TÉCNICO LISBOA

BikeApp - Detecting Cyclists Activity and Location using Bluetooth Low Energy Technology

Andriy Zabolotnyy

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Dr. Paulo Jorge Pires Ferreira

Examination Committee

Chairperson: Prof. Dr. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Dr. Paulo Jorge Pires Ferreira
Member of the Committee: Prof. Dr. Miguel Filipe Leitão Pardal

November 2017

Acknowledgments

Firstly, I would like to thank my thesis supervisor Professor Paulo Jorge Pires Ferreira, for the opportunity to work in this project and the guidance provided during the time of research and writing of this thesis.

Second, I would like to thank my parents and my sister for their love and for all the support given.

Last but not the least, I would like to thank my friends, for all the fun we have had in the last years.

Abstract

In urban environments, cycling may help to solve traffic delays, interruptions and parking, while at the same time being a cleaner and healthier mean of urban transportation. Several approaches are being taken to increase cycling in urban areas. One such approach is taking an advantage of the availability of smartphones and developing applications that somehow improve and reward the cycling activity. To promote and motivate cycling in cities we propose *BikeApp*, a cycle rewarding smartphone application. *BikeApp* detects when the user starts cycling and makes him eligible for rewards that can be claimed at the shops. When the user gets close to or enters a shop, *BikeApp* detects it and allows the user to easily claim the benefit. The implementation focuses on cycling and location detection mechanisms, and its integration within the application. Both are implemented using Bluetooth Low Energy (BLE) technology. Our *BikeApp* application, implemented targeting the iOS platform, can accurately detect the cycling activity using a BLE bicycle-mounted sensor and determine the shop the user enters using BLE location beacons. The location detection shows to be effective even in worst-case scenarios, with location beacons located in side to side shops. We also tested *BikeApp* in terms of its power consumption, and there is no significant overhead introduced with the BLE communication.

Keywords

(BLE, Bluetooth Low Energy, GATT, App, iOS)

Resumo

Ciclismo em meio urbano pode ajudar a resolver problemas de trânsito, problemas de estacionamento e ser ao mesmo tempo, um meio de transporte ecológico e benéfico. Várias abordagens estão a ser seguidas para aumentar o ciclismo em meios urbanos. Uma dessas abordagens tira partido da grande disponibilidade de smartphones e propõe aplicações que de alguma forma melhorem e recompensem o ciclismo. Para promover e motivar o ciclismo em cidades, propomos BikeApp, uma aplicação que recompensa o ciclismo. BikeApp detecta quando o utilizador anda de bicicleta e torna-o elegível para benefícios em lojas. Quando o utilizador se aproxima ou entra em certas lojas, BikeApp detecta a loja e permite que o utilizador reivindique facilmente o seu benefício. A implementação da nossa solução foca-se em mecanismos de detecção de ciclismo e de localização, e na integração dos mesmos na aplicação. Ambos os mecanismos são implementados usando a tecnologia BLE. A nossa aplicação, implementada para iOS, detecta com exatidão a actividade de ciclismo usando um sensor BLE instalado na bicicleta, e determina a localização em loja usando *beacons* BLE de localização. A detecção de localização mostra ser eficiente até em piores cenários, com duas lojas adjacentes. BikeApp foi também testado em termos de consumo de energia e este não é substâncial.

Palavras Chave

(BLE, Bluetooth Low Energy, GATT, App, iOS)

Contents

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Goals . . . 3

1.3 Current Solutions . . . 5

1.4 Contributions . . . 5

1.5 Document Structure . . . 5

2 Related Work **7**

2.1 Introduction . . . 9

2.2 WiFi . . . 9

2.2.1 Discovery and RSSI retrieval . . . 9

2.3 Classic Bluetooth . . . 10

2.3.1 Discovery and RSSI retrieval . . . 10

2.3.2 Range . . . 11

2.4 Bluetooth Low Energy . . . 11

2.4.1 Advertising and RSSI retrieval . . . 12

2.4.2 Discovery and RSSI retrieval . . . 12

2.4.3 Range . . . 12

2.4.4 Generic Attributes (GATT) . . . 13

2.5 Indoor Positioning . . . 14

2.5.1 Global Positioning Systems (GPS) . . . 14

2.5.2 Proximity detection . . . 15

2.5.2.A Radio-frequency identification (RFID) . . . 15

2.5.2.B Near Field Communication (NFC) . . . 15

2.5.2.C WiFi and Bluetooth. . . . 16

2.5.3 Triangulation and trilateration . . . 16

2.5.3.A Time-based distance estimation. . . . 17

2.5.3.B Radio propagation model-based distance estimation. . . . 18

2.5.3.C Log-distance path loss model. . . . 18
2.5.3.D Fingerprinting . . . 19
2.6 Activity Recognition . . . 21
2.6.1 Supervised learning . . . 21
2.6.2 Mobile device and body-worn sensors . . . 22
2.6.3 Bicycle-mounted sensor . . . 23
2.6.4 Commercially available sensors . . . 24
2.7 Summary . . . 24
3 Solution **29**
3.1 Introduction . . . 31
3.2 Solution Overview . . . 31
3.2.1 Web application . . . 32
3.2.2 Mobile application . . . 32
3.2.3 Backend . . . 34
3.2.4 Cycling and location detection using BLE devices . . . 34
3.3 Architecture of Biklio . . . 35
3.4 Architecture of BikeApp . . . 37
3.4.1 Cycling detection . . . 38
3.4.2 Location detection . . . 41
3.5 Summary . . . 43
4 Implementation **45**
4.1 Introduction . . . 47
4.2 Development environment . . . 47
4.2.1 Equipment . . . 48
4.3 Project Structure . . . 48
4.3.1 Module communication . . . 48
4.4 BLE Manager module . . . 49
4.4.1 BLE Manager platform-specific implementation . . . 51
4.4.2 Scanning . . . 51
4.4.3 Communication with the sensor . . . 53
4.5 Cycling Detector module . . . 55
4.6 Beacon Monitor module . . . 57
4.6.1 Beacon protocol . . . 58
4.6.2 Beacon Monitor platform-specific implementation . . . 59
4.6.3 Beacon discovery time . . . 60

4.7 Location Detector module . . . 60
4.8 Summary . . . 62

5 Evaluation **65**
5.1 Introduction . . . 67
5.2 Advertising Intervals and Discovery Times . . . 67
5.2.1 Methodology . . . 68
5.2.2 Results . . . 68
5.3 Estimated vs real distance . . . 68
5.3.1 Methodology . . . 69
5.3.2 Results . . . 70
5.4 Location Detection . . . 72
5.4.1 Methodology . . . 73
5.4.2 Results . . . 73
5.5 Cycling Detection . . . 75
5.5.1 Methodology . . . 75
5.5.2 Results . . . 76
5.6 Smartphone Battery Consumption . . . 76
5.6.1 Methodology . . . 76
5.6.2 Results . . . 77
5.7 Usability Evaluation . . . 78
5.7.1 Results . . . 78
5.8 Summary . . . 81

6 Conclusion **83**
6.1 Future Work . . . 86

A B'TWIN Sensor Instructions **93**

List of Figures

2.1 GATT scheme . . . 13
2.2 Angulation based positioning . . . 17
2.3 Positioning based on circular lateration (trilateration) . . . 17
2.4 Fingerprinting Online and Offline phases (adapted from Li et al. [1]) . . . 20

3.1 System overview of BikeApp . . . 31
3.2 Available Spots presentation . . . 33
3.3 Location detection notification and benefit claim . . . 33
3.4 Complete system overview of *BikeApp* containing BLE devices . . . 35
3.5 Biklio modules . . . 36
3.6 BikeApp modules . . . 38
3.7 Sensor managing . . . 39
3.8 Region defined with a shared UUID, and unique Major and Minor . . . 42

4.1 BLE bicycle sensor and location beacons . . . 48
4.2 IBLEManager interface . . . 50
4.3 Raw advertised data structure . . . 53
4.4 ICyclingDetector interface . . . 56
4.5 IBeaconMonitor interface . . . 57
4.6 iBeacon and Eddystone-UID protocol packet structure . . . 58
4.7 ILocationDetector interface . . . 61

5.1 Beacon discovery times with 100 ms Advertising Interval . . . 69
5.2 Beacon discovery times with 300 ms Advertising Interval . . . 69
5.3 Beacon placement and RSSI retrieval . . . 70
5.4 Real and estimated distances at 0 dBm *transmission power* . . . 71
5.5 Real and estimated distances at -12 dBm *transmission power* . . . 71
5.6 RSSI at -12 dBm *transmission power* . . . 72

5.7 RSSI at 0 dBm *transmission power* 72
5.8 Shops and location beacons placement 73
5.9 Location detection starts few meters away of both shops before entering shop B 74
5.10 Location detection starts upon entering shop B 74
5.11 Characterization of the users 78
5.12 Question 1: Spot presentation rating 79
5.13 Question 2: Spot presentation rating 79
5.14 Question 3: Bicycle sensor installation 79
5.15 Question 4: Bicycle sensor pairing 80
5.16 Question 5: Benefit claim easiness 80
5.17 Question 6: Shop detection usefulness 80
5.18 Question 7: Overall application usability 81
5.19 Question 8: Required Bluetooth connectivity opinion 81

List of Tables

2.1 Bluetooth transmitted output and range by class 11
2.2 Overview of positioning techniques, with numeric scale from 1 (*Very Low*) to 5 (*Very High*) 27

3.1 Example of a beacon-shop correspondence table 43

5.1 Battery consumption when smartphone is idle 77
5.2 Battery consumption when cycling 77

Acronyms

AoA	Angle of Arrival
AP	Access Point
API	Application Programming Interface
BLE	Bluetooth Low Energy
CSC	Cycling Speed and Cadence
FHSS	Frequency-Hopping Spread Spectrum´
GATT	Generic Attributes
GPS	Global Positioning Systems
HCI	Host Controller Interface
ISM	Industrial Scientic and Medical
MAC	Media Access Control
NFC	Near Field Communication
PC	Personal Computer
PCL	Portable Class Library
PDA	Personal Digital Assistant
RPM	Revolutions per minute
RFID	Radio-frequency identification
RSSI	Received Signal Strength Indicator
SDK	Software Development Kit

SIG Special Interest Group

STA Station

SSID Service Set Identifier

TDoA Time Difference of Arrival

ToA Time of Arrival

UI User Interface

UML Unified Modeling Language

UUID Universally Unique Identifier

WLAN Wireless Local Area Networks

1

Introduction

Contents

1.1 Motivation 3
1.2 Goals 3
1.3 Current Solutions 5
1.4 Contributions 5
1.5 Document Structure 5

1.1 Motivation

The use of a bicycle as a regular mode of transport in the urban environment is constantly increasing, as it may solve a lot of problems such as traffic delays, interruptions, parking, and also contributes to the reduction of greenhouse gas emissions and brings health benefits. Great effort was put into cycling promotion over the past years, for example, expansion of bicycle lane network and bicycle racks across the city.

Currently, a smartphone is a widely used communication device, with a rich set of built-in sensors such as an accelerometer, digital compass, gyroscope, GPS and camera. These sensors enable new applications across multiple domains, such as healthcare, social networks, safety, etc [2].

With the availability of powerful smartphones with embedded sensors many personal sensing applications appeared, focused on data collection and analysis. A typical scenario is tracking the user's exercise routines and encourage the user to reach his personal fitness goals. Most of the cycling applications follow the same idea of being a personal fitness trainer, but they also have a goal of contributing to the cycling promotion.

Endomondo [3] and KAPPO [4] are two examples of available applications that motivate users to cycle. Endomondo is a fitness tracker and personal trainer, providing a large set of interesting features like tracking routes with GPS, allowing users to set cycling goals and guiding the user with an audio coach to achieve them. KAPPO uses gamification instead, where users play a multi-player cycling game. This application turns the cycling routine into a game and motivates the users to cycle even more. Both applications have social aspects, making them communication, competition and sharing platforms between its users.

1.2 Goals

The goal of this project is to design, implement and evaluate a smartphone cycling application to encourage citizens to ride their bikes. The application provides users a way of being rewarded for cycling, with benefits (e.g. discounts) at shops that join the rewarding program and also benefit from cycling customers.

To guarantee correct rewards, the application needs to detect the cycling activity, and determine the exact user location, to ensure that a user is within a given shop. Indoor localization is challenging, because accurate GPS positioning is not possible inside the buildings (indoor spaces), and even for outdoor location the accuracy of GPS is sometimes not enough, so alternative techniques should be used. Activity detection is also problematic and several challenges arise, such as battery consumption, response time or strong requirements of wearable devices.

The solution must fulfill the following functional requirements:

- ***Detect cycling activity***
- ***Detect indoor locations***: detect user's presence within a certain shop
- ***Fully offline***: require no Internet connection, consequently all the computations should be performed locally at the mobile device, without any computation on a centralized server side. Opportunistic Internet connection updates the data stored on the device, required for offline operation.

Regarding the non-functional requirements, our solution must fulfill the following:

- ***High accuracy and precision on cycling activity detection***
- ***High accuracy and precision on indoor location detection***
- ***Scalable location detection***: positioning coverage area should be easily extended and do not limit simultaneous usage by multiple users
- ***Ease of deployment***: system should be easily deployed and without complex and time consuming initial configuration
- ***Minimal maintenance***: require minimal future maintenance or calibration
- ***User-friendly and minimally intrusive***: easy to use and require minimal user input
- ***Low battery consumption***: used technologies should be chosen taking into consideration smartphone's and additional device's power consumption
- ***Responsiveness***: user should receive quick feedback

Our proposed solution named *BikeApp*, is a mobile application intended to be used by cycling users. The application shows the user nearby shops where the user can receive benefits for cycling. To detect the cycling activity, *BikeApp* uses a BLE bicycle-mounted sensor. After cycling to a given shop, *BikeApp* is capable of detecting the shop and prepares the User Interface (UI) to allow the user easily claim his benefit. The location detection uses BLE location beacons deployed at the shops.

This thesis is integrated within a *EU Horizon 2020* project named *TRACE* [5][6], which has a goal of assessing the potential of movement tracking services to better plan and promote walking and cycling in cities.

BikeApp implementation uses *Biklio*[7] as its base project. *Biklio* is a production application of the *TRACE* project, with the previously described cycling rewarding capabilites, but with alternative and less accurate cycling and location detection mechanisms. Thus, our main focus is the implementation of the cycling and location detection mechanisms using BLE devices, and its integration within *Biklio*.

1.3 Current Solutions

Currently, there are several indoor positioning techniques. However, many of them are not applicable to our solution, because of their particularities which conflict with our system requirements. The following enumeration specifies some of such techniques, together with the reasons why they are not suitable:

- **NFC -** not all smartphone models support NFC (see Section 2.5.2.B);
- ***Proximity detection using Wireless Local Area Networks (WLAN) -*** may have poor accuracy and requires periodic calibration (see Section 2.2)
- ***Fingerprinting -*** complex initial setup and high computation complexity (see Section 2.5.3.D)

Regarding the cycling detection, there are also some existing solutions, but most of them use the accelerometer data collected from the smartphone or some wearable sensors, and the cycling activity is inferred using supervised learning (see Section 2.6.1). This approach requires heavy computations in a continuous fashion, which consumes a lot of battery power and does not meet our requirement.

1.4 Contributions

This work makes the following contributions:

- Architectural design of the cycling detection mechanism using BLE bicycle sensors;
- Architectural design of the location detection mechanism using BLE location beacons;
- Implementation of both mechanisms;
- Integration of the mechanisms within *Biklio*, resulting in a new application version called *BikeApp*;
- Evaluation of configurable BLE beacon's parameters to determine the optimal system configuration to meet our requirements;
- Evaluation of the cycling and location detection mechanisms;
- Evaluation of the usability of *BikeApp* with end-users.

1.5 Document Structure

The rest of this document is organized as follows. Chapter 2 provides a background on existing location and cycling detection techniques, detailedly analyzing its applicability within our solution. Chapter 3 starts by providing a solution overview, then introduces the architecture of *Biklio* and *BikeApp*, together

with the cycling and location detection mechanism explanations. Chapter 4 addresses the implementation details of our proposed solution. Chapter 5 covers the taken evaluation process. Finally, Chapter 6 concludes this work and presents the future work.

2

Related Work

Contents

2.1 Introduction 9
2.2 WiFi 9
2.3 Classic Bluetooth 10
2.4 Bluetooth Low Energy 11
2.5 Indoor Positioning 14
2.6 Activity Recognition 21
2.7 Summary 24

2.1 Introduction

This section overviews the state-of-the-art of the project's main features: location detection (indoor positioning) and cycling activity detection. Although analyzed separately, the project results in a single system supporting those previously mentioned features. This means that technologies used to implement each of them should be compatible. Related work addressed is based on projects implemented with off-the-shelf equipment, thus excluding any systems deployed with custom-made devices.

WiFi and Bluetooth are both ubiquitously available technologies integrated in any smartphone. Consequently, several positioning studies appeared using WiFi or Bluetooth transmitter's Received Signal Strength Indicator (RSSI). Important WiFi and Bluetooth insights related to the following sections, are initially discussed. Bluetooth is divided in *classic* Bluetooth and BLE, where *classic* refers to the Bluetooth before the BLE version.

2.2 WiFi

WiFi[8] is a technology that allows electronic devices to connect to a WLANs. WLAN infrastructures are based on IEEE 802.11 standards. Multiple frequency bands are supported but mostly 2.4 GHz Industrial Scientic and Medical (ISM) band is used. Connected components to the wireless network are called Station (STA) and are identifiable by unique Media Access Control (MAC) address. STA can be either clients or Access Point (AP). APs are base stations for the wireless network that facilitate the communication between clients within the network.

WLAN became ubiquitously available in commercial and home settings, over the past two decades. Computing devices such as personal computers and smartphones have WLAN capabilities as well, so an advantage can be taken of already and widely deployed APs in urban areas to build cost-effective indoor positioning systems.

2.2.1 Discovery and RSSI retrieval

WLAN positioning works using beacon exchanges between mobile client and APs. Beacon exchange can be performed in two ways: active scanning and passive scanning. In passive scanning mode the client station moves the radio into each channel and waits to listen for periodic (typically 100ms) *beacon frame* advertisements by surrounding APs, containing their information. In alternative scanning mode, the client sends *probe request frames* on each channel. These probe requests may contain Service Set Identifier (SSID) of a specific WLAN that the station is looking for or the probe requests can also do a full scan to find out all the SSIDs in the proximity of the client. Nearby APs respond to probe requests that they receive on a channel they work with *probe response frames*, the contents of which are similar

to beacon frames [9].

Trilateration and *fingerprinting* indoor positioning techniques, addressed in Section 2.5, use a signal parameter that correlates with the distance. The RSSI correlates with the distance and represents how well a device can hear a signal from an AP. It can be retrieved from beacon frame or probe response sent by APs.

2.3 Classic Bluetooth

Bluetooth[10] is a short range wireless communication technology standard, operating over 2.4-2.485 GHz ISM band. Bluetooth technology is embedded in most digital devices, including all mobile devices, due to its low cost, short range and low power wireless transmissions using secure protocol between electronic devices. This section overviews Bluetooth particularities for devices before BLE introduction in Bluetooth 4.0.

Bluetooth follows a master-slave structure, where a device can be either a slave or a master, which can support a network (known as *piconet*) with up to seven active slaves. A slave can only be connected to one master, and a master can send data to any of its slaves and request data from them as well, but slaves cannot talk to other slaves in the piconet.

Since there are several other technologies (including WiFi) using the 2.4 GHz band, to avoid interference Bluetooth uses a radio technology called Frequency-Hopping Spread Spectrum (FHSS), a method of transmitting radio signals by rapidly switching a carrier among many frequency channels, using a pseudorandom sequence known to both transmitter and receiver[10].

2.3.1 Discovery and RSSI retrieval

A pairing is necessary before the data transmission takes place between the devices. To initiate the pairing the master performs a discovery procedure called inquiry, to detect nearby Bluetooth devices. During this procedure, inquiry messages are broadcast to Bluetooth devices in the proximity and devices in discoverable mode, that are listening for such a request and allow a connection to be established, respond to that inquiry with its information. Having this information, a master can initiate a pairing and establish a connection.

The inquiry procedure is lengthy and can even reach a duration of 10.24 seconds [10], which slows down the performance of applications that require quick update of nearby Bluetooth devices (for example real-time positioning or tracking applications). Naya et al. [11] proposed a system capable of quicker device discovery, compared to standard inquiry time. This was achieved through direct manipulation of inquiry parameters in Host Controller Interface (HCI) [10], but those parameters are not manipulable in smartphones from user applications, so the problem of long device scanning remains.

Table 2.1: Bluetooth transmitted output and range by class

Class	Max. transmitted power	Theoretical range
1	20 dBm	100 m
2	4 dBm	10 m
3	0 dBm	1 m

Bluetooth has two types of RSSI: connection-based and inquiry-based. The first measures RSSI of data packets after a connection is established, while inquiry-based measures the RSSI of inquiry packets in inquiry mode during device discovery. Subhan et al. [12] experimentally analyzed and came to the conclusion that inquiry-based RSSI correlates better with the distance than connection-based RSSI. Additionally, inquiry-based RSSI does not limit the simultaneous positioning of several mobile devices, because Bluetooth locator devices can respond to any device that sends an inquiry message. In case of connection-based RSSI, once a Bluetooth locator device is in connection with a mobile device (master), it can not be in connection with another mobile device. This would limit the positioning to only one device at a time.

2.3.2 Range

The range of transmitters, that represents the distance at which the emitted signal is available, is an important aspect in positioning systems.

Bluetooth devices are divided in three classes, where each class specifies the maximum signal output power and the corresponding theoretical range. The effective range that can be achieved depends on signal propagation, that can be affected by several factors, especially in indoor environments. Thus, the effective range in most cases is lower than the theoretical.

2.4 Bluetooth Low Energy

In 2010 a new variation of Bluetooth technology appeared in Bluetooth 4.0 version, called Bluetooth Low Energy (BLE). BLE introduced significant changes compared to the classic Bluetooth (previous versions). BLE is aimed at peripheral devices which operate on batteries, do not require high data rates or constant data transmission, such as proximity sensors, heart rate monitors, fitness devices. Furthermore, application implementation complexity is reduced with BLE Application Programming Interface (API)s and developer support resources provided by the mobile device platforms and BLE manufacturers[13][14][15][16].

An important improvement to mention in BLE over classic Bluetooth is its coexistence with the WiFi, which also operates in the same frequency band and both can interfere. The broadcasting of advertise-

ments occurs on three channels chosen to not collide with the three most commonly used WiFi channels, which was experimentally confirmed by Silva et al.[17].

2.4.1 Advertising and RSSI retrieval

One of the most important features of BLE is its periodical advertisement of small packets that mobile devices can periodically listen to. The periodicity of those advertisements can be configured in a range of 20 ms to 10.24s [18]. BLE devices wake up from sleep mode, where they reside most of the time, to advertise and go back to sleep mode, decreasing significantly their power consumption. The advertised packet contain BLE beacon's Universally Unique Identifier (UUID) and other useful information, such as battery level or some manufacturer specific sensor data, which depends on the peripheral type.

Retrieved RSSI values from received advertising packets has good correlation to the distance and can be used to estimate the distance to the BLE device using a radio propagation model (see Section 2.5.3.B).

2.4.2 Discovery and RSSI retrieval

Discovery of new devices in proximity is done with a scanning procedure. In BLE, scanning can be either active or passive. In active mode, a mobile device sends a *Scan Request* packet to which a BLE device responds with a *Scan Response*. In passive mode, a mobile device just listens to advertised packets that are periodically sent by nearby BLE devices.

Scanning performed on mobile devices can also be configured by setting the scan window duration (T_w) and the scan interval (T_i) parameters, putting a mobile device in scanning state for T_w every T_i. Short scanning times and long intervals between the scans results in a low average power consumption but also limits the probability of discovering the BLE devices; so, an appropriate advertising period value should be chosen to ensure that a BLE device is discovered by a mobile device. Advertising and scanning duty cycles set a burden of power consumption of the mobile device and BLE peripherals , so they need to be chosen carefully[19]. Adjusting the scanning, mobile devices can perform a faster periodical discovery of BLE devices, which plays an important role in applications needing continuous and quick update of nearby BLE devices.

