuing a read request for the GATT characteristic containing the student number allows it to record the student's ID and then terminate the Bluetooth connection. This process is repeated for each student detected by the tracker within a certain window of time. It

then sleeps before repeating this for a second pass, recording each student number a second time. As mentioned in Section 3.1, this is to avoid spoofing one's attendance at a lecture by simply performing the sign in once from outside the lecture room and then not attending.

By the end of the lecture, the attendance tracker will have obtained a list of student numbers, as well as the number of times it recorded each student number (either once or twice). This data is sent to the scheduling API by sending a HTTP POST request to **/lectures/:id**, where **:id** is the unique ID for lecture as returned by the scheduling API. The student numbers list is serialized in JSON format and attached as the body of the POST request.

3.3 Attendance & Scheduling API

The attendance and scheduling API is responsible for persisting and making available all lecture schedules and student attendance data. The goal of this web service is to provide a way for a Raspberry Pi to update its lecture schedule remotely, as well as publish its attendance data for a lecture once it has been recorded. The intention is that this particular component would act as a proxy between the Raspberry Pi and the college's existing scheduling and attendance recording systems. This has the advantage of providing one central API for each Raspberry Pi to communicate with rather than multiple disjoint systems.

This communication is performed via HTTP GET requests in the case of retrieving lecture schedules, and HTTP POST requests when uploading student attendance data to the API. All data sent to and from the API is serialized in JavaScript Object Notation (JSON) format. An SQL database stores all lectures and displays them in a schedule for a specified room when requested.

3.3.1 Data Models

There are two primary data models in the API - *Lecture* and *AttendanceRecord*. A *Lecture* represents a single lecture slot in a room at a certain time, identified by a unique ID. A schedule for room is simply an aggregated chronological view of all *Lecture* objects with that particular room code. This is returned as JSON when a schedule is requested for a room. An *AttendanceRecord* confirms that a student (identified by their student

number) was successfully recorded as having attended a certain lecture. As seen in Figure 3.2, each record is uniquely associated with a lecture through a foreign key. This easily allows for finding a list of all students who attended a particular lecture, which is discussed in further detail in Section 4.2.

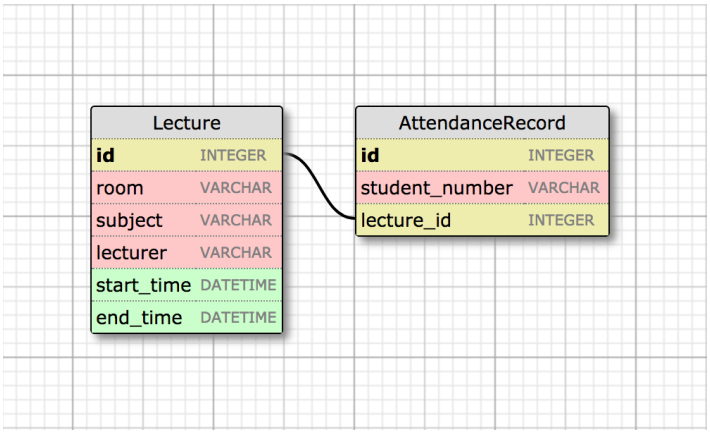


Figure 3.2: Data model relationships in scheduling API

3.4 Mobile Application

Each student records their attendance at a lecture through the mobile application on their phone. On the first boot of the app, the student will log in using their college credentials so that the application can save their student number. In the context of Trinity College, the intention is that this process would use the student's existing "My TCD" portal login. For the purposes of this dissertation, the student simply enters their student number into a text box as the application is not integrated with the college's existing login system.

At the beginning of the lecture, the student starts the application. As mentioned in Section 3.2, this will begin broadcasting their presence by transmitting BLE advertising packets. The Attendance Tracker can use these to initiate a BLE connection to the phone. Once connected, the tracker may read the student's ID by reading the appropriate GATT characteristic.

The application defines a custom GATT profile named **"Student Attendance Profile"** for the purpose of transferring the student number to the attendance tracker. A GATT profile may have one or more GATT Services, each of which can define a number of GATT Characteristics, i.e. values to be either read or written. Only one Service is needed, the **"Student Number Service"**, which contains one read-only

Characteristic, the **"Student Number"**. Each Service and Characteristic is assigned its own Universally Unique Identifier (UUID) for which the attendance tracker queries when it connects to the phone.

After the tracker scans the student number for the first time, the mobile application will stop advertising and the app will sleep. The app reawakens in time for the attendance tracker's second scan of the student number, after which the advertising process will terminate itself again and the attendance tracking will be complete.

Chapter 4

Implementation

The following chapter describes technical details of the prototype implementation of the attendance tracking system. It follows a similar structure to the previous chapter, discussing the hardware and software used in each component of the implementation, as well as the description of important algorithms or processes. Some justification is also given to hardware and software choices where relevant. This chapter aims to provide a more in-depth technical understanding of the system implementation.

4.1 Attendance Tracker

4.1.1 Hardware

A small, Bluetooth LE-enabled hardware device is required in each lecture room to run the attendance tracking software. A Raspberry Pi Zero W was chosen for this purpose as it is a very small, affordable computer that can run Linux. It is smaller in size than a debit card (Figure 4.1) and can be purchased for roughly 10 Euro. Raspbian (based on Debian) is the default Linux distribution for Raspberry Pi models and can be installed via a flashed MicroSD card. The Zero W also has built in WiFi and Bluetooth Low Energy support. A combination of this device's small form factor, ease of configuration, and connectivity capabilities are the primary reasons why it was chosen as the hardware for the Attendance Tracker.

An important point to note is that the Raspbian uses Bluez[1], the official Linux Bluetooth stack, to provide access to Bluetooth (LE) protocols and functionality. The Python implementation of the Attendance Tracker makes use of Bluez through a li-

[1] `http://www.bluez.org/about/`

brary, meaning that the implementation is not restricted to the Raspberry Pi Zero W. The attendance tracker implementation can run on any other Linux device which uses Bluez. This would grant a college the freedom to upgrade to more powerful devices or interchange between different hardware solutions as they see fit.

Figure 4.1: A Raspberry Pi Zero W[2]

4.1.2 Lecture Scheduling

Each attendance tracker is assigned to a lecture room so it requires a schedule of the lectures for the room in which it will be located. It must also have the ability to enable itself automatically in accordance with this schedule. This schedule is provided by the Scheduling API which is explained in further detail in the next section.

The implementation achieves this automatic scheduling by defining a startup service using systemd. Systemd is a Linux startup manager that, among other things, allows programs and daemons to be started up automatically at boot time. This startup service is described in a service file and then enabled and set to autostart using systemd's command line interface. The service below simply starts the Python attendance tracking software once all networking interfaces are up and running:

```
[Unit]
Description=Attendance Tracker Service
```

[2] https://www.raspberrypi.org/app/uploads/2017/05/PI-Zero-W-1-1620x1080.jpg

```
[Service]
Type=idle
ExecStart=/usr/bin/python3 /home/conor/attendance/scan.py

[Install]
WantedBy=multi-user.target
```

As previously mentioned, the attendance tracking software itself is implemented in the Python programming language (specifically, Python 3). This program acts as a daemon that, once started by systemd, runs infinitely until the system is shut down. This behaviour is intended so that the device can run autonomously and persist across power cycles. The program will be restarted upon every boot and will not require the institution or lecturer to manually enable it. The program executes in an infinite loop which can be seen in pseudocode form below:

```
while True:
    lectures = pull_lecture_timetable()

    for lecture in lectures:
        schedule(lecture)

    sleep(SLEEP_TIME_SECONDS)
```

The lecture schedules must be updated on start-up by querying the API with a GET request. A simple Python API wrapper was created for sending HTTP GET/POST requests using the Requests library and parsing the returned JSON response. An advantage of the Python language is that its "dictionary" data structure (akin to a hash map/associative array) has a virtually one-to-one mapping to JSON. This makes JSON (de)serialization trivial.

