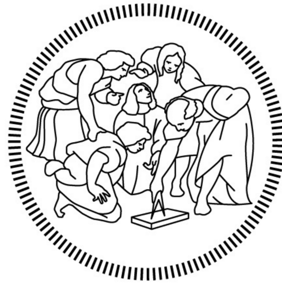

POLITECNICO DI MILANO

DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

A distributed framework for proximity-aware Android applications

TESI DI LAUREA DI

Luca BAGGI

Matr. 814638

Marco MEZZANOTTE

Matr. 816479

RELATORE

Prof. Sam Jesus Alejandro GUINEA MONTALVO

Anno Accademico 2014/2015

Ai nostri genitori.

Abstract

During last few years smartphones evolution affected our lives, changing real world interaction such as innovative ways to get in touch with other people using social networks, and providing internet network connectivity almost all the time. This evolution led smartphones to embed more and more sensors (e.g. motion, location sensors, etc.) useful to infer - previously unavailable - user context information: the latters were exploited by smart applications to properly change their behaviour; these applications are called context-aware apps.

In this thesis we focused on a particular category of context-aware apps, the location-aware ones, taking into account indoor proximity data: this information is currently used to customize apps user experience on the basis of user surroundings, as in targeted advertising tasks.

What we propose is a framework, called Android Distributed Proximity Framework (ADPF), supporting Android developers to implement this kind of applications. ADPF provides a set of API to subscribe to proximity events about application actors (e.g. the relative distance between them) and additional functionalities, such as actors grouping, to enrich information on these events. Using ADPF developers can easily build proximity-aware apps transparently with respect to technologies producing proximity information, through a common high-level interface.

Opposite to existing solutions, the framework doesn't rely on a central computational node and only needs a standard WiFi network to work.

Sommario

Nel corso degli ultimi anni l'evoluzione degli smartphone ha influenzato fortemente le nostre vite, cambiando il modo di interagire con il mondo reale, in particolare a livello sociale con l'avvento dei social networks, e permettendo di essere sempre (o quasi) connessi a Internet. Durante questa evoluzione si è verificata una continua integrazione di sensori (di movimento, di posizione ecc.) utili a ottenere sempre più informazioni sul contesto nel quale agisce l'utente: ciò ha permesso lo sviluppo di nuove applicazioni, in grado di modificare il proprio comportamento sulla base del contesto stesso; tali applicazioni sono dette context-aware apps.

La nostra tesi si concentra su una particolare categoria di context-aware apps, le cosiddette location-aware apps, prendendo in considerazione informazioni relative alla proximity in ambienti indoor: esse sono attualmente utilizzate per personalizzare l'esperienza dell'utente sulla base di ciò che si trova in prossimità rispetto ad esso, come nel caso della pubblicità mirata.

La nostra proposta è un framework, chiamato Android Distributed Proximity Framework (ADPF), per supportare gli sviluppatori Android nell'implementazione di tali applicazioni. ADPF offre un insieme di API che permette di sottoscriversi ad eventi di proximity riguardanti gli "attori" dell'applicazione (es. la distanza relativa fra di loro) e altre funzionalità, come la possibilità di raggruppare questi ultimi, in modo tale da arricchire le informazioni su tali eventi.

Grazie ad ADPF, un programmatore può sviluppare una proximity-aware app in modo totalmente trasparente rispetto alle tecnologie utilizzate per produrre

informazioni di proximity, utilizzando un'interfaccia comune di alto livello.
Al contrario di altre soluzioni simili esistenti, il framework non necessita di un nodo centrale di elaborazione, ma il tutto avviene in un contesto distribuito; pertanto, per funzionare necessita soltanto di una rete WiFi standard.

Ringraziamenti

Ringraziamo Sam Guinea, relatore di questa tesi, per il continuo supporto al nostro lavoro, la grande disponibilità e la professionalità dimostrate. Grazie inoltre agli amici universitari e non per averci sostenuto nei momenti difficili durante il nostro percorso di studi.

Luca e Marco

Ringrazio la mia famiglia, per la continua fiducia riposta in me e per tutti i sacrifici fatti per permettermi di raggiungere questo traguardo. Grazie a Sonia, per essere sempre stata al mio fianco in questo percorso, e per avermi aiutato, sostenuto e motivato in ogni momento di difficoltà. Grazie a tutti i compagni con cui ho affrontato questi anni di università: sarebbero troppi i nomi da citare, ma sono certo che ognuno di loro saprà riconoscersi leggendo queste parole.
Infine grazie a Marco, con cui ho condiviso gioie e dolori di questa tesi: non avrei potuto desiderare compagno migliore.

Luca

Un ringraziamento speciale va alla mia famiglia che da sempre ha creduto in me e mi ha sostenuto in ogni scelta fatta, dandomi fiducia anche quando pochi l'avrebbero fatto. Grazie a Selena per avermi aiutato a superare i momenti di sconforto e le difficoltà di questi ultimi anni. Infine un grazie ai miei compagni Filippo e Tommaso per questi 5 anni di università passati assieme: affrontarli con voi è stata tutta un'altra storia.
Non saprei come ringraziare Luca per il lavoro svolto fianco a fianco, la passione e la determinazione dimostrate.

Marco

Contents

List of Figures **xiii**

List of Tables **xv**

Listings **xvii**

1 Introduction **1**

2 State of the art **5**

2.1 Technological overview 6
2.1.1 WiFi 6
2.1.2 Radio Frequency - Ultrasonic signals 8
2.1.3 Bluetooth 8
2.1.4 Geo-Magnetism sensing 14
2.1.5 LTE Direct 15
2.2 Platform overview 16
2.2.1 Redpin 16
2.2.2 ActiveBat 17
2.2.3 Cricket 18
2.2.4 Navizon 19
2.2.5 Insiteo 19
2.2.6 Allseen - Alljoyn 20
2.2.7 Google Nearby 22
2.3 Conclusions 23

3 Our framework **25**

3.1 Problem analysis 26
3.2 Use cases 27
3.3 Functional requirements 29
3.4 Non-functional requirements 30
3.5 Framework abstractions 30
3.5.1 Entity 30
3.5.2 SelfEntity 31
3.5.3 DistanceRange 31
3.5.4 POI 32
3.5.5 Group 33

3.5.6 Events . . . 33
3.6 Framework APIs . . . 34
3.7 The distributed approach . . . 35
3.8 Communication . . . 35
3.9 Security . . . 36
3.10 Framework working principles . . . 37

4 Library architecture 43
4.1 Architecture overview . . . 44
4.2 Group-Entity Interface . . . 45
4.2.1 Utility classes and interfaces . . . 46
4.2.2 Abstractions objects . . . 49
4.2.3 Subscription events . . . 52
4.2.4 POI Events . . . 54
4.2.5 ClientProximityAPI . . . 55
4.2.6 Framework messages . . . 56
4.2.7 GroupEntityManager . . . 62
4.3 Technology Interface . . . 73
4.3.1 Utility classes and interfaces . . . 74
4.3.2 TechnologyManager . . . 76
4.4 Technology level . . . 78
4.4.1 BLEProvider . . . 80
4.4.2 GeofenceProvider . . . 81
4.5 Security layer . . . 82
4.6 Network layer . . . 84
4.6.1 MQTT . . . 85

5 Results and evaluation 87
5.1 Test and tuning parameters . . . 88
5.1.1 Test parameters . . . 88
5.1.2 Tuned parameters . . . 88
5.2 Experimental setup . . . 89
5.2.1 Log messages . . . 89
5.3 Testing tools . . . 93
5.3.1 LoggerService . . . 93
5.3.2 TcpServer . . . 94
5.3.3 MySQL database . . . 94
5.3.4 Ntp Server . . . 95
5.3.5 TestClient . . . 95
5.4 Tests and results . . . 98
5.5 Conclusions . . . 116

6 Conclusion and future works 119

Bibliography 123

List of Figures

2.1 iBeacon packet schema 10
2.2 Eddystone packet schema 13
2.3 Alljoyn network architecture 21

3.1 Framework network schema 36
3.2 Proximity computation 39

4.1 Architecture overview 44
4.2 Simplified class diagram for Group-Entity Interface layer 46
4.3 `Group.evaluate` method flow chart 51
4.4 Topology example: devices “far” from beacon 70
4.5 Topology example: devices “near” to beacon 70
4.6 Simplified class diagram for Technology Interface layer 74
4.7 Providers passing data through `PassiveTechnologyListener` . 78
4.8 `TechnologyManager` passing requests to providers 79
4.9 Technology level simplified class diagram 80
4.10 Security Layer class diagram 84

5.1 Toy example - Logging 90
5.2 TestClient window 96
5.3 Average message count 113

List of Tables

3.1 DistanceRange table 32

5.1 Dispatching time table 101
5.2 Routing time table 103
5.3 Routing time table 104
5.4 Average dispatching time for geofence events 106
5.5 Entity object memory schema 107
5.6 Time intervals between CHECK_OUT false and PROPERTIES_UPDATE 108
5.7 Time intervals between SYNC_REQ and SYNC_RESP - before fix . . 110
5.8 Time intervals between SYNC_REQ and SYNC_RESP - after fix . . . 110
5.9 Message count table 112
5.10 Packet loss rate table 115
5.11 Routing time comparison table 116

Listings

4.1 CHECK_IN message 57
4.2 CHECK_OUT message 57
4.3 PROPERTIES_UPDATE message 58
4.4 PROX_BEACONS message 59
4.5 PROXIMITY_UPDATE message 60
4.6 SYNC_REQ message 60
4.7 SYNC_RESP message 61
5.1 REC_MSG log 91
5.2 SENT_MSG log 91
5.3 EVENT log 92

Chapter 1

Introduction

During the last few years we lived a revolution due to the advent of smartphones which play an increasingly central role in everyday life. Not only these devices have changed the social interaction rules, providing broad access to social network platforms and introducing new information spreading paradigms, they have enabled always on network connectivity.
Smartphones also gained lots of sensors through which it is possible to infer new information about a user's surroundings: for example, most smartphones now have embedded daylight sensors, motion sensors, location sensors, etc.
These embedded sensors enable new kinds of apps: the so called **context-aware apps**, smart applications which adapt their behaviour on the basis of actual real world context. A particular category of context-aware apps are location-aware apps: they change their behaviour depending on the user's geolocation, as retrieved by GPS sensors or other network services.
Location-aware apps that use outdoor location information have been very effective. As a result, several organizations started showing interest for these kinds of applications, and began trying to reach similar results in indoor environments. The main issues with indoor location-aware apps is the technological limit of the involved sensors: in particular, GPS sensors don't work indoors, and the geolocation information that can be inferred by cell towers needs to be provided by Internet deployed services which are not as accurate as some

applications might.

In most applications location context data are used to get the relative distance between two application actors: for example, in the case of proximity marketing, couponing and similar applications, the need of precise absolute location is not imperative and the cost needed to get it wouldn't be motivated. A direct outcome is that leading companies are developing new technologies and standards in this field: Bluetooth Low Energy technology (BLE) and LTE-Direct are only a couple of them. A key aspect is that there is also a strong need to maintain good power efficiency.

Current solutions mainly provide absolute location data using centralized architectures: this usually leads to high deployment costs and further constraints, such as the need for internet access in the case of cloud-based solutions. Furthermore, almost every solution we analyzed relies on a single proximity technology, limiting possible scenarios[1, 2, 3, 4].

With our work we aim to provide a technology-independent interface through which applications that need to operate indoor can retrieve proximity information. In addition, we want to keep our solution as simple as possible to deploy and maintain.

We provide a framework for **proximity-aware apps** that mainly focuses on indoor environments called **Android Distributed Proximity Framework** (ADPF). It uses a distributed approach and a simple WiFi network infrastructure to keep deployment costs as low as possible. Our target was to make developers' work easier through simple abstractions, hiding low level technologies and the complexities of data integration. In particular, we aim to provide periodic proximity and "raw presence" updates, namely entry and exit events from considered indoor environment.

The approach follows a publish-subscribe pattern, suitable for event based scenarios like the ones ADPF has to deal with.

In order to also provide targeted updates on group of entities we offer grouping functionalities. To do this we abstract application actors as virtual entities

with their application-specific properties.
ADPF provides three different types of subscriptions based on events the app needs: *group subscriptions* signal when someone leaves or joins a group, together with network presence data; *proximity subscriptions* are used to receive relative distance updates; finally *geofence subscriptions* allow the programmer to be notified of entry or exit events related to geofences that are built around entities.
In order to automatically manage the available indoor proximity technologies, the framework introduces ad-hoc objects called Points Of Interest (POI): they are entity containers, and are used as triggers to enable indoor ADPF notifications, meaning that upon a user enters in a POI the framework makes the client app aware of this entry event and starts to provide proximity information.
In our first release, the framework supports Bluetooth Low Energy for indoor proximity data, while GPS is used to get geolocation data (useful for POI related events). The rapid evolution of technologies and standards, especially for indoor environments, motivated us to make ADPF easily extendable, be able to exploit new technologies as quickly as possible and integrate their proximity data capabilities.

ADPF is suitable for several indoor proximity contexts: from classic retail store tasks, such as couponing and custom offers, to emergency support activities. ADPF however also tackels device-to-device proximity scenarios, enabling smart interactions between app users. This aspect can be really useful in social-oriented contexts, such as exhibitions or conventions, especially if combined with user profiling capabilities.