2.4.3 Range

BLE devices are not divided in classes like classic Bluetooth. Existing BLE beacons on the market have typical theoretical range up to 200 meters, which directly depends on device's *Transmission Power* (also known as *Tx Power)*. In classic Bluetooth devices range depends on the class they belong to, as in Table 2.1. In BLE, *Transmission Power* parameter can be totally configurable, setting a desired device's

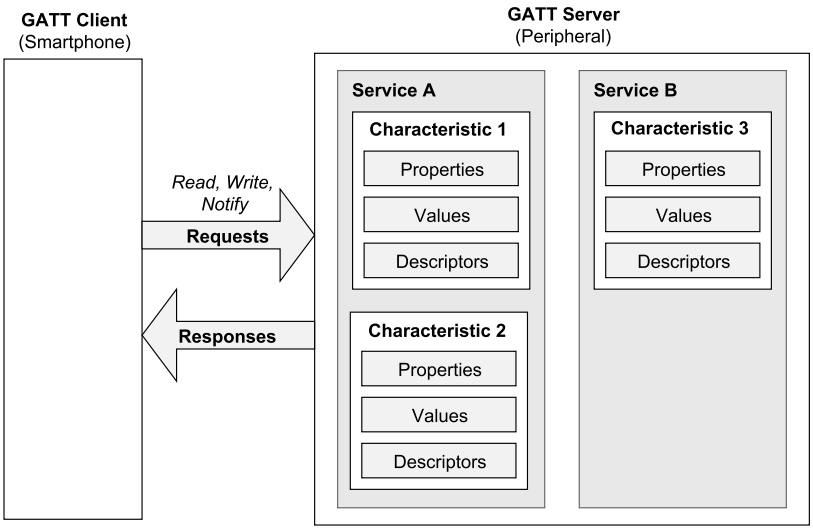


Figure 2.1: GATT scheme

range. Logically, greater *Transmission Power* consumes more battery power, so its value should be chosen wisely, considering the desired range at which a beacon should be visible.

2.4.4 GATT

The communication between BLE devices is based on Generic Attributes (GATT) [20]. GATT cover precise device functionalities and roles, and ensure interoperability between devices from different vendors, by specifying the way that two BLE devices transfer data. This data is organized in hierarchical data structure containing *Profile*, *Service* and *Characteristic* entities.

The top level of the hierarchy is a *Profile*, which has at least one *Service* to fulfill a use case. A *Service* can have one or more *Characteristics*. A *Characteristic*, the lowest level concept in GATT, consists of a set of *properties* indicating the operations the characteristic supports and a set of permissions relating to security, a *value* and optionally one or more *descriptors*, which contain manufacturer specific metadata or configuration flags. Figure 2.1 shows the structure of the data. This lookup data is stored at the GATT server (peripheral device). A GATT client (central device) can perform three requests to the GATT server: *read*, *write* and *notify*. A *read* request asks the peripheral to send back the current value of a characteristic. A *write* request modifies the value of the characteristic, often used as commands, for example telling a peripheral to turn something on. Finally, a *notify* asks the peripheral to continuously send updated values of the characteristic, without the central having to constantly ask for it. [20][21][22].

Bluetooth Special Interest Group (SIG)[23] contains a large list of GATT standard services supported by BLE devices. This contributes to the interoperability between the devices, for example a fitness smartphone application can use any BLE heart rate monitor sensor, since all of them they offer the required *Heart Rate Service*.

2.5 Indoor Positioning

Indoor positioning systems are used to locate people and objects within buildings and closed environments. Indoor localization has become very important and crucial in a large number of applications, designated by *Location Based Services*, that depend on the user's location to provide information, services or control features [24].

Obtaining indoor position information is not easy, as several issues exist, such as presence of moving people, obstacles that cause high wireless signal attenuation. Consequently, many positioning systems were designed using alternative technologies over the years, and several survey articles were written that overview and provide better understanding about existing wireless indoor location estimation mechanisms [25, 26, 27, 28].

There is no single solution that works well in all scenarios, being the aim of this section to analyze existing solutions and technologies used according to the requirements specified in Section 1.2.

Wireless location technologies can be either *mobile-based* (also known as *terminal-based*) or *network-based*. In *mobile-based* location systems the location is calculated by the mobile device using signals received from transmitters. The second approach, called *network-based* [29], relies on existing networks, for example WLAN or *cellular network*, whose stations measure mobile device's signal parameters and send to a central server for further processing and mobile device location estimation. The latter approach has also an advantage of mobile device not being involved in location calculation. However, since we need the location estimation information on mobile devices and a completely offline positioning system, not requiring any Internet connection, the *mobile-based* approach needs to be followed.

This section describes in detail possible positioning techniques and analyzes its applicability in indoor environments, taking into account our requirements.

2.5.1 GPS

GPS is the most well-known positioning technology used that provides geolocation to a GPS receiver in any weather conditions and anywhere in the world, as long there is a unobstructed line-of-sight to at least four satellites. GPS receivers intercept information periodically transmitted by satellites, about its position and current time, and perform a calculation of the distance to each of those satellites. Once the distance to each of satellites is known, GPS receiver can identify its physical location employing *trilateration* technique (see Section 2.5.3).

However, closed indoor environments affect satellite signal propagation, making less satellites being "visible". Low satellite coverage results in a decreased location accuracy, or even in an impossibility of detecting the location, in case of not satisfying the four satellite signal reception requirement. This makes GPS totally unsuitable for indoor location estimation.

2.5.2 Proximity detection

Proximity detection is considered the simplest method to estimate mobile device location. This technique determines the position of the mobile device based on existing wireless transmitters at known locations. The idea behind this method is that radio transmitters have a limited range and if a receiver, smartphone in our case, receives signal from a transmitter, it is within the coverage range of that transmitter.

2.5.2.A RFID

RFID technology allows automatic identification and tracking of tags, that contain electronically stored information. Typical RFID systems contain a computer or a mobile device equipped with an RFID reader, and tags (i.e., transceivers) that can be either passive or active.

Passive tags have no internal power source, so they wait for a signal from an RFID reader and use its electromagnetic energy to power their internal circuit and transmit their ID and data to the RFID reader. Active tags have built-in batteries that makes them energy independent and being able to periodically and autonomously broadcast their own signal, that can be captured by RFID readers. Passive tags are smaller, cheaper and do not require any battery replacement, but have lower read distance range when compared to active ones that can reach tens of meters, due to its own power source.

2.5.2.B NFC

NFC is a wireless technology that allows NFC enabled devices communicate when bringing them close, typically up to 5cm apart. Being a specialized subset within the family of RFID technology, makes it possible to read some kinds of passive RFID tags as well.

Mainetti et al. [30] categorized RFID localization in reader localization and tag localization, depending on what needs to be localized. In reader localization the indoor environment, in which we want to locate, needs to be deployed with RFID tags containing location information. A person with a hand held reader reads the closest tag and obtains the physical location within that indoor space. In case of tag localization, several RFID readers are covered over indoor environment instead. Logically tag localization is more expensive than reader localization, as RFID readers are considerably more expensive than tags.

Reader localization can be a implemented using a NFC enabled mobile device as reader and a covered indoor environment with location information RFID passive tags, but it will not satisfy the requirement of not requiring any user interaction to detect its indoor location. Since NFC is short-ranged, it will require the user to manually read the RFID passive tag. Another drawback is the fact that not all mobile devices have NFC technology, that would limit the positioning system to a set of NFC enabled

smartphone models only.

2.5.2.C WiFi and Bluetooth.

It is also possible to apply this technique using WiFi or Bluetooth wireless transmitters covered within the indoor space. A user scans the available wireless devices, using a receiver, and RSSIs are collected. Users's position is set to the position of the transmitter with the strongest emitting signal, holding the highest RSSI value. Assuming that the wireless signal power decreases as the distance between transmitter and the receiver increases, the transmitter with the strongest signal is the nearest to the user.

2.5.3 Triangulation and trilateration

Triangulation can be defined as the process of determining the location of a point by forming triangles to it from known reference points. It can either direction-based or distance-based (known as *trilateration*).

Triangulation using Angle of Arrival (AoA) determines the position of a mobile device determining the angle of incidence at which signals arrive at the receiver. Figure 2.2 illustrates an AoA positioning, where *X* represents a user with a mobile device and *A* and *B* are two wireless receivers. The receivers capture the signal emitted by a mobile device, and determine the angle of incidence (θ_A and θ_B) of the received signal. Finally, the position of the mobile device is determined as the intersection of several pairs of angle direction lines, between the wireless transmitter and the receiver.

The disadvantages of AoA approach is that it requires specialized hardware to measure the angle of arrival at the receiver, by having directional antennas or antenna array. Schüssel [31] evaluated a system of tracking the location of a user with a smartphone using AoA calculation at WiFi APs. Quuppa [32] is a commercially available locating system that also uses signal AoA. Quuppa uses advanced antennas capable of measuring the radio signal transmitted by a tag and sends this information to a server, referenced as *Quuppa Positioning Engine* where the tag's location is estimated. However, both of them are *network-based* solutions; *mobile-based* solutions are not possible because smartphones only have omni-directional antennas, making impossible to measure the angle of arrival of the signals.

Lateration technique determines the position of a mobile device by its distance to known wireless transmitters. *Trilateration* and *multilateration* are special cases of the *lateration* using three and more transmitters, respectively. Figure 2.3 shows how trilateration is performed, where *A*, *B* and *C* correspond to wireless transmitters and *X* is a user with a mobile device acting as receiver. If we draw a circle with a radius ρ, that represents the measured distance between the mobile device and each wireless transmitter, the intersection area corresponds to a mobile device estimated location. More accurate distance measurement to each transmitter will result in a more accurate mobile device position estimation. Distance can be measured using either time-based methods or radio signal property based method, as follows.

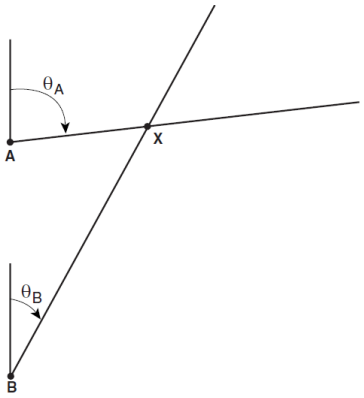


Figure 2.2: Angulation based positioning

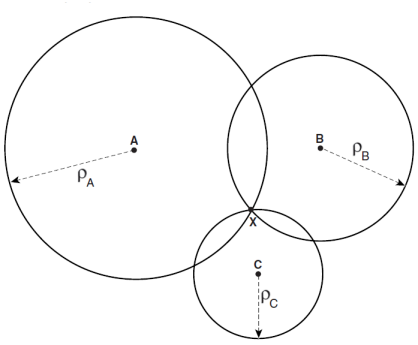


Figure 2.3: Positioning based on circular lateration (trilateration)

2.5.3.A Time-based distance estimation.

Time of Arrival (ToA) represents the travel time of the wireless radio signal from the transmitter to the mobile device (receiver). Since we know the speed radio signal travels at, the distance between transmitter and receiver can be determined. The problem of this time based approach is the extremely precise clock synchronization mechanism it requires between the mobile device and the wireless transmitters, because few nanoseconds clock skew makes a positioning error in hundred of meters, since radio signal travels at the speed of light. Time Difference of Arrival (TDoA) measures the difference in time of arrival at multiple pairs of receivers with known location. It uses relative time instead of absolute, so transmitter and receiver do not require to be synchronized, but receivers require time synchronization between each other. Each measured difference of arrival time produces a hyperbolic curve and the intersection of multiple hyperbolic curves estimate the position.

An AoA technique requires the signal travel time to be precisely measured when it is received in a mobile device. Using WiFi or Bluetooth signal transmitters and a smartphone as a receiver is impossible to apply this technique to calculate the distance between the transmitter and the smartphone. A 1μ time deviation equals to a 300m error in distance estimate and the instantaneous timing of packet transmis-

sion in Bluetooth may deviate up to 1μ from the average [33]. In TDoA systems the signal needs to be transmitted from a mobile device to several receivers whose differences in ToA are collected in a server, and then used to estimate mobile device's position. This does not follow our offline and *mobile-based* positioning system requirement, where the positioning needs to be fully calculated at the mobile device.

2.5.3.B Radio propagation model-based distance estimation.

Distance from mobile device to each transmitter can be also calculated using theoretical models of radio propagation and the RSSI feature, which measures the radio signal power received at the mobile device from the transmitter. This process is challenging due to complexity of indoor environments, where signal does not fall inversely proportional to the square of the distance between the transmitter and the receiver, as it would in perfect conditions (i.e., free space). In real indoor environment the signal acquires spatial and temporal properties that change over time due to channel impediments caused by moving objects within indoor space, such as shadowing and multipath [34]. This complexity of indoor environments make impossible the exact distance calculation between mobile device and wireless transmitter using RSSI, but the distance can be predicted using a radio propagation model that takes path loss into account.

2.5.3.C Log-distance path loss model.

Thaljaoui et al. [35] used one of several radio propagation models, called *Log-distance path loss*[36], to estimate the distance between a mobile device and BLE beacons. This model predicts the propagation loss for a wide range of indoor environments and has the following formula:

$$RSSI(d) = RSSI(d_0) - 10 \times n \times log(\frac{d}{d_0}) \tag{2.1}$$

RSSI(*d*) received signal strength at distance *d* (between transmitter and receiver)

RSSI(d_0) received signal strength at distance d_0 (usually d_0 = 1 meter)

n path loss exponent; normally in the range of 2 to 4 (where 2 is for propagation in free space, 4 is for relatively lossy environments)

For d_0 = 1 meter, and solving equation (2.1) for *d* we get:

$$d = 10^{\frac{RSSI(d_0) - RSSI(d)}{10 \times n}} \tag{2.2}$$

Path loss exponent is estimated with RSSI and known distances as follows:

$$n = \frac{RSSI(d_0) - RSSI(d)}{10 \times log(\frac{d}{d_0})} \tag{2.3}$$

The RSSI value measured at a given time and space depends not only on the distance but on many other factors too. So, a single RSSI measurement cannot be a reliable parameter to compute an accurate distance. To improve distance estimation accuracy and the instability of the RSSI, the proposed solution first measured 50 times the RSSI value at various distances (0.25m, 0.5m, 1m, 1.5m, 2m, 2.5m, 3m, 3.5m) and the average of ten maximum values for each distance *d* represent the *RSSI(d)*. Once a *RSSI(d)* is known, path loss exponent *n* can be estimated as an average of all values calculated for each distance using equation (2.3). Finally, *RSSI(d_0)* and *n* can be substituted in equation (2.2) and used to calculate the distance providing only the measured *RSSI(d)*.

2.5.3.D Fingerprinting

Fingerprinting positioning technique (also known as *pattern matching*) consists in two phases: *offline* and *online*.

During the *offline* phase a *radio map* is created, which stores RSSI from all nearby transmitters at specific *reference points* (or *anchor points*). The *Radio map* consists in a database holding the position of each *reference point* with an unique identifier of each nearby transmitter and their corresponding RSSI values.

In the *online* phase, the mobile device collects the RSSI of nearby transmitters, which represents the current fingerprint. This fingerprint is compared with the stored ones in the *radio map* database using a matching algorithm, resulting in a likeliest position of the mobile device in that indoor space.

Two important parameters must be taken in consideration when building a *radio map* that later, in *online* phase, will affect the accuracy of mobile device position estimation, namely the number of *reference points* and the number of RSSI samples collected at each *reference point*. The smaller the granularity and the more measurements for each *reference point*, the better. However, it creates a time-consuming initial configuration during the *offline* phase.

The whole process is illustrated in Figure 2.4, where during the *offline* phase a mobile device collects the RSSI measurements from several nearby APs at different *reference points*. These measured RSSI values are stored in a *radio map* database, together with the corresponding location. Later, during the *online* phase, a mobile device measures the RSSI values from surrounding APs and performs a matching process, with the available fingerprints at the *radio map*, through a pattern recognition technique.

Lashkari et al. [37] proposes a mobile application able to estimate the position of a user within a building using WiFI technology, taking advantage of already existing WLAN infrastructure. The positioning is based on *fingerprinting* technique. In the *offline* phase the *radio map* stores the RSSI value

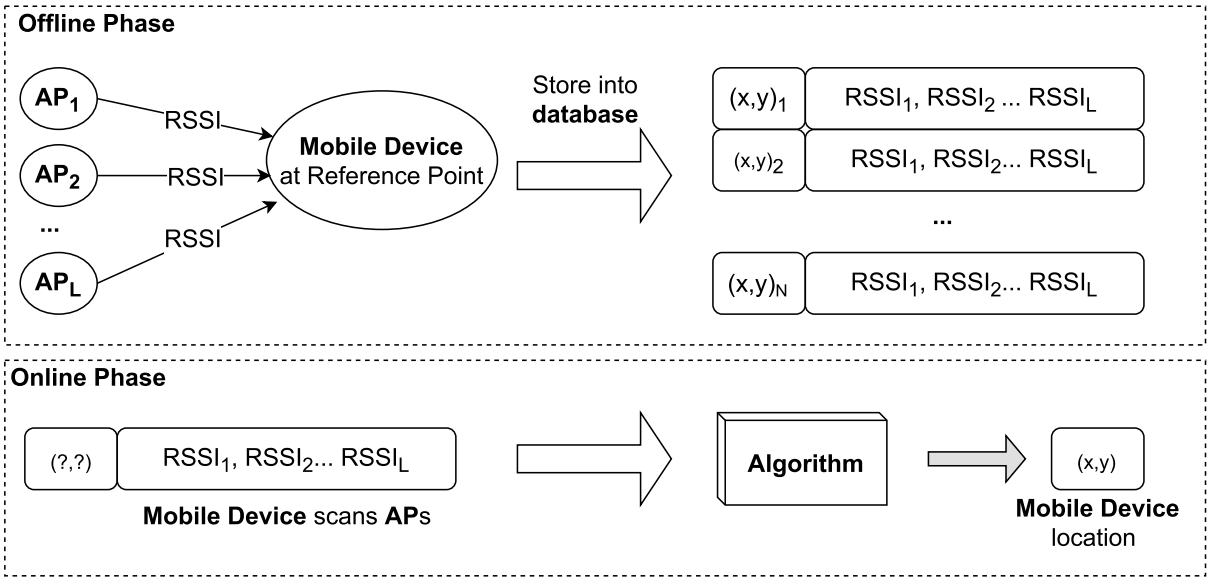


Figure 2.4: Fingerprinting Online and Offline phases (adapted from Li et al. [1])

together with the MAC address for each nearby AP, measured at specific positions. To reduce signal attenuation or interference error, four samples are taken, for each direction (north, south, east and west), and the mean values are recorded in the *radio map* for that position. In the *online* phase, a mobile application's sniffer collects RSSI values of nearby APs and use them to build the measurement vector of signal strength, which represents the obtained fingerprint. Finally, Euclidean distance calculation estimates the position, comparing the obtained fingerprint with the existing ones stored in the *radio map* database. The fingerprint vector with minimal Euclidean distance to the calculated one corresponds to the estimated position.

Li et al. [1] evaluates the effect of the granularity of reference points in system's accuracy. Five *radio map* database are generated with several decreasing number of reference points spread as evenly as possible and it is observed that when the granularity reduces (i.e., the number of reference points increases), the accuracy of estimated location increases. Training *offline* phase is an important and time consuming task if high accuracy is needed, since fingerprinting needs to be performed and stored in *radio map* for a large number of reference points. Instead of constructing the *radio map* database manually for large number of reference points, this database can be generated using interpolation based on a small number of reference points, simplifying the training phase. The results show that RSSI can be efficiently estimated from information of some of the reference points. Using generated data with only 16 existing reference points the estimated error is smaller than using 66 reference points, without any generation.

Fingerprinting is the most widely used and with good accuracy technique in indoor positioning. It does not require the exact position of wireless signal transmitters to be known, like it does in *trilateration*.

Also, an already existing wireless infrastructure can be used, which reduces the overall cost of the system since no additional hardware is needed. A drawback in this method is periodical calibration required, because indoor environments may change over time due to furniture and moving people, causing changes in propagation environment. Those changes may cause the RSSI, during positioning process, be significantly different from those modeled by the *radio map*, so the *radio map* needs to be periodically updated and even using interpolation, it is still time consuming. Additionally, it has a high computational cost compared to *trilateration* technique, because of the matching algorithm that is executed in the *online phase*.

2.6 Activity Recognition

Activity recognition plays an important role in context-aware ubiquitous computing and can be applied in many fields such as eldercare, healthcare, tracking applications. By knowing user's activity, a personalized and intelligent service can be provided.

Motion of a body usually consists of translational motion. To infer user's activity using movement-based sensors, accelerometers proved to be more suitable to infer human activity by identifying and classifying movements performed by the subjects [38] [39], since patterns in movements can be determined by measuring the linear acceleration. Thus, our main focus are accelerometer-based solutions.

2.6.1 Supervised learning

Most of accelerometer-based activity recognition systems use machine learning approaches. According to Incel et al. [39], the process of activity recognition can be summarized as determining a target set of activities, collecting sensor data and labeling the collected data to the appropriate activities. This is achieved using supervised learning methods, which consist of two main phases: *training* and *classification*.

The *training* phase uses a given set of examples or observations called *training set* to discover patterns. These examples have an associated activity (i.e., labeled) and usually this labeling is done manually, while performing each activity during which sensor data is collected (known as *raw data*). After the collection of labeled data *preprocessing* and *feature extraction* steps are followed, after which training models are built and training parameters are calculated according to the used machine learning technique. The *classification* phase, classifies the live stream of raw data gathered by the sensors with the computed training model, and the activity being performed is inferred.

The *classification* phase executes the following steps: (1) *preprocessing*, (2) *segmentation*, (3) *feature extraction* and (4) *classification*.

The *preprocessing* performs noise removal, because raw data received from sensors may contain noise that prevents accurate activity pattern detection.

The *segmentation* phase applied to a continuous stream of sensor data divides the signal into smaller time segments to make information extraction from continuous stream of data possible. Streaming data is divided into chunks before the classification starts.

The *feature extraction* phase reduces the large input sensor data to a smaller set of features called *feature vector*, that represents the original data.

Finally, the classification phase maps the extracted feature set to a set of activities. The classification technique may involve a simple thresholding scheme or a machine learning scheme based on pattern recognition or neural network.

2.6.2 Mobile device and body-worn sensors

Pärkkä et al. [40] evaluates an automatic activity recognition system consisting of a Personal Digital Assistant (PDA) and wireless motion bands. Wireless bands collect accelerometer data and transfer it to a PDA over a Bluetooth connection during five activities: lying, sitting/standing, walking, running and cycling. Wireless motion bands are attached to both ankles and wrists. Four features are extracted from raw accelerometer sensor data and a classifier is built, based on those features. An important issue is addressed in this study regarding mobile device's power consumption. If the activity recognition is done entirely in a mobile device, not requiring any computation offloading to a system having more resources, the complexity of the algorithm should be analyzed, as complex algorithm consumes more time, processing power and may require more memory. Taking that into consideration, *Binary decision tree* classification algorithm is chosen to infer the activity and proves to be effective in terms of consuming little battery power while detecting activities with high precision. Although the authors present this system as a "real-time activity recognition", no time constraints and requirements are specified that could demonstrate how responsive activity detection is. Another issue with this solution is the need of having four wireless bands attached to the body, that can create discomfort and without them the activity recognition becomes impossible.

Guiry et al. [41] describe a method of detecting human activity using accelerometer data collected from both: mobile device and a wireless sensor band placed on user's chest. Similarly to the previous study, the same target activities are recognized. Being capable of differentiating the activities, this solution still imposes the use of a additional component (wireless band) and has even stronger requirement: the smartphone should be in user's trouser pocket.

Understanding the drawbacks of prior works, where to recognize accurately the activity the mobile phone should be placed in a particular pre-determined way, some studies extend the locations at which the phone can be placed to recognize different activities. Kwapisz et al. [42] vary phone placement during

data collection including placement in hand, pants pocket, shirt pocket and handbag. The recognition of the performed activity, including cycling, becomes possible while mobile phone was kept in different orientations, but if a phone is carried in a different way, of the considered ones for data collection and resulted classifier, the system similarly may infer wrong activity.