Once the schedule has been retrieved, each individual lecture is scheduled. They are scheduled by using "Timer" objects, which are part of Python's built-in threading library [14]. A timer can be used to schedule an action to occur on a thread after a certain amount of time. This amount of time is simply the delta between the current time and the lecture start time. After this delay has elapsed, a thread will be assigned to perform the action specified in its creation. In this case, that action is to carry out the scanning for student numbers.

When all lectures have been scheduled, the main thread may sleep for an appropriate amount of time. This amount of time will differ depending on how often the lecture schedule should be updated and how far in advance the college would like to schedule them. It should also be noted that Python's sleep functions may be affected by other factors like the operating system's scheduler. For this reason, perhaps a long sleep time should not be relied upon to be accurate and a shorter one should be used instead.

4.1.3 Scanning Process

When the Timer wakes, it begins the process of scanning for students. An **AttendanceTracker** Python class was created to carry out this process. This class provides a simplified, generalised interface for handling all behaviour related to the attendance tracking process. By simply passing in a parameter for the lecture name, this work can be offloaded to the object on a different thread and the results returned to the main thread upon completion:

```
tracker = AttendanceTracker(lecture="Security & Privacy")
students_attended = tracker.record()
```

First Scanning Pass

As mentioned in Section 3.2, student numbers are scanned twice: one pass near the beginning of the lecture (e.g. 9:10), and one pass before the students leave (e.g. 9:45).

The AttendanceTracker itself begins scanning for devices in the room that are emitting BLE advertising packets. It makes use of the Bluepy library which provides a collection of classes and methods for interacting with the Bluez stack via Python code. A Bluepy **Scanner** object is instantiated which scans for a duration and returns a list of all nearby broadcasting devices. Since there may be other devices broadcasting in the general vicinity, the devices are filtered to identify which devices are students advertising their student number and which devices are irrelevant. Students' devices will be advertising the 128-bit UUID that identifies the **Student Number Service** - the BLE data structure which will contain each student's identification number.

As these particular advertisement packets indicate that the student's device is connectable, the AttendanceTracker creates a BLE connection with the device using the hardware address detected during the scanning process. Using the appropriate UUIDs, it queries the device for the "Student Number" Service and its "Student Number"

Characteristic before disconnecting. The tracker repeats this process for each device scanned, recording student numbers one by one and storing them in a dictionary. The dictionary is a Key-Value data structure which allows for simple look-ups of how many times a particular student number has been recorded. An outline of this scanning pass it presented in pseudocode form below:

```
class AttendanceTracker(object):

    def scan_for_students(duration):
        devices = Bluepy.Scanner.scan(duration)

        for device in devices:
            service = device.get_service(service_UUID)
            student_number = service.get_characteristic(characteristic_UUID)

            register_student(student_number)
```

In a final implementation, this tracker may need to carry out each pass in a "Scan-Connect" cycle for the duration of the pass. For example, if the first pass lasts for 10 minutes, it may be wise to alternate between scanning for devices for 30 seconds or so, then connecting to each device to record the student numbers. This has the advantage that those devices will then sleep once they have been recorded in the first pass, reducing the number of devices that are advertising. The reduced network congestion means that the AttendanceTracker will then have a new set of devices in the second Scan-Connect cycle instead of repeatedly detecting the same devices. For debugging purposes, this implementation uses one Scan-Connect cycle since the scan duration was generally kept short during testing.

Second Scanning Pass

After the first pass has completed, the AttendanceTracker sleeps until it is time for the second pass. This is almost identical to the first instance but the program has already stored the ID numbers of all the students who have been recorded once. When they are recorded for the second time, the count for the number of times they have been recorded is simply incremented to 2. Once this second round of scanning has concluded,

the attendance tracking process for that lecture is complete. The AttendanceTracker terminates and returns its list of results to the main program.

Recording of Results

When the attendance tracker records a list of the students who attended a lecture, it must persist that data somewhere. It does so by uploading it to the Attendance and Scheduling API to store it in its database. As previously illustrated in the data models diagram in Figure 3.2, each lecture contains a unique ID field. The tracker stored the ID of each lecture it scheduled when it retrieved the lecture schedule from the API. To upload attendance data for a lecture, the attendance tracker sends a HTTP POST request to **/lectures/:id**. The list of student numbers are serialized into JSON format and attached to the body of the request:

```
{
    "student_numbers": [
        "12345678",
        "45678912",
        "etc",
        "etc"
    ]
}
```

Once this upload is complete, the thread's actions are terminated and the program returns back to its infinite loop.

Asynchronous Connections

Initially, it was intended that during the scanning process, the AttendanceTracker would connect to multiple devices simultaneously to record their student numbers. This was attempted using Bluepy's **DefaultDelegate** class which allows BLE messages to be received by the scanner asynchronously [15]. In theory, this should have allowed the Scanner to process multiple connections at once, up to a limit imposed by the manufacturer of the Bluetooth chip. In practice, this method resulted in the scanner disconnecting intermittently and the student number never being recorded. After some

research, it appears to be a bug/limitation in the BLE Host Controller Interface[3], [4]. This bug resulted in active connections being disconnected when a second connection was pending. For this reason, this approach was abandoned in favour of the "Scan-Connect" cycling approach using sequential connections described above.

4.2 Attendance & Scheduling API

The Attendance and Scheduling API is the web service that enables each Attendance Tracker to update their lecture schedules remotely and additionally serves as a recipient for the attendance data they have recorded. An implementation of this API was developed using Flask, which is a "microframework" for writing web services in Python. It is a lightweight framework that allows for the mapping of URL routes to Python functions in order to display dynamic content from a database. It also ships with a built-in web server.

4.2.1 Data Storage

An overview of the data models designed for the API was presented in Section 3.3. An SQLite database is used for storing these records. This is a lightweight, embedded SQL database that does not have a separate server process and instead reads and writes directly to disk files [16]. It is Flask's default database and is the most popular database in the world according to its website.