ADPF is a framework built for the Android platform, and is implemented as a standard Android Service. It is kept always running in the background. This means that it reacts to POI events and fires application events, even if the application itself is stopped or paused. As previously stated a standard WiFi network is required in indoor environments, and a publish-subscribe paradigm is used to maintain good scalability performances. We adopted MQTT for this

purpose, a TCP based protocol with various free message brokers implementations.
We also took into account security and privacy aspects; they are explained in detail in chapter 4.

The last part of our work was focused on evaluating the framework's performance and scalability. To achieve this goal we implemented a Java application that simulates an environment with a high number of devices and beacons. Using it we were able to get performance metrics, such as average event "firing time" or messages routing time, and inspect the framework's behaviour with a varying number of devices.
Our evaluation show a good overall performance of the framework in several scenarios that are comparable to real world situations.

As possible future works, we pointed out ADPF implementations for other mobile platforms such as iOS or Windows Mobile, together with the integration of new proximity technologies. Furthermore, in lots of scenarios, could be useful to automatically manage also "active technologies" such as NFC and QRCode, involving direct user interaction.

Chapter 2

State of the art

Location-aware mobile apps in the past years were mainly focused on outdoor environments due to the built-in technologies smartphones had in that period. In particular the GPS sensor was the most used in these apps for this purpose: from road navigation systems to activity tracking applications it was able to fulfill all their needs.
Now we are living an evolution of these smart applications bringing their capabilities indoor: proximity marketing, home automation and indoor navigation are only few of the fields in which this capabilities are required.
This chapter describes the evolution in indoor-location, the currently adopted solutions and the ones available in the near future.
The first section describes several approaches used in last years from the technological point of view, with a brief evolution of its usage over time.
The second section examines solutions currently available in commercial products and the platform supporting them, with a brief description of the additional features offered.

2.1 Technological overview

In this section we'll see an overview of the currently available technologies for indoor location purposes and some projects where these technologies were used.

2.1.1 WiFi

The first experiments in indoor location field were based on wifi networks. The main advantage using wifi is the broad diffusion of this technology in most of interesting environments like offices, homes and several city areas.
In wifi-based location projects, three main approaches have been taken into account and we want to describe them below.

Fingerprinting

The idea behind fingerprint is mainly based on having a "discrete environment model" from the location point of view: this means that the whole environment considered for indoor-location activity is divided into predefined areas, for example a $3 * 3$ meters square: the systems using this technique most of the times return location results based on the aforementioned discrete model instead of a continuous one. For example in Redpin project's tests[5] when the user queries the system for his own location the system returns the room number having most similar fingerprints.
A fingerprint is usually composed by a tuple of signals parameters (such as network identifiers, signal strength, and access point information) from different access points in order to map this fingerprint with the real location where data were measured.
Most of the systems maintain a database with the set of all fingerprints measured for each "discrete area" in order to compute the similarity between user measures and stored ones and return a location best estimate.
The deployment of the application consists of two phases: a measurement

phase where several measurements are performed, for each area in the environment, in order to detect and store these fingerprints and their maps with areas, and a second phase where the running applications periodically computes the fingerprint resulting by consequent scans and look for a match on the previous phase mapping; if it finds a match the mapped real world area is considered the actual user/device location.
This technique led to poor results mainly due to WiFi RSSI measurements deviation: for example in case of the Redpin[5] project the requirement was to have room-level accuracy (where room's walls mitigate measurements variations over time) using WiFi, Bluetooth and GSM network fingerprints.

RSSI

In this case the idea is to maintain a list of the wifi APs in the considered environment with their own respective absolute location coordinates. Each device detects several signal strength levels from near APs and then uses a trilateration algorithm in order to identify its real absolute location.
As shown in [6] an algorithm has an average deviation of 0.85m considering an $8 * 7$ m^2 room with 3 APs inside.
The main problem in this approach is the signal path loss model that should take into account several possible scenarios: a line of sight between signal transmitters and receivers, or an interrupted path due to obstacles presence and also signals reflection caused by walls or objects surrounding both receivers and transmitters.

TDoA

In the idea of TDoA[7] (Time Difference of Arrival), the position of an object is calculated by measuring the time taken from a packet being sent from a transmitter device to being received by a receiver. By means of Wi-Fi, a client transfers out a time embossed signal which is received by the APs. From the dissimilarity in time between sending and receiving the signals, the distance

between the AP and the client can be designed. When 3 access points are been used, by the principle of triangulation, it's possible to establish location of a client to an accuracy of less than 5 meters[8].

2.1.2 Radio Frequency - Ultrasonic signals

Radio frequency and ultrasonic signals were used in first indoor location approaches: projects using them are mainly older ones because smartphones were not broadly diffused or they had limited capabilities, forcing the use of dedicated hardware to attach to mobile entities tracked.
In last months some projects as Google Nearby[9] brought it back to the front in proximity communication field thanks to newer smartphones computational capabilities: as explained better in the following subsection, Nearby uses an inaudible sound to transmit small data packets between devices; it exploits the fact that if two devices can exchange a valid data packet in this way, then they are one next to the other.

2.1.3 Bluetooth

Bluetooth is the mostly used technology in indoor-location field because of its broad diffusion on everyday life devices (such as smartphones, tablets, cars, etc.) and the interesting features offered by its latest implementations: for example Bluetooth 4 introduces new low energy capabilities, allowing new power and interaction modes and a service-based architecture through the attribute protocol (ATT).
Furthermore, the low cost of hardware platforms makes solutions based on Bluetooth the preferred ones for commercial applications.

First experiments

Some old experiments such as the one reported in Hallberg's work[10] started using as beacon whatever Bluetooth-enabled device, from PC keyboard

adapters to mobile phones. This method uses bluetooth connection's RSSI in order to compute the mobile unit's position (which is the information they wanted) through the distance from the beacon and beacon's position stored by a "positioning service". In this paper they reached an average position error of 1.7m +/- 1.7m.
Several applications were implemented on top of similar architectures (or even simpler ones, like exploiting only device to device detection), where distance accuracy was not so crucial as for other applications: applications like MobiTip[11] (a mobile proximity based recommender system) or social centered applications, usually need raw proximity information.
In latest years the "beacons approach" has shown to be successful in lots of application fields, and some companies, like Apple, made first steps in this field.

Bluetooth Smart and beacons

In 2010 Bluetooth version 4.0 was released and then completed in 2011 with the 4.1 version release. It introduced new features like a service-based architecture through the attribute protocol (ATT) and new power and interaction modes through Bluetooth Smart[12] (also known as Bluetooth Low Energy or BLE) specification: in particular low-energy mode and advertising communication mode are interesting for Bluetooth-powered beacons.
Bluetooth Smart beacons are Bluetooth Smart compliant devices and they act as packets emitter (see various formats in iBeacon - 2.1.3, UriBeacon - 2.1.3 and Eddystone - 2.1.3 sections): in fact, using low-energy features, they can periodically broadcast data packets that Bluetooth Smart (or Smart Ready) devices can listen[12] and use to perform context-specific actions; the advertising-only and low-power modes allow a beacon to have a really long battery life (even years in some cases). Furthermore smartphones can scan for advertised packets (also in a low-power manner) without the need to pair to transmitters, enabling ambient discovery tasks that is a key feature for BLE

beacons in context-aware applications.

The last features contributing to their broad diffusion is the low hardware cost they have: the average price is in a 5-40$ range depending on vendors, transmission power/range and battery life characteristics.

iBeacon

Apple was maybe the first making a move in the Bluetooth Low Energy beacons field developing a standard protocol called iBeacon that defines the broadcasted messages format: in particular it advertises a structured identifier that iBeacon compatible applications can use for trigger context-specific actions such as running background task, showing user notifications or launching apps.

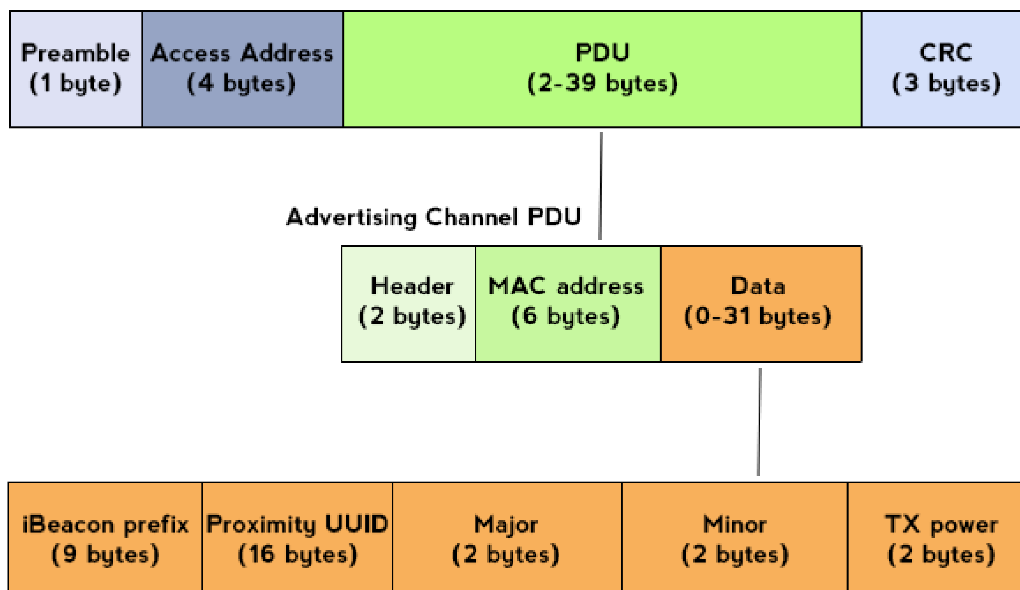

Figure 2.1: iBeacon packet schema

As shown in Figure 2.1, the broadcasted packet composes of several fields; we will examine the PDU Data chunk:

- iBeacon prefix: this field contains some information about the beacon itself, such as the company code, or about the packet such as the broad-

casted packet type (in case of iBeacon packets last 2 bytes have fixed value 0x0215)

- Proximity UUID: is the beacon's main identifier that most of BLE beacons libraries use to filter out detected beacons
- Major: is an identifier used to group beacons with the same UUID
- Minor: is an identifier used to identify individual beacons
- TX Power: is the strength of the signal measured at 1 meter from the iBeacon. This number is then used to determine how close you are to the iBeacon

An example could clarify the identifiers hierarchy: if Apple decides to deploy some beacons in each of its stores, it would define a unique UUID all beacons in all stores all around the world will broadcast; this will allow Apple devices to know when they entered an Apple store. In order to identify which Apple store they entered, the same major value would be broadcasted by all beacons in the same store. Finally, if the app would understand where the device is inside the store, all beacons in all stores would broadcast the same Minor value for the same location in the store.

UriBeacon

UriBeacon[13] is an open format specification for BLE broadcast packets. The project started in 2014 and its goal was to define a new way to discover nearby smart things and give a "web presence" to each of them (an IoT-oriented approach), proposing itself as a bridge between Open Web technologies and Bluetooth beacons.
A UriBeacon can broadcast both HTTP URLs (http and https), UUID URNs (128-bit universally unique identifiers) as well as other URIs.

Eddystone

Recently Google stepped into BLE beacons world with its own new open standard (available under the open-source Apache v2.0 license) called Eddystone (a kind of UriBeacon successor). As reported from Eddystone project's website *"Eddystone is a protocol specification that defines a Bluetooth low energy message format for proximity beacon messages. It describes several different frame types that may be used individually or in combinations to create beacons that can be used for a variety of applications."*
Respect to iBeacon this standard is more focused on broadcasting location information to nearby users (as beacon's latitude and longitude) and it's specifically designed to easily integrate applications in Google environment (using Google Nearby related services).
Furthermore Eddystone includes an ad-hoc module for beacons fleet management, called Proximity Beacons API: this API allows people who maintain beacons powered applications to manage large fleet of beacons in a centralized manner; this option enforces our beliefs in terms of beacons diffusion in the near future.
Eddystone differs from iBeacon because it's an open standard and because it allows its devices to broadcast three different frame types where iBeacon had a single frame type broadcasting an UUID, a major and minor number.

In Figure 2.2 you can see Eddystone packets schema. Here we report a brief description of Eddystone's frame types from specification[14]:

- **Eddystone-UID**: it's the iBeacon parallel frame but it broadcasts a 16byte identifier where 10 bytes act as UUID (called namespace) and 6 bytes as instance (similar to major and minor in iBeacon)
- **Eddystone-URL**: this frame broadcasts a URL using a compressed encoding format in order to fit more within the limited advertisement

packet, using the encoded URL to access remote resource stored in Google's cloud services.

- **Eddystone-TLM**: this frame broadcasts telemetry information about the beacon itself such as battery voltage, device temperature, and counts of broadcasted packets.