All previously mentioned accelerometer-based activity recognition systems work in a similar fashion, where a mobile phone executes an application that captures accelerometer data and infers the activity being performed. It is possible using this approach to detect the cycling activity, but it has to sense accelerometer data in a continuous way and perform complex computations, that would quickly drain smartphone's battery. Since one of our requirements is low power consumption, it would be preferable to have a system which minimizes the computation performed by mobile devices. Additionally, activity recognition using mobile device's accelerometer can be problematic because some Android phones are not capable of using the accelerometer in standby mode, when the screen is turned off, by the choice of manufacturer. In iOS the restriction is even higher since iOS 7 version because the applications are terminated after being 180 seconds in the background, if there is no *background execution mode*[43] declared, and the use of the accelerometer data is not one of the supported modes.

2.6.3 Bicycle-mounted sensor

A more efficient alternative approach of detecting the cycling activity is to detect bicycle pedaling. In cycling, pedaling rate is called *cadence* and represents the number of Revolutions per minute (RPM). Bicycle's cadence is related to wheel speed and consequently cycling speed, but typically all bicycles have many gears making impossible to infer bicycle speed knowing its cadence. The cadence also characterizes cycling style and is used by cyclists to minimize muscular fatigue, maintaining RPM at specific ranges. A system measuring cadence with a bicycle-mounted wireless sensor is discussed in this subsection.

Okugawa et al. [44] implement a system that generates feedback sound accordingly to the pedaling rate. The system uses a wireless sensor containing a 3-axis accelerometer and 3-axis gyroscope attached to the right pedal crank of the bicycle with a sampling frequency of 50Hz. The sensor sends accelerometer and gyroscope data via Bluetooth to a Personal Computer (PC) that calculates the crank angle and the pedaling speed. Then, a PC plays a short feedback sound when the crank angle is at specific values. Results show that a strong relationship exists between accelerometer values and the crank angle, that is directly related to pedaling. When acceleration *x-axis* value is 0 *mG*, the crank angle is in position of 0 or 180 degrees. When pedaling, acceleration waveforms appear as sine waves and even without a gyroscope sensor, it is possible to determine the crank RPM.

2.6.4 Commercially available sensors

Several cycling applications support an integration with wireless sensors, which enables additional functionalities and improve cycling performance. This led to a grow in cadence and speed sensors availability on the market. These sensors communicate with the mobile device over standard interfaces, mostly by BLE or ANT [45] connection, providing real-time data.

A BLE cadence sensor can be used to detect the cycling activity. This approach follows the GATT communication, discussed in Section 2.4.4. The cadence sensor works as GATT server holding lookup data with a *Cycling Speed and Cadence* [46] service. The GATT client would be a smartphone application with Bluetooth GATT functionality. Once a connection with the sensor is established, the GATT client can perform a characteristic *notify* request, after which it starts to receive sensor updated data. It is immediate then to infer the cycling activity with instantaneous cadence and speed values, determined with the data provided by the sensor.

2.7 Summary

After describing the state-of-the-art of indoor positioning techniques and cycling detection mechanisms, we present an overview and discuss how they meet our requirements as well the potential of being integrated in our project.

As already discussed in subsection 2.4, BLE has significant improvements over classic Bluetooth in power consumption, discovery time and adjustable *Transmission Power* in BLE beacons. Thus, BLE, instead of classic Bluetooth, is considered in the positioning technique comparison presented in the Table 2.2. GPS and RFID solutions are not included in the comparison because GPS has poor indoor satellite signal coverage and not all smartphones have NFC technology, which act as RFID readers. Since smartphone WiFi and Bluetooth antennas are not capable measuring the wireless signal direction, AoA technique is discarded too. Also, time-based techniques for distance estimation between transmitters and mobile device are not possible, since they require clock synchronization between the transmitter and the receiver (smartphone).

Table 2.2 compares *proximity detection*, *trilateration* and *fingerprinting* techniques taking into account our requirements. Each technique is divided in two: using existing WLAN infrastructure or with installed BLE transmitters.

Using already existing WLAN infrastructure within an indoor environment require no additional hardware, but this approach has the problem of needing a periodical calibration since we have no control over existing APs and if some changes occur, invalid position may be estimated. Using existing WLAN may also result in a weaker accuracy, as it depends on the availability and positioning of the APs within a given indoor space. Additionally, *proximity detection* and *trilateration* techniques can only be possible

if the position of existing APs is known.

Computation complexity represents the computational cost that is required and *fingerprinting* technique holds the higher computational cost because of the heavy matching algorithms during the *online* phase. Since computational cost is directly related to mobile device power consumption, lower computational complexity is preferred.

Coverage scalability means the easiness of expanding the positioning coverage area. When using existing WLAN infrastructure, the coverage depends on the existence of APs in the new area we want the positioning system to work in. The inexistence of APs makes the positioning in that area not possible. Using BLE beacons this problem does not occur, since they are manually positioned in the places we want the positioning to be covered, making BLE beacon positioning easily scalable.

Periodic calibration is the maintenance necessary to keep the positioning system at the desired accuracy level. Fingerprinting technique has the biggest calibration, because periodical updates of radio map are necessary due to indoor environmental changes. In *proximity detection* and *trilateration* techniques calibration means inspection of available APs.

Analyzing the Table 2.2 we can see that *proximity detection* technique using BLE beacons better satisfies our requirements. Although, *fingerprinting* has higher accuracy, it has significant drawbacks in terms of required initial configuration, periodic calibration and complexity of computation, making it not suitable positioning technique for our project. *Trilateration* using BLE also satisfies most of the requirements but would require more BLE beacons, thus making the system more expensive. *Proximity detection* has enough accuracy to guarantee that a user is within a given indoor space and in a shopping center scenario can accurately detect that a user is at the store *X* and not at nearby stores *Y* or *Z*.

Regarding cycling detection three approaches were presented. In the first approach, accelerometer data is collected from the smartphone or some wearable sensors, and the activity is inferred using that accelerometer *raw data*. This approach can detect cycling activity, but sensing accelerometer and executing heavy computations in a continuous fashion would consume a lot of battery power, which does not satisfy our requirement of low power consumption.

The second presented approach was using accelerometer data from bicycle-mounted sensor, and as we saw, it is possible to infer the pedaling rate (i.e., *cadence*) which tells whether or not a user is cycling. There are some commercially available BLE devices with built-in accelerometer sensors, like *motion beacons* from Estimote [47], but they are not prepared to be mounted on a bicycle.

The last presented solution was using a commercially available cadence sensor. In this approach the sensor transfers real-time pedal revolution information to the smartphone application over a BLE connection. Then, the calculated instantaneous RPM automatically tells whether or not the user is cycling, with no additional complex computations required like in the previous approaches.

Additionally, by using BLE in both location and cycling detection has the advantage of using only

one radio component, saving power when compared with a two radio component solution (WiFi and Bluetooth).

Table 2.2: Overview of positioning techniques, with numeric scale from 1 (*Very Low*) to 5 (*Very High*)

Technique	Accuracy	Coverage Scalability	Initial Setup	Periodic Calibration	Computation Complexity	Hardware Cost	Assumption
Proximity Detection (existing WLAN)	2	Depends on existing WLAN infrastructure	2	Yes	1	-	Position of APs is known
Proximity Detection (BLE)	4	Good	2	No	1	1	-
Trilateration (existing WLAN)	3	Depends on existing WLAN infrastructure	2	Yes	2	-	Position of APs is known
Trilateration (BLE)	4	Good	2	No	2	4	-
Fingerprinting (existing WLAN)	5	Depends on existing WLAN infrastructure	5	Yes	4	-	-
Fingerprinting (BLE)	5	Good	5	Yes	4	5	-

3

Solution

Contents

3.1 Introduction 31
3.2 Solution Overview 31
3.3 Architecture of Biklio 35
3.4 Architecture of BikeApp 37
3.5 Summary 43

3.1 Introduction

As already mentioned in Section 1.2, the purpose of *BikeApp* is to reward users for cycling. There are two entities within our system: cyclists and shops. The connection of the two entity groups is achieved through a mobile device application for cycling users, and a web application for shop owners to register their shops. The mobile application shows information about nearby registered shops, which includes the reward that each shop offers the users for cycling. *BikeApp* detects the cycling activity using a bicycle mounted BLE sensor and determines when the user is close to or within a shop using BLE location beacons.

This Chapter describes our solution with a focus on its architecture. First, in Section 3.2, we provide a solution overview, presenting its main components, such as the *web application*, the *mobile application*, the *backend* and the BLE devices. As previously mentioned in Chapter 1, *BikeApp* uses *Biklio* as its base project. In Section 3.3 we first present the architecture of *Biklio*. Then, in Section 3.4, we present the architecture of *BikeApp* with a focus to the cycling detection mechanism using bicycle BLE sensors, and the location detection mechanism using BLE location beacons.

In this Chapter we mention two applications: *Biklio* and *BikeApp*. When we do not mention one in particular, what is said applies to both.

3.2 Solution Overview

As previously mentioned, there are two main entities within our system: cyclists and shops. Considering the goals introduced in Section 1.2, our solution contains a set of shops that give rewards to users that cycle to their shop. The communication between these two entities is achieved through a *web application*, a *mobile application*, and a *backend*, as shown in Figure 3.1.

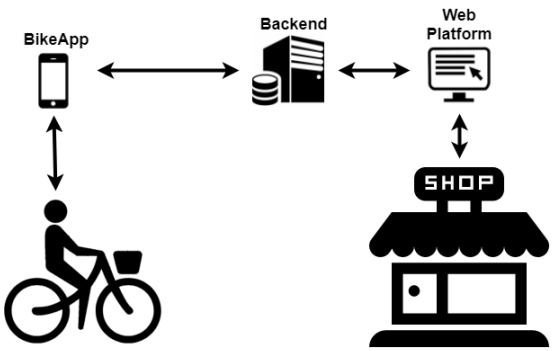


Figure 3.1: System overview of BikeApp

3.2.1 Web application

The *web application* allows shop owners to register their shops, designated as *spots* within the application, and become part of the *rewarding program*. During the registration, basic spot information is requested, such as name, spot schedule, spot type, address and spot logo. The shop owner must also specify the rewards his shop is able to offer to cycling customers.

Two types of rewards can be set: *cycle-to-spot* and *cycled-distance*. *Cycle-to-spot* is mandatory, and represents a low valuable benefit, e.g. 5% discount, for users that cycle to a spot. *Cycle-to-spot* benefits can be *claimed* multiple times. *Cycled-distance* is optional, and represents a valuable benefit, e.g. a high discount or a free product, and requires users to cycle a determined distance. *Cycled-distance* benefits can only be claimed once and require an active internet connection, as described in the following section.

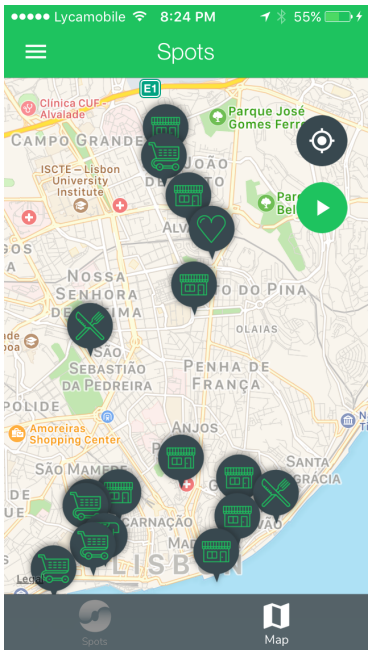
3.2.2 Mobile application

The *mobile application*, intended to be used by cyclists, provides information about nearby spots, that joined the *rewarding program*. Figure 3.2a shows the main screen that users see upon opening the application. This screen presents a list of nearby spots, with its *cycle-to-spot* partial reward description and the distance to the spot location. By opening the *Map* tab, the application presents the screen shown in Figure 3.2b, where the user sees a map with the nearby spots on it. The map screen also contains a *Record* button, which starts/stops tracking the user's route. Additional spot information can be found in Figure 3.2c, which is opened by clicking on the *View Spot* button from the spot listing screen or on the spot marker from the map screen. In this screen, the user can see detailed spot information, together with the full description of both *cycle-to-spot* and *cycled-distance* rewards.

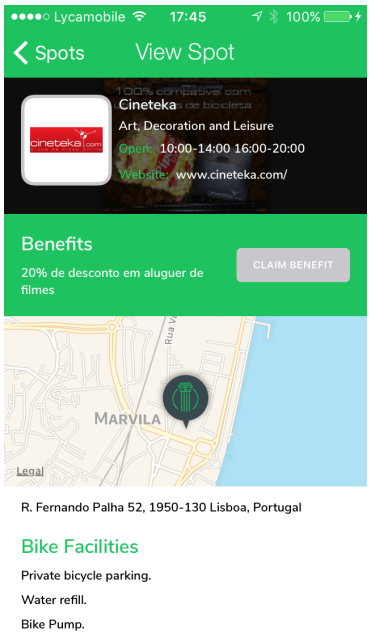
The application has a notion of *eligibility* for each type of reward; this means that the reward can be *claimed* at the spot. The user becomes eligible for *cycle-to-spot* rewards after the application detects the cycling activity during a period of 1.5 minutes. The eligibility for a specific *cycled-distance* reward is activated after the user cycles the required distance (that depends on each shop challenge). *BikeApp* uses the distance tracked over multiple routes.

When the user becomes close to or enters one of the spots, and is *eligible* for any of its rewards, *BikeApp* notifies him that there is an available *benefit* to *claim*, as shown in Figure 3.3a. Then, upon opening the application a benefit claim screen pops-up. This screen is shown in Figure 3.3b and contains the information about the detected spot and the reward description. If there is eligibility for both *cycle-to-spot* and *cycled-distance* rewards, the latter is chosen, since it is more valuable. To claim the benefit, all the user needs to do is press the *Claim Your Benefit* button and show the claimed screen to the spot cashier, who will give the user the benefit. Figure 3.3c shows the claimed benefit screen, which contains

(a) Spots listed by distance

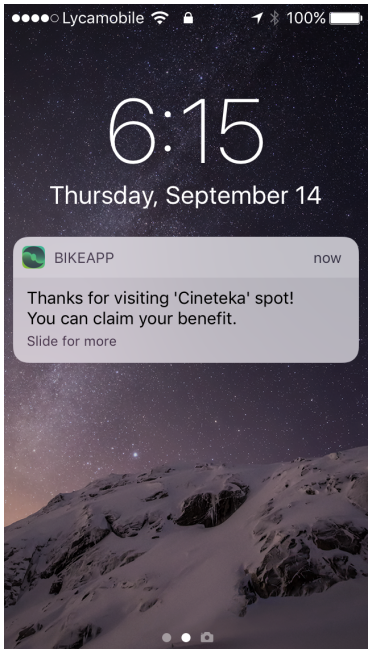


(b) Spots on map

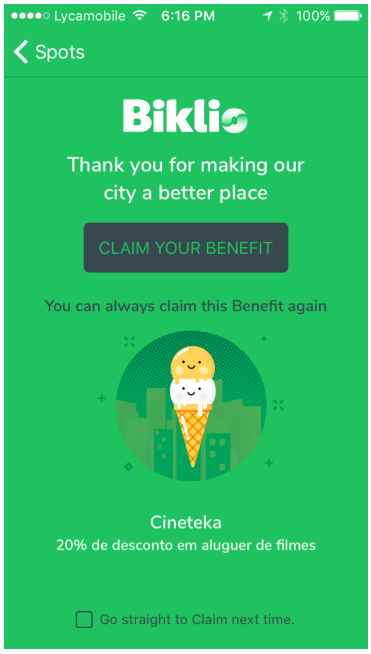


(c) Spot information

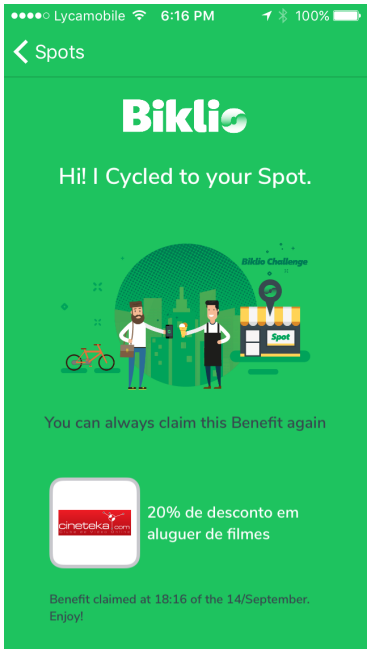
Figure 3.2: Available Spots presentation



(a) Claimable benefit notification



(b) Benefit claim



(c) Screen to be shown

Figure 3.3: Location detection notification and benefit claim

the spot logo, the reward description and the moment at which the benefit was claimed. If the reward is *cycled-distance*, an active internet connection is required to confirm with the *backend* that the reward

was not claimed yet.

The mobile application stores all the data it receives from the *backend*, to guarantee offline functioning. The data is stored persistently using *SQLite* [48] database engine.

3.2.3 Backend

The *backend* plays an intermediary role, since it handles requests from both *web application* and *mobile application* sources. It stores all the application relevant data, such as its users, registered spots information, etc. *BikeApp* oportunistically synchronizes with the *backend*, when there is an active Internet connection. For example, if the user uses the application in offline mode and a route is tracked, the registered route trajectory (of GPS coordinates) is only uploaded to the *backend* when internet connection becomes available.

Regarding the shops information, the application and the backend maintain its data version. Everytime there is any new data (e.g., a new registered shop) or any data change (e.g., shop information change), the backend associates a freshly incremented version number to that data. The mobile application stores persistently all the received data from the backend, and its version. When the application requests the backend for nearby shops, it provides its data version number, so that the backend only sends the data with higher version numbers. Thus, only the data that application does not have yet is sent, which helps to decrease the amount of data sent over the network, and is very important when using cellular data connectivity.

There is another aspect which reduces the amount of received data from the backend. Upon requesting the nearby shops, we specify the current user's location and a desired radius. This allows the backend to send only the shops located geographically within the circular area centered at the user's location and the specified radius.

3.2.4 Cycling and location detection using BLE devices

The cycling and location detection are two crucial features, since they are required for making the user eligible for rewards, and preparing the benefit claim upon entering the spot. However, as we already discussed in Sections 2.5 and 2.6, it is quite challenging to find high precision and accuracy solutions.

BikeApp solves this problem by detecting the cycling activity and the user's location with an introduction of two types of BLE devices: a bicycle sensor and a location beacon. Both communicate with *BikeApp* through BLE (see Figure 3.4). More details about the cycling detection and location detection mechanism are covered in Section 3.4.1 and 3.4.2, respectively.

The cycling activity detection is required to make the user *eligible* for *cycle-to-spot* and *cycled-distance* benefits. Eligibility means that the *benefit* can be *claimed* at the spot.

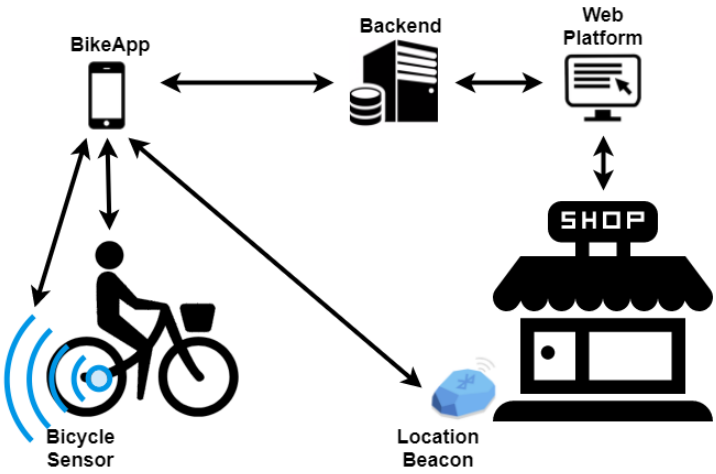


Figure 3.4: Complete system overview of *BikeApp* containing BLE devices

To summarize, our proposed solution is composed of the following components and responsabilities:

BikeApp - mobile application running in a BLE enabled device

Location beacons - BLE beacons deployed at the shops, used by *BikeApp* to determine the shop the user is located in.

Bicycle sensor - BLE sensor attached to the bicycle. Sends cadence and speed data to the smartphone, which is used by the *BikeApp* to detect the cycling activity.

Web Application - allows shop owners to register their shops.

Backend - handles mobile client and web application requests. Stores all the system data that is requested and stored by *BikeApp* to be able to work in offline mode.

3.3 Architecture of Biklio

Once again, *Biklio* is a production version of our solution, available on both iOS and Android platforms, but with less accurate cycling and location detection mechanisms. Because *BikeApp* uses *Biklio* as its base project, it is important to analyze the architecture of *Biklio* to later understand the architecture of *BikeApp* and the changes due to the use of the BLE sensors for better cycling and location detection.

The cycling detection of *Biklio* (iOS version) uses an algorithm that relies on the average and exceeded above a given threshold speeds during a trip, to infer whether or not the user is cycling. The average speed is calculated with the the travelled distance and elapsed time, by periodically acquiring the user's GPS position. To minimize the battery consumption from the GPS adapter's usage, this mechanism is only activated when the user is moving. The smartphone stillness is determined from

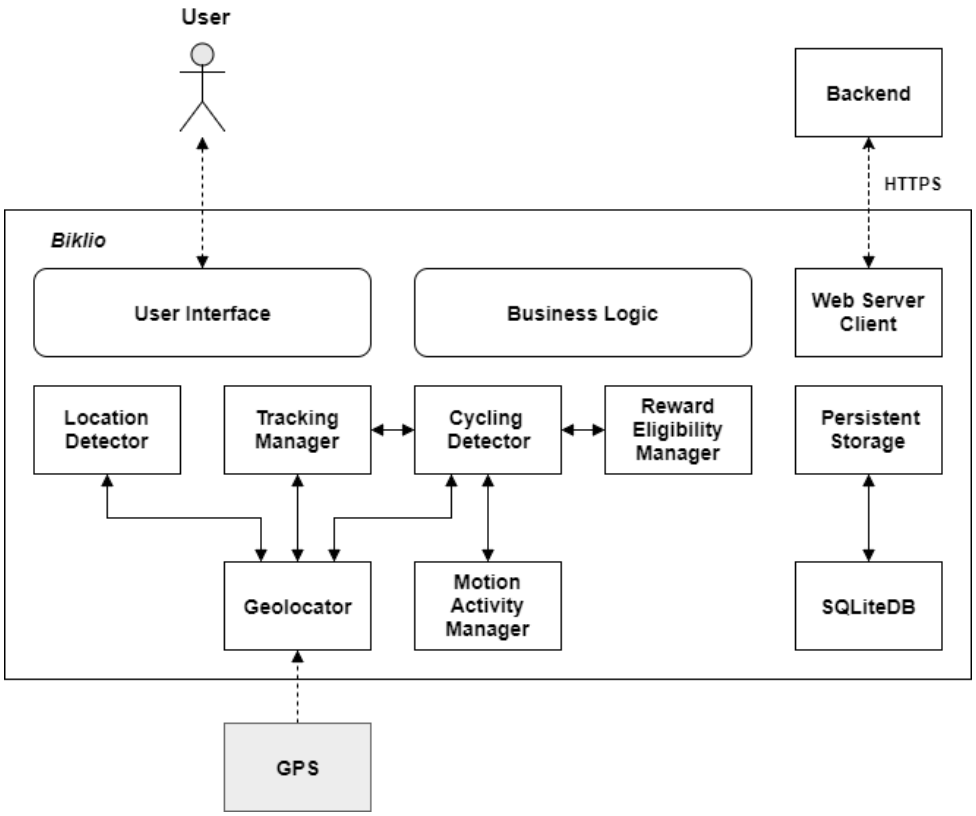


Figure 3.5: Biklio modules

the smartphone's accelerometer sensor data, by using the *activity manager* provided by the Software Development Kit (SDK), which returns the *stationary* activity when the phone is still. For example, if during a trip the average speed is 12 km/h and the speed does not exceed 40 km/h in 80% of times, it is considered that the user is cycling during that trip. Relying on trip speeds may introduce false positives, if certain conditions are met, even if the user is not cycling. This makes the mechanism not accurate.