Rather than creating an manipulating the data records directly through SQL statements, SQLAlchemy was used. This is an Object Relational Mapper (ORM) that allows SQL records to be defined as Python objects. Here is an example of the Lecture model defined using SQLAlchemy:

```
class Lecture(Model):
    __tablename__ = "lectures"


    id = Column(Integer, primary_key=True, nullable=False)
    room = Column(String(128), nullable=False)
    subject = Column(String(128), nullable=False)
```

[3] https://github.com/IanHarvey/bluepy/issues/126
[4] https://github.com/IanHarvey/bluepy/issues/57

```
    start_time = Column(DateTime(timezone=True), nullable=False)
    end_time = Column(DateTime(timezone=True), nullable=False)
    lecturer = Column(db.String(128))

    students = relationship('AttendanceRecord', backref='lecture', lazy='dynamic'
```

SQLAlchemy provides an intermediary layer which removes the need for writing SQL queries, greatly simplifying database related code. Foreign key relationships and associations are also greatly simplified in SQLAlchemy as these associations are pre-loaded and accessible through attributes on the Python object. This can be seen in the above code listing in which a **students** attribute is defined. This is defined as a relationship, which can be viewed as a virtual database column containing all the **AttendanceRecord** objects referring to this **Lecture** object via foreign key. For example, accessing a lecture and all the students who attended that lecture can be easily done:

```
lecture = Lecture.query.get(id)      # Lecture object
students = lecture.students          # List of AttendanceRecord objects
```

4.2.2 API Routes

The API exposes a number of URL routes (or "endpoints") which are accessible over HTTP. Flask uses a pattern called the Decorator pattern to define these routes. This uses Python decorators to annotate a function, mapping a HTTP URL to this function call. The function returns a response JSON response object which Flask ensures is returned to the client as a HTTP Response (along with a status code) by Flask's web server. Flask makes any query parameters or headers available as Python variables. From these functions, the database can be queried or modified using the SQLAlchemy definitions of the data models. As is typical with a web application or API, each function performs one of the *Create, Read, Update* or *Delete* (CRUD) operations on the database records. The following function handles a request for the schedule for a room, triggered by a GET request to **/schedules/:room** where **:room** is the unique room code (e.g. "LB04").

```
@app.route('/schedules/<room>', methods=['GET'])
def get_schedule(room):
```

```
    room_schedule = Lecture.query.filter_by(room=room).all()
    response = {
        'room': room,
        'schedule': room_schedule
    }
    return jsonify(response), 200
```

Scheduling

As can be seen in the endpoints in Table 4.1, the API exposes a URL for accessing the schedule for a particular lecture room. The API does not create a **Schedule** model, as a schedule in this context is simply an aggregated list of all of the lectures in a room. The code listing above shows the function that returns said schedule. The function is decorated with its corresponding route and the **room** variable is passed in as a function argument. The list of the room's lectures are queried and parsed into JSON. Below is an example of the JSON response that this function would return when retrieving the schedule for "LB04":


```
{
  "room": "LB04",
  "schedule": [
    {
      "id": 1,
      "subject": "Advanced Telecommunications",
      "lecturer": "Dr. Joe Bloggs",
      "room": "LB04",
      "start_time": "Mon, 09 Apr 2018 10:00:00 GMT",
      "end_time": "Mon, 09 Apr 2018 11:00:00 GMT",
      "status": "FINISHED"
    },
    {
      "id": 4,
      "subject": "Security & Privacy",
      "lecturer": "Dr. Joe Bloggs",
      "room": "LB04",
```

```
      "start_time": "Wed, 11 Apr 2018 16:00:00 GMT",
      "end_time": "Wed, 11 Apr 2018 18:00:00 GMT",
      "status": "FINISHED"
    }
  ]
}
```

Note that the JSON response also contains a **status** field, indicating whether the lecture is yet to occur, in progress, or finished at the time of the request. This field is generated at request time instead of storing it in the database. A utility method in the Lecture object returns the status based on if the current request time is before, during, or after the lecture start and end time.

HTTP Verb	**Endpoint URL**	**Description**
GET	/schedules	Get all lectures in the database (debugging purposes)
GET	/schedules/**:room**	Get lecture schedule for **:room**
GET	/lectures/**:id**	Get lecture summary and attendance information for a lecture by id
POST	/lectures/**:id**	Upload recorded list of students who attended lecture **:id**

Table 4.1: Endpoints for the Scheduling API

Attendance

The API supplies an endpoint for AttendanceTrackers to upload the students who attended a lecture via HTTP POST. Flask allows for convenient parsing of the attached JSON body into a Python dictionary. The dictionary will have a key named "studentnumbers" which contains the list of students. For each student in the list, a new AttendanceRecord object is created containing the student number and the lecture id. At this point, these objects are just Python objects and the database remains unchanged.

Using SQLAlchemy, changes to the database are done in the context of a session [17]. The new AttendanceRecord object is added to the session. To persist this data, the data is "committed" atomically so that either all session changes are put into

effect, or none are (in the case of a failure). If this operation fails, the database is automatically rolled back. An outline of this process is shown below.

```
lecture = Lecture.query.get(id)
student_numbers = params.get('student_numbers')

for number in student_numbers:
    record = AttendanceRecord(student_number=number, lecture=lecture)
    db.session.add(record)
    db.session.commit()
```

The above occurs upon a POST request to **/lectures/:id**. A GET request to the same URL defines a different endpoint for viewing detailed information about a lecture. If the lecture has finished, then the attendance data for that lecture will be returned in the response. While this implementation simply returns them in the JSON body, this could be exported to a spreadsheet or uploaded to the college's servers now that the data has been stored. Another potential use case for this data is to make it available to a separate dashboard web application where students may view their own attendance figures.

An example of this recorded attendance data is presented in the following JSON snippet, with the lecture information shortened for brevity:

```
{
    "status": "success",
    "lecture": {
        "id": 1,
        "etc": "etc",
        "status": "FINISHED"
    },
    "students": [
        "1234",
        "5678",
        "9012"
    ]
}
```

4.2.3 Configuration and Deployment

In the case of a college or university implementing this system, it is likely that they would deploy the API on their own servers within their network. This would allow them to integrate it with any of their own timetabling and attendance storage software, as well as enabling them to restrict traffic to the API. For the purposes of this research, this implementation was deployed using Amazon Web Services (AWS). AWS delivers a cloud hosting service for provisioning virtual private servers in the cloud. The API was hosted on a single Elastic Compute Cloud (EC2) instance.

Docker and Pip are responsible for the configuration and dependency management of the project. Docker is a containerisation platform that runs configured Linux containers on a host machine. Pip is a popular package manager for Python, used to install the external libraries used by the project.

Using Docker, the process for configuring and installing the API can be entirely automated and scripted with a "Dockerfile". If the API is installed onto a fresh machine, Docker will install Python and Pip, install any dependencies with Pip, and start the server and database. An annotated Dockerfile can be seen in Appendix A.

4.3 Mobile Application

The mobile application is the component of the system that students directly interact with using their smartphones. This implementation of the mobile application was developed in Android, using the Java programming language. The application was physically deployed and tested on a Samsung Galaxy S6, running Android 7.0 Nougat (API Level 24). An iOS application will likely need to be developed for Apple iPhone users. However, this Android application is sufficient for the purposes of demonstrating how a student will sign in during a lecture.

Both Android and iOS support BLE - Android from version 4.3 (API Level 18)[5], and iOS from version 5.0 (and iPhones 4S and above)[6].

[5] `https://developer.android.com/guide/topics/connectivity/bluetooth-le`

[6] `https://developer.apple.com/library/content/documentation/DeviceInformation/Reference/iOSDeviceCompatibility`

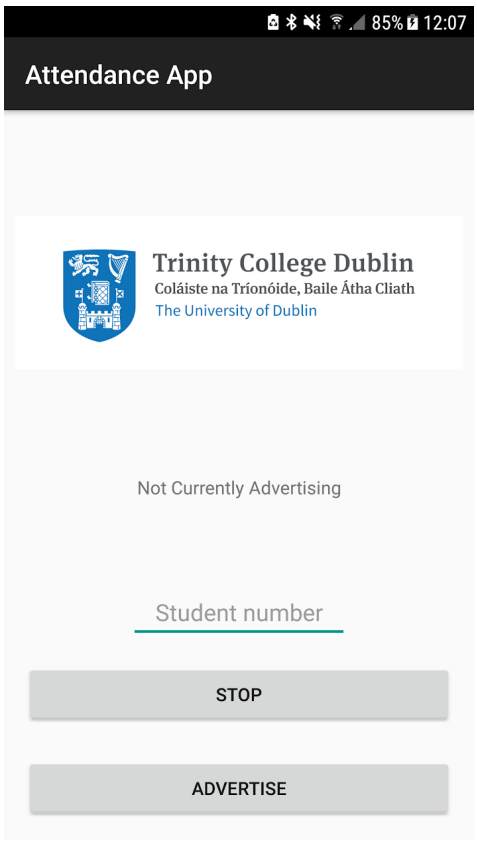


Figure 4.2: The Main Activity of the Android application.