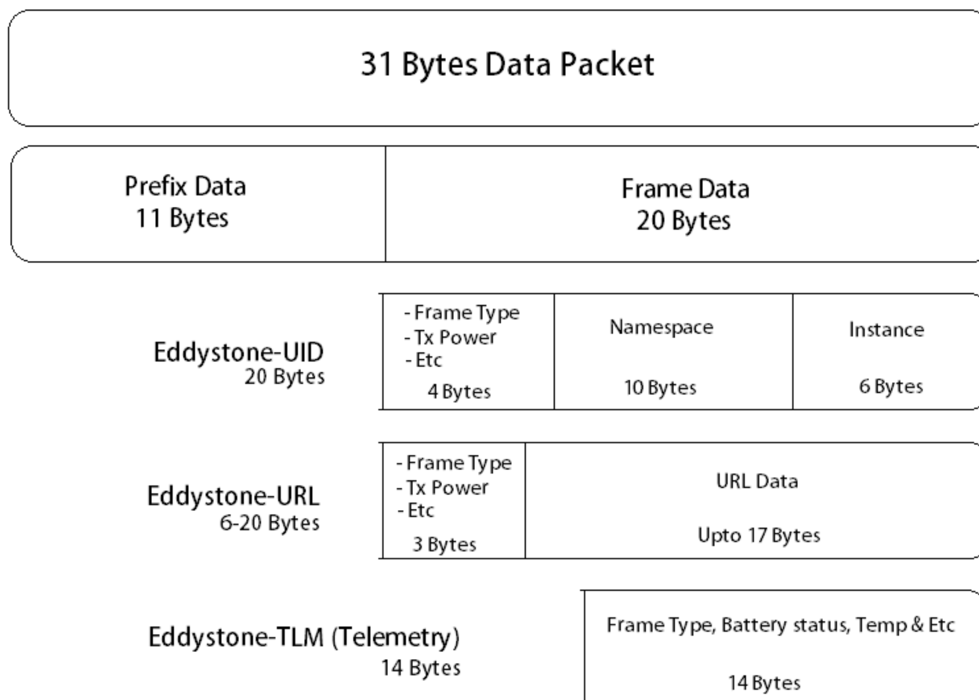

Figure 2.2: Eddystone packet schema

Google will include Eddystone support since Android Marshmallow 6.0 and will release an SDK for iOS devices in order to allow developers to create Eddystone powered apps: in fact Eddystone can be used in a "standalone" manner, as with iBeacon, listening for nearby beacons and react to their proximity on the basis of UID frames; anyway, its power is shown in Google environments and open source projects such as Physical web[15].

2.1.4 Geo-Magnetism sensing

Some studies reported quite accurate results obtained in indoor environments location using geo-magnetism sensors.
The paper Indoor Location Sensing Using Geo-Magnetism[1] describes this approach: in brief this method exploits geomagnetic field distortions caused by steel and concrete skeletons of buildings.
Paper authors investigated the magnetic-field inside a building in order to map magnetic field distortion. Using an electronic compass mounted onto a rotating robotic arm, they moved along a straight line (a corridor) and for each position they measured heading values; they stored these values with the corresponding real headings. They noticed that the heading error varied along the corridor, so they suspected that the error was due to the building's structure.
The proposed system uses a fingerprint technique (see section 2.1.1) where a pre-deployment phase is performed in order to map indoor locations' geomagnetic records with their corresponding real environment location. The clients, collecting the signature during their run, ask for a fingerprint match with a real location to the server that eventually reports back the matched location. In this system they used specific hardware with size constraints similar to smartphones size.
The mean error (of the paper proposed system) reported in case of atriums with radius greater than 15m is about 3m.
The limitation of the system is that the chance of error increases with the size of the fingerprint map, and the cost of mapping every space in a building may require significant effort and time using the presented method.
The advantages respect to other indoor-location systems like beacons-based (either WiFi or Bluetooth) is the lower deployment cost especially in large buildings and the immediate total covering of the building (obviously after the mapping phase).

2.1.5 LTE Direct

LTE is mainly known as a new wireless technology for data exchange but it's also expanding into new frontiers as proved by its evolutions like LTE Advanced and LTE Direct[16].
LTE Direct uses the LTE licensed spectrum and is part of the Release 12 of the 3GPP standard. It is particularly interesting by the proximity point of view because it aims to provide an "always-on device-to-device proximity service" that scales on large number of devices, all in a distributed fashion. LTE Direct proposes itself as a reference technology to overcome main limits of current proximity services approaches using Bluetooth beacons or geolocation: in fact they either use centralized architectures (e.g. in location-based approaches) with privacy issues due to user tracking data, or have a small working range (e.g in case of Bluetooth beacons approaches a beacon range is about 40m) and are not suitable for "always-on modes". LTE Direct instead allows a device to discover nearby devices or services in approximately 500m proximity range, making apps continuously aware of device surroundings in a battery efficient manner; furthermore it aims to work also in crowded environments, discovering 1000s of devices/services[16].
Mobile applications can instruct LTE Direct to monitor for nearby services on other devices or to broadcast their own services through Expressions, filtering data at the device level, without the needs of a centralized architecture or network pings. Moreover applications don't need to keep running in order to react to Expressions but they can be notified of them directly by the device.
LTE Direct-enabled devices broadcast local running services and requests through "Expression": these are service-layer identifiers representing the offered or requested service, identity, interest or location. Other devices wake up periodically (e.g. every 10 seconds) listening for Expressions and broadcasting their own: for each Expression received, the latter is filtered on the basis of applications requests, and eventually notifies them of the proximity event.

2.2 Platform overview

In this section we are going to analyze main features of some platforms offered by several companies operating in indoor-location and mobile fields together with some academic projects developed. The aim of this overview is to understand indoor-location services history and new trends in this field.
We tried to group platforms by technology used with the same order defined in previous subsection.

2.2.1 Redpin

The Redpin project[5] is an open source fingerprint-based indoor location system specifically designed to run on standard mobile devices like smartphones. The aim of the project is to obtain room level location accuracy with a deployment easier than previous proposed solutions: in particular here they exploit final users collaboration for training phase.
For each room mapped they record more than one measurement: each fingerprint is the mapping between the room where measurement take place and the signal strength of the currently active GSM cell, the signal strength of all WiFi access points as well as the Bluetooth identifier of all non-portable Bluetooth devices in range.
The room level accuracy requested helped them to reduce measurement errors or fingerprint mismatch during application running, due to rooms' walls signals absorption.
Redpin consists of two components: a Sniffer that runs on a mobile device and collects signals and fingerprints and a Locator component, that runs on a separate server. The Sniffer scans for signal sources and create the fingerprint while the application code gets the fingerprint and queries the Locator through the network for a corresponding location. The server instead keeps track of all collected fingerprints and their map with real locations (in this case with rooms names) and sends back to the Sniffer query results produced by the

location algorithm implemented.
The location algorithm tries to get a best match out of the given fingerprint: the algorithm uses a threshold value on the distance calculated as a similarity between identifiers and signal strengths; if the distance is not above the threshold the fingerprint is considered as "not matched".
When Redpin application starts, it sniffs for GSM active cell, Wifi access points (recording respective signals strengths) and nearby Bluetooth non-portable devices' IDs: sniffed data are sent to a central server that tries to locate the device given all know fingerprints. The Redpin application continuously sniffs for aforementioned data and queries the server. When a fingerprint match is found it displays to the user the detected location. If no fingerprint in the database matches the given one, than the user can suggest its current position in order to allow the central server to store the new fingerprint in the database: this collaborative approach can simplify the training phase that in other projects could take also some hours for large environments.

2.2.2 ActiveBat

ActiveBat[2] is an ultrasonic-based location system that uses the principle of trilateration.
System's main components are: Bats, which are ultrasound receivers, and a central controller that coordinates the Bats and the receiver chains. Bats are ultrasound transmitters attached to tracked objects having the size of a car key; they are identified by a unique 48bit code and are linked with the fixed location system infrastructure through a radio link. Receivers are placed on the ceiling in a square grid, 1.2m apart, and are connected by a high-speed serial network.
When a Bat is to be located, the controller instructs it to send a short pulse of ultrasound, and its times-of-flight to receivers is measured. A DSP board executes the location algorithm and sends results to the application (running on a PC).By finding the relative positions of two or more Bats attached to an

object, they can calculate its orientation.
The main problem of the system is the large number of receivers required to be mounted on the environment ceiling, and their precise alignment.

2.2.3 Cricket

The Cricket system[3] is an indoor location system developed by MIT taken as reference in lots of academic projects on indoor-location: it has an architecture similar to modern Bluetooth beacons based systems.
Cricket consists of “nodes”, ad-hoc hardware platforms with a transceiver and a microcontroller onboard able to send signals over Radio frequency (RF) and ultrasonic band (US), and able to interface with a host device. There are two types of nodes: beacons that act as fixed reference points of the location system (usually attached to walls) and listeners attached to fixed or mobile objects that need to determine their location.
Beacons transmit their own location as coordinates, but when they are deployed they don’t know their coordinates: to solve this issue a listener attached to a host roams in the environment computing the distance between all beacons; the host then computes, on the basis of several measurements, a unique inter-beacon distances set used to build a uniquely coordinates set that defines the real beacons topology. Finally the host computes coordinates for each beacon such that they resemble real disposition.
Beacons periodically transmit a short ultrasonic pulse followed by an RF message containing beacon’s information like absolute location (in the form of aforementioned coordinates set), a unique beacon identifier, the physical space associated with the beacon, etc.
There’s no central coordination of beacons transmissions. Each listener listens to beacon transmissions, measures its distance from nearby beacons and uses this information to compute its currently absolute position: the distances are measured using a TDoA approach (see TDoA description in section 2.1.1) on RF and ultrasound signals (using the US pulse sent just before a beacon RF

message). If the listener has multiple ultrasonic sensors it can use distance difference information to compute its orientation.
The aim of the system was to have high location accuracy and ease of deployment: the testing outcomes results in a 10cm position error with listeners fixed on a parallel plan with respect to beacons' plan.

2.2.4 Navizon

Navizon[4][17] is a company that offers indoor-location services based on WiFi technologies. In particular they exploit smart WiFi nodes (similar to regular access points): each of them periodically detects WiFi-enabled devices such as smartphones, tablets, laptops or WiFi tags and records their MAC addresses and signal strength, pushing these data to a remote server hosted on a private or public cloud platform. The server estimates devices' locations and records them in local. Each device can query the server through a REST API in order to obtain its own location.
In this case hardware nodes have to be deployed and at least one node must be connected to the internet (nodes form an ad-hoc network in order to allow each of them to push data to the remote server). For a whatever small environment, the company states that at least 5 nodes have to be deployed.
An accuracy of 2-3m is claimed by the company with a 15-20m spacing between adjacent nodes.
The platform provides a Proximity Engine through which the system administrator can instruct the server to notify another "listener server" (using a POST message) about proximity events fired: these events can be defined as signal strength thresholds on a specific node of a specific site.

2.2.5 Insiteo

Insiteo is a company that offers indoor-location services such as navigation, geofencing and indoor maps. They have a configuration platform through

which devices can download setup data in order to run properly. Its solutions mix WiFi access points data with Bluetooth beacons (iBeacon compatible) and devices' sensors such as accelerometers or pedometers in order to obtain better performance (claimed 2m by company website).
The platform they sell allows analytics tasks based on location data collected by devices, indoor map loading and management and devices configuration.

2.2.6 Allseen - Alljoyn

Allseen[18] is an open world-wide consortium that is trying to develop a global standard for IoT communications called Alljoyn.
Alljoyn is a framework that provides an ad-hoc proximal network to several devices running on different platforms and operating systems through a distributed bus: in this way all devices connected to the same WiFi network (in the future other "transport technologies" could be supported) can communicate between each others using the provided interface.
The Alljoyn power is its capability to adapt to lots of scenarios thanks to service discovery functionalities offered in order to make applications aware of devices or services available on the network: a developer can easily build its own service, defining the Actions other peers can perform on it and the signals it can send out (Events); in this way developers can instruct applications with if-this-then-that rules for smarter interactions, even application-to-application interactions.
All communications are performed by a local or remote router instantiated by the framework on behalf of the application layer: for example in case of embedded devices with low computational capabilities the router used is a remote one that resides on a more powerful peer in the network (see figure 2.3).