The *Biklio*'s location detection mechanism just uses the GPS adapter to fetch the user's position and considers that the user is within a spot area if the distance to the spot is less than a certain value. For example, when the user approaches a spot, and his distance to that spot is less than 40 meters, *Biklio* notifies the user about the spot nearby. Because of the GPS accuracy known problems, this mechanism is not accurate and does not work in indoor scenarios.

Figure 3.5 presents the modules contained within the *Biklio* project, and each module has the listed below responsabilities. The solid arrows represent the communication between the inner application modules, while the dashed arrow represents the communication with the external entities to the application. The *User Interface* and *Business Logic* modules have curved edges and no interaction arrows because these modules are implicit within every mobile application, and are not the main focus of our solution.

User Interface - provides the means by which the user interacts with the mobile application.

Business Logic contains the domain model objects and the application business rules.

Web Server Client - communicates with the backend and stores the requested from the backend information into the application.

Geolocator - provides the means of interaction with the GPS adapter.

Location Detector - detects the user's location using the GPS coordinates provided by the *Geolocator* module.

Tracking Manager - keeps track of the path traveled by the user using the GPS coordinates provided by the *Geolocator* module.

Motion Activity Manager - informs whether the user is walking, running, in a vehicle, or stationary.

Cycling Detector - detects the user's cycling activity using the *Geolocator* and the *Motion Activity Manager*.

Reward Eligibility - maintains the user's eligibility state using the *Cycling Detector*. Acts accordingly when the eligibility state changes.

SQLiteDB - interacts with the device's *SQLite*.

Persistent Storage - saves and retrieves the data persistently using the *SQLiteDB* module

3.4 Architecture of BikeApp

As already mentioned, *BikeApp* solves the previously mentioned problems in *Biklio*'s cycling and location detection by introducing alternative accurate cycling and location detection mechanisms. Figure 3.6 shows the modules of *BikeApp*. Since *Biklio* served as its base project, some of its modules were already implemented, and these are represented as uncolored boxes. The yellow boxes represent the modules that required some changes, and the green boxes represent the new or replaced modules.

The *Cycling Detector* module was completely replaced, and detects the user's cycling state using bicycle mounted sensor data. The communication with the sensor is handled by the newly added *BLE Manager* module. Similarly, the *Location Detector* module was also completely replaced. It determines the user's location considering BLE location beacons in range. The beacons are monitored by the newly added *Beacon Monitor* module. These modules are our main focus, therefore the following sections explain how the detection mechanisms work and the modules are involved.

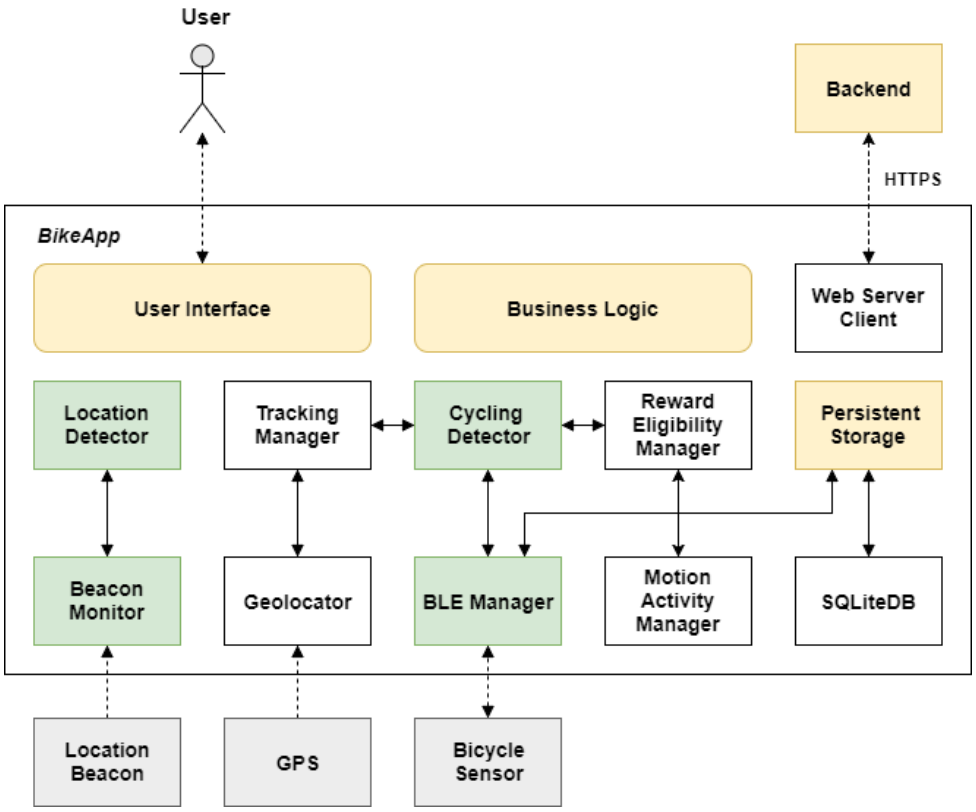
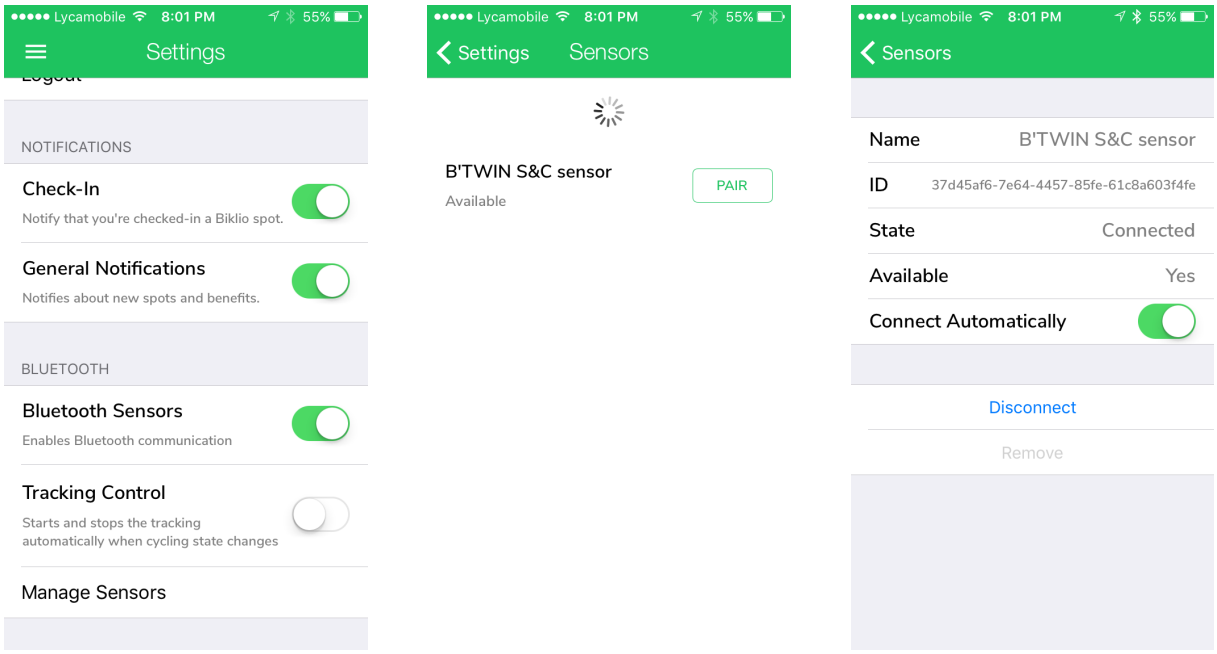


Figure 3.6: BikeApp modules

3.4.1 Cycling detection

As previously mentioned, *BikeApp*'s cycling detection mechanism uses a BLE bicycle sensor. The sensor is attached to the bicycle frame and contains two components: one magnet attached to the pedal and another attached to the rear wheel spoke. The sensor is powered with a single coin cell battery (type CR2032). Because the BLE is efficient in terms of energy consumption, the battery is enough to power the device up to 12 months (with a 1 hour per day use), as stated by many manufacturers. The sensor is powered off after few minutes of inactivity, which is 4 minutes in the sensor used during the development. This value is set by each manufacturer and is not configurable. The sensor powers on as soon as there is a crank or wheel revolution.

To enable the cycling detection mechanism it is required that the user pairs a BLE bicycle sensor. The sensor pairing is done by navigating to the *Manage Sensors* page of the *BikeApp*, available at the Bluetooth grouping of application's Settings, as shown in Figure 3.7a. This grouping contains two more elements: a switch that enables the Bluetooth communication and a switch that enables *BikeApp* to automatically start and stop the route tracking with the cycling detection. After clicking on *Manage Sensors*, a list with currently available sensors is presented, as shown in Figure 3.7b. To pair the sensor, the user just needs to press the *Pair* button, and the connection with the sensor is established.



(a) Bluetooth Section **(b)** Available sensors **(c)** Paired sensor details

Figure 3.7: Sensor managing

The pairing is only required to be done once, because the application maintains a state of paired devices and the sensor is automatically connected as soon as it becomes available. After pairing the sensor, the *Pair* button is replaced with an information icon. Clicking on that icon opens a page represented in Figure 3.7c, where the user can control the sensor's connection or remove the sensor.

Even though the connection with the sensor already means the user is nearby the bicycle and the sensor was activated through pedal or wheel revolution, there is no actual guarantee that the user started cycling (e.g., he could be standing by the bicycle). To detect the cycling activity we use the instantaneous cadence and speed values, determined with the received from the sensor data (as described later).

The *BLE Manager* module manages the connection and the communication with the BLE sensors. It scans for nearby BLE devices, establishes a connection with a specific device and receives device updated data. After a connection is established, the *BLE Manager* starts receiving sensor data and routes it to the *Cycling Detector* module. The latter has no knowledge about the sensors and has the function of interpreting the sensor data to determine the cycling state.

The sensor data contain four cumulative values: the total counts of crank and wheel revolutions, and the times the last crank and wheel revolutions occurred. Let's designate them *Crank Revolutions*, *Wheel Revolutions*, *Last Crank Event Time*, and *Last Wheel Event Time*, respectively. All the counters start at 0 when the sensor is activated for the first time after purchasing it, and are always incremented. As previously mentioned, there is a magnet attached to the pedal and another to the rear wheel. When the user cycles, each pedal revolution makes the magnet pass near the sensor, which increments the *Crank Rev-*

olutions counter by 1. Similarly, the sensor increments the *Wheel Revolutions* counter with each wheel revolution. The *Last Crank Event Time* and *Last Wheel Event Time* are incremented with the elapsed time, in units of 1/1024 of a second, everytime there is a crank and wheel revolution, respectively. For example, if a revolution takes 2 seconds, the value is incremented by 2048.

The sensor periodically sends its four cumulative counters within a single data packet. Using simple calculation involving the elapsed time between two sequential revolutions and the number of revolutions that occurred during that time, allow the *Cycling Detector* module to determine instantaneous wheel and crank revolution speeds, as follows. Let us consider two consecutive received data packets, and use "previous" and "current" to identify the counters contained within each packet. The following expressions show how we determine the number of revolutions and the elapsed time, between two sequential received packets, by subtracting their counters.

$$wheelRevolutions = currentWheelRevolutions - previousWheelRevolutions$$

$$crankRevolutions = currentCranksRevolutions - previousCrankRevolutions$$

$$wheelEventElapsedTime = currentLastWheelEventTime - previousLastWheelEventTime$$

$$crankEventElapsedTime = currentLastCrankEventTime - previousLastCrankEventTime$$

To determine the instantaneous wheel speed, we multiply the previously calculated *wheelRevolutions* by the wheel circumference (in m), which gives us the traveled distance, and then divide it by *wheelEventElapsedTime*. Because the time is in units of 1/1024 of a second, and the traveled distance in meters, we need to multiply by 1024 and 3.6 to get the instantaneous speed in km/h. To determine the instantaneous cadence value, we divide *crankRevolutions* by the *crankEventElapsedTime* and finally, multiply by 1024 and 60 to get the cadence in RPM, as follows:

$$instantaneousSpeed = \frac{wheelRevolutions \times CIRCUMFERENCE}{wheelEventElapsedTime} \times 1024 \times 3.6$$

$$instantaneousCadence = \frac{crankRevolutions}{crankEventElapsedTime} \times 1024 \times 60$$

We use a value of 210cm for a wheel circumference in the previously presented instantaneous speed calculation formula. This value corresponds to a typical mountain bike 26-inch wheel size. If a smaller bike is used, there would be more wheel revolutions, which results in higher instantaneous speed values, but this does not affect our cycling detection.

Cycling Detector considers that the user is cycling when the instantaneous speed and cadence values exceed 30 RPM and 11 km/h during a determined period of time. Those were approximately the values gathered from cycling at a slow speed but faster than walking. The time period within which the instantaneous cadence and speed values should be higher than the thresholds is just to prevent cycling detection when, for example, the user abruptly pushes the bicycle, which would result in high instantaneous values, but did not, in fact, start cycling. We experimented with a period of 5 seconds and consider it suitable.

Cycling Detector detects also that the user is no longer cycling, when the module does not receive updated sensor data for a determined period of time. This timeout avoids taking actions too early when the user just stops cycling for a few seconds (e.g., in a traffic light), which for example would result in finishing the tracking when the user did not actually end the trip. Because the non-cycling detection only controls the tracking, there is no significant importance on the chosen timeout value. Event if the timeout elapses, the tracking will be finished, but will instantly start again as soon as the user resumes cycling. The total tracked distance will still be the same, not affecting the eligibility for the *cycled-distance* rewards. We conclude with our experimental data that a value of 30 seconds is a good fit.

3.4.2 Location detection

BLE location beacons are devices that broadcast their identifiers to nearby devices at regular periods. Such beacons also contain important configurable parameters, such as *advertising interval*, *transmission power* and the *protocol*. An analysis and practical evaluation on *advertising interval* and *transmission power* parameters is presented in Section 5.2 and 5.3.

The beacon *protocols* are standards of BLE communication and each protocol describes the structure and the content of the data packets beacons advertise. Section 4.6.1 presents two widely used protocols, supported by many beacon manufacturers, including the protocol we use in our solution: the *iBeacon* protocol. Beacons set to broadcast *iBeacon* protocol packets have three configurable *identifiers*: UUID, *Major* and *Minor*. A UUID is a 128-bit value, while *Major* and *Minor* are unsigned integer values between 0 and 65535.

Our solution requires a location beacon to be placed within each shop, configured with a specific UUID, and unique *Major* and *Minor* identifiers. All the shop beacons have the same UUID. The beacons are installed by the shop owners, and configured with the parameters and generated identifiers given by the web application during the shop registration. Because the *web application* submits the shop registration to the *backend*, the latter contain all the information about the beacons installed within the shops. Such information is sent to the *BikeApp* mobile clients when they request the nearby shops. The beacon configuration is done through mobile applications. These applications depend on the beacon manufacturers and are available in the application stores (*Google Play* and *App Store*). The configu-

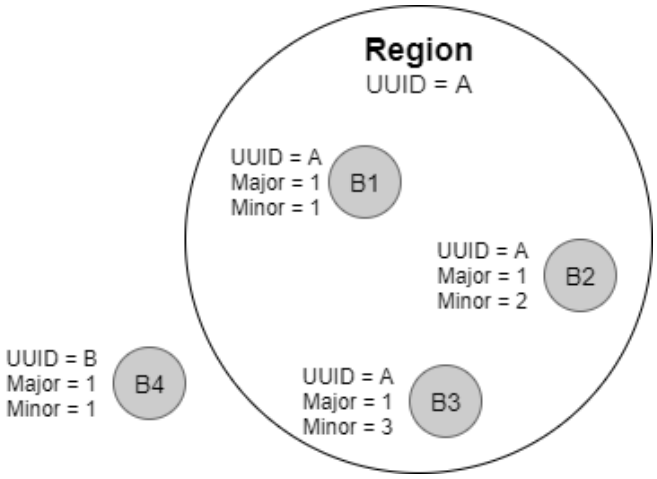


Figure 3.8: Region defined with a shared UUID, and unique Major and Minor

ration process is simple and applications are easy to use, which makes the configuration possible by someone with no knowledge in BLE technology.

The *Beacon Monitor* module is responsible for monitoring BLE beacons. To do so, it has a notion of monitored beacon regions, identified by a UUID, which contain beacons that have the same UUID. The *Beacon Monitor* module detects whenever the mobile device enters or leaves a region, i.e., the signal range of region's beacons. Those region boundary crossings are designated *region entered* and *region left* events. Both are events that *Beacon Monitor* exposes and other entities can subscribe to. When *region entered* event occurs, the subscribed entities are notified and receive information about the beacons in range. This information includes the UUID, *Major* and *Minor* identifiers, and also an estimated distance to the beacon.

Figure 3.8 shows an example of a monitored beacon region identified by UUID = A. The beacons *B1*, *B2* and *B3* are contained within the region because they share the same UUID. If the mobile device appears in the range of the beacon *B1*, for example, *Beacon Monitor* triggers its *region entered* event, and sends information about the beacons in range to the subscribed entities, more specifically the identifiers of the beacon *B1* and the estimated to the beacon distance.

The *Location Detector* module encapsulates all the location detection logic. The used location detection method is straightforward, because it is simply based on a correspondence table between each beacon and the shop it is placed in, as shown in the Table 3.1. For example, by knowing that the closest in range beacon has the (UUID, *Major*, *Minor*) identifiers equal to (A, 1, 1), it easily determines that the user visited the "Bob's Shop" through a table lookup. As previously mentioned, when *BikeApp* requests the nearby shops, it also receives the information about its beacons. Since the *BikeApp* persistently stores all the received from the backend data, *Location Detector* can always construct its correspondence table relying on the stored data, and is fully functional in offline mode.

Table 3.1: Example of a beacon-shop correspondence table

Beacon	**Shop**
UUID: A, *Major*: 1, *Minor*: 1	”Bob’s Shop”
UUID: A, *Major*: 1, *Minor*: 2	”Patrick’s Shop”
UUID: A, *Major*: 1, *Minor*: 3	”Linda’s Shop”

To determine the closest in range beacon, *Location Detector* uses the *Beacon Monitor* module. The *Location Detector* module requests *Beacon Monitor* to monitor a beacon region with a specific UUID (that all the shop beacons have), and subscribes to its *region entered* event. If the user appears in range of any of the shop beacons, *Beacon Monitor* triggers the *region entered* event, and sends to *Location Detector* the identifiers and the estimated distance of all the beacons that are in range. If multiple beacons are in range, the *Location Detector* module uses the estimated distance to determine the closest beacon. Finally, *Location Detector* just consults its beacon-shop correspondence table and determines the visited shop.

3.5 Summary

In this chapter we present our *BikeApp* solution and its main components. Our solution provides a way of rewarding users for cycling, with benefits at specific shops. It contains a mobile application for cycling users, a web application for shop owners and a backend.

Two core aspects of *BikeApp* are its cycling and location detection mechanisms, since the correct functioning of the application depends on their accuracy. Our solution uses BLE devices in both mechanisms. The cycling detection uses a bicycle mounted BLE sensor. When connected with the smartphone, this sensor sends crank and wheel revolution data, with which the application calculates instantaneous cadence and cycling speed values. These instantaneous speed values are then used in simple arithmetic comparisons to detect the cycling activity. It is considered that the users starts cycling when any of the instantaneous values is above a certain threshold, during a determined period of time. If the mechanism stops receiving updated sensor data, it is considered that the user stops cycling.

The location detection mechanism uses BLE location beacons mounted within the shops that join the rewarding program. Location beacons periodically advertise its unique identifiers, which enables to automatically infer the location by knowing the beacon’s identifiers and the places they are deployed at. When shop owners register their shop at the web application, they are given with instructions to configure the deployed at their shop beacon with specific identifiers and parameters. The shop beacon identifiers are stored at the backend, and sent to the mobile application clients upon requesting the

shops nearby. Thus, *BikeApp* contains information about the shops and its beacons, and constructs a beacon-shop correspondence table to be used by the location detection mechanism. Upon detecting any of the shop beacons in vicinity, the mechanism just lookups the correspondence table to determine the visited shop. When multiple shop beacons are in range, which may happen when there are two side by side shops within the rewarding program, the mechanism uses estimates of the distance to the beacons distances to determine the closest beacon, and consequently the visited shop.

4

Implementation

Contents

4.1 Introduction 47
4.2 Development environment 47
4.3 Project Structure 48
4.4 BLE Manager module 49
4.5 Cycling Detector module 55
4.6 Beacon Monitor module 57
4.7 Location Detector module 60
4.8 Summary 62

4.1 Introduction

In the previous Chapter we present our solution, which is a mobile application to be used by cycling users. This mobile application provides a way of discovering shops and getting rewards at those shops for completing cycling challenges.

The two core features where we mainly focused are implementing the cycling and location detection mechanisms. These are already presented in the previous chapter, but in this Chapter we describe its implementation details and important decisions done during the implementation process.

During the implementation, we only developed the application targeting the iOS platform. However, because the implementation was done as a cross-platform project (covered in the next Section), we abstracted our solution in a way that is possible and requires little implementation effort to expand it targeting also the Android platform.

In the next section we start by introducing the Xamarin cross-platform development solution and the equipment used during the development. Then, in Section 4.3 we present how the project is structured and modules communicate with each other. Section 4.4 shows the implementation of the module responsible for the communication with the BLE bicycle sensor. Section 4.5 explains how the previously mentioned module is used to implement the cycling detection, and presents how the cycling detection mechanism is used within the application. Section 4.6 describes the implementation of the module which detects BLE location beacons, and similarly, the following Section 4.7 explains how the implementation of the location detection uses this module, and how the location detection mechanism is used within the application

4.2 Development environment

BikeApp is developed in a cross-platform fashion using Xamarin [49]. Xamarin allows developers write, using the C# programming language, native Android, iOS and Windows apps with native user interfaces and share code across multiple platforms. Using Xamarin has the advantage over developing separate native application because of the shared code-base and there is no need to duplicate the code when targeting multiple platforms. However, we must follow a required project structure.

BikeApp is developed using Xamarin's Portable Class Library (PCL) project. To encourage code-sharing, it contains two types of project: *core project* and *platform-specific project*. The *core project* contains the greater part of the implementation, to enable code sharing. The *platform-specific project* contains the native part of the implementation. The linking between the two projects its done through an interface or a base class defined in the core project and implemented or extended in the platform-specific projects. This allows us to write generic code within the core project without worrying about the target platform details.

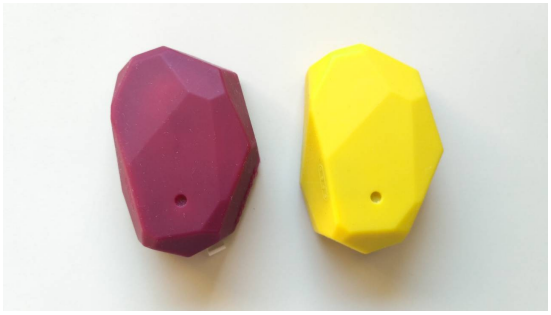
4.2.1 Equipment

During the implementation and evalution we used two Estimote [47] BLE location beacons, shown in Figure 4.1b. These beacons are configured using their official apps available for both Android and iOS platforms.

For cycling detection, we used a bicycle attachable BLE sensor from B'TWIN [50]. Figure 4.1a shows the sensor already mounted on the bicycle.

The application was tested on an iPhone model 5s with iOS version 10.3.2.

(a) B'TWIN BLE cadence and speed sensor



(b) Estimote BLE beacons

Figure 4.1: BLE bicycle sensor and location beacons

4.3 Project Structure

Now that we know that there are *core project* and *platform-specific project* types of project, we present how we structure our solution. To avoid digging deep into implementation details we only present the contract that our main modules follow, through their *interfaces* on Unified Modeling Language (UML) class diagrams, and specify which parts of the modules were implemented in the *core project* and which in the *platform-specific application project*. Because UML does not cover some of the C# programming language elements, such as *properties* and *events*, we follow IBM recommended mappings [51] between C# elements and UML elements. These are represented with *stereotype*, which is an extension mechanism that broadens the vocabulary of the UML and gives it more specific meaning.