4.3.1 Main Activity

The entry point to the application is the **Main Activity**, which can be seen in Figure 4.2. An activity can be thought of as a screen in the application with a user interface. Activities are developed using a combination of Java and XML. XML is used to define the UI layout and its various components (buttons, text boxes, images, etc.). A Java class controls the behaviour and lifecycle of the activity.

Since the aim of this application is to be seamless and easy to use, only one Activity is needed. The objective was to keep the UI minimal, with an EditText box for the student to enter their student number, and a button for beginning the attendance broadcasting process. This implementation has both a start and stop button for debugging purposes, but in practice a student would have just a start button as the application controls starting and stopping itself dynamically. If a college were to integrate this attendance tracking system, then the student number EditText box would be replaced with a login form. No such integration currently exists so the "login" is in the form of this simple text field.

To begin tracking their attendance, a student will open the app, enter their student

number and tap the start button. At this point the Main Activity will create a Service object.

4.3.2 Background Service

A Service is used to keep a long running operation executing in the background. If the attendance tracking behaviour were implemented in the Main Activity, then it would be stopped or interrupted when the student opened a different app, returned to their home screen, or locked their phone's screen. Since the app must be as unobtrusive as possible, a Service must be used. The **Advertiser Service** was created for this purpose.

When initially started by the MainActivity, the AdvertiserService calls an initialisation method, which performs three primary actions:

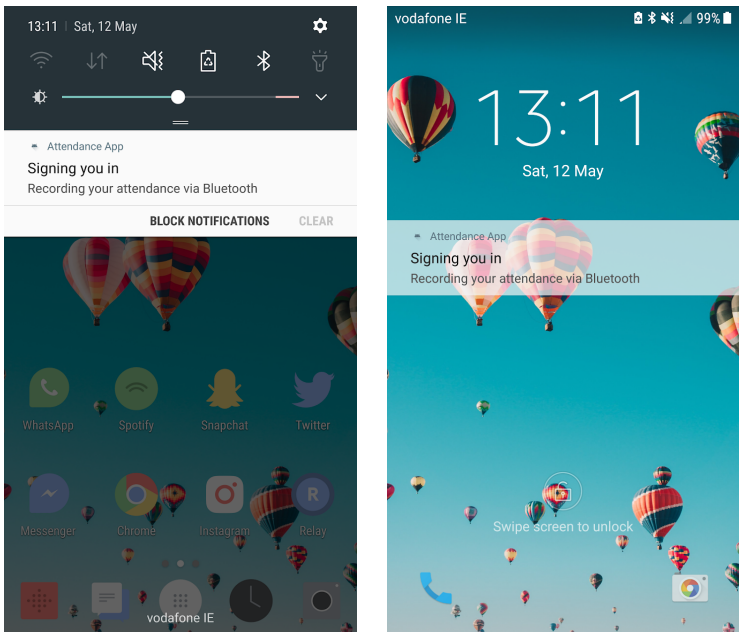
- Read the value of the student number from the MainActivity. It retrieves this through an "Intent Extra" message created by the MainActivity. Extras are string messages that an Activity can pass to other Activities or Services.
- Start a GATT server containing the Student Number Service that will respond to requests about this service during a BLE connection.
- Begin advertising its presence via BLE, indicating that the device is connectable and that there is a GATT server running.

Once this initialisation method returns, the service is considered started, so any set-up or configuration code should be called here.

```
public int onStartCommand(Intent intent, int flags, int startId) {
    String intentExtra = intent.getStringExtra(MainActivity.STUDENT_NUMBER);
    if (intentExtra != null)
        studentNumber = intentExtra;

    advertise();
    startGATTServer();

    return START_NOT_STICKY;
}
```



(a) Home screen (b) Lock screen

Figure 4.3: The attendance tracking notification displayed by the AdvertiserService

Advertising

As this service will be running in the background and services do not have a UI attached to them, it is helpful for the student to have some indication or visual feedback that their attendance is being recorded. Hence, when the Service begins advertising, a notification is displayed on the student's smartphone. This notification also has the benefit of moving the service to the foreground, so that Android will not kill the application while performing memory management. This means that the attendance tracking process can be guaranteed to run in its entirety, while still allowing the service to run without a UI and while the screen is locked. This is displayed in Figure 4.3.

When the service begins advertising, a number of parameters must be set:

Advertise Mode - If the advertising mode should be low power, low latency (and high power), or balanced. This affects the visibility range of the advertising packet and the power consumption.

Tx Power Level - If the transmission power level should be ultra-low, low, medium, or high. This also affects the visibility range of the advertising packet.

Connectable - Whether or not the device is connectable to other devices who scan the advertisement.

Timeout - An optional timeout delay after which advertising will terminate.

The parameters are set to Balanced mode, High Power, Connectable. The range needs to be maximised but it attempts to strike a balance between range and power consumption.

Finally, the data in the packet itself is set. This data simply includes the UUID for the Student Number Service, along with the random device address that Android generates itself. This randomised address is to avoid privacy issues where a user can be tracked via their device's physical address. This is explained in further detail in the security considerations in Section 5.2. The AdvertiserService obtains a reference to the device's Bluetooth adapter and then calls a method of Android's **BluetoothLeAdvertiser** class to start transmitting advertising packets in the vicinity.

GATT Server

The mobile application acts as a GATT server which responds to the AttendanceTracker's requests for the student number once a BLE connection has been established. Section 3.4 describes the **Student Number Service** - the GATT Service implemented by this GATT server. A callback is supplied to this GATT server during its instantiation. This callback is a Java class that contains methods for responding to requests by the remote client (the tracker) about this device's GATT services during a connection with that client.

The two trigger events implemented in the callback are when a remote client connects/disconnects via a one-to-one BLE connection, and when a remote client issues a read request for a GATT characteristic during a connection. The former simply logs the connection state change and device address for debugging purposes, while the latter is how the application communicates the student number to the AttendanceTracker. The behaviour for this method is as follows (the full callback handler can be seen in Appendix B):

- The app checks if the UUID in the request matches the UUID of the Student Number characteristic. If not, it responds with an error.