The communication can happen in different formats like synchronous messages, asynchronous signals (used for notification purposes), etc.
As mentioned before the Alljoyn project is growing more and more with several

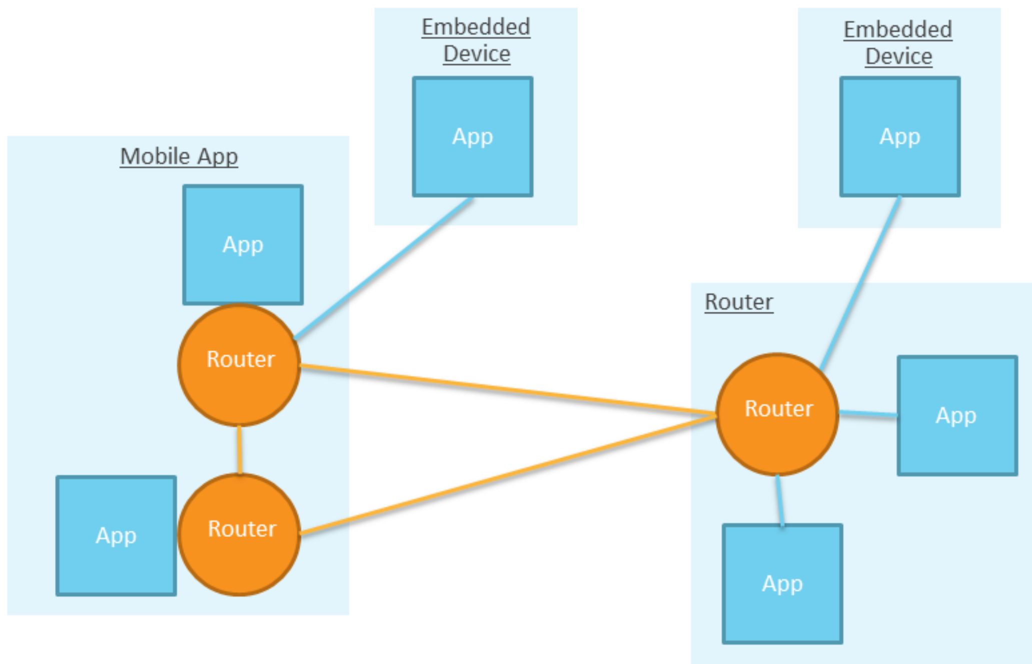

Figure 2.3: Alljoyn network architecture
source: `https://allseenalliance.org`

big companies interested in it. Due to the wide range of applications such a standard can have, some projects were started as "Alljoyn subprojects", developed by approved working groups, in order to implement some services useful to lots of applications: as an example a notification service has been developed in order to send text, audio or images notifications to peers in the network, a configuration service that allows peers to set a property of another peer through the network, etc. Between these projects there's a Location Service Project[19]. It introduces the concept of Entity as a general item applications in the network are interested in and it consists of four different services we briefly describe here:

- **Presence**: this service announces the entry or exit of an entity to the proximal network. Network peers can query for the presence of an entity in the network or they can subscribe to entry or exit events
- **Proximity**: this service provides proximity data on the relative distance between an entity and a proximity service host. A peer can query for

a distance relative to an entity or subscribe to distance updates with minimum update interval or proximity variation constraints

- **Location**: this service provides absolute location data about entities both in a synchronous or asynchronous fashion (always with interval and location updates constraints)
- **Containment**: this service allows peers to query for the containment relationship between two entities. For example a peer can perform a query to know whether an entity A is contained by an entity B or vice versa

2.2.7 Google Nearby

Nearby[9] is a Google project that provides a Proximity API for iOS and Android developers; thanks to this API, Android and iOS devices can discover and communicate with each other, as well as with beacons.
In proximity field, Nearby uses a combination of WiFi, Bluetooth and inaudible sound (using the device's speaker and microphone) to establish proximity and to communicate with devices in proximity.
The "handshake" between two devices in proximity is possible using Google Cloud Services for authentication purposes: this doesn't means that only Google users can use the service, but the process transparently uses random generated tokens stored in the cloud to perform a kind of challenge-response mechanism and notify two involved devices of their proximity (without a distance estimate).
The programmer can instruct the APIs with an indirect range setup: in fact he can limit the "scanning range" to Bluetooth and ultrasound or to ultrasound only range.
To communicate, Nearby exposes publish-subscribe methods, meaning that an app publishes a payload that can be received by subscribers in proximity. So, it is possible to build applications for:

- messages sharing (using Nearby Messages API): it is useful for interactions such as resource sharing or collaborative editing
- real-time connections between devices (using Nearby Connections API): it is useful for local multiplayer gaming and multi-screen gaming.

2.3 Conclusions

In this chapter we analyzed currently available solutions for location-aware and proximity-aware applications. As you can notice the focus is more on providing platforms location-oriented than proximity-oriented ones. We think the main motivation is the higher flexibility provided by location data: in fact proximity data can be inferred by location data these systems provide, motivating companies interests to develop location-oriented products for marketing reasons.

Furthermore almost all systems examined exploit centralized architectures and cloud features to provide smart interaction modes and data analysis or aggregation features not always needed.

As reported in previous sections, last years saw an always growing interest in location-aware scenarios from different points of view: providing new smart interaction modes and context-aware information quickly became a must for successful applications in whatever field, from "simple" smartphone applications to data collect tasks and software.

Nowadays we are seeing this interest moving from technological development (protocols, sensors and technologies) to platforms and smart services offered (see section 2.2.7).

Chapter 3

Our framework

The ADPF framework aims to define common abstractions suitable for proximity-aware applications, ensuring at the same time good flexibility and performance for current and future application fields.
Its focus is to provide new high level objects representing all kinds of actors involved, define a unique data representation suitable for different technologies, and hide low level details to more easily face future evolutions from a technological point of view.

In section 3.1 we provide a general overview of the problems addressed by ADPF. Section 3.2 presents some general use cases pointing out key features and real world scenarios where ADPF could be deployed.
Sections 3.3 and 3.4 define functional and non-functional requirements as inferred from the use cases we present.
Section 3.5 gives a high level description of the abstractions used by the framework together with the framework's main APIs which are explained in section 3.6.
Sections 3.7 and 3.8 describe ADPF's architectural choices, while section 3.9 focuses on security and privacy aspects.
Finally section 3.10 illustrates ADPF's working principles in order to give the reader an overall description of the idea behind framework's implementation.

3.1 Problem analysis

Lots of location-aware applications, especially in those apps that use indoor-location data, need to know the relative distance between the device they're running on and other devices or application specific "actors": for example, in the case of proximity marketing or couponing an app needs to know when the user entered a shop in order to provide products offers, or custom advices.
The key information used by the apps' business logic is the **proximity** between these actors.
Nowadays location or indoor proximity information are mainly made available using a specific technology: for example, a shop can develop its own mobile app that uses the bluetooth beacons or WiFi based devices deployed therein for indoor-location purposes. Each of these technologies has its own library the programmer has to understand and integrate inside his app, taking into account all aspects not managed by the library itself. But what if an application scenario involves more than one technology?
Although our work is mainly focused on indoor environments, we also defined a "bridge" with outdoor technologies (exploiting the well-known macro-location facilities that are offered off-the-shelf by the development platform) so that the programmer can maintain continuous control over user context transitions.
Section 2 pointed out some interesting facts about the indoor-location field: the recent development and release of new standards and technologies (see Google Eddystone in section 2.2), the importance of device-to-device proximity awareness (see Google Nearby and LTE-Direct in sections 2.2.7 and 2.1.5) and the agreement on event-based communication between location services and applications. Furthermore, we thought it could be useful for a programmer to have device and object grouping functionalities.
As a result our framework takes into account all mentioned aspects and tries to face related problems providing straightforward approaches, as easy to use and as safe as possible. A distributed approach to the problem reduces technology

constraints for ADPF-powered applications, limiting costs and effort for their deployment.
The following sections explain all these features in detail, through some toy examples and application scenarios.

3.2 Use cases

In this section we are going to define some "general use cases" that encapsulate several scenarios and examples: they are not meant to show a particular application for indoor proximity but to motivate our framework's functionalities in different situations.

Proximity notifications

Use cases belonging to this class are maybe the most known and most common because they represent the "101 toy examples" about indoor proximity and BLE beacons: retail stores, cinemas and stations are only a few of the contexts for these kinds of examples; couponing and targeted advices are the most valuable outcomes.
We would like to have the same functionalities in other situations, where proximity driven notifications can be powerful. Some examples might be the delivery of custom evacuation instructions to people in a building, in the case of emergency, depending on their current floor or room; the delivery of tracking notifications to warehouse employees; and queue waiting time estimates in public offices reported upon user arrival.
Besides this, in some situations it could be useful to automatically deal with surrounding changes: for example, a user can ask to be notified about the proximity to entities that have specific properties that evolve over time (e.g. "tell me when I'm near a vending machine that was refilled less than 2 days ago").

Device-to-device proximity awareness

In the last few years social networks have been enriched with smarter functionalities. Some exploits user context data, from a post's location to photo geotagging and event tagging of whatever content a user publishes.
Several academic projects and companies have started analyzing possible solutions to provide user proximity information in a reliable manner (see section 2.2.7): in the case of social networks, matching profiles or friends in a user's surroundings can be exploited to provide "immersive" interaction modalities. For example, in a job fair users might want to be aware of whether they are near a company that has open positions matching their skills, or a company might want to be notified about the proximity of new candidates with certain knowledge.
In the smart ticketing field (for events or public transport) a ticket controller could use an app to discover people nearby without a valid ticket in a quick and easy manner, even in crowded situations. Furthermore, virtual leashes can easily be set in order to track pets or children (so that we are notified when they leave a defined space - e.g. a mart playground or a park).

Proximity triggers

In the growing IoT and smart home markets (see section 2.2.6), the chance to have smart appliances take actions on the basis of simple "if-this-then-that" rules seems almost a basic requirement. The machine-to-machine communication that triggers these actions actually give smart products a competitive edge on the market.
The integration of proximity information as a triggering conditions can be useful in lots of cases: to enable stove security when children are nearby, to turn on the lights when someone enters in the house, to have an appliance emit a "task finished" sound only if somebody is in its surroundings, and so on.
Triggers can be used to manage and generate whatever type of data. For example, they can be used to detect people nearby and collect statistics on

the basis of their properties. This kind of application can be useful for trends discovery and data forecasting.

3.3 Functional requirements

In this section we are going to define the functional requirements of the framework, as identified from the use cases presented in section 3.2.

- *Common abstract entity*

  In order to keep our framework as flexible as possible, we use an abstract entity (with common description fields) to map each real world actor involved in an application. This way the framework can access all the actors through the same interface, distinguishing them only by framework relevant information.

- *Device profile*

  Each device in the network has to maintain a description so that it can be identified in device-to-device proximity discovery, as well as advertise it in framework level communications.

- *Proximity data*

  The framework has to provide proximity information to devices in the network, in terms of presence and relative distance.

- *Always-running*

  The framework has to offer a wake up mechanism in order to react to proximity/location events even when the client app using it is in the background or stopped. An activation mechanism is needed so that energy consumption doesn't waste the devices' batteries.

- *Notification mechanism*

  In the real world proximity events happen with no standard frequency or predefined instants. The framework has to deal with both synchronous

and asynchronous events based on the actors' request types. For example a request could be to "give me all the entities within proximity" (synchronous) or "tell me when I arrive at my office" (asynchronous).

- *Grouping*

  The framework has to offer the possibility to choose interesting entities so that it only receives relevant proximity events.

3.4 Non-functional requirements

We identified two main non-functional requirements: first of all, since we can connect devices and broadcast potentially sensible profile data, the framework has to allow the user to decide what and when to share information. Furthermore, the programmer should be able to encrypt communications if activated by the application's context. Secondly, we need to hide the complexities of proximity technologies so that future solutions can be easily integrated in the framework, and to lower the effort required of a programmer to manage his/her entities.

3.5 Framework abstractions

In this section we present the framework components we identified to abstract the above requirements. For the technical detail please refer to Chapter 4.

3.5.1 Entity

To fulfill the requirement of having a common descriptor for every actor, we defined a general component `Entity`. An `Entity` is represented by four fields:

1. *entityID*

   This attribute uniquely identifies an `Entity`. It is not possible to define two entities with the same `entityID`.

2. *entityType*

   This field is used to distinguish entities by their type. We currently support two types: `DEVICE` and `BEACON`. The framework offers the possibility to easily extend the list of possible types (see section 4.2.1 for more details).

3. *distanceRange*

   This represents a relative distance for the `Entity`. See section 3.5.3 for more details about `DistanceRange`.

4. *properties*

   This attribute is used to describe the properties that are typical of each `Entity`. These are only relevant from an application's point of view. For example, the developer of an app for sports centers will add the fact that a user is interested in basket.

3.5.2 SelfEntity

`SelfEntity` is a particular `Entity` built by the framework; it describes the device where the framework is running. It is used to address the need of a device profile, in order for the entity to be identified during framework communication (see section 3.3).

3.5.3 DistanceRange

To deal with the current technological limits in terms of distance computation accuracy[20], we decided to use a set of relative distances, each one representing a range of meters. In particular, we identified these distance ranges:

DistanceRange	Description
IMMEDIATE	0m - 0.5m
NEXT_TO	0.5m - 2m
NEAR	2m - 4m
FAR	4m - 8m
REMOTE	more than 8m
SAME_BEACON	entities are seeing the same beacon
SAME_WIFI	entities are in the same network
UNKNOWN	no distance information available

Table 3.1: DistanceRange table

Latest three values reported in the table above are returned when there is no useful information to estimate relative distance between entities: SAME_BEACON means that two entities (A and B) detect the same BLE beacon but it's impossible to compute their proximity, SAME_WIFI instead is used when A and B have detected no common BLE beacons but they are in the same environment (meaning that they can communicate). Eventually, UNKNOWN is used when there is not distance information at all.

3.5.4 POI

A `POI`, acronym for Point Of Interest, represents a real world location where there are entities that are deemed useful for the application. In practice this means that a `POI` identifies a location where there are beacons.
It is described by a name, a latitude, a longitude and a radius so that a geofence can be created, and a beaconUuid that is used to identify the beacons of interest. The idea is that a BLE scan (and Bluetooth activation) only occurs after the entry in the geofence. This way, the framework does not waste the device's battery scanning for BLE beacons when it is useless.

3.5.5 Group

A `Group` is a set of entities that have something in common: to define it a programmer must provide an "entity prototype" that all group members have to match.
This prototype is described by:

- an entityID: if this field is defined, the group can be composed only by zero or one entity.
- an entityType: with this field, the programmer can decide to create a group based on entity type; the type can be `DEVICE`, `BEACON` or `ALL` (to select both devices and beacons).
- a propertiesFilter: through this attribute the programmer can define a filter on `Entity` properties. Thanks to propertiesFilter it is possible, for example, to create a group with all the entities that are interested in "western movies".