4.3.1 Module communication

In software design it is a good practice to structure the architecture so that modules follow *high cohesion* and *low coupling*. High functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module. With low coupling, modules have little or no knowledge of the inner workings of others and can be developed independently without having to worry about how other modules have to interact with this module.

.NET framework offers an event handling feature, which allows an entity to *listen* (or *subscribe*) to a specific *event*, and be notified by the *sender* (or *publisher*) when the event is raised, executing its own custom code. The sender can also send custom data to its subscribers when the event triggers.

To communicate using events, the sender entity declares an event of type *EventHandler* as class field (see Listing 4.1) and specifies its generic type parameter, i.e. the data that the event pushes to the listeners.

```
1 public event EventHandler<MyDataEventArgs> SomethingHappened;
```

Listing 4.1: Sender's entity event declaration as class field

The listener entities subscribe a callback to the declared by the senders events (see Listing 4.2), which is invoked when the sender raises the event.

```
1 SenderEntity.Instance.SomethingHappened += (s, e) => {
2     // Execute custom action
3 };
```

Listing 4.2: Event callback subscription by receiver entity

To achieve high cohesion and low coupling, each of our modules have a well-defined responsibility and a set of exposed events, providing a way of interaction to the other modules in a *event-driven* fashion.

4.4 BLE Manager module

The *BLE Manager* module is responsible for discovery and communication with BLE devices. Figure 4.2 shows the interface that the implementation of *BLE Manager* module follows and other modules use to interact with it. The following property, events and operation descriptions appear by the order they appear in the interface.

Settable properties *ScanInterval* and *ScanWindow* specify the waiting time and the duration of scans.

Gettable properties *IsAdapterAvailable*, *IsAdapterConnected* and *IsScanning* indicate the hardware adapter availability, if there is an established connection and if there is an active scanning.

The *AddedSensors* and *DiscoveredSensors* are lists containing the paired sensors and the result of the last sensor scan, respectively. Both are of type *ObservableCollection* because this dynamic collection allows the entities that use it be aware and notified when changes to the collection occur. For example, if we create a *ListView* UI component, which is a view that displays a collection of data as a vertical list, and specify an *ObservableCollection* collection as its data source, the UI component would be automatically refreshed as contents of the source list change.

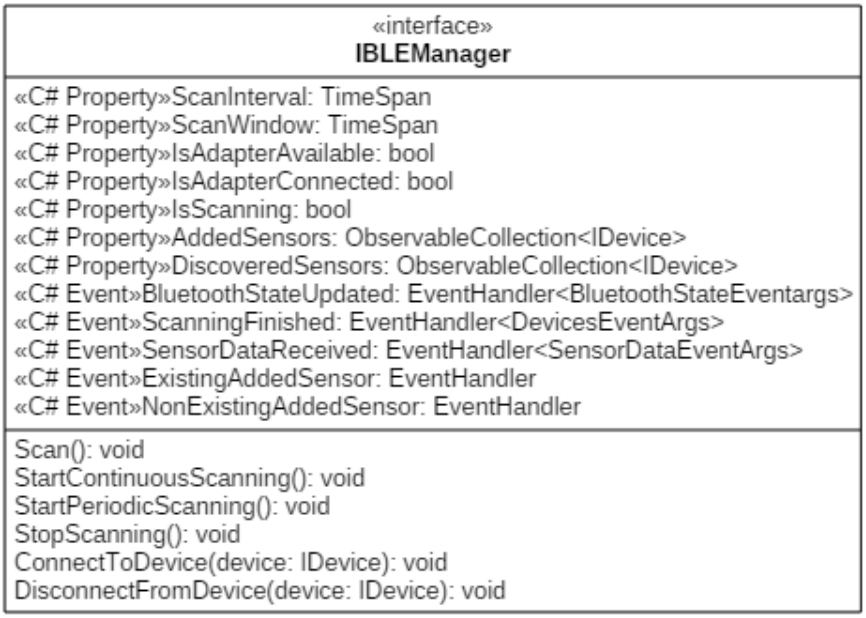


Figure 4.2: IBLEManager interface

BLE Manager provides a set of events that other application modules can subscribe to, to be notified on the following occurrences:

BluetoothStateChanged - Event raised when adapter's state changes (e.g. user turns off the Bluetooth), providing the new state to the subscribed callbacks.

ScanningFinished - Event raised when a scan cycle finishes, providing the scan result to the subscribed callbacks.

SensorDataReceived - Event raised upon receiving new data from the connected sensor, providing the sensor data to the subscribed callbacks.

ExistingAddedSensor - Event raised when the first sensor is paired at the application settings.

NonExistingAddedSensor - Event raised when the unique paired sensor is removed from the application settings.

The module also provides the following operations:

Scan - Performs a single scan for bicycle BLE sensors, with a duration specified by the *ScanWindow* property.

StartContinuousScanning - Starts scanning continuously, until *StopScanning()* is called.

StartPeriodicScanning - Starts scanning for *ScanWindow* every *ScanInterval*, until *StopScanning()* is called

StopScanning - Stops the scanning

ConnectToDevice - Establishes an connection with a given device.

DisconnectFromDevice - Ends an established connection with a given device.

To increase code sharing between the platforms and avoid code duplication, part of the interface's implementation is done within the *core project* and the other part within the *platform specific project*. The implementation from the platform side is required because native SDK must be used to interact with BLE devices.

4.4.1 BLE Manager platform-specific implementation

To implement the platform-specific part of the *BLE Manager* module, we use the *Core Bluetooth*[52] framework, more specifically its *CBCentralManager* [53] class. This class provides the required methods to discover and communicate with BLE devices.

On *CBCentralManager* instantiation, we pass a delegate class instance as one of its arguments. The delegate class extends the *CBCentralManagerDelegate* class, and implements the following of its relevant methods:

DiscoveredPeripheral - invoked for every discovered scanned peripheral;

ConnectedPeripheral - invoked when a connection with a peripheral establishes;

DisconnectedPeripheral - invoked when a connection with a peripheral ends;

FailedToConnectPeripheral - invoked when a connection with a peripheral fails.

The following sections present important implementation details of the scanning and connection with peripherals using the *CBCentralManager* class, where during the explanation we refer the aforementioned delegate class overriden methods.

4.4.2 Scanning

Every BLE peripheral advertises a list of *services* it contains. Each *service* has an assigned UUID. To start scanning for a specific type of peripheral, we start the scanning providing UUIDs of the *services* they advertise. The *CBCentralManager* class contains a *ScanForPeripherals* operation, which we invoke

with the UUID of Cycling Speed and Cadence (CSC) [54] *service* as parameter, to scan exclusively for bicycle sensors. To stop the scanning operation, we use the *StopScan* operation. After starting the scanning, the previously listed *DiscoveredPeripheral* method of the delegate class is invoked when a peripheral is discovered. It provides a peripheral object of type *CBPeripheral* as an argument, which represents the discovered peripheral, and is used whenever we want to establish or end a connection with the peripheral. As we present in the next section.

As mentioned in Section 2.4.2, BLE scanning devices usually can configure its scanning through *scan window* and *scan interval* parameters. This controls the Bluetooth adapter's usage, and decreasing its usage also decreases the scanning device's power consumption. However, *Core Bluetooth* does not provide a way of controlling these parameters, which are only controlled by the iOS.

BLE Manager module provides three types of scanning. A single scan, a continuous scanning and a periodic scanning, controllable by the *ScanInterval* and *ScanWindow* properties. To implement each of those, we just invoke the *ScanForPeripherals* and *StopScan* methods from the *CBCentralManager* native class. The single scan implementation invokes the *ScanForPeripherals* method, waits the period of time specified by the *ScanWindow*, and then invokes the *StopScan* method. The continuous scanning just invokes the *ScanForPeripherals*. Finally, the periodic scanning invokes the *ScanForPeripherals*, waits the *ScanWindow* period of time, invokes the *StopScan*, waits *ScanInterval* period of time, and repeats the process. Both continuous and periodic scans keep scanning until the *StopScanning* operation from *BLE Manager* module is invoked, which native implementation simply invokes the *StopScan* method.

Starting from iOS version 7, an iOS application is only allowed to run for 180 seconds after being backgrounded, unless the application uses any of the background execution modes. To avoid this, and to allow *BikeApp* scan in the background, we declare the *Uses Bluetooth LE accessories background execution mode* [43]. To prevent the application of being terminated, we start the continuous scanning when the application goes into the background. This sounds highly inefficient in terms of battery consumption, but as we already mentioned, iOS automatically manages its Bluetooth scanning, adjusting its *scan window* and *scan interval* parameters when the application goes into the background, to avoid the smartphone's battery draining. In Section 5.6 we also analyze the battery consumption overhead introduced with the background BLE scanning.

When the application is in the foreground, we switch to the periodic scanning, to reduce the Bluetooth adapter's usage. We decided to use *ScanWindow* and *ScanInterval* set to 1 and 10 seconds, respectively, because a scan with a duration of 1 seconds is always enough to discover the bicycle sensor, and the waiting period of 10 seconds between the scans minimizes the adapter's usage and consequently the power consumption. The chosen scanning interval only affects the discovery time of the bycycle sensor when the application is in the foreground, but since it is more likely that the users turn off the

screen before starting cycling, which activates the continuos scanning, the sensor detection time does not depend on the chosen scanning interval of 10 seconds.

4.4.3 Communication with the sensor

During the implementation of the communication with the sensor, the *Core Bluetooth Programming Guide* [55] can be followed to understand all the required steps to start receiving sensor's data.

BLE cadence sensors share information with the mobile phone through GATT communication, which we explained in section 2.4.4. The sensor acts as a GATT server, and defines a hierarchical data structure that is exposed to connected GATT clients. Its hierarchy is composed of a single *profile* containing a mandatory CSC [54] *service*. The CSC service contains a *CSC Measurement* [56] characteristic, which provides crank and wheel revolutions data.

The sensor only sends its data to connected devices that subscribes to its *CSC Measurement* characteristic. Once the subscription to the characteristic is done, the sensor starts periodically sending its *value*, with a data packet structure represented in Figure 4.3. The first data byte contains flags field and the first two bits indicate whether the value contains wheel revolution data, crank revolution data or both. Wheel revolution data, if present, consists of a 32-bit cumulative count of wheel revolutions and a 16-bit value representing the time the last wheel event was measured, in units of 1/1024 of a second. Crank revolution data consists of a 16-bit cumulative count of revolutions of the crank, and a similar 16-bit last event time value.

We are already familiar with these values from the explanation of the cycling detection mechanism presented in Section 3.4.1. Let's see all the steps that platform-specific implementation of *BLE Manager* follows to start receiving the sensor data.

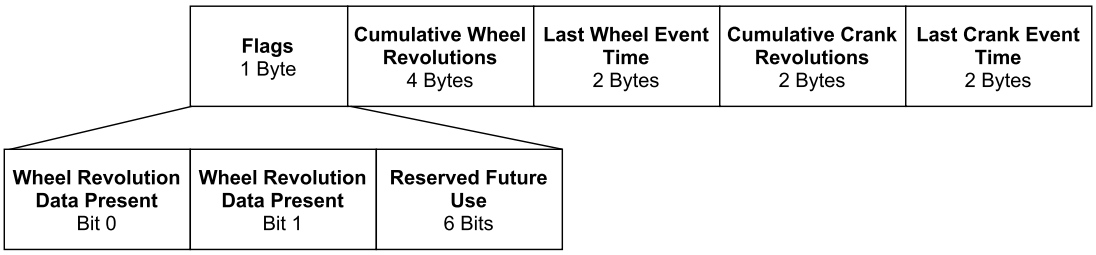


Figure 4.3: Raw advertised data structure

The first thing that is required, is to establish a connection with the sensor. This is done by invoking the *ConnectPeripheral* method from the *CBCentralManager*, with the sensor's *CBPeripheral* object as an argument. As previously mentioned, this object is obtained through the scanning results, provided by the delegate's *DiscoveredPeripheral* invoked method.

When the connection establishes, delegate's *ConnectedPeripheral* is invoked and passes a *CBPe-*

ripheral object of the connected peripheral as argument. In this method we create an instance of a sensor delegate class, which extends from *CBPeripheralDelegate* [57] class, and overrides its *DiscoveredSensor*, *DiscoveredCharacteristic* and *UpdatedCharacteristicValue* methods. Then, we assign this delegate instance to *CBPeripheral* object's *Delegate* property. Finally, we invoke the *DiscoverServices* method over the *CBPeripheral* object, which requests all the sensor's GATT services.

Requesting the peripheral services results in its delegate's *DiscoveredSensor* invocation, which has the implementation represented in Listing 4.3. Because we only need the CSC service, we loop through all the offered by the sensor services, and find it by checking the service's UUID. Then, we request CSC service's characteristics, by invoking the *DiscoverCharacteristics* method over the peripheral object and passing the CSC service object as argument.

```
1 public override void DiscoveredService(CBPeripheral peripheral, NSError error){
2
3     foreach(var service in peripheral.Services) {
4
5         if(service.UUID == CSC_SERVICE_UUID){
6             peripheral.DiscoverCharacteristics(service);
7         }
8     }
9 }
```

Listing 4.3: Extracting values from raw advertising data

Similarly, requesting the peripheral characteristics also results in its delegate's method invocation, called *DiscoveredCharacteristic*. Listing 4.4 shows its implementation, where we loop through all the characteristics contained within the service, compare its UUID to find the *CSC Measurement* characteristic, and subscribe to it, by setting its *notify* value to *true*. From this moment, sensor delegate's *UpdatedCharacteristicValue* method is called every time the sensor sends its data. Now, let's see how we extract the meaningful values from the raw advertised sensor data.

```
1  public override void DiscoveredCharacteristic(CBPeripheral peripheral,
2                                                 CBService service, NSError error){
3
4      foreach(var characteristic in service.Characteristics) {
5
6          if(characteristic.UUID == CSC_MEASUREMENT_UUID)
7              service.Peripheral.SetNotifyValue(true, characteristic);
8          }
9      }
10 }
```

Listing 4.4: Extracting values from raw advertising data

Listing 4.5 shows the implementation of the invoked *UpdatedCharacteristicValue* method, when sensor data is received. First, we copy the data contained within the *Value* parameter into a byte array (lines 4-9). Then we check the flags bits, contained within the first byte, to determine if wheel revolution and crank revolution data are present (lines 14-15). Finally, using the C# *BinaryReader* and considering

the previously described packet structure, we read and store the meaningful values into variables (lines 23-36). After this, we invoke the *SensorDataReceived* event of *BLE Manager* and its argument passes the just extracted values. This way, the subscribed to this event entities can access and interpret the sensor data.

```
1  unsafe public override void UpdatedCharacterteristicValue(CBPeripheral peripheral,
2                                  CBCharacteristic characteristic, NSError error) {
3
4      var dataSize = (int) characteristic.Value.Length;
5      byte[] data = new byte[dataSize];
6
7      // Preparing raw advertised data
8      Marshal.Copy(characteristic.Value.Bytes, data, 0, dataSize);
9
10     // Checking which information is included in the flags
11     // First bit - Wheel Revolution data presence
12     // Second bit - Crank Revolution data presence
13     var wheelRevPresent = (data[0] & 0x01) > 0;
14     var crankRevPreset = (data[0] & 0x02) > 0;
15
16     int wheelRevolutions = 0;
17     int lastWheelEventTime = 0;
18     int crankRevolutions = 0;
19     int lastCrankEventTime = 0;
20
21     // Building a stream reader to ease data reading
22     var reader = new BinaryReader(new MemoryStream(data));
23
24     // Advancing the reader's offset by 1 byte (flags)
25     reader.ReadByte();
26
27     if(wheelRevPresent) {
28         wheelRevolutions = (int) reader.ReadUInt32();
29         lastWheelEventTime = reader.ReadUInt16();
30     }
31
32     if(crankRevPreset) {
33         crankRevolutions = reader.ReadUInt16();
34         lastCrankEventTime = reader.ReadUInt16();
35     }
36
37     // Retrieved sensor values ready to be used
38 }
```

Listing 4.5: Extracting values from raw advertising data

4.5 Cycling Detector module

The *Cycling Detector* module is entirely implemented within the core project. Figure 4.4 presents the interface this module follows, which serves as an entry point to use the cycling detection mechanism. The property *IsCycling* indicates whether or not cycling is being detected. The events *CyclingStarted* and *CyclingStopped* allow other application modules to be notified when the cycling state changes. For example, we use the subscription to those events to automatically start and stop the user tracking.

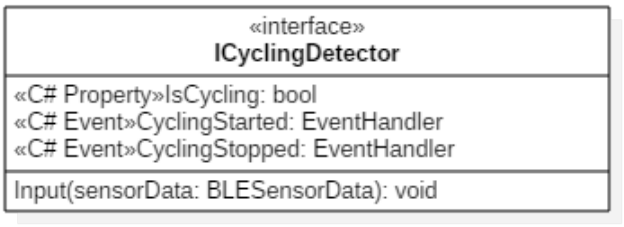


Figure 4.4: ICyclingDetector interface

As already mentioned in the Section 3.4.1, *Cycling Detector* uses the received sensor data to determine the cycling activity. The sensor data is obtained through *SensorDataReceived* event subscription of *BLE Manager*, as shown in Listing 4.5. This event starts to be triggered after *BLE Manager* establishes a connection with a sensor. The subscribed callback invokes the *Input* operation, passing it the sensor data, namely the *Crank Revolutions*, *Wheel Revolutions*, *Last Crank Event Time* and *Last Wheel Event Time* counters. Then, the *Input* method applies the logic previously described in Section 3.4.1 to determine the cycling state. When the cycling activity is detected, *CyclingStarted* event is raised.

```
1 BLEManager.Instance.SensorDataReceived += (s,e) => Input(e.Data);
```

Listing 4.6: Sensor data subscription

In order to use the *Cycling Detector* module within our application, we just use its *CyclingStarted* and *CyclingStopped* events and the *IsCycling* property. Listing 4.7 shows the event subscriptions that are required to make the *TrackingManager* automatically start and stop the tracking with the cycling detection state changes.

```
1 CyclingDetector.Instance.CyclingStarted += (s,e) => {
2     TrackingManager.Instance.StartTracking();
3 }
4
5 CyclingDetector.Instance.CyclingStopped += (s,e) => {
6     TrackingManager.Instance.StopTracking();
7 }
```

Listing 4.7: Automatic tracking enabling

Users are eligible for *cycle-to-spot* rewards after cycling for a period of 1.5 minutes. This is also easily achievable through an event subscription, as we can see in Listing 4.8. The subscribed callback is called when the cycling activity is detected, waits the required period of time, then checks if the user is still cycling and confirms to the *RewardEligibilityManager*.

```
1 CyclingDetector.Instance.CyclingStarted += async (s,e) => {
2
3     await Task.Delay(TimeSpan.FromMinutes(1.5));
4
```

```
5       if(CyclingDetector.Instance.IsCycling){
6           RewardEligibilityManager.Instance.Input(EligibilityCondition.ConfirmCycling);
7       }
8   }
```

Listing 4.8: Turning users eligible after cycling 1.5 minutes

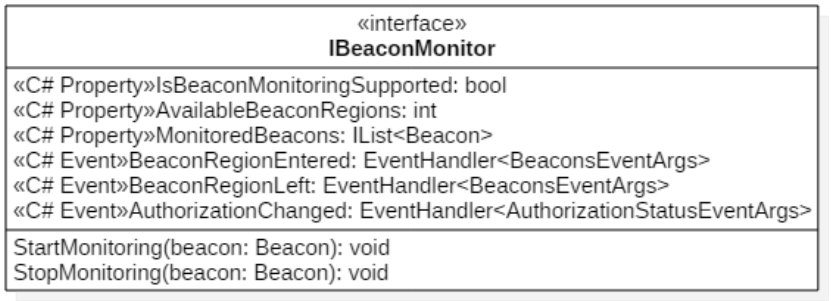


Figure 4.5: IBeaconMonitor interface

4.6 Beacon Monitor module

The *Beacon Monitor* module is responsible for monitoring the BLE location beacons, and its implementations follows the interface presented in Figure 4.5. Its items have the following purposes.

IsBeaconMonitoringSupported indicates whether or not beacon monitoring is supported by the hardware. *AvailableBeaconRegions* specifies how many more regions can be monitored. This parameter exists because in iOS there is an limitation of 20 simultaneous monitored regions. The last *MonitoredBeacons* provides a list of the beacon regions being monitored.

There are three events that can be subscribed by other applications modules to fully take advantage of its functionality: *BeaconRegionEntered*, *BeaconRegionLeft* and *AuthorizationChanged*. *BeaconRegionEntered* is triggered when a monitored beacon region appears in range. It provides information about the beacons in range to the subscribed callbacks. This information includes the beacon's UUID, *Major* and *Minor* identifiers, and an estimated distance to the beacon. *BeaconRegionLeft* is triggered when the monitored beacon region becomes out of range. Finally, *AuthorizationChanged* event is triggered when the location permissions are changed in the iOS device settings.

Beacon Monitor provides two operations, to start and to stop the monitoring of a given beacon region. Both operations receive a *Beacon* object that represent the target beacons, for example, if we invoke the *StartMonitoring* operation and pass it an instance of a *Beacon* class with its *UUID* parameter set to a given value, then all the beacons that have this UUID are also monitored, not considering their *Major* and *Minor* identifiers. When a beacon is being monitored, the *BeaconRegionEntered* event is triggered if the mobile device appears in range of that beacon. Similarly with the *BeaconRegionLeft* event. This

approach allows us to monitor all the shop beacons, which are configured with the same UUID, by requesting a single monitoring.

Similarly to the *BLE Manager* module, *Beacon Monitor* module was also developed within the core and platform-specific projects. In the next section we discuss the protocol the beacons use and the used framework. Then, in the following section we present some platform-specific implementation aspects regarding the monitoring feature provided by the *Beacon Monitor* module.

4.6.1 Beacon protocol

There are many BLE protocols used by the location beacons. Some of them are manufacturer's proprietary (e.g., *Estimote Telemetry* [58]), and are created to advertise manufacturer specific data. However, using them limits the system to strictly work with specific manufacturer beacons. To ensure that the system works with a larger variety of beacons from different manufacturers, Apple's *iBeacon* [59] or Google's *Eddystone* [60] protocols should be used. This is possible because many beacons available on the market do support both protocols.

In Figure 4.6 we can see the packet structure of *iBeacon* and *Eddystone-UID* protocol advertised packets. UUID, *Major* and *Minor* fields uniquely identify each *iBeacon* beacon. In *Eddystone-UID*, same is achievable through *Namespace ID* and *Instance ID* fields value concatenation. Both protocols send a factory calibrated RSSI value measured at 1 meter in *iBeacon* and 0 meters in *Eddystone-UI*.

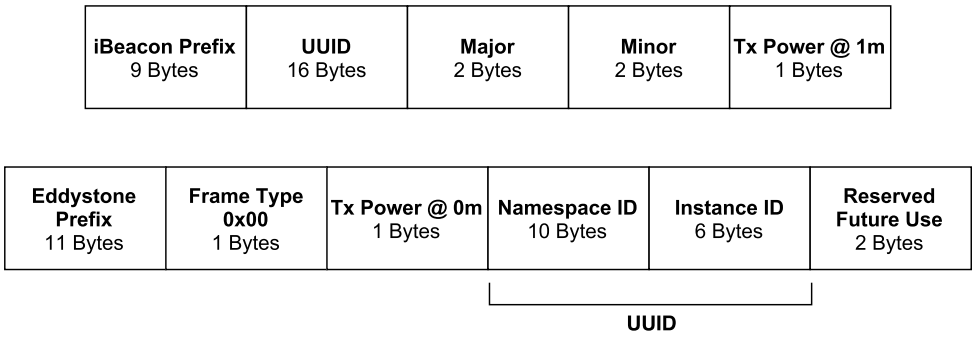


Figure 4.6: iBeacon and Eddystone-UID protocol packet structure

To not depend on any specific protocol, our first approach was to use the *Core Bluetooth* framework, which as we already saw provides the classes needed to communicate with an BLE wireless devices. However, the UUID property of the discovered *CBPeripheral* beacon object does not match the real beacon's UUID, advertised in the data packets [61]. Instead, for security reasons, it is randomly generated from device to device. This means that the same beacon has different associated UUID values when scanned in different devices.

iOS SDK contains another framework which interacts with BLE beacons, named *Core Location* [62]. With *Core Location* it is possible to detect the proximity to monitored beacons by their real UUID, and

obtain a distance estimation to the beacons in range. However, the framework works exclusively with beacons using the *iBeacon* protocol. We do not consider this a big limitation, because this protocol is widely used and many beacons from different manufacturers support it. Additionally, Android's SDK provides full access to the raw beacon advertised data, making our solution using *iBeacon* also compatible. Furthermore, there are native and cross-platform Xamarin libraries, allowing Android devices to use beacons much like iOS devices do, which makes the expansion of our solution to the Android platform easy.