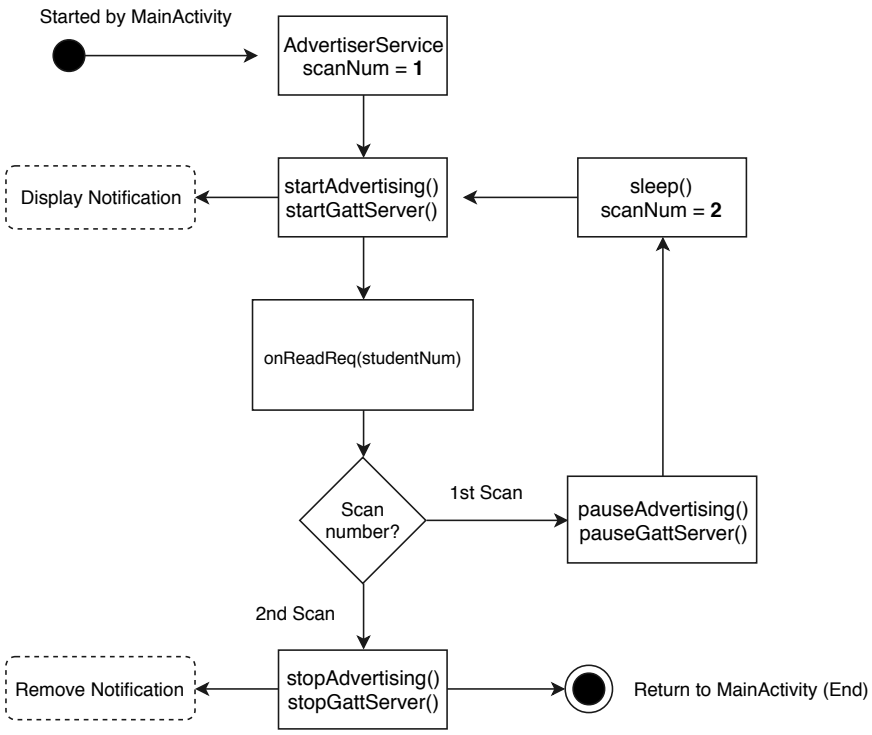


Figure 4.4: The flow of state implemented by the Android application.

- If the UUID is correct, the application responds with the student's student number stored in the characteristic's value field, along with a success status code.
- If this is the first time that the student number has been read by the tracker, the app then sleeps until the second scanning pass, stopping the GATT server and the BLE advertising to conserve battery.
- However if this read request is the second time the student number has been recorded, both advertising and the GATT server are stopped and the service terminates.

A state diagram is presented in Figure 4.4 to further illustrate the control flow of the Android application during the attendance tracking process.

Chapter 5

Evaluation & Results

This chapter investigates the performance and functionality of the implementation through a combination of experimentation and further research. Some possible security considerations are also outlined, along with any relevant countermeasures that could be taken to combat them. Finally, a summary of the findings is provided which analyses the results.

5.1 Experimentation

5.1.1 Recording Speed

Due to time constraints, the system performance was untested in a lecture scenario with a large number of smartphones. In lieu of this, the aim of this experiment is to determine the average length of time it takes to record one student number, with a view to extrapolating that result to give a rough estimate of the time needed to track all students in one pass. The test was performed using a Samsung Galaxy S6 running the mobile application and the Raspberry Pi Zero W implementation of the attendance tracker.

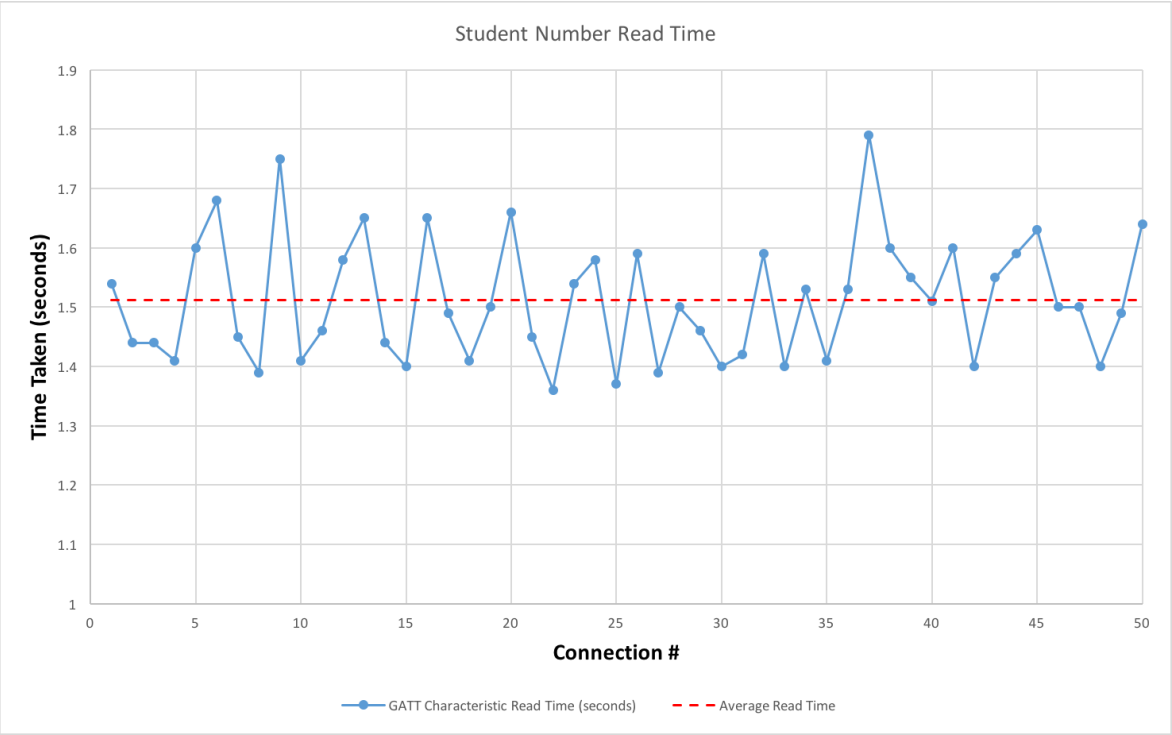


Figure 5.1: Student number read times (in seconds). The mean time (1.51 seconds) is shown in red.

A chart of all the recorded times is presented in Figure 5.1. Table 5.1 provides a summary of the results, with the mean read time (in seconds) of 1.51 ± 0.03 (95% CI). These numbers can be used to estimate how long it would take to scan each student's number in a large class. In a class size of 200 it would take on average $1.51 * 200 = 302$ seconds (roughly 5 minutes) to read each student number in sequence (with no concurrent connections). This number is an extrapolation and will likely increase in a room with increased interference and range. However, it still gives a useful indication of how long each scanning pass must last to ensure all students can be recorded in time.

Number of connections	50
Shortest time (s)	1.36
Longest time (s)	1.79
Standard Deviation	0.1
Mean time (s)	**1.51**
95% Confidence Interval	± 0.03

Table 5.1: Connection times

5.1.2 Full System Cycle

Without integration of the Attendance Tracker, Scheduling API, and Mobile Application, this implementation would exist as a series of disjoint components rather than a proof of concept. The aim of this experiment is to test the integration of the components in the implementation by performing a full end-to-end cycle.