It is important to notice that a `Group` only defines which entities are part of it. There's no direct binding between a group and its entities (i.e. a group does not "physically" collect entities that belong to it). This is due to the fact that the `Entity` set changes continuously. Entities can change their properties, or they can enter(exit) in(from) the `POI`. Keeping an always updated list of entities for each group can be a time and energy consuming task, definitely infeasible in terms of memory consumption and energy expensive synchronization tasks.

3.5.6 Events

Here, we define events in a more fine-grained manner.
In particular, the framework identifies three different types of events:

1. *Group events*

   These are all the events that are related to group changes. They are

produced when an entity changes its properties, or when an entity enters or leaves a `POI`.

2. *Proximity events*

   These are the events that are strictly related to proximity. Every time an entity is in proximity to another, whether device or beacon, a proximity event is fired.

3. *Geofence events*

   These are events that are fired when a geofence entry/exit happens. The geofence can be built around a beacon, or around a single device.

3.6 Framework APIs

In this section we explain how the framework interacts with the programmer. We explain the framework's APIs from a conceptual point of view. For technical details, please refer to chapter 4.
Given that there are three different types of events (see section 3.5.6), the framework offers the programmer three different types of subscription:

1. *Group subscription*

   This is the subscription a programmer has to do to be notified about group events. A group subscription requires only one parameter, the `Group` G the programmer is interested in to receive notifications: every time an entity enters(exits) in(from) G, a group event for that subscription is fired.

2. *Proximity subscription*

   This is the subscription related to proximity events. The programmer has to pass an `Entity` E (which can be only selfEntity - see 3.5.2 - or a beacon), and a `Group` G: every time an entity belonging to G is in proximity with respect to E, a proximity event for that subscription is fired.

3. *Geofence subscription*

   This represents the subscription for geofence events. In this case, the parameters required are an `Entity` E, a `Group` G and a `DistanceRange` D; the framework builds a geofence of radius D around E: every time an entity belonging to G enters/exits in the geofence, a geofence event for that subscription is fired.

The three aforementioned subscriptions are obviously asynchronous: the programmer subscribes for an event and, at a certain point, he will be notified of the firing of the latter.
ADPF offers also a synchronous method, to retrieve all the entities (eventually belonging to a `Group`) in proximity with respect itself in a certain instant of time. This could be useful for the programmer to have an overview of the application state in a certain moment, or to retrieve entities of interest.
Finally, the framework provides obviously a function that permits the programmer to unsubscribe from previously made subscriptions.

3.7 The distributed approach

We decided to store all the business logic needed to compute proximity, fire events and so on, on client side, i.e. on devices, without the help of a centralized server. This distributed approach differentiates ADPF from the totality of solutions currently available on the market (which are explained in section 2.2).
We opted for a distributed solution in order to reduce deployment costs and physical hardware needed for developers who are going to use our framework for their applications.

3.8 Communication

As mentioned in non-functional requirements (see section 3.4) we needed a way to permit communication among devices: we decided to use WiFi, as it is

supported by almost all Android devices on the market and, due to its broad diffusion, makes sense the hypothesis of its presence in most of real use cases (furthermore, whenever it is absent should be easy to deploy).
Communication happens through well-defined messages (see section 4.2.6) using the publish-subscribe pattern, so that the centralized broker is very lightweight and can be deployed also on cheap machines, in accordance with the choice of a distributed approach (as said above in section 3.7).
A high level network schema is reported in Figure 3.1: devices connects to the broker in order to be able to send and receive framework related messages and infer proximity information about different actors in the `POI`.

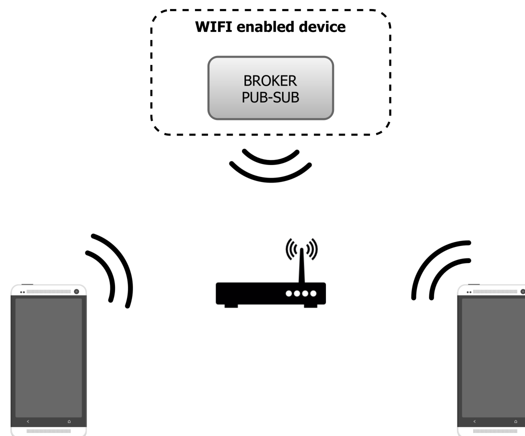

Figure 3.1: Framework network schema

3.9 Security

To address the problem of sharing potentially sensible information, we introduced a security module that permits to the user to select what can be sent on the network by the framework, and what cannot. In particular, a user can decide (only before framework starting, in configuration phase) to send:

- only messages related to proximity and geofence events;
- only messages related to group events;
- no messages.

Furthermore a user can decide to switch on/off the “Ghost Mode” at runtime: enabling Ghost Mode, the device becomes invisible from other users point of view.

Finally, in configuration phase it is also possible to decide whether to send messages using SSL or not.

3.10 Framework working principles

In the previous sections we defined all components and concepts needed to understand how the framework works from a conceptual point of view. So, in this section we are going to explain how ADPF interacts with programmers and applications that use it.

Initial configuration and start

The first and fundamental thing a developer has to do, is to give to ADPF a list of `POI`s (see section 3.5.4); in this way, the framework is able to know when turning on Bluetooth and starting BLE beacons ranging. Furthermore, the framework needs a security configuration (see section 3.9) in order to know what messages it can send, and in which way (i.e. using or not SSL).
At this point, the framework is ready to run: when the application is started, ADPF starts to retrieve position using GPS and Android location functionalities, in order to find out when the user enters in one of `POI`s’ geofences.

Network entry

After a `POI`’s geofence entry, ADPF turns on device’s Bluetooth, and begins to scan for BLE beacons with UUID defined for that `POI`. At the same time, it notifies the client app about the entry event and the app itself can perform an attempt to connect to the broker. If the attempt goes fine, the framework configures the network module for publish-subscribe (connection to broker, subscription to topics, etc.) and announces himself to other entities publish-

ing a message (the `CHECK_IN` message) of network entry: in this way, entities interested in group changes on groups selfEntity (see section 3.5.2) belongs to receive a notification of check in (conceptually equal to a group join). Instead, in case the connection attempt goes wrong, the client app is notified about it and could for example retry a connection.
Upon the connection to the broker is established, the user can subscribe to events is interested in: for every subscription made, ADPF returns to programmer a corresponding object `Subscription` (see technical details in section 4.2.2), on which he can register a callback function that has to be executed once the associated event is fired.

BLE scanning and communication

Up to this point, ADPF is connected to network and is scanning for BLE beacons. Every time it scans, it updates the list of last beacons seen and verifies if some of them matches some proximity or geofence subscription: for every match, it fires the corresponding event.
Furthermore, after a fixed number of scans, the framework broadcasts a message (`PROX_BEACONS` message) with the list of last detected beacons and the computed `DistanceRange` (see section 3.5.3) for each of them, that is used by other devices to find out if they are in proximity.

Proximity computation

When a `PROX_BEACONS` message is received from another device, ADPF compares message beacons with the last detected ones by selfEntity: if at least one is in common, and corresponding `DistanceRange`s are useful (see section 4.2.7), the framework is able to say that selfEntity and the message sender are in proximity: so, it sends to message sender a `PROXIMITY_UPDATE` message, containing information about that computed proximity.
If instead ADPF has sent a `PROX_BEACONS` message and has received a corresponding `PROXIMITY_UPDATE` message by someone, let's say `Entity` E, it

checks if there's some proximity or geofence subscription between selfEntity and a group containing E, and for each matching subscription it fires the corresponding event.
In figure 3.2 is shown proximity computation with a simplified schema, in order to better understand the aforementioned process.

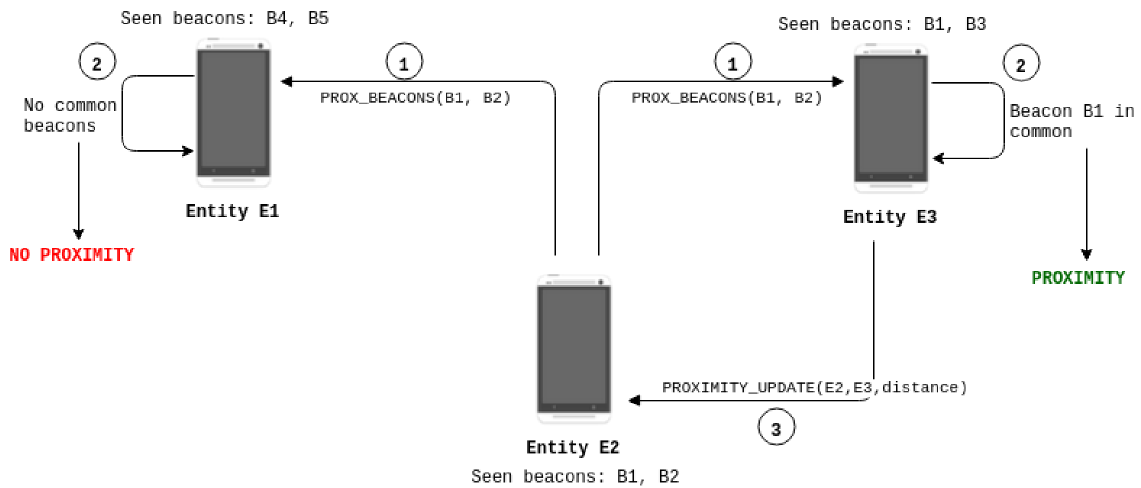

Figure 3.2: Proximity computation

Properties update

At a certain point, an `Entity` can decide to update its properties (see section 3.5.1); in this case, framework has to communicate this change to other entities, because an update of properties could mean a group leave/join. For example, if a user wants to change his interests from "comic movies" to "action movies", he is leaving the `Group` defined by entities interested in comic movies, and joining the `Group` defined by entities interested in action movies.
Then, a properties change is followed by a `PROPERTIES_UPDATE` message sending, containing selfEntity information, old and new properties. When a similar message is received, ADPF checks if there is some group subscription with a `Group` matching old or new properties, and in that case it fires the corresponding group join/leave event.

Synchronous proximity request

As said in section 3.6, a programmer has the possibility to execute a synchronous request to retrieve all entities belonging to a `Group` G in proximity (i.e. with a `DistanceRange` less or equal to the defined one in the request). Also this operation happens in a distributed way: the `Entity` interested broadcasts a message (`SYNC_REQ` message) containing all information about the request (the `Group`, the `DistanceRange`, the last seen beacons). Every device receives this message, checks if it matches the `Group`, and in this case computes its proximity with respect to that entity: if they are in proximity and the computed `DistanceRange` is less or equal to the requested one, the device sends to the entity who sent the request a `SYNC_RESP` message containing information about himself and the `DistanceRange` obtained.
At this point entities already filtered (on group and distance) are ready to use.

Ghost Mode

As explained in section 3.9 the user has the possibility to switch on/off the Ghost Mode at runtime: when Ghost Mode is enabled, ADPF simulates a network disconnection (broadcasting an appropriate `CHECK_OUT` message) and stops sending any kind of message. In this way, the device becomes "invisible", but continues to receive messages from other entities. The user can than decides in every moment to turn back "visible" disabling Ghost Mode: in this case the framework simulates a network entry (with a `CHECK_IN` message) and restart sending messages, according with security configuration set on startup.

Network exit

When a device exits from the network (for whatever reason, from `POI` exit to network error) ADPF broadcasts a `CHECK_OUT` message, containing information about selfEntity. Entities receiving the message check for matching group subscriptions (i.e. the `Entity` that sent the `CHECK_OUT` message belongs to the

`Group` they subscribed for), and for each match they fire a check out event (conceptually equal to a group leave).

Chapter 4

Library architecture

In this chapter we present the ADPF framework architecture in deep, explaining how internal components work, how they interact between them and providing detailed motivations for choices made: obviously main focus is kept on ADPF core components for each layer.
Furthermore components implementation details and inter-device communication are examined together with internal classes and interfaces.

Section 4.1 provides a coarse-grained description of ADPF internal divisions and components.
Sections 4.2, 4.3 and 4.4 describe ADPF layers pointed out in previous overview, with internal components and abstractions analyzed in deep from the point of view of functionalities offered and implementation.
Section 4.5 gives an overview of security options and configuration ADPF provides to programmers in order to fulfill privacy and consistency requirements.
Finally section 4.6 explains in detail choices made for devices communication tasks and components addressing this problem. Furthermore it briefly describes MQTT protocol used for message exchange purposes.

4.1 Architecture overview

The framework is composed of several different layers, each one involved in different business logic tasks, that interact among them to offer functionalities described in chapter 3.

Figure 4.1 shows an overview of the architecture.