4.6.2 Beacon Monitor platform-specific implementation

The monitoring functionality provided by the *Beacon Monitor* is implemented using two features available in *Core Location* framework, namely *region monitoring* and *region ranging*. To use them, we use an instance of the *CCLocationManager* [63] class and subscribe callbacks to the following of its events:

RegionEntered - event raised when any of the monitored beacon region appears in range;

RegionLeft - event raised when the detected monitored beacon region becomes out of range;

DidRangeBeacons - event raised every second, providing beacons in range;

When the *StartMonitoring* or *StopMonitoring* methods of *Beacon Monitor* module are invoked, their native implementations invoke the *StartMonitoring* and *StopMonitoring* methods of the *CCLocationManager* class. Both methods receive an instance of *CLBeaconRegion* as an argument, which represents the target beacon region, and is identified by a UUID.

When a monitored beacon appears in range, *RegionEntered* event is triggered, and our subscribed callback is invoked. This event tells the UUID of the entered region, but provides no information about the *Major* and *Minor* identifiers of the beacon that is in range and triggered the *RegionEntered* event. To get these identifiers we use the ranging.

Ranging continuosly scans for beacons and provides results every second. To start the ranging we invoke the *StartRangingBeacons* operation, also contained within the *CCLocationManager* class. Similarly to the monitoring, we provide it an argument of type *CLBeaconRegion* to identify the beacon region to be ranged with a UUID. After starting the ranging, the *DidRangeBeacons* event starts to be triggered every second, providing a list of *CLBeacon* objects, which represent the available in the vicinity beacons. *CLBeacon* object contains *Major* and *Minor* identifiers, and also an estimated to the beacon distance as *Accuracy* parameter. Just considering the first ranging result is unreliable, because beacon's RSSI fluctuates over time, and the estimated distance directly depends on RSSI. Thus, we keep the ranging running for a few seconds and consider the calculated average *Accuracy* value as

beacon's estimated distance. After this, we trigger the *Beacon Monitor*'s *BeaconRegionEntered* event and provide it information about the beacons in range, such as their UUID, *Major* and *Minor* identifiers, and an estimated to the beacon distance.

We performed a set of tests to our location detection mechanism with the ranging being run for 10 seconds and consider this time period enough to correctly identify the closest beacon in situations where two beacons are in range, which happens where there are two shops next to each other. The experiment is explained detail in Section 5.4.

4.6.3 Beacon discovery time

In the previous section we presented the Core Location framework and how it allows to monitor BLE beacons. However, we did not mention anything about beacon discovery times. Now, that we introduced the *Core Bluetooth* framework, let's see how it relates to the beacon discovery using *Core Location*.

The *Core Location* API for BLE beacon monitoring is a thin layer on top of *Core Bluetooth*. In order to detect beacons, iOS performs a brief BLE scan and checks if any discovered device contains the monitored beacon signature. Thus, the time to discover a monitored beacon, which triggers the *region entered* event, directly depends on how often iOS does a BLE scan. This could be problematic if only the *Core Location* framework was used, because it would only rely on the scans performed by the iOS. However, since we perform continuous scanning in the background, we get the best Core Location performance.

Because the iOS reduces the scanning duty cycle when the application is brought into the background, reasonable *advertising interval* should be set on BLE beacons to guarantee optimal discovery times. Apple recommends setting the beacons with 100 ms advertising interval when monitoring with *Core Location*. In Section 5.2 we analyze the background discovery times of a beacon with *100 ms* and *300 ms* advertising intervals.

4.7 Location Detector module

The *Location Detector* modules is responsible for detecting when the user approaches or enters any of the shops. This module provides the interface represented in Figure 4.7, for interaction with the location detection mechanism. It contains a settable *BeaconSpotTable* property, which is a C# *Dictionary* collection containing beacon identifiers as a *key*, and the corresponding shop object from the application's business model as a *value*. This property is initialized before using the module and updated whenever *BikeApps* receives new shops from the backend.

Location Detector module is entirely implemented within the core project and its implementation uses the *Beacon Monitor* module. *Location Detector* requests the *Beacon Monitor* to start monitoring for all

Figure 4.7: ILocationDetector interface

the shop beacons, by invoking the *StartMonitoring* operation and passing it a *Beacon* object with the UUID set to the shop beacon's UUID. Additionally, the *Location Detector* subscribes its *BeaconRegionEntered* event, to be notified whenever a shop beacon appears in range. Finally, *Location Detector* determines the visited shop, using its *BeaconSpotTable*, by the steps previously described in Section 3.4.2.

When *Location Detector* determines the shop, it triggers its *SpotEntered* event, which notifies the subscribed entities and provides the object representing the visited shop. This object can also be consulted with the *CurrentSpot* property.

```
LocationDetector.Instance.SpotEntered += (s,e) => {

    if(!RewardEligibilityManager.Instance.IsEligible){
        return;
    }

    ShowSpotEnteredNotification(e.Spot);

    var challenges = e.Spot.Challenges;

    if(challenges.Count() >= 1){

        var cycledDistanceChallenge = challenges.Where(c => c.IsCycledDistance).First();

        if(cycledDistanceChallenge.RemainingDistance == 0){
            var challengeVM = new ChallengeViewModel(cycledDistanceChallenge);

            // Showing 'cycled-distance' reward claim page
            Navigation.PushAsync(new ClaimPage(challegeVM));
            return;
        }
    }

    var challengeVM = new ChallengeViewModel(challenges[0]);

    // Showing 'cycle-to-spot' reward claim page
    Navigation.PushAsync(new ClaimPage(challegeVM));
}
```

Listing 4.9: Notifying and opening claim page upon shop detection

The integration of the *Location Detector* module within the application is also quite simple. We just subscribe a callback to the *SpotEntered* event, as shown in Listing 4.9, to present the notification and automatically open the reward claiming page. The first thing the executed callback checks is if the user

is eligible, which is enabled when the cycling activity is detected, as we previously saw. If the user is eligible then we present the notification and check how many *challenges* there are. *Challenge* is the class that represents both *cycle-to-spot* and *cycled-distance* rewards, and each spot has always one *cycle-to-spot* and optionally a *cycled-distance*. Because the *cycled-distance* is more valuable, we first check if it exists and can be claimed. This is done by checking if its object's *RemainingDistance* property equals zero (lines 11-15), which means that all the required distance was already cycled and the reward can be claimed. If the *cycled-distance* reward exists and complete, then we show its claiming page (line 19), otherwise we show the 'cycle-to-spot' reward claiming page (line 27).

4.8 Summary

In this chapter we presented the implementation details of our solution, more specifically how the application was developed and how the cycling and location detection modules were implemented.

Developing using Xamarin increases the productivity when the same application needs to be developed for multiple platforms, because it avoids duplicate code in separate native implementations by implementing in the *core project*. *BikeApp* was only implemented for the iOS platform, however the implementation is done in a way to make the extension to the Android platform also possible and easy.

Even though *Core Bluetooth* framework scans BLE devices, the scanned beacon's UUID does not correspond to its real UUID, advertised in the packets. Thus, we use the *Core Location* framework to detect the beacons. This framework has a limitation of only interacting with beacons that follow the *iBeacon* protocol, which limits our solution to work strictly with this protocol. However, we do not consider this a problem because this protocol is supported by many beacon manufacturers.

The *region monitoring* feature provided by the *Core Location* framework does not provide information about the beacon's *Major* and *Minor* identifiers, when the monitoring is requested only by its UUID. We overcome this problem by combining the *region monitoring* with the *region ranging*, and start ranging to actively scan for nearby beacons where any of the monitored beacons appear in range. By scanning for a couple of seconds we can determine the nearby beacon's *Major* and *Minor* identifiers, and obtain an average value of the estimated to the beacon distance.

The *Core Bluetooth* framework, used in the native implementation of the cycling detection mechanism, also contains some limitations. It does not provide a way of configuring the BLE *scan window* and *scan interval* parameters. Additionally, the only way of getting the application to scan in the background is by starting the continuous scanning. iOS adapts the BLE scanning parameters automatically, making the smartphone scanning with a lower duty cycle, to reduce the Bluetooth adapter usage and consequently the battery consumption. However, this process is officially undocumented and the only way of controlling the peripheral discovery times during the background scanning, is by adjusting its *advertising*

interval parameter.

Following the *event-driven* design makes the integration of the cycling and location detection mechanism very easy. The interaction with both is done exclusively by the subscription to the events offered by the *Cycling Detector* and *Location Detector* modules, requiring no application-specific adaptation or changes.

5

Evaluation

Contents

5.1 Introduction 67
5.2 Advertising Intervals and Discovery Times 67
5.3 Estimated vs real distance 68
5.4 Location Detection 72
5.5 Cycling Detection 75
5.6 Smartphone Battery Consumption 76
5.7 Usability Evaluation 78
5.8 Summary 81

5.1 Introduction

The present chapter describes the evaluation methodology as well as the experiments performed to understand the viability of our solution and determine optimal conditions to achieve the system requirements listed in Section 1.2.

We evaluate the location detection, cycling detection and smartphone's battery consumption, based on the results collected during the performed experiments. The usability of *BikeApp* is evaluated based on the feedback given by the users on surveys, after using the application and performing the required tasks.

First, we analyze how the configurable beacon's *advertising interval* parameter affects the beacon discovery time, to determine an optimal value to be used in our solution. Then, in Section 5.3 we analyze how the estimated to the beacon distance correlates with the real distance. We also try different *transmission power* parameters to determine if the correlation holds. The next Section 5.4 analyzes the behavior of our location detection mechanism in a real word worst-case scenario, with shops side to side. Section 5.5 presents an evaluation on the cycling detection mechanism in terms of sensor connection time, cycling detection time and detection accuracy. Section 5.6 evaluates our solution in terms of its power consumption when the smartphone is idle and when cycling. We compare the power consumption to the smartphone's "base consumption", i.e., when there is no running application in the background, and also to *Biklio*'s consumption. Finally, in Section 5.7 we show the results of *BikeApp* usability evaluation with end-users.

5.2 Advertising Intervals and Discovery Times

This section analyzes beacon discovery times when scanning beacons with different *advertising intervals*. This is required, to determine an optimal beacon's *advertising interval* value, as a trade-off between beacon discovery time and its battery life. Apple does not document how iOS adjusts the background scans, and high advertising intervals (e.g., 1000 ms) may result in too long beacon discovery times or even its nonexistence. On the other hand, setting a very low *advertising interval* value (e.g., 20 ms) will quickly drain the beacon's battery, which is inconvenient for shop owners that should not be worried about frequent battery replacements.

Apple recommends setting the 100 ms *advertising interval* for the *iBeacon* profile. We analyze the discovery times of beacons with this recommended configuration, then we analyze how the beacon discovery time changes over time by slightly increasing the *advertising interval* to 300 ms. Greater values (e.g., 500 ms) are not considered, because they significantly degrade the system's scanning performance.

5.2.1 Methodology

In this experiment we set our smartphone to scan for BLE beacons. The scanning was performed in the background and with a duration of 1 hour for each beacon *advertising interval*. We choose this duration to understand how the discovery time varies over time with different *advertising interval* configurations, because the application can remain in the background for long periods of time and we want to minimize the time to discover a beacon.

In the first trial we configure the *advertising interval* to the 100 ms value. In the second trial we increase it to 300 ms.

During the scanning we registered each discovery time into a logfile to be able to analyze how the discovery time varies by increasing the *advertising interval* and try to understand how iOS changes its background scanning.

5.2.2 Results

Figure 5.1 and Figure 5.2 show the beacon discovery times over 1 hour of background scanning, using 100 ms and 300 ms *advertising interval*, respectively. The y-axis contains the time in seconds that each beacon discovery took. The x-axis contains the total scanning duration.

In both interval configurations the majority of discoveries are within the 0-10 seconds range. But we can not observe any pattern or rule by which the discovery time varies. The discoveries with a higher *advertising interval* may sometimes take a little bit longer, as expected. However, both contain unpredictably long discoveries, even during the first minutes of scanning. This is result of Apple's way of adjusting the *scan window* and *scan interval* parameters, not controllable by the developers.

We also noticed that the beacons are always immediately discovered when the phone is woken up by pressing the "Home" or "Power" button. This means that the beacons in vicinity would be instantly discovered as soon as the users start using the smartphone.

Regarding the question which *advertising interval* should be used in beacons, as we can see on the presented charts, there is no substantial advantage of using the recommended 100 ms value over the 300 ms interval. This means that the beacons can be configured to 200 ms or 300 ms and benefit from prolonged battery life.

5.3 Estimated vs real distance

Transmission power is the power with which the beacon broadcasts its signal. Logically, the beacon's range directly depends on the *transmission power*, but greater *transmission power* values decrease the beacon's battery lifespan.

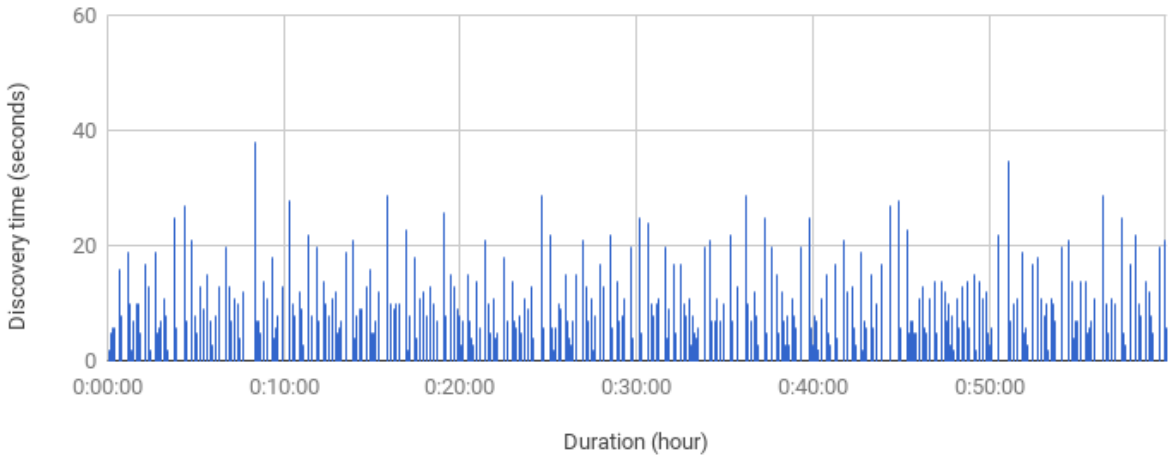


Figure 5.1: Beacon discovery times with 100 ms Advertising Interval

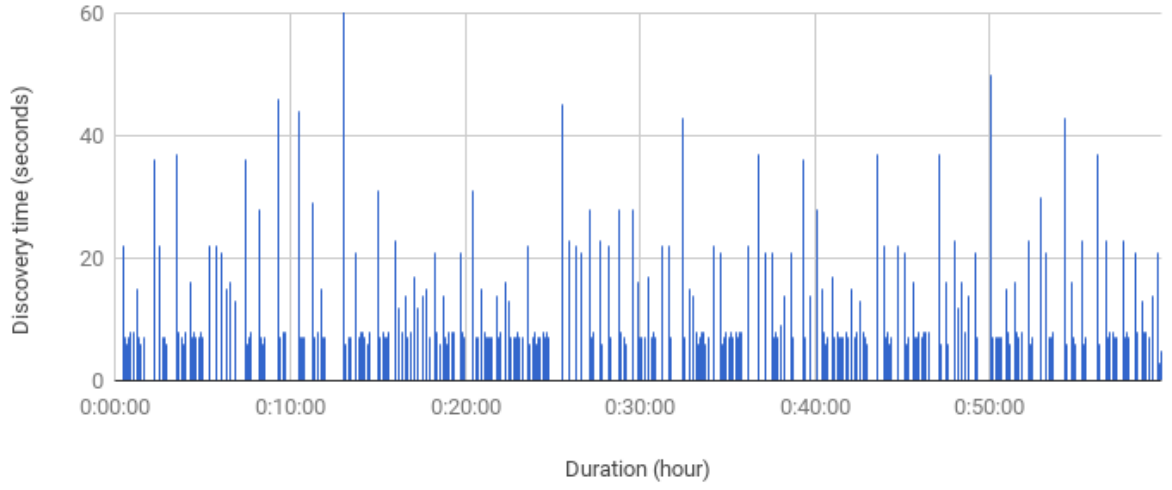


Figure 5.2: Beacon discovery times with 300 ms Advertising Interval

As mentioned in 4.6.2, the *beacon ranging* result provides a list of beacon object representing the beacons in vicinity. Each beacon object is of type *CLBeacon* and contains a RSSI and an *Accuracy* parameter, representing an estimated to the beacon distance.

We collected the beacon's RSSI and estimated distance at different distances, to understand how the signal strength and the given estimated distance correlate with the real distance.

5.3.1 Methodology

To ensure the same RSSI retrieval conditions, we placed our advertising beacon at a height of 2 meters, and collected its RSSI and estimated distance values with a line-of-sight to the beacon for a

Figure 5.3: Beacon placement and RSSI retrieval

duration of 2 minutes at each of the following distances: 2m, 5m, 10m and 15m, as shown in Figure 5.3.

The experiment was repeated with -12 and 0 dBm *transmission power* configurations, which have a theoretical range of 15 and 50 meters, respectively. These values were chosen because depending on the shop's dimensions, different beacon ranges may be needed, and we need to confirm that the reported estimated distance values have good correlation with the real distance when scanning beacons with different *transmission power* configurations.

5.3.2 Results

Figures 5.5 and 5.4 illustrate the correlation between the estimated and the real distances to the beacon. Y-axis contains the measured distance (in meters), while the x-axis contains the duration of the experiment (in minutes). The red line represents the real distance, at which the beacon's signal was captured, and the blue line represents the *CLBeacon*'s *Accuracy* parameter value. Indeed, we can see that the *Accuracy* parameter does represent a beacon's distance estimation, showing an acceptably good correlation. The *Pearson correlation coefficient* values are 0.958 and 0.928, respectively for 0 dBm and -12 dBm *transmission power*. Additionally, the correlation is strong until transitioning from the 5 to 10 meters distance, which occurred at the time of 04:00 minutes.

Figure 5.6 and 5.7 line charts are graphical representations of the collected RSSI values. They

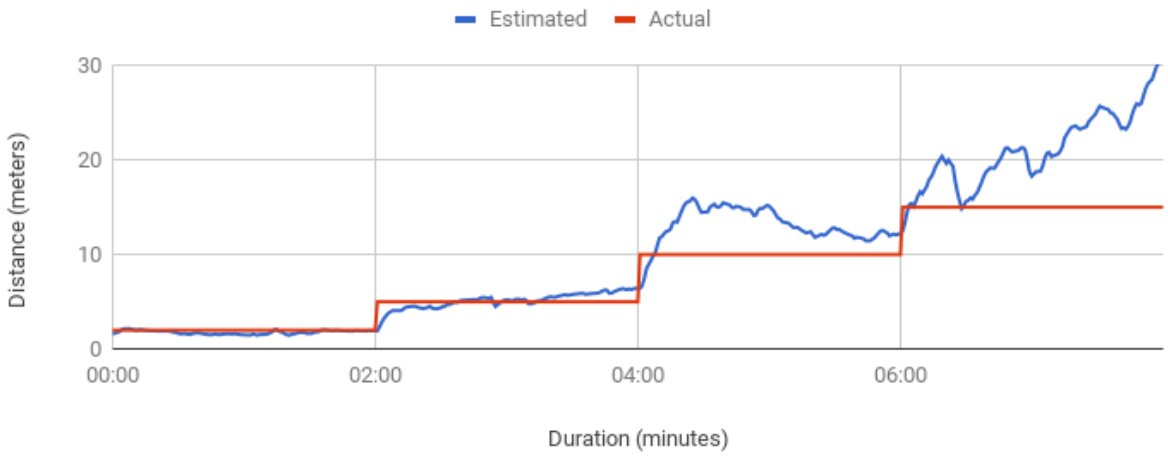


Figure 5.4: Real and estimated distances at 0 dBm *transmission power*

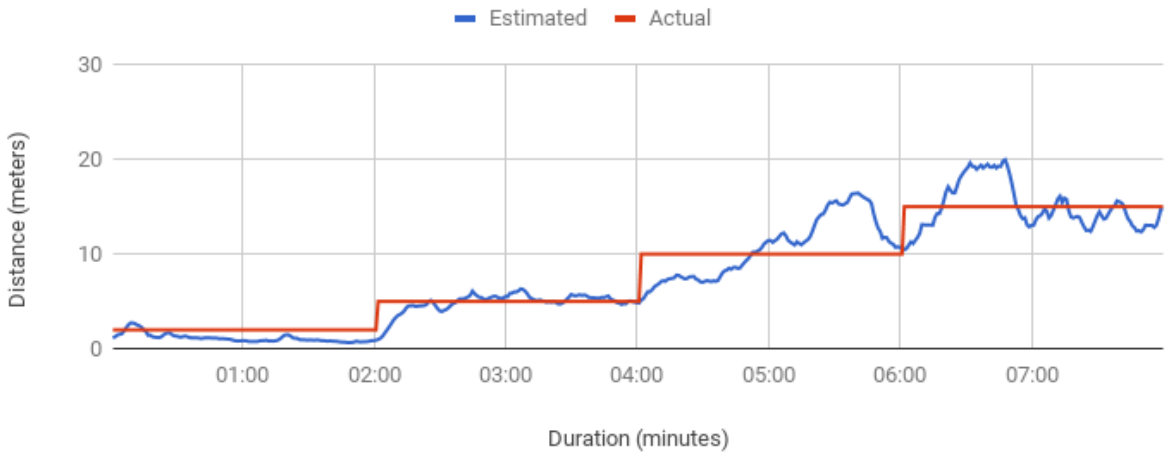


Figure 5.5: Real and estimated distances at -12 dBm *transmission power*

contain the RSSI value (in dB) on the y-axis and the duration of the experiment (in minutes) on the x-axis, showing how the RSSI varies over time with two different *transmission power* configurations. As we can see, the signal strength gets weaker when we move further from the beacon, which happened every 2 minutes. Additionally, the RSSI fluctuates even when standing at the same position and sometimes peaks may be observed, which affects the reported estimated distance. But the important thing is that there is always a considerable difference in the RSSI value when being closer (0-5m) or further away (10-15m) to the beacon, and consequently in the estimated distance. Let's consider two shops close to each other, let's say A and B. Being inside shop A, the smartphone may also receive the signal from the neighboring shop B, but since its beacon is more distant, and there is a wall separating both, estimated distance to beacon B would be way greater than to beacon A.