In order to assist with this experiment, a test endpoint was created in the API for creating temporary lectures on demand. A GET request to this endpoint returns a temporary dummy lecture that is scheduled to start 30 seconds from the request time. This response contains some fake lecture information, but a valid lecture ID associated with a lecture in the database. This means that the start time of a lecture can be dynamically configured for evaluation purposes without needing to create a new lecture for each test. When the Attendance Tracker is automatically started at boot time using the systemd startup service, it queries this test endpoint to update its timetable and schedules this temporary lecture with an ID of 1. The response returned by the API is as follows:


```
{
  "schedule": [
    {
      "end_time": "Tue, 15 May 2018 20:36:48",
      "id": 1,
      "lecturer": "Dr. Test Testington",
      "room": "LB08",
      "start_time": "Tue, 15 May 2018 20:36:18",
      "subject": "TEST LECTURE"
    }
  ],
  "status": "success"
}
```


The student number "12345678" is entered in the mobile application and it begins broadcasting. The attendance tracker then proceeds with the attendance tracking process described in Section 4.1, scanning for two passes with each scan lasting 45 seconds. The full log from the attendance tracker can be seen in 5.2. These logs show

the lecture being scheduled, the two rounds of scanning, and the student numbers that were detected.

```
pi@raspberrypi:~ $ journalctl -u attendance.service
-- Logs begin at Tue 2018-05-15 18:23:17 UTC, end at Tue 2018-05-15 18:34:16 UTC. --
May 15 18:29:59 raspberrypi systemd[1]: Started Attendance Tracker Service.
May 15 18:32:13 raspberrypi python3[383]: start: 2018-05-15 19:32:20, delay: 14.829487, now: 2018-05-15 19:32:05.170513
May 15 18:32:13 raspberrypi python3[383]: Scheduled Lecture 1 [TEST LECTURE, LB08] @ Tue, 15 May 2018 19:32:20
May 15 18:32:13 raspberrypi python3[383]: 2018-05-15 19:32:05.170513
May 15 18:32:13 raspberrypi python3[383]: Main Thread Woken Up
May 15 18:33:58 raspberrypi python3[383]: Starting lecture TEST LECTURE on thread <Timer(Thread-1, started -1246325648)>
May 15 18:33:58 raspberrypi python3[383]: Recording attendance for: TEST LECTURE
May 15 18:33:58 raspberrypi python3[383]: Scanning attendance for 1st time...(1/2)
May 15 18:33:58 raspberrypi python3[383]: Service <uuid=4cc28e11-4465-4136-b84c-ab34109b3d87 handleStart=40 handleEnd=65535>
May 15 18:33:58 raspberrypi python3[383]: [<bluepy.btle.Characteristic object at 0xb5be0af0>]
May 15 18:33:58 raspberrypi python3[383]: Characteristic <8fb63c83-2aac-44dc-8fb6-6be9257cbdd1> 42 READ  True
May 15 18:33:58 raspberrypi python3[383]: 1.636045217514038 seconds
May 15 18:33:58 raspberrypi python3[383]: Student Num: 12345678
May 15 18:33:58 raspberrypi python3[383]: 1 student numbers scanned during first scan
May 15 18:33:58 raspberrypi python3[383]: Sleeping
May 15 18:33:58 raspberrypi python3[383]: Scanning attendance for 2nd time...(2/2)
May 15 18:33:58 raspberrypi python3[383]: Service <uuid=4cc28e11-4465-4136-b84c-ab34109b3d87 handleStart=40 handleEnd=65535>
May 15 18:33:58 raspberrypi python3[383]: [<bluepy.btle.Characteristic object at 0xb5be0a30>]
May 15 18:33:58 raspberrypi python3[383]: Characteristic <8fb63c83-2aac-44dc-8fb6-6be9257cbdd1> 42 READ  True
May 15 18:33:58 raspberrypi python3[383]: 1.5012283325195312 seconds
May 15 18:33:58 raspberrypi python3[383]: Student Num: 12345678
May 15 18:33:58 raspberrypi python3[383]: 1 student numbers scanned during second scan
May 15 18:33:58 raspberrypi python3[383]: {'12345678': 2}
May 15 18:33:58 raspberrypi python3[383]: Finished lecture TEST LECTURE
```

Figure 5.2: Logs from attendance service full cycle

When the student number is successfully read from the mobile application for the second time, it stops all BLE communication and displays a notification indicating that the student's attendance has been recorded successfully. The attendance tracker sends a POST request to the API at **/lectures/1** containing a JSON body with the list of recorded students. In this evaluation, the list contains a single student number (12345678). The success of this upload is confirmed by the API's response, which is presented below. This JSON response shows a completed lecture with 12345678 as the only student in the database with their attendance recorded for the lecture.


```
{
  "lecture": {
    "id": 1,
    "end_time": "Mon, 09 Apr 2018 11:00:00 GMT",
    "lecturer": "Dr. Joe Bloggs",
    "room": "LB04",
    "start_time": "Mon, 09 Apr 2018 10:00:00 GMT",
    "status": "FINISHED",
    "subject": "TEST LECTURE"
  },
  "status": "success",
  "students": [
```

```
    "12345678"
  ]
}
```

5.1.3 Advertisement Detection

While evaluating the research, an observation was made that the Raspberry Pi implementation had a relatively poor detection rate of advertising packets. After some investigation it was noted that just over 50% were detected on average. This was tested three times with scan durations of 15, 30, and 45 seconds. Out of 30 advertising packets, 14-17 (46-56%) were successfully scanned by the attendance tracker during each test. For comparison, the same test was performed on a 2015 Macbook Pro running the attendance tracker in a Debian virtual machine. The Macbook successfully scanned over 93% of advertisements, meaning the poor result is not related to the attendance tracking software but to the hardware chosen to run the implementation. There are likely a number of contributing factors to the Raspberry Pi's poor detection rate:

- The most likely factor is hardware limitations such as the Bluetooth card model, RAM, or CPU not being sufficiently powerful to consistently detect advertisements consistently. This means that the Raspberry Pi Zero W is simply not a capable enough choice of hardware for consistent advertisement detection.

- The Raspberry Pi supports an older version of Bluetooth Low Energy (4.1), compared to BLE 4.2 in the Macbook pro. While 4.2 is backwards compatible, it introduced some improvements to packet capacity and data transfer speeds that the Raspberry Pi is unable to take advantage of.

- Another potential factor is the advertising interval of the mobile application. Since advertising depends on the scanner and advertiser randomly overlapping, there are instances where advertisements won't be detected. Android does not permit changing this advertising interval manually, other than through the "Advertising Mode" variables discussed in Section 4.3. After inspecting Android's source code, it seems that the minimum available interval available to the developer is 100 ms. For reference, Apple's Bluetooth accessory design guidelines [18] recommend an initial advertising interval of 20 ms for the first 30 seconds, with an interval increase in specific steps if the advertisement is not detected.

However, as the change in scanning hardware resulted in a 93% detection rate, the advertising interval is unlikely to be the most significant factor affecting these results.

5.1.4 Power Consumption

The power consumption of the mobile application was also investigated through a combination of research and experimentation in order to assess the impact it would have on a student's battery. An experiment was carried out on the test device (Samsung Galaxy S6) that measured the battery drain of the application while constantly advertising on the least power efficient settings (100 ms advertising interval, high transmission power, no time out period).

Experiment

All background apps and processes were closed, WiFi and mobile data were disabled, and the screen remained off with the attendance application running. During this test, the application drained an average of 1.5% of the phone's battery per hour. For comparison, with no applications running the phone's normal battery drain under the same circumstances was 1% per hour on average, meaning the application drained an extra 0.5% while advertising constantly. While testing on a wider range of hardware is needed to say conclusively, these results are promising and indicate that excess energy consumption by the smartphone application is of minimal concern.

Related research

Some further research was conducted to investigate the authenticity of Bluetooth Low Energy's claims regarding low power consumption. For the most part, it does indeed seem to be quite energy efficient which aligns with the results of this experiment. A study that tested BLE's power consumption using an nRF51 System-On-Chip reported current consumption in an hour of between 0.038 mA and 0.461 mA depending on advertising interval and duration [19]. This would drain a 230 mAh battery in roughly between 6000 and 500 hours respectively. A similar study using a Texas Instruments CC2541 tested power consumption while in a continuous BLE connection and found the average current to be 0.024 mA [20]. This would drain the same 230 mAh battery in approximately 9583 hours. While these tests use power-constrained devices, their

results as well as the results of the experiment on the Android test device indicate that BLE is a suitable choice of technology for this dissertation's use case in terms of energy efficiency.