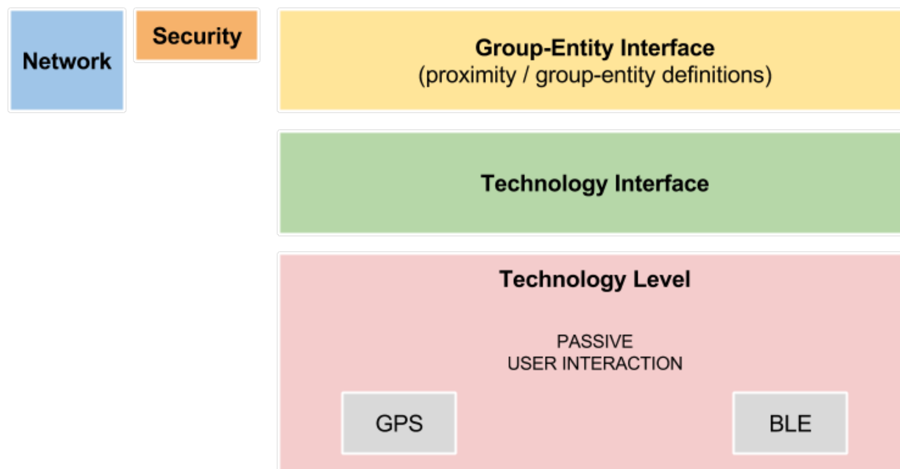

Figure 4.1: Architecture overview

Starting from bottom, we have:

- the **Technology level**. It is the lowest level of the framework and it is the interface between framework and Android OS services it uses. It is composed of technology specific objects providing proximity information and usually managing available sensors.
- the **Technology Interface**. It creates a common abstraction for all the Technology level resources, so that the superior level can handle different technologies in the same way. This means that, from Group-Entity Interface point of view, the technology used to perform application-specific functionalities is transparent and the interaction APIs are the same for every kind of technology, whether it's BLE, GPS or whatever technology the framework will support in the future.

- the **Group-Entity Interface**. It represents the interface exposed to the developer: here are defined the APIs through which the programmer can retrieve Proximity and Geolocation information.

Furthermore, at the same level of Group-Entity Interface we can find two modules involved in specific non-functional requirements: the **Security** module, that defines user's privacy and security policies in terms of what and how share proximity information, and the **Network** module, which manages all the network operations such as connection, disconnection, messages sending and receiving.
In following sections we will inspect each layer in detail.

4.2 Group-Entity Interface

As said above, Group-Enitity Interface is the layer that defines proximity APIs provided to the programmer. These functions are exposed through the `ClientProximityAPI` interface, which is implemented by the `GroupEntityManager` class.
`GroupEntityManager` is the core component of the level: in fact, it interacts with Network and Security modules in order to handle messages and their corresponding business logic, and also it communicates with the lower level (see section 4.3) to retrieve information about proximity and geolocation.
Furthermore, this layer defines high-level abstractions such as `Entity`, `Group`, `POI` and `Subscription` logically explained in section 3.5 while here we describe their implementation details.

Figure 4.2 shows a simplified UML class diagram for this layer.

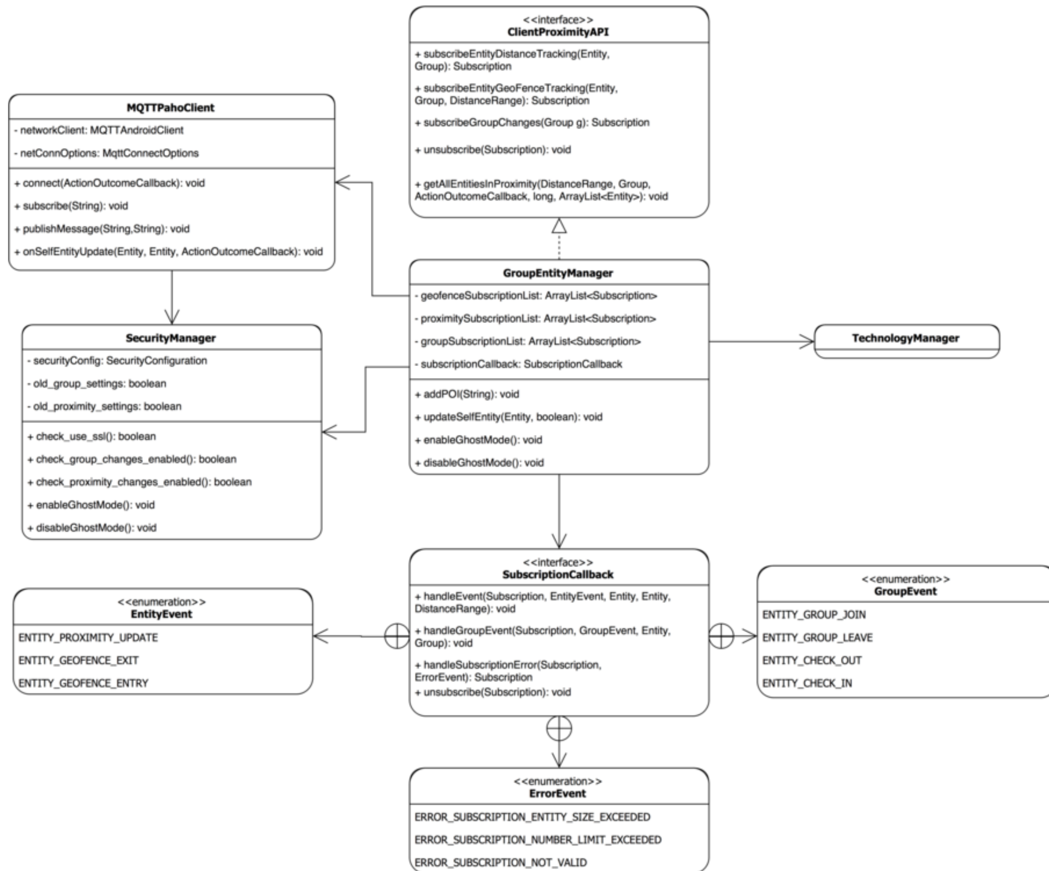

Figure 4.2: Simplified class diagram for Group-Entity Interface layer

4.2.1 Utility classes and interfaces

Before starting with abstraction descriptions we report here a brief description of classes (or subclasses) and interfaces used by framework or the client application.

PropertiesFilter

This class allows the framework to easily define filters on entities' JSON properties such that flexible grouping functionalities can be offered to programmers. An additional feature useful for the framework is the possibility to serialize and deserialize from/to string format the filter built: in fact a `PropertiesFilter` object is internally composed of a JSON object that acts as a filter descriptor; this descriptor can be serialized to a `String` and can be sent attached to net-

work messages when needed and, upon received, the JSONPath[21] equivalent filter can be built and used to evaluate properties objects.

Entity.Type

This class defines possible entities types the framework can deal with and enable the programmer to filter on events related only over specific types. Other than real types a “special type” make easier the filters building.
Types defined (in first release) are:

- `BLE_BEACON`: this type represents Bluetooth beacon real entities.
- `DEVICE`: this type represents devices in the network.
- `ALL`: this type is a wildcard used to easily build filters involving all types of entities in the surroundings.

The framework uses this class to standardize entity types (and filters building) and to easily extend managed types in the future.

SubscriptionCallback

This interface defines the callback methods called by the `GroupEntityManager` in order to notify client application of events fired or subscription errors occurred in case of geofence limits (see test 5 in section 5.4 for limit values) reached. Obviously the app has to implement this interface and pass the reference of the implementation to the `GroupEntityManager`.
All methods take as parameter the `Subscription` object to which the event relates: this object is the same returned by `GroupEntityManager` when the client app submitted the subscription, so that it can determine tasks to perform on the basis of the subscription itself.
Another parameter all methods take is the event related to the method call: in fact `SubscriptionCallback` interface defines three types of events respectively `EntityEvent`, `GroupEvent` and `ErrorEvent` used by the app to properly react

to them.

In particular methods the programmer has to implement are:

- `handleEvent`: this method is called when a proximity event is fired (depending on a previously submitted proximity or geofence subscription); additional parameters passed are the two `Entity` objects involved in the event and the `DistanceRange` value this event wants to notify to the client app.

- `handleGroupEvent`: this method is called when a group event is fired (depending on a previously submitted group subscription); additional parameters passed are the `Entity` involved in the event itself and the `Group` object matching the `Entity`.

- `handleSubscriptionError`: this method is called whenever an error occurred during geofence subscription submission due to geofence subscriptions limits.

ConnectionStateCallback

This interface is used by the client application to be aware of network connection and disconnection events. Methods defined are:

- `onConnected`: this method is called whenever a connection to the network broker having address specified by the client app at `GroupEntityManager` creation time.

- `onDisconnected`: this method is called whenever an unexpected disconnection from the network occurred, for example if the user switch off the device's WiFi connectivity or if a "connection lost" event occurred.

- `onConnectionFailed`: this method is called upon a connection attempt performed, fails. The client app can then decide whether to retry a connection to the network broker or not (maybe on the basis of WiFi status information retrieved from Android system).

The client application can implement this interface and properly react to these events occurrences (for example retry a connect).

4.2.2 Abstractions objects

After the description of classes used by the abstraction objects we present here the latters in detail.

Entity

`Entity` object is the most important DAO in the framework due to the fact that each instance abstracts a real world object or device.
It contains following fields:

- `entityID`: this field is a `String` representing the `Entity`'s identifier used to distinguish whatever entity in the network or in device's surroundings.
- `entityType`: this value is a constant defined by `Entity.Type` enum class (subclass of `Entity`) and used to determine this `Entity` type (see above for `Entity.Type` description).
- `distanceRange`: this field is a `DistanceRange` defined value (see section 3.5.3) and is used to piggyback proximity values from lower levels.
- `properties`: this field is a `JSONObject`[22] object representing entity properties used by `GroupEntityManager` to filter on events and messages.

This class contains also a `Builder` subclass helping programmers and lower levels to build correctly `Entity` instances.

Group

This class defines `Group` instances used to filter on entities in events subscriptions: each instance represents a kind of `Entity` "prototype" used to determine whether an `Entity` belongs to the group or not. Fields contained are:

- `groupDesc`: this field has been included as a human-readable description of the group alternative to `toString` representation defined by the framework.

- `filter`: this field represents a `PropertiesFilter` object used to evaluate the `Entity` properties (see previous subsection to a detailed `PropertiesFilter` description).

- `entity_id`: this field is a `String` object used by the `Group`'s `evaluate` method to filter entities on their identifiers.

- `type`: this field is an `Entity.Type` object used to filter on entities' type.

As for `Entity` class, a builder subclass is provided in order to help programmers to correctly build `Group` instances.
The most important method of this class is the `evaluate` method that permits to determine whether an `Entity` instance belongs to the group or not.
Evaluation task flowchart is represented in figure 4.3.

POI

This class defines objects representing `POI`s the framework has to monitor. Data contained are:

- `name`: this field is a `String` object included as a human-readable name of the `POI` object but in most of the cases is useful for the programmer to use it as an identifier, so that he can easily determine upon entry or exit events happen which `POI` they are referring to.

- Geofence data: `POI` center geolocation data (maintained as latitude and longitude values) and geofence radius value defined by client application.

- `beaconUuid`: this field represents an iBeacon UUID identifying Bluetooth beacons entities within the `POI` advertises. The fact that only one identifier can be set is motivated by the meaning of iBeacon UUID identifier (see section 2.1.3 for iBeacon format and identifier details).

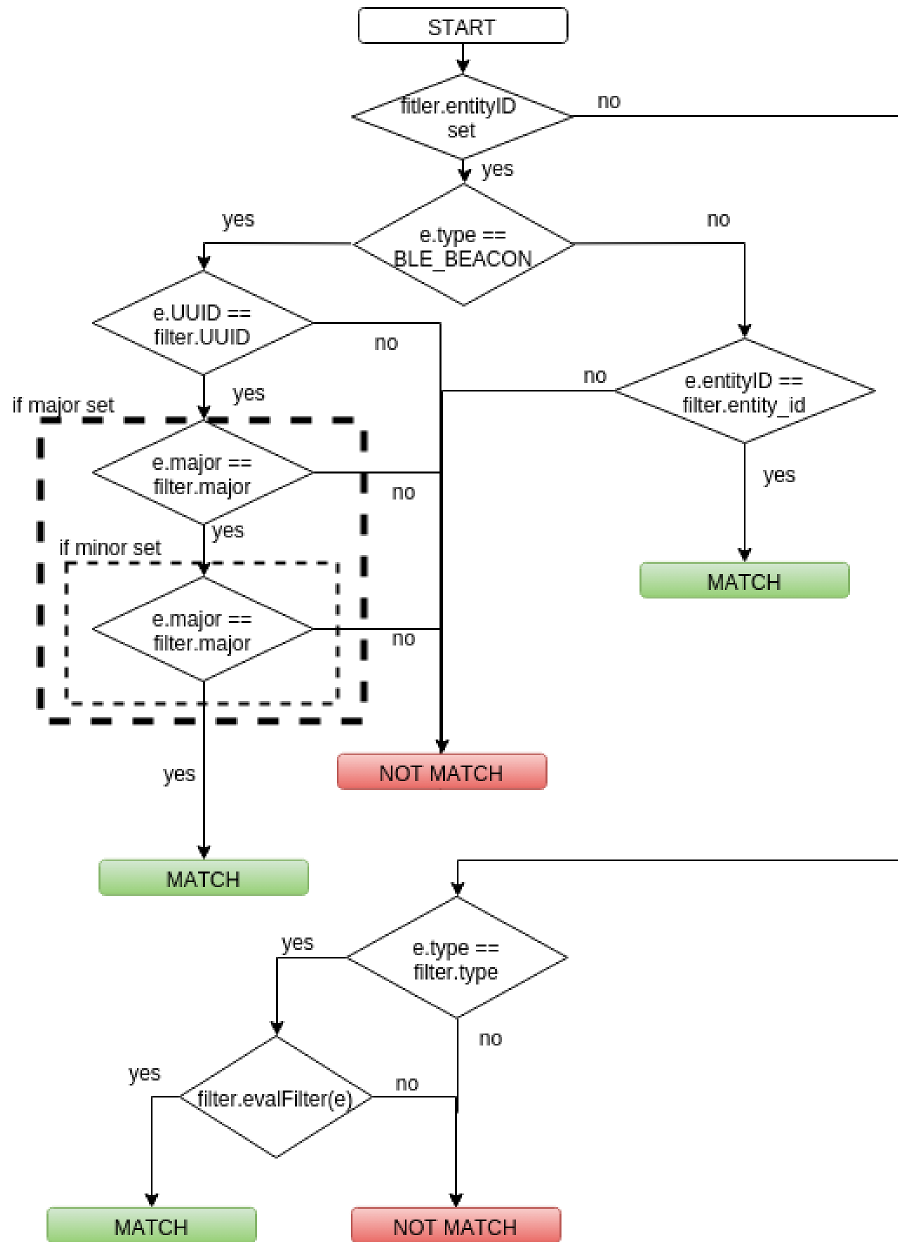

Figure 4.3: `Group.evaluate` method flow chart

Subscription

This class defines the `Subscription` object created by the framework and used both by the framework and the app: in fact when the latter subscribes to whatever type of event (proximity, group or geofence), the framework builds a `Subscription` object and returns it back to the client app so that it can determine in `SubscriptionCallback`'s methods to which subscription the fired

event relates.
Fields contained are:

- g1,g2: Group objects representing two groups involved in this subscription [1].