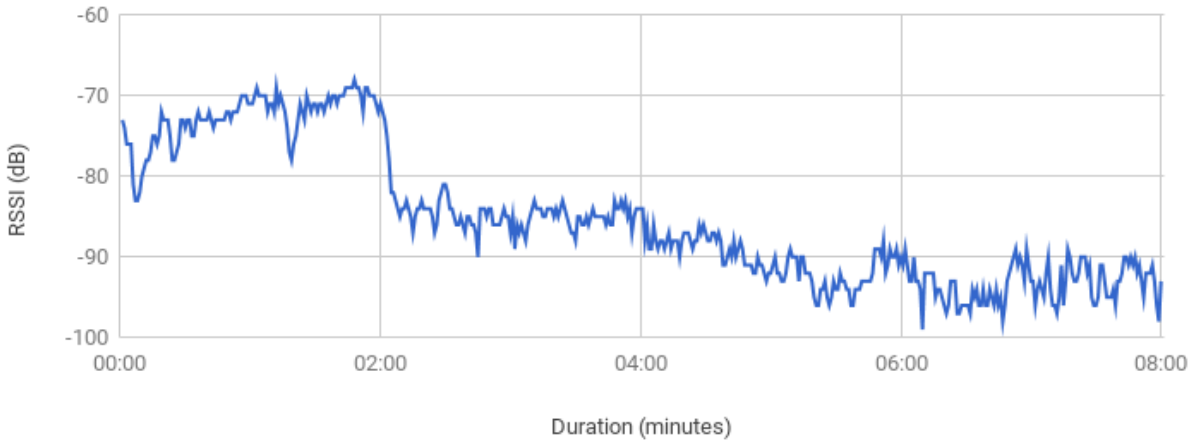


Figure 5.6: RSSI at -12 dBm *transmission power*

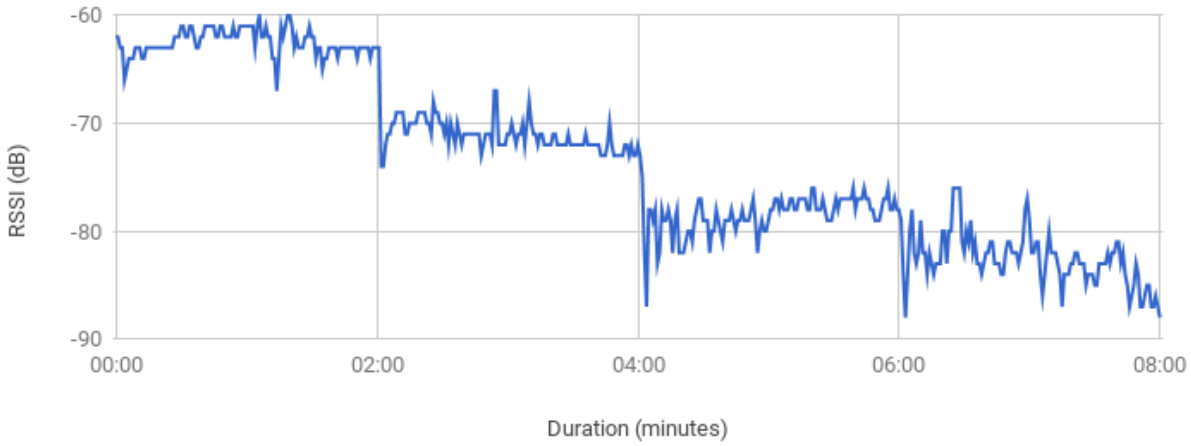


Figure 5.7: RSSI at 0 dBm *transmission power*

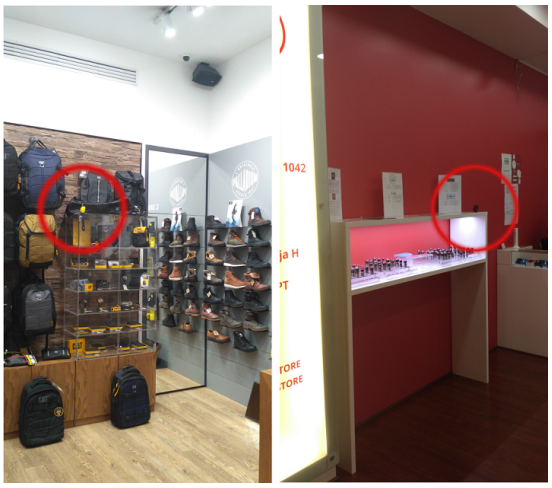
From the presented charts and observed RSSI fluctuations, we can clearly see the importance of not relying on a single *beacon ranging* result, but to take an average of this value over a period of a few seconds.

5.4 Location Detection

Now that we analyzed how the scanning behaves with different *advertising interval* values, and how the given estimated distance correlates with the real distance at different *transmission power* configurations, it is time to evaluate how our location detection mechanism would work in a real-world scenario.

To evaluate the location detection mechanism we decided to collect estimated distance values of two beacons placed in a worst-case scenario, within shops adjacent to each other, and see how the

(a) Adjacent shop A and B chosen for testing



(b) Beacon placement within the shop A and B

Figure 5.8: Shops and location beacons placement

estimated distance varies when entering one of the shops.

5.4.1 Methodology

Figure 5.8 shows the shops chosen for testing and their beacon placement, marked with red circles. We placed the red beacon closer to the wall on purpose, to represent a worst-case scenario where the beacons are close to each other, and to analyze the signal attenuation from the wall. The distance between the beacons, including the wall, was approximately 3-4 meters.

The beacons were configured to broadcast with a 300 ms *advertising interval* and with *transmission power* set to -12 dBm. Each trial consisted in approaching both shops (standing at the same distance from both) and then entering one of them, while collecting the estimated distance values of the beacons placed within each shop.

5.4.2 Results

It is important to mention that there is a serious bug in the *Core Location* framework, which appeared with the iOS 10 update. The bug was reported many times [64], however it still has not been fixed in the version we tested (iOS 10.3.2). The bug consists in no nearby beacons being detected when running the *beacon ranging*, even when standing next to the broadcasting beacon. Additionally, some of the reported beacons may have a 0 dB RSSI and a -1 m estimated distance. This problem happens sometimes and is temporarily fixed when turning on and off the Bluetooth module, or restarting the device. Because of this bug, our location detection mechanism failed sometimes at detecting the beacons nearby. However, the mechanism worked as expected when the *beacon ranging* properly reported the nearby beacons,

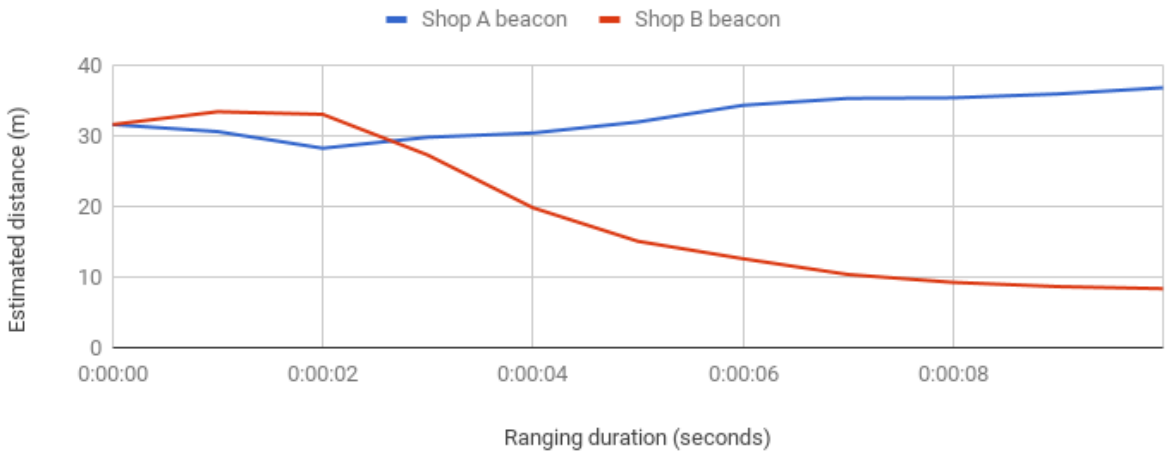


Figure 5.9: Location detection starts few meters away of both shops before entering shop B

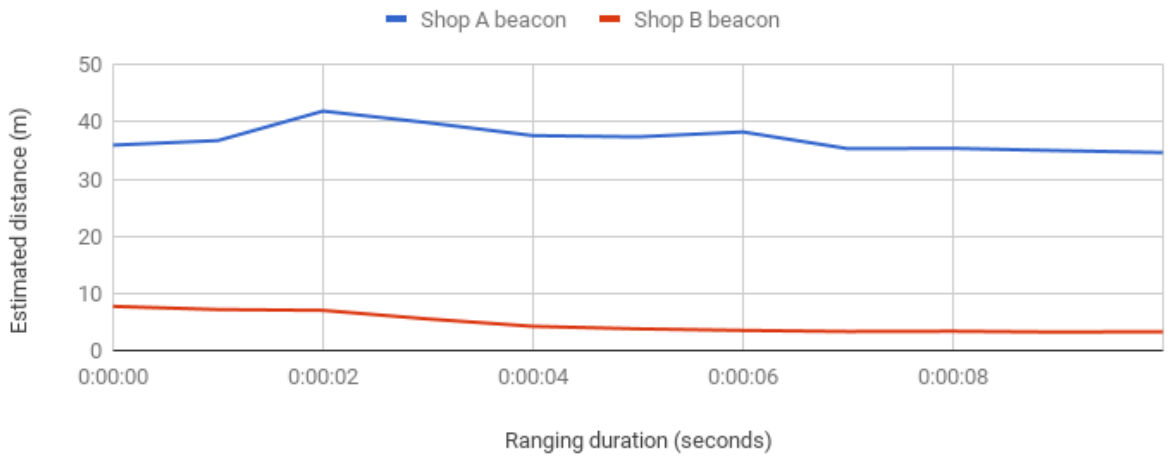


Figure 5.10: Location detection starts upon entering shop B

together with their RSSI and estimated distance values.

Analyzing the estimated distances collected from the beacons located within each shop, we observe two situations. The first situation happens when the *region entered* event is triggered when being few meters away from both shops, where signal from both beacons is received. At this time our location mechanism starts the *beacon ranging* and collects the estimated distance values during a period of 10 seconds. Figure 5.9 represents one of this trials and we can see that the estimated distance to both beacons is approximately the same when the *beacon ranging* starts, but as long as we walk towards shop B, its beacon's estimated distance considerably decreases, while the estimated distance to beacon A slightly increases. Thus, the calculated average of the estimated distance from both beacons indicates the shop we enter.

The second situation happens when the *region entered* event is only triggered upon entering the destination shop. We can see from the estimated distance values shown in Figure 5.10, that during the whole period of 10 seconds, *beacon ranging* always reported estimated distances with values less than approximately 8 meters for the beacon of the shop we entered, and values greater than approximately 35 meters for the beacon of the neighboring shop. Logically, the average of the estimated distance values have a considerable difference, which makes the location detection mechanism correctly determine the entered shop.

The first situation happens because the chosen *transmission power* of -12 dBm has a theoretical range of 15 m and since the shops were of small dimensions, the beacon's signal was received even when being outside and few meters away from both shops. It shows the importance of selecting an appropriate *transmission power* value and the beacon's placement within the shop on a way that the shop indoor area is within the beacon's range and the beacon signal coverage outside the shop is minimized. For example, a slightly lower *transmission power* would be a better fit for shops A and B, because the beacon's signal will be mainly detected when entering the destination shop, thus decreasing the first presented situation, were the location detection starts outside of the both shops.

It is also important to note that the signal strength is highly attenuated because of the wall that separates the shops, as we can see from the presented estimated distance variation charts.

Except the cases where the previously described bug manifested itself, our mechanism always correctly determined the entered shop, in all 10 trials. Thus, we consider that the *beacon ranging* period of 10 seconds is a good candidate for our solution.

5.5 Cycling Detection

When *BikeApp* is in the foreground, the sensor is always instantly connected as soon as it becomes available, i.e., with a crank or wheel revolution, and starts sending its data.

We decided to test also the connection time in an alternative scenario, in which the application is scanning in the background and the sensor becomes activated. Thus, to evaluate the cycling detection mechanism we take into consideration the time it takes to connect to a deactivated sensor when starting cycling, the time the mechanism takes to detect the cycling activity and the accuracy of detection.

5.5.1 Methodology

We have done several separate bike rides to collect the results. Each ride was started with the sensor deactivated (i.e. not advertising its packets), in order to register the time it becomes activated by starting cycling. To register the ride start time we created a *Start Ride* button in our test/debug page, which registers a ride start time after a 1 minute countdown. After pressing the button we immediately shut

down the smartphone screen to ensure the application is in the background, and waited the remaining time before starting cycling. The ride start, connection and cycling detection times were registered into a logfile.

5.5.2 Results

During the first 10 trips we observed an average and maximum connection times of 9 and 17 seconds, respectively. The connection times showed to be very similar to the beacon discovery times, where the majority of discoveries are within the first 10 seconds, but sometimes this time may be greater (see Section 5.2). But this only applies to the connection when the application is in the background, the connection in the foreground is instant.

This cycling detection time represents the elapsed time until the instantaneous cadence and speed values exceed determined thresholds during a period of 5 seconds. After the established connection with the sensor, our mechanism took on average 8 seconds to detect the cycling activity over the 10 trips.

Regarding the accuracy, our mechanism proved to be 100% accurate on detecting the cycling activity. The sensor was always connected and the instantaneous cadence and speed values reached the required thresholds, resulting in cycling detections.

5.6 Smartphone Battery Consumption

One of our system requirements is the low battery consumption. Because our mobile application uses the Bluetooth adapter to detect the location beacons and interact with the bicycle sensor, it is important to analyze the power consumption overhead introduced with the required continuous scanning when the application is in the background, and the communication with the sensor when the user is cycling.

First, we determine the power consumption of the smartphone without running any application (standby), to have a reference point. Then, we determine the power consumption of *BikeApp* when the smartphone is idle and when cycling. Similarly, we determine the power consumption of *Biklio* (which uses GPS instead of BLE), when the smartphone is idle and when cycling.

5.6.1 Methodology

To measure the power consumption we used the iOS *UIDevice* class, from the *UIKit* [65], to obtain the information and notifications about changes to the battery's charge level. After requesting the battery

monitoring, a subscribed event notifies everytime the battery level is updated (i.e., drops by 1%). We just register a *timestamp* and the battery percentage into a logfile on each battery level update.

We left the smartphone overnight registering the battery level changes into the logfile for the following scenarios: not running any application, running *BikeApp* when smartphone is idle, and running *Biklio* when the smartphone is idle.

When cycling, we determined the amount of time required to drop the battery level by 2% on both applications, and estimated a consumption per hour.

In all scenarios the smartphone had cellular communication enabled, and WiFi together with the mobile data disabled. Additionally, when running *BikeApp*, the smartphone also used Bluetooth connectivity.

5.6.2 Results

In all three scenarios, we consider the battery consumption over a period of approximately 12-13 hours, and obtain the following consumptions represented in Figure 5.1. The results show that when the smartphone is idle, *BikeApp* has considerably less power consumption than *Biklio*, with a difference of ≈ 2% consumption per hour. Additionally, the continuous background BLE scanning does not introduce a big power consumption overhead, the consumption just increases by 0.34%.

Table 5.1: Battery consumption when smartphone is idle

Running	Battery consumption per hour
No application	0.40%
BikeApp	0.74%
Biklio	2.75%

Regarding the battery consumption when cycling, as we can see in Table 5.2, our solutions has a slightly lower battery consumption than *Biklio*'s. The communication with the sensor introduces an overhead of ≈2.2% per hour. However, since the majority of cycling trips in urban environments do not last hours, a power consumption from an occasional trip of 10-15 minutes is unnoticeable from the user perspective.

Table 5.2: Battery consumption when cycling

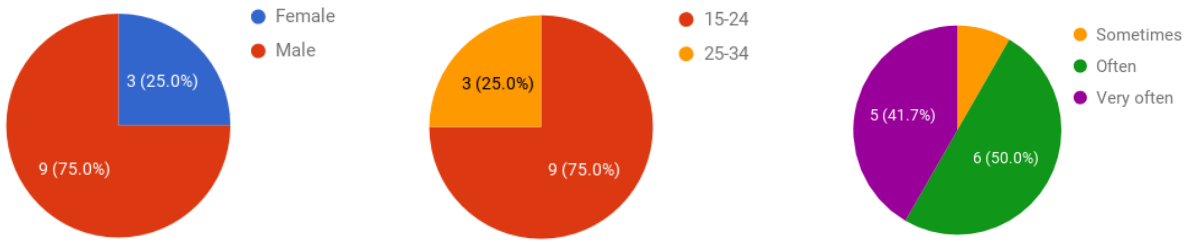
Running	Time to drop 2% battery level	Battery consumption per hour
BikeApp	≈45 minutes	≈2.6%
Biklio	≈37 minutes	≈3.2%

5.7 Usability Evaluation

To finish the *BikeApp* evaluation, we evaluate the usability of our solution with end-users, by asking them to interact with the application and answer a survey. In this section we describe the usability evaluation process. During the application usage session the users used the application by themselves and we just guided them through the session by asking them to do a specific action and answer the required questions.

5.7.1 Results

We performed *BikeApp* application usage sessions with 12 users in total. Figure 5.11 shows the characterization of the users in terms of their gender, age and mobile applications usage frequency. The majority of the users were relatively young and with high frequency of using mobile applications on a daily basis.



(a) Gender **(b)** Age **(c)** Mobile apps usage frequency

Figure 5.11: Characterization of the users

The first two questions evaluate the presentation of nearby shops within the application, which are presented in form of a list or on a map, as already shown in Section 3.2. We asked the users to explore the application for a while, before answering the questions. The first question asks the user how easy nearby shops can be found within the application, and the second asks how easy benefits offered by a given shop can be consulted. From the answers shown in Figure 5.12 and Figure 5.13, we can see that the UI was intuitive and the users experienced no difficulties.

The next set of two questions evaluate the easiness of using the bicycle mounted sensor within our solution. Because the sensor was already installed on the bicycle used during the evaluation, we asked the users to take a careful look at the sensor and estimate the easiness of installing it by following the installation instructions given by the sensor manufacturer, which are shown in Appendix A. Then, we asked the users to pair the sensor in *BikeApp* settings. Answers from Figure 5.14 and Figure 5.15 show that the users did not consider difficult the sensor installation, and the Bluetooth section within the application settings allow easy sensor pairing.

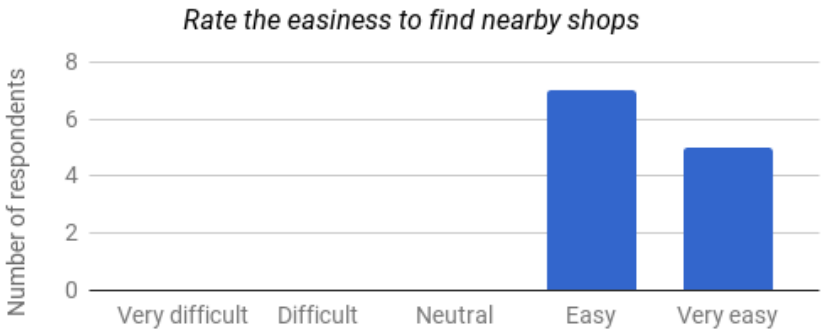


Figure 5.12: Question 1: Spot presentation rating

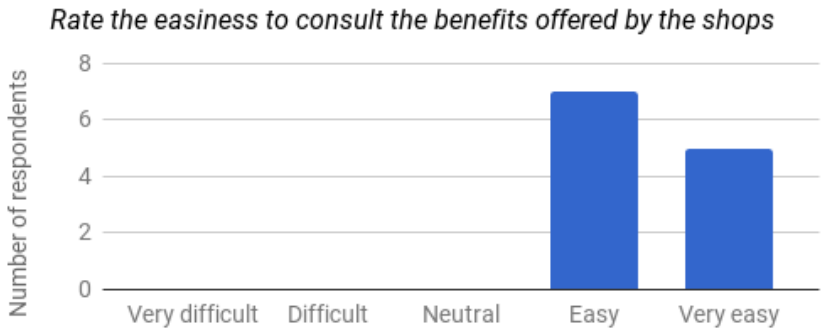


Figure 5.13: Question 2: Spot presentation rating

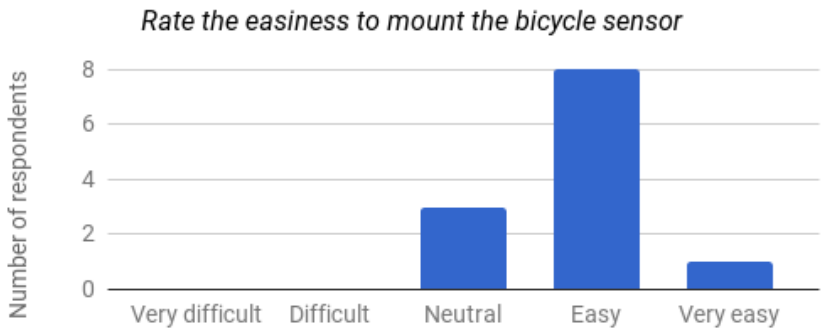


Figure 5.14: Question 3: Bicycle sensor installation

Next, we evaluated the benefit claiming process. To facilitate the testing we decided to manually enabled the location beacon when required, instead of making the users to enter a shop. We explained users that the beacons are placed within the shops and manually enabling it has the same effect as entering the shop. To make the users experience the whole process, we asked them to cycle for a while around the neighborhood first, and when they arrived, we enabled the location beacon so that they can claim their benefit. Then, the users answered a question to rate the easiness to claim the benefit, and to rate the usefulness of automatic shop detection and benefit claim page opening. Figure 5.16 shows that all the users rated the claiming process as easy. This confirms that the automatic shop detection

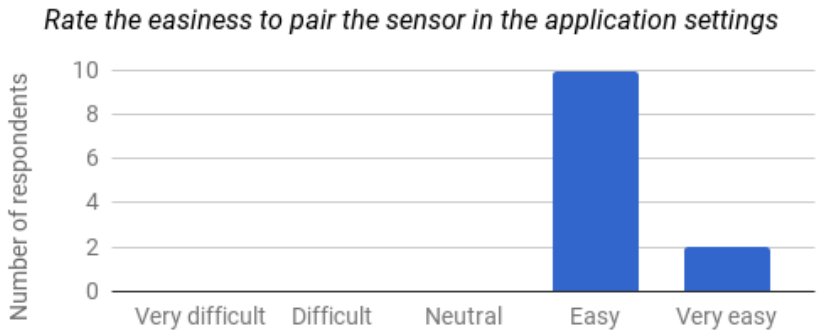


Figure 5.15: Question 4: Bicycle sensor pairing

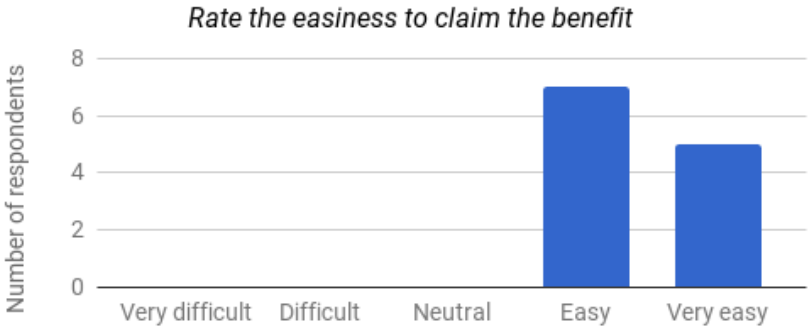


Figure 5.16: Question 5: Benefit claim easiness

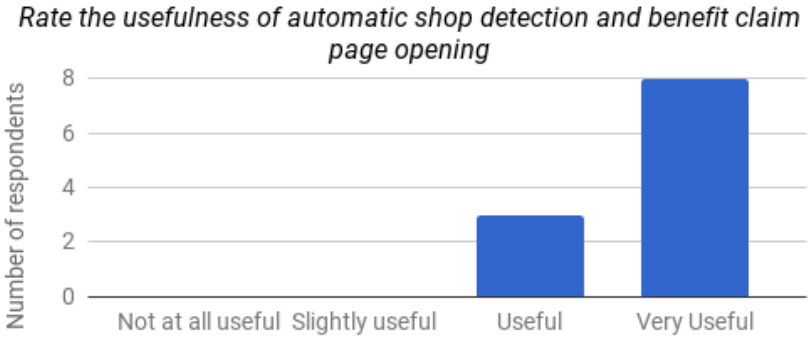


Figure 5.17: Question 6: Shop detection usefulness

and the appearing of the claim page upon entering a shop simplifies a lot the claiming process, because all that is required from the users is to press the claiming button and show the screen to the cashier. Regarding the usefulness of the shop detection mechanism, a very positive feedback was received as we can see from the answers presented in Figure 5.17.

The following questionnaire question asked users to rate the overall application usability, and as we can see from the responses represented in Figure 5.18, the feedback was also positive.