5.2 Security Considerations

5.2.1 Secure API Payloads

Each Raspberry Pi communicates with a central API when pulling the lecture schedules for its room and when uploading the attendance figures it has recorded. If this communication were to happen over HTTP, the system would be vulnerable to a Man-In-The-Middle-Attack. This attack could be an active attack or a passive attack. A passive attack could mean an attacker would eavesdrop on the student numbers being sent to the API and be able to tell which students attended which lectures. An active attack could involve an attacker substituting their own messages to and from the API, falsifying attendance information or lecture schedules. Using HTTPS will prevent these attacks and ensure that (i) - the Raspberry Pi is communicating securely and privately with the API server, and (ii) - the payloads sent between the Raspberry Pi and the server have not been altered. This will also require the Raspberry Pi to verify that the TLS certificate provided to it by the server is valid and signed by a trusted Certificate Authority. The project prototype implementation uses the Requests package for Python which verifies TLS certificates for HTTPS requests by default [21]. However, the API implementation was not deployed with HTTPS.

5.2.2 Client Authentication

While HTTPS would provide payload encryption and integrity, an attacker could still send their own HTTP POST requests to the API containing false student numbers. The attendance system requires that only the Raspberry Pis in each lecture room may communicate with the API, so an additional layer of security is necessary. TLS offers a possible solution to this issue in the form of client certificates. This provides a way for the API server to authenticate the client (the Raspberry Pi) and assure that only clients with a valid certificate can access the API. As these Raspberry Pis will be installed manually by the college, it is safe to assume that a client certificate may be loaded onto each one.

However, this solution is not optimal as it would require the certificate of each device to be updated manually when it expires or if it is revoked. In a college setting, where there may be a device in each one of possibly hundreds of lecture rooms, this solution will not scale efficiently. A more appropriate client authentication system may be a login-based system whereby each device would authenticate with the API using a unique login. This login could be configured once upon setup of the Raspberry Pi through some registration interface and could be managed remotely by the college with an administrator login. This would have the added benefit that if one device's login needed to be revoked or altered, it would not affect the rest of the devices. A full implementation of such a system (including roles, login management, etc.) is beyond the scope of this dissertation topic. However it is nonetheless an important aspect to consider in the design of the system.

5.2.3 API Denial-of-Service

The central API is a vital component of the attendance tracking system in order for it to function as intended. As a result, a successful "Denial-of-service" (DoS) attack or "distributed DoS" (DDoS) attack would impact the system as a whole. If the API were brought down, then the Raspberry Pis would be unable to update their schedules or to upload any attendance data they have recorded. The impact of such an attack could be reduced by a number of factors.

- Assuming that both the API server and each Raspberry Pi are located inside the college, the college could configure their network and firewall to restrict API access exclusively to traffic originating from inside the college. This will mitigate the effectiveness of an external DDoS attack.

- The attendance tracking software can be implemented with this as a consideration. Due to the repetitive nature of lecture schedules, the lecture schedule may be pulled infrequently, meaning the device does not need constant communication with the API in the event that it does go down. Similarly, any attendance data unable to be uploaded to the API could be stored locally and re-uploaded when the Raspberry Pi next detects that the API is functional.

- In extreme cases, a third party DDoS protection service such as Cloudflare [22] may be employed. However, if the aforementioned measures were implemented,

it may be unlikely that the setup and cost of Cloudflare will be warranted, given the reduced probability of such an attack.

5.2.4 Bluetooth Security

As the successful recording of student numbers via BLE is the core function of this system, it is important that this process is also secure and private. Similarly to the API communication portion of the system, this Bluetooth communication between the phone and the Raspberry Pi should be encrypted to prevent MITM attacks and for the student's identity to remain private.

BLE encrypts communication between two devices by exchanging a Temporary Key (TK) which both sides use to generate the actual encryption key called the Short Term Key (STK) [9]. The method with which the STK is generated is called the "pairing method", with each pairing method offering varying levels of security. The "Just Works" and "Passkey" pairing methods are both susceptible to MITM attacks where the attacker can brute force the TK and derive the STK. The "Out of Band" method uses another communication medium such as NFC in order to exchange the keys, which is MITM-resistant as long as the other band is. From BLE 4.2 onwards, the TK and STK are replaced with a Long Term Key (LTK) which is used to encrypt the connection. Elliptic-curve Diffie-Hellman was introduced in order to generate the LTK, providing added protection against MITM attacks [23]. Such an attack has a low probability of occurring in the context of this topic as it would require an attacker to be present in the lecture room and for the communication to be using one of the older, insecure pairing methods.

Recent versions of Android (6.0 onwards) also randomize the hardware address of the phone by default when using any of their Bluetooth APIs such as BLE advertising or scanning [24]. This ensures a student cannot be tracked from location to location based on their Bluetooth device address by making it appear as a different device each time the scanner is turned on.

5.3 Summary

This chapter presented an evaluation of the prototype implementation of the attendance tracking system. The system was not evaluated under a trial lecture scenario

due to time constraints so single device testing was performed to try and give an indication of its feasibility.

The system was shown to take an average of 1.51 seconds to read each student number. When extrapolated to estimate the time taken to record an entire class, the resulting ranges were acceptable. The individual system components were deployed and tested in an end-to-end evaluation of a full system cycle, with the components interoperating successfully. Upon experimentation, the system was found to have a poor detection rate of advertisement packets (roughly 50%). The most likely factor for this was deemed to be hardware limitations of the Raspberry Pi as the system performed significantly better using a more capable machine.

Some investigation into the smartphone application's power consumption was also carried out to assess how suitable BLE technology is for this use case. It was found to consume a relatively small amount of excess battery, due mainly to the implementation's ability to run without UI and with the screen disabled. Finally, some security considerations of the design were discussed, with particular attention being paid to their probability, potential impact, and mitigation techniques.

Chapter 6

Conclusion

6.1 Overview

This dissertation aimed to create a system for lecture attendance tracking that alleviated the flaws of existing systems by being seamless, easy to use, accurate, and scalable. A prototype was developed using an Android application capable of communicating with a small device in the lecture room using Bluetooth Low Energy. This system uses a combination of Bluetooth Low Energy Advertisements and Connections to record the student's ID number multiple times over the course of a lecture.

Through use of Android services, the attendance process occurs entirely in the background and does not distract the student by requiring manual interaction once started. This allows the system to meet the desired objectives of being seamless and easy to use, with minimal disruption to the student's learning.

The range of Bluetooth and the process of recording a student's presence multiple times during the lecture helps to improve upon accuracy flaws of existing systems both by verifying that a student was (a) in close proximity to the attendance device, and (b) present for the full duration of the lecture. In the current prototype, this could be circumvented by a student typing another student's number into the application. However, the intention is that the application would be integrated with a college's own student login systems so that a student may only log in once using their own credentials.

Due to time constraints, the system was not tested in a lecture scenario with large numbers of smartphones. While considerations were made (remote schedule configuration, sleeping to reduce congestion) to ensure the system could be scaled to perform

in such a scenario, it cannot be said definitively if the goal of system scalability was met. The system's performance was evaluated and its speed and power consumption were found to be acceptable. Detection of advertisement packets was found to be inconsistent on the chosen hardware but its potential was demonstrated on a more capable device. While some future work is needed, the initial work presented in this dissertation achieved most of its objectives and the resulting system demonstrates the feasibility of such an attendance tracking solution with some promising results.