- distanceForGeofence: this value is a DistanceRange class value defining size of the geofence (if this Subscription is a geofence subscription).

- e1: this field contains the Entity center of this Subscription (if it's a geofence subscription).

- detectedEntities: this field is a list of Entity objects containing (if this is geofence Subscription) all entities within the geofence described by above fields (entities for which an entry event has been received).

4.2.3 Subscription events

Here we are going to describe events client application can be notified of (and properly react) and their meanings: in order to handle them the SubscriptionCallback interface (see section 4.2.1) has to be implemented by the app. All events reported are relative to subscription (of corresponding type) submitted by the client application.
Possible events the framework provides are divided by type in following subsections.

Proximity events

These events are fired as consequence of a proximity or geofence subscription submitted to the GroupEntityManager.

ENTITY_PROXIMITY_UPDATE is the event fired when the framework computed

[1] Although subscription relate only an Entity and a Group, to easily extend the framework in the future supporting group-group subscription, the framework only builds Subscription objects having g1 matching the Entity defined in subscription method

a fresh relative distance value (expressed as `DistanceRange` value) between two entities E_1, E_2 where E_1 is the one defined at subscription time and E_2 matches the subscriptions' `Group` filter.

`ENTITY_GEOFENCE_ENTRY` is the event fired when an `Entity` E entered a geofence defined by a geofence subscription S submitted such that E matches `Group` defined by S.

`ENTITY_GEOFENCE_EXIT` is the event fired when an `Entity` E exited from a geofence defined by a geofence subscription submitted such that E matches `Group` defined by S.

Group events

These events are fired as consequence of a group subscription (with defined `Group` matching the `Entity` involved in the event itself) submitted and they aim to notify client application about group related "facts" involving a single `Entity` of type `DEVICE`.

`ENTITY_CHECK_IN` is the event fired when an `Entity` enters the network (connecting to network broker) where selfEntity is already in.

`ENTITY_CHECK_OUT` is the event fired when an `Entity` leaves the network where the selfEntity is already in, due to both an unexpected or wanted disconnection.

`ENTITY_GROUP_JOIN` is the event fired when an `Entity` E joins a certain group G. It is fired on a device where selfEntity submitted a group subscription on entities matching G. E joins G changing its properties from P_1 to P_2 where P_2 matches G's filter while P_1 doesn't.

`ENTITY_GROUP_LEAVE` is the event fired when an `Entity` E leaves a certain group G. It is fired on a device where selfEntity submitted a group subscription on entities matching G. E joins G changing its properties from P_1 to P_2 where P_1 matches G's filter while P_2 doesn't.

Error events

These events are related to subscription submission errors: in fact the framework defines boundaries (see test 5 in section 5.4) on the number of geofence subscription selfEntity can submit and the constraints on entities passed as parameter.

`ERROR_SUBSCRIPTION_ENTITY_SIZE_EXCEEDED` is the event fired when the maximum number of entities the framework can manage for a single geofence subscription is reached.

`ERROR_SUBSCRIPTION_NUMBER_LIMIT_EXCEEDED` is the event fired when the maximum number of simultaneous geofence subscriptions the framework can manage is reached.

`ERROR_SUBSCRIPTION_NOT_VALID` is the event fired when an `Entity` passed as subscription parameter is neither of type `DEVICE` nor `BLE_BEACON`

4.2.4 POI Events

This section describes poi events the client app can subscribe to: they are entry and exit events available as Android Broadcasts so that the client app can register a `BroadcastReceiver` to handle them.
`GroupEntityManager` provides a method to get the action string related to these events and extras' keys containing additional events information (such as an entry or exit event flag).

The Broadcast mechanism has been used in order to make client application aware of these events even when it is not running.

4.2.5 ClientProximityAPI

In chapter 3 we introduced framework APIs provided to the programmer from a logical point of view (see section 3.6). Here, we report APIs with a more technical approach, showing their signature and briefly describing their behaviour.

In particular, provided APIs are:

1. ```
   public Subscription subscribeGroupChanges(Group g);
   ```

   Through this method, the programmer can subscribe to group events. Passing the required `Group` as parameter, ADPF returns a `Subscription` object so that the programmer can bind the `SubscriptionCallback` for the event with the corresponding `Subscription`.

2. ```
   public Subscription subscribeEntityDistanceTracking(Entity e1
       , Group g2);
   ```

   This is the method that a programmer can use to subscribe to proximity events. The method accepts as parameters an `Entity` e1 (selfEntity or a beacon) and a `Group` g2: in this way, every time an `Entity` belonging to `Group` g2 is in proximity with respect e1, the programmer is notified with an event. As for the previous method, it is returned a `Subscription` object with the same aim as before.

3. ```
   public Subscription subscribeEntityGeoFenceTracking(Entity e1
       , Group g1, DistanceRange distance);
   ```

   This method is the one the programmer has to call to subscribe to geofence events. A new parameter is required: the `DistanceRange` repre-
   ```

senting the geofence radius. Every time an `Entity` belonging to `Group` g1 enters or exits the geofence created on e1, a geofence event is fired. As before, the method returns a `Subscription` object.

4. ```
public void getAllEntitiesInProximity(DistanceRange distance,
    Group g, ActionOutcomeCallback callback, long millisec,
   ArrayList<Entity> result);
```

   This is the "synchronous" method to retrieve all entities belonging to `Group` g at a `DistanceRange` less or equal then `distance`, passed as parameter. In this case, selfEntity sends a request message and waits for responses from entities in proximity. Other parameters are: the `ActionOutcomeCallback` representing the callback function to be executed on completion, the time (in milliseconds) to wait for responses from entities, and the `ArrayList` of entities which will be filled with entities in proximity.

5. ```
public boolean unsubscribe(Subscription s);
```

   Finally, this is the method used to unsubscribe from a `Subscription` passed as parameter: method returns `true` if the operation was successful, otherwise `false`.

4.2.6 Framework messages

Before describing the `GroupEntityManager`, we need to describe framework messages in detail, as they are essentials to understand the Group-Entity Interface mode of operation. For this reason, in this section we report all messages, with their structure and a brief description of where we make use of them.
The common characteristic is that they are in `JSON` format, because of its ease of parsing, use and conversion to `String` objects.

CHECK_IN message

This is the message an `Entity` sends when enters the network. It is used to announce himself, in order to fire `ENTITY_CHECK_IN` events for interested entities. The message structure is:

```
{
    "message":{
        "sender":"senderID",
        "msg_type":"CHECK_IN",
        "entity":{
            "entity_id":"entityID",
            "entity_type":"DEVICE",
            "distance_range":"distance",
            "properties":{
                "properties":{
                    "tags":["tag1","tag2"],
                    "prop1":"p1"
}...}
```

Listing 4.1: `CHECK_IN` message

CHECK_OUT message

In parallel with `CHECK_IN` message, this is the message sent when an `Entity` exits the network. It permits the firing of `ENTITY_CHECK_OUT` events, and its structure is the following:

```
{
    "message":{
        "sender":"senderID",
        "msg_type":"CHECK_OUT",
        "entity":{
            "entity_id":"entityID",
            "entity_type":"DEVICE",
            "distance_range":"distance",
            "properties":{
```

```
                    "properties":{
                        "tags":["tag1","tag2"],
                        "prop1":"p1"
                    }...},
                "valid":true
        }...}
```


Listing 4.2: CHECK_OUT message

The boolean field `valid` is used to distinguish between a real network disconnection and a fake one; the latter is used when an `Entity` wants to change its properties, as explained in detail in the next sections.

PROPERTIES_UPDATE message

This is the message sent when an `Entity` updates its properties, in order to fire `ENTITY_GROUP_JOIN` and `ENTITY_GROUP_LEAVE` events. It's structured in this way:


```
{
    "message":{
        "sender":"senderID",
        "msg_type":"PROPERTIES_UPDATE",
        "entity":{
            "entity_id":"entityID",
            "entity_type":"DEVICE",
            "distance_range":"distance",
            "current_properties":{
                "properties":{
                    "tags":["tag1","tag2"],
                    "prop1":"p1"}
            },
            "old_properties":{
                "properties":{
                    "tags":["tag1","tag2","tag3"],
                    "prop1":"old_p1",
                    "prop2":"old_p2"
```

```
}...}
```

Listing 4.3: `PROPERTIES_UPDATE` message

PROX_BEACONS message

The `PROX_BEACONS` message is sent by an `Entity` to notify others which are the last detected beacons, and at which distance it is from them. When it is received, is used to understand if two entities are in proximity. The structure is:

```
{
    "message":{
        "sender":"senderID",
        "msg_type":"PROX_BEACONS",
        "entity":{
            "entity_id":"entityID",
            "entity_type":"DEVICE",
            "distance_range":"distance",
            "properties":{
                "properties":{
                    "tags":["tag1","tag2"],
                    "prop1":"p1"}
        }...},
        "beacons":[
            {   "beacon_id":"beaconID_1",
                "distance_range":"distance_1"},
            {   "beacon_id":"beaconID_2",
                "distance_range":"distance_2"}]
}...}
```

Listing 4.4: `PROX_BEACONS` message

PROXIMITY_UPDATE message

This message is sent by an `Entity` to notify another one that they are in proximity, at a certain `DistanceRange`. It permits the firing of every proximity

event (see section 4.2.3), and its structure is the following:


```
{
    "message":{
        "sender":"senderID",
        "msg_type":"PROXIMITY_UPDATE",
        "entity1":{
            "entity_id":"entityID_1",
            "entity_type":"DEVICE",
            "distance_range":"distance_1",
            "properties":{
                "properties":{
                    "tags":["tag1","tag2"],
                    "prop1":"p1"
        }...},
        "entity2":{
            "entity_id":"entityID_2",
            "entity_type":"DEVICE",
            "distance_range":"distance_2",
            "properties":{
                "properties":{
                    "tags":["tag1"],
                    "prop1":"p4",
                    "prop2":"p3"
        }...},
        "distance_range":"distance"
}...}
```


Listing 4.5: `PROXIMITY_UPDATE` message

SYNC_REQ message

This message is used to perform the synchronous request of all entities in proximity. It contains the topic on which other entities has to reply, the last detected beacons such that other entities can compute proximity, and the `Group` and the `DistanceRange` required.

```
{
    "message":{
        "sender":"senderID",
        "msg_type":"SYNC_REQ",
        "topic_reply":"topic",
        "beacons":[
            {   "beacon_id":"beaconID_1",
                "distance_range":"distance_1"},
            {   "beacon_id":"beaconID_2",
                "distance_range":"distance_2"}],
        "group":{
            "filter":{
                "tags":["tag1", "tag2"],
                "prop1":"p1"},
            "entity_id":"entityID",
            "entity_type":"DEVICE",
            "group_descriptor":"description"},
        "distance_range":"distance"
}...}
```

Listing 4.6: SYNC_REQ message

SYNC_RESP message

It is the message sent by entities matching a SYNC_REQ message, that is entities belonging to Group requested in the synchronous request, in proximity with respect to Entity which sent the message. It has this structure:

```
{
    "message":{
        "sender":"senderID",
        "msg_type":"SYNC_RESP",
        "entity":{
            "entity_id":"entityID",
            "entity_type":"DEVICE",
            "distance_range":"distance",
            "properties":{
```

```
                    "properties":{
                        "tags":["tag1","tag2"],
                        "prop1":"p1"
}...}
```

Listing 4.7: SYNC_RESP message

4.2.7 GroupEntityManager

As written before, `GroupEntityManager` is the core component of the Group-Entity Interface layer: it implements almost all the high-level business logic (proximity computation, events firing etc.), it communicates with superior and lower layers (respectively the application using the framework and the Technology Interface), and also interacts with Network and Security modules in order to coordinate their operations.