Since our solution requires Bluetooth connectivity for both cycling and location detection mecha-

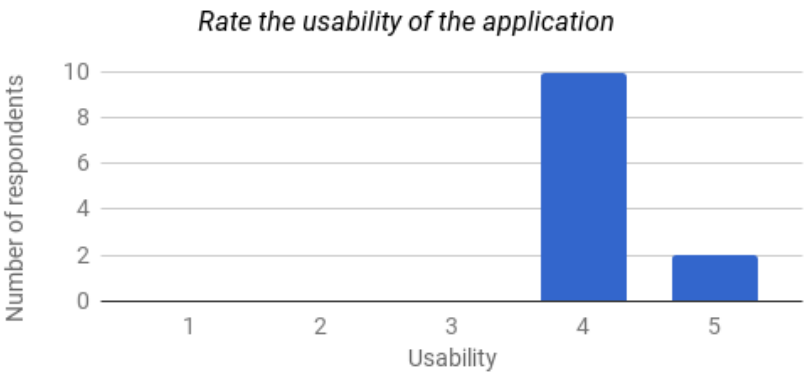


Figure 5.18: Question 7: Overall application usability

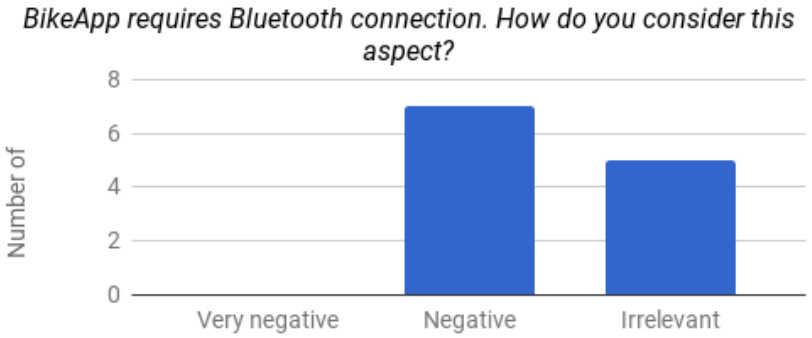


Figure 5.19: Question 8: Required Bluetooth connectivity opinion

nisms, we decided to ask users if they consider this aspect as *very negative*, *negative* or *irrelevant*. Unfortunately, the majority of the users answered *negative*, as we can see in Figure 5.19. Some users mentioned that they think that having a connectivity with the sensor would quickly drain the smartphone's battery.

5.8 Summary

In this chapter we present the evaluation process of *BikeApp*, and the following aspects can be summarized.

Although Apple recommends an *advertising interval* value of 100 ms for location beacons, in practice there is no considerable difference in discovery time when comparing to a 300 ms value. The majority of discoveries are still within the 0-10 seconds range when using the greater *advertising interval* value, and the beacons can benefit from a greater battery life, by advertising three times less often.

Regarding the estimated distance, we experimentally confirm that the estimated distance value obtained on every *beacon ranging* result has reasonably good correlation with the real distance when being close to the beacon, and only starts to fluctuate if the beacon becomes further than $\approx$10 meters, due

to the RSSI fluctuations. Thus, is important to calculate an average estimated distance over a period of few seconds. After testing the location detection mechanism in a worst-case scenario, with beacons deployed in side to side shops, we confirm that the wall which separates the shops causes a strong signal attenuation, which yields to high estimated distance values. When being inside one of the shops, the given estimated distance to the neighboring shop beacon is greater by 25-30 meters, which results in correct location detection if determining an average value of the estimated distance over a period of 10 seconds.

When testing the cycling detection mechanism we observed that the sensor connection time depends basically on the sensor discovery time, because the connection to a discovered peripheral is almost instantaneous. When scanning in the background, the sensor discovery time varies, but is usually within the first 10 seconds after the sensor activation. Regarding the communication with the sensor, no issues were experienced and the cycling detection mechanism worked always deterministically, as expected.

Regarding the power consumption, *BikeApp* introduces an overhead of 0.34% battery consumption per hour, when the smartphone is idle, and an overhead of 2.2% battery consumption per hour, when the smartphone keeps the communication with the sensor while cycling.

The questionnaire answers from the user testing sessions show that the users did not experience any problem in finding relevant information or pair a sensor within the application settings. By following the bicycle sensor's manufacturer installation instructions, the majority of users rate the installation as easy. The reward claims are also considered easy, and the majority of users find the automatic shop detection as very useful. However, without knowing the particularities of BLE in terms of power consumption, many users consider the fact that the application requires Bluetooth connection as negative, thinking that it drains the smartphone's battery.

6

Conclusion

Contents

6.1 Future Work 86

This project introduces new location and cycling detection mechanisms. The cycling detection mechanism is able to accurately detect the cycling activity by using a bicycle-mounted BLE sensor, and the location detection mechanism is able to accurately determine the location by using BLE location beacons. We integrated those mechanisms in *Biklio*, replacing its less accurate cycling and location detection mechanisms. The new version of the application (*BikeApp*) requires users to pair their bicycle's sensor in the application Settings, and the shop owners to install a location beacon within their shops. *BikeApp* detects when the user starts cycling and makes him eligible for rewards that can be claimed at the shops. When the user gets close to or enters a shop, *BikeApp* detects the shop and shows a screen to allow the user claim the benefit.

BikeApp implementation was done in a cross-platform fashion using Xamarin, but only targeting the iOS platform. The implementation of the cycling detection mechanism is composed by two modules: *BLE Manager* and *Cycling Detector*. The *BLE Manager* module is responsible for the communication with the bicycle sensor, while the *Cycling Detector* module uses the sensor data provided by the *BLE Manager* and infers the cycling activity. Similarly, the location detection mechanism is also composed by two modules: *Beacon Monitor* and *Location Detector*. The *Beacon Monitor* module is responsible for detecting the presence of BLE beacons, and the *Location Detector* module determines the location using the information provided by the *Beacon Monitor*. The modules are implemented in a way that its integration is effortless, requiring few event subscriptions to provide cycling and location detection mechanisms to the application.

After the implementation, we evaluate the system. First, we conduct an experiment to determine optimal *advertising interval* value to be configured in location beacons, and how this parameter affects the beacon discovery time. Then, we analyze the correlation that exists between the estimated (based on RSSI) and the real distances at which the beacon is located. The results show that the correlation is enough to determine if the beacon is located closer (0-5 meters) or further (10-15 meters) from the smartphone. The next experiment evaluates the location detection behavior in a worst-case scenario: with shops side to side. This experiment allows to understand that the wall that separates both shops has a strong attenuation effect on the beacon's signal, and even in this situations our mechanism always detected the correct entered shop. Regarding the cycling detection, we present an experiment in which we do several cycling trips and measure the sensor connection and cycling detection times. The results show that when the application is in the background, the sensor connection varies, because it depends on the discovery of the peripheral. The cycling detection always deterministically detects the cycling activity. Next, through power consumption measurements, we conclude that *BikeApp* introduces a small overhead of 0.34% when the smartphone is idle, and an acceptable overhead of 2.2% when the smartphone keeps the communication with the sensor while cycling. In both scenarios *BikeApp* has less power consumption than *Biklio*, especially whe the smartphone is idle, with a difference of $\approx 2\%$

consumption per hour. Finally, the usability of *BikeApp* was evaluated with end-users and a questionnaire. The results allow us to conclude that users do not consider difficult the bicycle sensor installation and pairing within the application. Users consider useful the location detection mechanism and consider easy the process of claiming the benefit.

6.1 Future Work

In order to achieve a public release, there are some aspects that need to be checked and implemented. From the mobile application side, we suggest to add an introductory tutorial to appear upon opening the application for the first time. In this tutorial it would clarify the use of bicycle-sensor and briefly explain that the BLE communication is effective in terms of power consumption. Since the application was implemented in cross-platform, we also suggest as future work to expand its implementation to the Android platform.

Because the tested version of iOS (10.3.2) has a bug in its *Core Location* framework [64]. Before releasing *BikeApp* it is important to check that in the newer versions of iOS, the bug is fixed, to guarantee correct application functioning.

The *web application* is already implemented for the *Biklio* project. However, it is required to update its shop registration page, to include a section about location beacons. This section should explain how the location beacons work and contain instructions to configure them.

Bibliography

[1] B. Li, J. Salter, A. Dempster, and C. Rizos, “Indoor positioning techniques based on wireless LAN,” *Conference on Wireless*, pp. 13–16, 2006.

[2] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, “A survey of mobile phone sensing,” *IEEE Communications Magazine*, vol. 48, no. 9, pp. 140–150, Sept 2010.

[3] Endomondo. Accessed: 13-12-2016. [Online]. Available: https://www.endomondo.com

[4] KAPPO Bike. Accessed: 5-1-2017. [Online]. Available: http://www.kappo.bike/

[5] European Comission - TRACE. Accessed: 14-10-2017. [Online]. Available: https://ec.europa.eu/inea/en/horizon-2020/projects/H2020-Transport/Urban-Mobility/TRACE

[6] Trace - Walking and Cycling Tracking Services. Accessed: 28-12-2016. [Online]. Available: http://h2020-trace.eu

[7] Biklio. Accessed: 28-12-2016. [Online]. Available: https://www.biklio.com

[8] P. Roshan and J. Leary, *802.11 Wireless LAN Fundamentals*, 2010.

[9] Cisco - 802.11 Fundamentals. Accessed: 26-11-2016. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Borderless_Networks/Unified_Access/CMX/CMX_802Fund.pdf

[10] A. S. Huang and L. Rudolph, *Bluetooth Essentials for Programmers*, 1st ed. New York, NY, USA: Cambridge University Press, 2007.

[11] F. Naya, H. Noma, R. Ohmura, and K. Kogure, “Bluetooth-based indoor proximity sensing for nursing context awareness,” *Proceedings - International Symposium on Wearable Computers, ISWC*, vol. 2005, no. November, pp. 212–213, 2005.

[12] F. Subhan, H. Hasbullah, A. Rozyyev, and S. T. Bakhsh, “Analysis of Bluetooth signal parameters for indoor positioning systems,” *2012 International Conference on Computer and Information Science, ICCIS 2012 - A Conference of World Engineering, Science and Technology Congress, ESTCON 2012 - Conference Proceedings*, vol. 2, pp. 784–789, 2012.

[13] Bluetooth Low Energy — Android Developers. Accessed: 27-11-2016. [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth-le.html

[14] iBeacon for Developers - Apple Developer. Accessed: 27-11-2016. [Online]. Available: https://developer.apple.com/ibeacon

[15] Beacon Tech Overview - Estimote Developer. Accessed: 27-11-2016. [Online]. Available: http://developer.estimote.com

[16] Kontakt.io Dev Center. Accessed: 27-11-2016. [Online]. Available: https://developer.kontakt.io

[17] S. Silva, S. Soares, T. Fernandes, A. Valente, and A. Moreira, "Coexistence and interference tests on a bluetooth low energy front-end," in *2014 Science and Information Conference*, Aug 2014, pp. 1014–1018.

[18] Core Specification - Bluetooth v4.0. Accessed: 28-11-2016. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[19] J. Liu, C. Chen, Y. Ma, and Y. Xu, "Energy Analysis of Device Discovery for Bluetooth Low Energy," no. August 2015, 2013.

[20] Generic Attribute Profile (GATT) Specification. Accessed: 5-1-2017. [Online]. Available: https://www.bluetooth.com/specifications/generic-attributes-overview

[21] GATT — Introduction to Bluetooth Low Energy. Accessed: 5-1-2017. [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt

[22] Cypress. Bluetooth Low Energy (BLE). Accessed: 5-1-2017. [Online]. Available: http://www.cypress.com/file/139326/download

[23] GATT Services. Accessed: 5-1-2017. [Online]. Available: https://www.bluetooth.com/specifications/gatt/services

[24] P. Bellavista, A. Küpper, and S. Helal, "Location-based services: Back to the future," *IEEE Pervasive Computing*, vol. 7, no. 2, pp. 85–89, 2008.

[25] H. Koyuncu and S. H. Yang, "A Survey of Indoor Positioning and Object Locating Systems," *International Journal of Computer Science and Network Security (IJCSNS '10)*, vol. 10, no. 5, pp. 121–128, 2010.

[26] H. Liu, H. Darabi, P. Banerjee, and J. Liu, "Survey of wireless indoor positioning techniques and systems," pp. 1067–1080, 2007.

[27] Z. Farid, R. Nordin, and M. Ismail, “Recent advances in wireless indoor localization techniques and system,” *Journal of Computer Networks and Communications*, vol. 2013, 2013.

[28] K. Al Nuaimi and H. Kamel, “A survey of indoor positioning systems and algorithms,” *2011 International Conference on Innovations in Information Technology, IIT 2011*, no. May 2011, pp. 185–190, 2011.

[29] A. H. Sayed, A. Tarighat, and N. Khajehnouri, “Network-based wireless location: Challenges faced in developing techniques for accurate wireless location information,” *IEEE Signal Processing Magazine*, vol. 22, no. 4, pp. 24–40, 2005.

[30] L. Mainetti, L. Patrono, and I. Sergi, “A survey on indoor positioning systems,” *2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 111–120, 2014.

[31] M. Schüssel, “Angle of Arrival Estimation using WiFi and Smartphones,” no. October, pp. 4–7, 2016, accessed: 26-11-2016. [Online]. Available: http://www3.uah.es/ipin2016/usb/app/descargas/223_WIP.pdf

[32] Quuppa Intelligent Locating System. Accessed: 26-11-2016. [Online]. Available: http://quuppa.com/

[33] A. Kotanen, M. Hannikainen, H. Leppakoski, and T. D. Hamalainen, “Experiments on local positioning with bluetooth,” in *Proceedings ITCC 2003. International Conference on Information Technology: Coding and Computing*, April 2003, pp. 297–303.

[34] A. N. Venetsanopoulos, A. Kushki, and K. N. Plataniotis, *WLAN Positioning Systems*, 2012, no. 2000.

[35] A. Thaljaoui, T. Val, N. Nasri, and D. Brulin, “BLE localization using RSSI measurements and iRingLA,” *2015 IEEE International Conference on Industrial Technology (ICIT)*, no. June, pp. 2178–2183, 2015.

[36] T. Rappaport, *Wireless Communications: Principles and Practice*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[37] A. H. Lashkari, B. Parhizkar, and M. N. A. Ngan, “WIFI-based indoor positioning system,” *2nd International Conference on Computer and Network Technology, ICCNT 2010*, no. November 2016, pp. 76–78, 2010.

[38] B. G. Mathie, M. J., Coster, A. C., Lovell, N. H., & Celler, “Accelerometry : providing an integrated , practical method for long-term , ambulatory monitoring of human movement,” *Physiological measurement*, vol. 25, no. 2, p. R1, 2004.

[39] O. D. Incel, M. Kose, and C. Ersoy, “A Review and Taxonomy of Activity Recognition on Mobile Phones,” *BioNanoScience*, vol. 3, no. 2, pp. 145–171, 2013.

[40] J. Pärkkä, L. Cluitmans, and M. Ermes, “Personalization algorithm for real-time activity recognition using PDA, wireless motion bands, and binary decision tree,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 5, pp. 1211–1215, 2010.

[41] J. J. Guiry, P. van de Ven, J. Nelson, L. Warmerdam, and H. Riper, “Activity recognition with smartphone support,” *Medical Engineering and Physics*, vol. 36, no. 6, pp. 670–675, 2014.

[42] J. R. Kwapisz, G. M. Weiss, and S. a. Moore, “Activity recognition using cell phone accelerometers,” *ACM SIGKDD Explorations Newsletter*, vol. 12, no. 2, p. 74, 2011.

[43] iOS Programming Guide - Background Execution. Accessed: 31-7-2017. [Online]. Available: https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html

[44] R. Okugawa, K. Murao, T. Terada, and M. Tsukamoto, “Training system of bicycle pedaling using auditory feedback,” *Proceedings of the 12th International Conference on Advances in Computer Entertainment Technology - ACE ’15*, pp. 1–4, 2015.

[45] THIS IS ANT: The Wireless Sensor Network Solution. Accessed: 30-11-2016. [Online]. Available: https://www.thisisant.com/

[46] GATT Service: Cycling Speed and Cadence. Accessed: 5-1-2017. [Online]. Available: https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.service.cycling_speed_and_cadence.xml

[47] Estimote. Accessed: 26-7-2017. [Online]. Available: https://www.estimote.com

[48] SQLite. Accessed: 26-7-2017. [Online]. Available: https://www.sqlite.org

[49] Xamarin. Accessed: 26-7-2017. [Online]. Available: https://www.xamarin.com

[50] B’TWIN. Accessed: 26-7-2017. [Online]. Available: https://www.btwin.com/en/home

[51] IBM Knowledge Center - Mappings between C and UML elements. Accessed: 28-7-2017. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSCLKU_7.5.5/com.ibm.xtools.viz.dotnet.doc/topics/r_cs_mapping.html

[52] Apple Developer Documentation - Core Bluetooth. Accessed: 28-7-2017. [Online]. Available: https://developer.apple.com/documentation/corebluetooth

[53] Core Bluetooth - CBCentralManager Class. Accessed: 28-7-2017. [Online]. Available: https://developer.xamarin.com/api/type/CoreBluetooth.CBCentralManager

[54] GATT - Cycling Speed and Cadence Service. Accessed: 28-7-2017. [Online]. Available: https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.service.cycling_speed_and_cadence.xml

[55] Core Bluetooth Programming Guide. Accessed: 28-7-2017. [Online]. Available: https://developer.apple.com/library/content/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts

[56] GATT - CSC Measurement Characteristic. Accessed: 28-7-2017. [Online]. Available: https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.csc_measurement.xml

[57] Core Bluetooth - CBPeripheralDelegate Class. Accessed: 28-7-2017. [Online]. Available: https://developer.xamarin.com/api/type/CoreBluetooth.CBPeripheralDelegate

[58] Estimote Developer - Estimote Telemetry Packet. Accessed: 2-8-2017. [Online]. Available: http://developer.estimote.com/sensors/estimote-telemetry

[59] Apple Developer - iBeacon. Accessed: 2-8-2017. [Online]. Available: https://developer.apple.com/ibeacon

[60] Google Developers - Eddystone. Accessed: 28-7-2017. [Online]. Available: https://developers.google.com/beacons/eddystone

[61] Radius Networks Developer Blog - CoreBluetooth Doesn't Let You See Beacons. Accessed: 29-7-2017. [Online]. Available: http://developer.radiusnetworks.com/2013/10/21/corebluetooth-doesnt-let-you-see-ibeacons.html

[62] Apple Developer Documentation - Core Location. Accessed: 28-7-2017. [Online]. Available: https://developer.apple.com/documentation/corelocation

[63] Core Location - CLLocationManager Class. Accessed: 28-7-2017. [Online]. Available: https://developer.xamarin.com/api/type/CoreLocation.CLLocationManager/

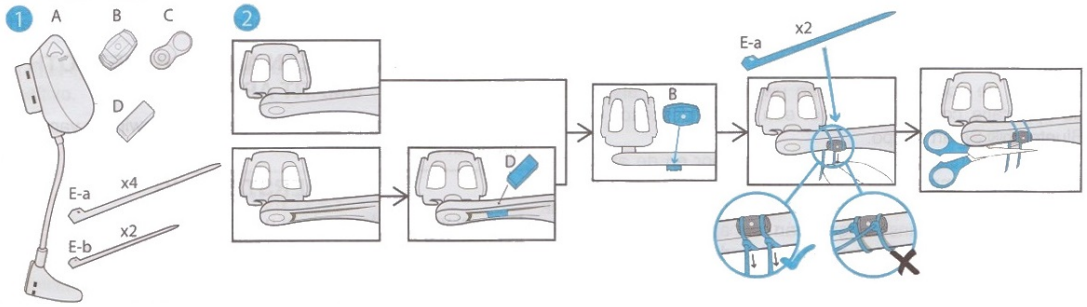
[64] iOS Core Location Bug Report. Accessed: 16-9-2017. [Online]. Available: http://www.openradar.me/31886860

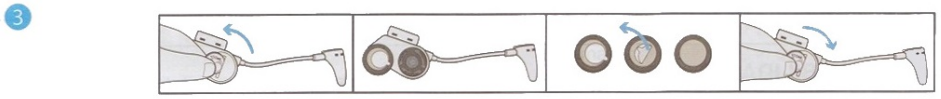
[65] Apple Developer Documentation - UIKit. Accessed: 21-9-2017. [Online]. Available: https://developer.apple.com/documentation/uikit

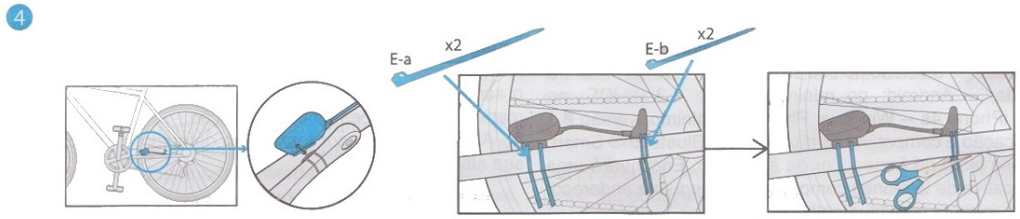
A

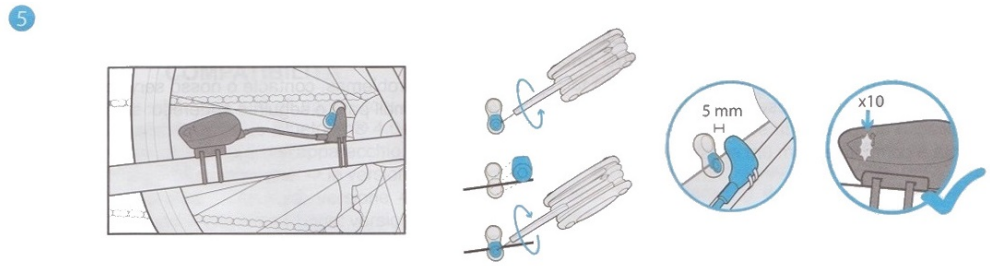
B'TWIN Sensor Instructions

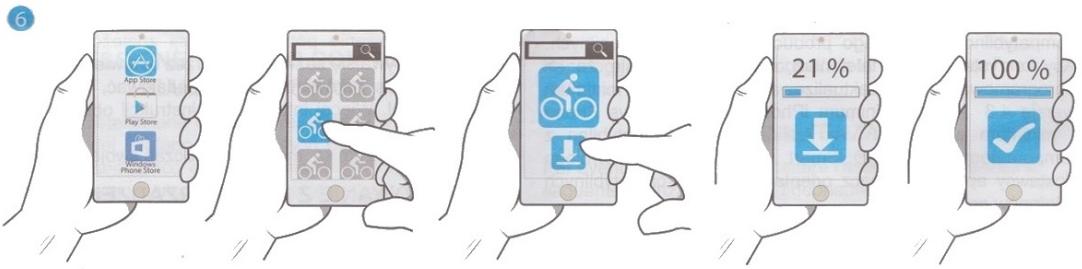
The following figures contain the bicycle sensor installation instructions, provided by the sensor manufacturer. The users rated the sensor installation easiness by following these instructions.

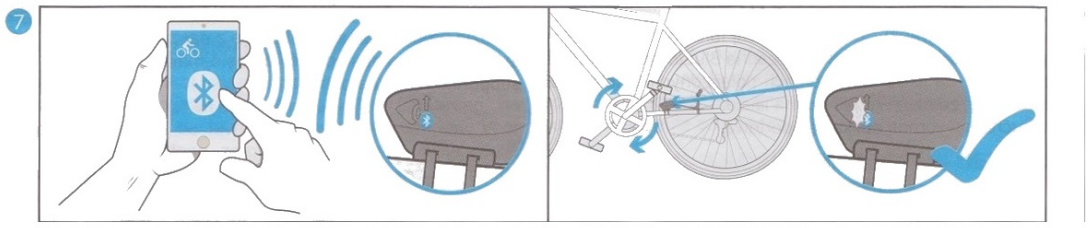

1
A
B
C
D
E-a
x4
E-b
x2
2
E-a
x2
B
D


3


4
E-a
x2
E-b
x2


5
5 mm
x10


6
App Store
Play Store
Windows Phone Store
21 %
100 %


7