6.2 Future Work

6.2.1 Hardware

Due to its detection of roughly 50% of advertisements, the Raspberry Pi Zero W is too inconsistent to be chosen as the hardware for the implementation of the Attendance Tracker component. More capable hardware will reduce the possibility of a student's attendance being missed by the system. A possible alternative could be its more powerful counterpart, the Raspberry Pi 3 b+, which also includes support for BLE 4.2. As discussed in section 4.1, the software of the Attendance Tracker will run on any Linux machine that supports BLE (including with a Bluetooth dongle), so this could broaden the hardware possibilities even further. It would allow a college to swap out or upgrade hardware as they see fit and also allow the lecturer to run the system on their own laptop if desired.

6.2.2 College Integration and Trial

A vital piece of future work would be to trial the system in a college lecture scenario. While the evaluation performed in the previous chapter can give an indication into the system's functionality, this is not a substitute for a larger scale test in a trial lecture. This will give more accurate insights into the system's speed, range, and overall performance when placed under the type of load that cannot be emulated by single device testing. Such a trial would be the only way conclusively determine the current implementation's scalability, making it perhaps the most important area of future work.

6.2.3 Bluetooth 5

Bluetooth 5 was released by the Bluetooth SIG in late 2016. This iteration of the wireless standard boasts a number of features which could vastly improve the Bluetooth LE communication portion of this dissertation. It introduces improved bandwidth, increased range, and larger advertising payloads. The new specification offers choices to developers to sacrifice some of these features in favour of large improvements to the features most relevant to their application. In the context of this system, the increased range seems an attractive feature for future work. Using the new "LE Coded" version of the PHY, the bit rate can be reduced to 125 kb/s in exchange for a 4x increase in range [25]. This would be a useful trade-off for the attendance tracking system as the amount of data transmitted is very small but with a large potential range required. Hardware support for Bluetooth 5 is still relatively uncommon, but current iterations of flagship Android phones such as the Samsung Galaxy S9 and OnePlus 6 are currently being released with support for the specification.

Appendix A

Annotated Dockerfile Example

The **Dockerfile** used to by the AWS EC2 instance to configure and install the Attendance and Scheduling API. The general process is as follows, with each line explained with a comment. The result is a Docker container running a publicly accessible instance of the API.

```
# use a pre-configured Linux image with Python 3.6.1 installed
FROM python:3.6.1

# set working directory of the application
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# add Pip requirements (to leverage Docker cache)
ADD ./requirements.txt /usr/src/app/requirements.txt

# install requirements/dependencies using Pip
RUN pip install -r requirements.txt

# add app to working directory
ADD . /usr/src/app

# run server and start app
CMD python -u manage.py runserver -h 0.0.0.0
```

Appendix B

Android BLE GATT Server Callback

The Bluetooth GATT Server callback implemented by the Android application. This abstract class handles read requests from the AttendanceTracker during a BLE Connection.

```
private BluetoothGattServerCallback gattServerCallback = new BluetoothGattServerCallback() {

    @Override
    public void onCharacteristicReadRequest(BluetoothDevice device, int requestId, int offset,
                                            BluetoothGattCharacteristic characteristic) {

        // Attendance tracker has requested the Student Number characteristic
        if (characteristic.getUuid().equals(StudentAttendanceProfile.STUDENT_NUMBER)) {
            boolean success = gattServer.sendResponse(device,
                    requestId,
                    BluetoothGatt.GATT_SUCCESS,
                    0,
                    studentNumber.getBytes());

            // Student number has been successfully read by the scanner
            if (success && numberOfReads < 2) {
                numberOfReads++;

                if (numberOfReads == 1)
                    sleepService();
                else if (numberOfReads == 2)
                    stopSelf();
```

```
            }

        } else {
            Log.d(LOG_TAG, "Invalid characteristic");
            gattServer.sendResponse(device,
                    requestId,
                    BluetoothGatt.GATT_FAILURE,
                    0,
                    null);
        }
    }
};
```

Bibliography

[1] Bluetooth SIG. Radio versions: The right radio, for the right job. `https://www.bluetooth.com/bluetooth-technology/radio-versions`.

[2] Bolivar A. Senior. Correlation between absences and final grades in a college course. In *International Proceedings of the 44th Annual Conference*. Associated Schools of Construction, 2008.

[3] Anna Lukkarinen, Paula Koivukangas, and Tomi Seppl. Relationship between class attendance and student performance. *Procedia - Social and Behavioral Sciences*, 2016.

[4] H. Paul LeBlanc III. The relationship between attendance and grades in the college classroom. *17th Annual Meeting of the International Academy of Business Disciplines*, 2005.

[5] D. Deugo. Using beacons for attendance tracking. In *FECS'16 - The 12th International Conference on Frontiers in Education: Computer Science and Computer Engineering*. Worldcomp, 2016.

[6] Bluetooth SIG. The rise of beacon technology, 2017. `https://blog.bluetooth.com/the-rise-of-beacon-technology`.

[7] Ben Lynch. Collaborative attendance tracking using bluetooth low energy. Master's thesis, University of Dublin, Trinity College, 2017.

[8] Robin Heydon. *Bluetooth Low Energy: The Developer's Handbook*. Prentice Hall, 2012.

[9] Kevin Townsend, Carles Cufi, Akiba, and David Robertson. *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. O'Reilly, 2014.

[10] Microchip Technology Technical Training Group. Bluetooth low energy physical layer. http://microchipdeveloper.com/wireless:ble-phy-layer.

[11] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 2012.

[12] Kevin Townsend. Introduction to bluetooth low energy, 2015. https://learn.adafruit.com/introduction-to-bluetooth-low-energy.

[13] Bluetooth SIG. Gatt overview. https://www.bluetooth.com/specifications/gatt/generic-attributes-overview.

[14] Python Software Foundation. 17.1. threading thread-based parallelism, 2018. https://docs.python.org/3/library/threading.html#timer-objects.

[15] Ian Harvey. The defaultdelegate class - bluepy 0.9.11 documentation, 2014. http://ianharvey.github.io/bluepy-doc/delegate.html.

[16] SQLite. About sqlite. https://www.sqlite.org/about.html.

[17] Miguel Grinberg. *The New And Improved Flask Mega-Tutorial*. Independently Published, 2017.

[18] Apple Inc. *Accessory Design Guidelines for Apple Devices, Release R5*, 2018.

[19] Bikash Shrestha. Measurement of power consumption of ble (bluetooth low energy). Master's thesis, Helsinki Metropolia University of Applied Sciences, 2016.

[20] Sandeep Kamath and Joakim Lindh. *Measuring Bluetooth Low Energy Power Consumption*. Texas Instruments.

[21] Kenneth Reitz. Advanced usage: Ssl cert verification - requests documentation, 2018. http://docs.python-requests.org/en/master/user/advanced/#ssl-cert-verification [Accessed: 24th April 2018].

[22] Cloudflare. Protect against ddos attack, 2018. https://www.cloudflare.com/ddos/ [Accessed: 24th April 2018].

[23] Matthew Bon. A Basic Introduction to BLE Security - EEWiki, 2018. https://eewiki.net/display/Wireless/A+Basic+Introduction+to+BLE+Security [Accessed: 26th April 2018].

[24] Google. Android 6.0 changes - android developers, 2018. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-hardware-id [Accessed: 25th April 2018].

[25] Bluetooth SIG. *Bluetooth Core Specification - v5.0*, 2016.