Initial setup: ADPF as a Service

In functional requirements section (see 3.3), we said that ADPF has to be always-running, meaning that also when applications using the framework are in background, ADPF has not to change its behaviour, continuing to send messages and notify events to the user.
For this reason, we decided to implement ADPF as an Android Service (called `ProximityService`), to which an application has to bind in order to take advantage of its functionalities. Creation of `GroupEntityManager` is delegated to the `ProximityService`, which creates a single instance of it and returns that to the programmer: in this way, a single `GroupEntityManager` instance is create.
Summarising, in order to make ADPF operative, the programmer has to start it as a Service, and than retrieve from it the created `GroupEntityManager` instance.
When `GroupEntityManager` is created, many operations are performed:

- selfEntity and security configuration (passed as parameters from programmer) are set.
- the `TechnologyManager` instance (see section 4.3.2), which is the Technlogy Interface component used to communicate with the Technology Level, is created.
- the network component (`networkClient`) used to connect to the broker is created, and then two callback functions of this `networkClient` are implemented: the one to handle received messages, and the one related to a successfull connection (subscription to topics, etc.).
- a `Scheduler` object is initialized; the `Scheduler` is a component that handles a thread pool: in this way, every time we need to perform a concurrent operation, all we have to do is to schedule a new `Thread` into the `Scheduler`.

Communication settings

As said above, on creation the `GroupEntityManager` "configure" network interface. In particular, we identified four different topics for the publish-subscribe model:

1. **BROADCAST**: on this topic are sent and received messages related to check_in and check_out events, namely `CHECK_IN` and `CHECK_OUT` messages, and the `SYNC_REQ` message, which is used to realise synchronous request of entities in proximity.
2. **GROUP**: on this topic are sent and received messages related to group events, then `PROPERTIES_UPDATE` messages.
3. **PROXIMITY**: here are sent and received all `PROX_BEACONS` messages.
4. ***entityId***: this is the topic where every `Entity` receives `PROXIMITY_UPDATE` messages involving it.

From a technical point of view, the first three topics are all broadcast topics, while only the latter is different for each entity. We opted for three different broadcast topics in order to maintain the logical separation between messages, and also to try to distribute them not creating a unique queue for all messages.

Adding POIs

Up to this point, `GroupEntityManager` is created and ready to run, but it needs at least one `POI` in order to start low-level proximity updates. Then, the `GroupEntityManager` offers a method `addPOI` that accepts as paramater a `String` representing a `JSON` file containing all `POIs` with related information. A utility method parses this `String` and, for each `POI` extracted, `GroupEntityManager` creates a new `ProximityData` object, implementing `ProximityListener` callback functions (see section 4.3.1 for logical explanation of them). See next section for callbacks implementation in detail.

ProximityListener callbacks implementation

`ProximityListener` callback functions are implemented as follow:

1. `onProximityChanged`
   This callback has as parameter a `ProximityResult` containing results from some proximity provider. Currently only results from BLE provider are handled: so, an execution of this method corresponds to a new BLE scan, and `ProximityResult` contains a `List` of detected beacons. First of all, for each of them `GroupEntityManager` checks if some proximity or geofence subscription is matched.
   Then, a variable representing the counter of BLE scans is increased: if the counter has reached the constant value `DEFAULT_BLE_SCAN_COUNTER` a new `PROX_BEACONS` message is built and sent on the network; in this way we don't send a `PROX_BEACONS` message for every BLE scan, but every 60s, as `DEFAULT_BLE_SCAN_COUNTER` is equal to 6, while the time

between two BLE scans is equal to 10s (see test 1 in section 5.4 for reasons about these parameter values and the architectural choice of not sending a message for every scan).
In the end, the `List` of last detected beacons is updated: to preserve memory occupation, at most 3 beacons are saved for each BLE scan; in particular, we decided to store the two nearest beacons, and the farthest one.

2. `onEnterGeofenceArea`
   This callback represents a `POI` geofence entry: `GroupEntityManager` creates an ad-hoc `Intent` and broadcasts it, so that a `BroadcastReceiver` implemented by the programmer can intercept and handle it.

3. `onExitGeofenceArea`
   Opposite of the previous one, this callback represents a `POI` geofence exit: also in this case, an ad-hoc `Intent` is sent in broadcast in order to be caught by a `BroadcastReceiver` implemented by the programmer.

Subscriptions handling

To manage subscriptions, the `GroupEntityManager` maintains three distinct `ArrayList<Subscription>`, one for each type of `Subscription`. Every time the programmer submits a new `Subscription` using the related API (see section 4.2.5), `GroupEntityManager` adds it to the corresponding `ArrayList`.
For geofence and proximity subscriptions, `GroupEntityManager` checks that the `Entity` passed as parameter was selfEntity or a beacon: if not, a `ERROR_SUBSCRIPTION_NOT_VALID` event is fired.
Only for geofence subscriptions, an additional control is needed: in fact, for memory occupation and performance reasons we decided to limit the possible number of geofence subscriptions to 5 (see test 5 in section 5.4); for this reason, before adding the new geofence subscription to the corresponding `geofenceSubscriptionList`, the `GroupEntityManager` checks the

`geofenceSubscriptionList` size: if it has already reached the limit, a `ERROR_SUBSCRIPTION_NUMBER_LIMIT_EXCEEDED` event is notified.

Entity check_in

When an `Entity` enters in the network, a `CHECK_IN` message is sent to every device on the `BROADCAST` topic. How is processed the message from the `GroupEntityManager`, once it is received?
A `CHECK_IN` could fire two types of event, on the basis of subscriptions: in case a proximity subscription was made for a `Group` containing the incoming `Entity`, an `ENTITY_PROXIMITY_UPDATE` with `DistanceRange` `SAME_WIFI` has to be fired; instead, in case a group subscription is matched, an `ENTITY_CHECK_IN` event has to be notified. For this reason, once a `CHECK_IN` message is received, two methods are called:

- `evaluateProximity`, to check the former case: this method receives as parameters two entities and a `DistanceRange`; for each matching proximity subscription in `proximitySubscriptionList`, an `ENTITY_PROXIMITY_UPDATE` event is fired with the `DistanceRange` passed.
- `evaluateCheckIn`, to check the latter case: this method instead receives an `Entity` as parameter and checks if it matches some group subscription; for every match, an `ENTITY_CHECK_IN` event is fired.

Entity check_out

Opposite to the previous section, this is the case when an `Entity` exits from the network. Obviously it is impossible to send a message notifying the check_out after the network exit, furthermore the network disconnection can happen for several reasons, and in some of them the framework cannot know the moment in which the `Entity` is going out (otherwise, the `CHECK_OUT` message is sent on disconnection), in order to notify the check_out immediately

before the network exit.

For this reason, we used an helper from the protocol we used for publish-subscribe (see 4.6.1), the so-called "**will message**": a message that can be set on network connection, and is sent in broadcast by the broker once the client is unexpectedly disconnected, for example because of a network error. So, we bound a `CHECK_OUT` message with the "will message", in order to notify other devices once an `Entity` is disconnected following a network error.

When `GroupEntityManager` receives a `CHECK_OUT` message, it calls the `evaluateCheckOut` method, that fires an `ENTITY_CHECK_OUT` event.

Properties update

As previously said, the only way to notify a `ENTITY_CHECK_OUT` event is through the "will message", which is set on network entry and cannot be modified anymore. So, changing `Entity` properties means necessarily a re-connection of the `Entity` as a new actor, with same `entityID` but different properties, in order to maintain the consistency of the "will message", and thus the consistency of the associated `ENTITY_CHECK_OUT` events.

Let's clarify with a toy example:

- Alice subscribes to group changes for group G with filter "tags contains expo";
- Bob enters in the network with properties "tags = [expo]", so Alice is notified with an `ENTITY_CHECK_IN` event;
- on Bob entry, a `CHECK_OUT` message is built and associated with "will message", containing all the information about Bob, including its properties;
- Bob changes its properties in "tags = [basket]" without reconnection, so the `CHECK_OUT` associated with the "will message" does not change, becoming inconsistent (it's still containing old properties);

- Bob disconnects from the network: Alice will see an `ENTITY_CHECK_OUT` related to Bob due to the inconsistent `CHECK_OUT`, but Bob was not in the group Alice is interested in anymore.

Then a reconnection is needed to maintain consistency, but a new problem arises: we don't want that the temporary disconnection due to a properties update was seen by other entities as an `ENTITY_CHECK_OUT` event. For this reason, we introduced the boolean "valid" field in `CHECK_OUT` message (see section 4.2.6): when the valid bit is set to `true`, `GroupEntityManager` interprets the message as an `ENTITY_CHECK_OUT` event, otherwise it understands that it is a "fake" disconnection for a properties update.
So, let's assume `GroupEntityManager` receives a `CHECK_OUT` message with valid equal to `false`, what is it supposed to do? In this case, we decided to use a `Timer` (that we call **check_out_timer**, see test 6 in section 5.4 to see how we tuned this parameter), which is set on `CHECK_OUT` message receiving, and waits for a corresponding `PROPERTIES_UPDATE` message: a reception of the latter will stop the timer; otherwise, an expiration of **check_out_timer** is seen by `GroupEntityManager` as an `ENTITY_CHECK_OUT` event, assuming that the updating `Entity` disconnected, but had failed to reconnect with new properties, as it had not sent the `PROPERTIES_UPDATE` message.

To summarise, `GroupEntityManager` offers a method to update properties: this method disconnects the `Entity` from the network sending a `CHECK_OUT` message with valid field set to `false`, creates a new `Entity` with same `entityID` and new properties, connects it to the network with a consistent "will message" and sends the corresponding `PROPERTIES_UPDATE` message.
When a `CHECK_OUT` with valid = `false` is received, `GroupEntityManager` calls the `evaluateCheckOut` method, that sets the **check_out_timer** and the event to be fired in case the latter expires (`ENTITY_CHECK_OUT`). If the corresponding `PROPERTIES_UPDATE` is received in time, **check_out_timer** is stopped and the method `evaluateGroup` is called, in order to fire potential group events such as `ENTITY_GROUP_JOIN` or `ENTITY_GROUP_LEAVE`.

Proximity computation

In section 3.10 we described at high-level how proximity between entities is computed; here, we explain how `GroupEntityManager` implements this functionality.

First of all, we have to distinguish between a device-to-beacon proximity and a device-to-device proximity. The former is computed after every BLE scan or `PROX_BEACONS` message received: for every beacon detected/in the message, the `evaluateProximity` method is called, and if some proximity subscription is matched, the associated event is fired using the `DistanceRange` obtained by scan/message.

The latter is computed every time a `PROX_BEACONS` message is received: `GroupEntityManager` computes the intersection between last seen beacons and the ones of the message, then calls the `evaluateProximityD2D` method passing the obtained beacon-in-common set. Up to this point, if there is at least one beacon in common, the aforementioned method can compute the distance between the devices; to calculate the best distance, we made some important assumptions:

- if both devices are "far" with respect to the common beacon (their `DistanceRange` from it is greater or equal than `NEAR`), we cannot say anything about their distance, so we can only state that they are at distance `SAME_BEACON` (see figure 4.4).
- if at least one device is "near" to the common beacon (`DistanceRange` less or equal than `NEXT_TO`), we can approximate the device position to the beacon position: in this way, we can assume that the distance between devices is the distance between the furthest device and the beacon. (see figure 4.5).

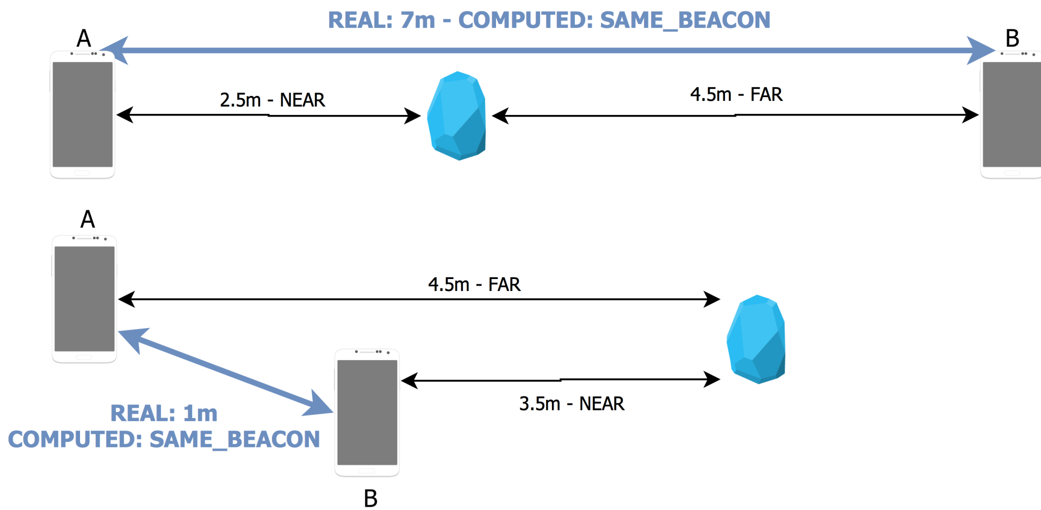

Figure 4.4: Topology example: devices "far" from beacon

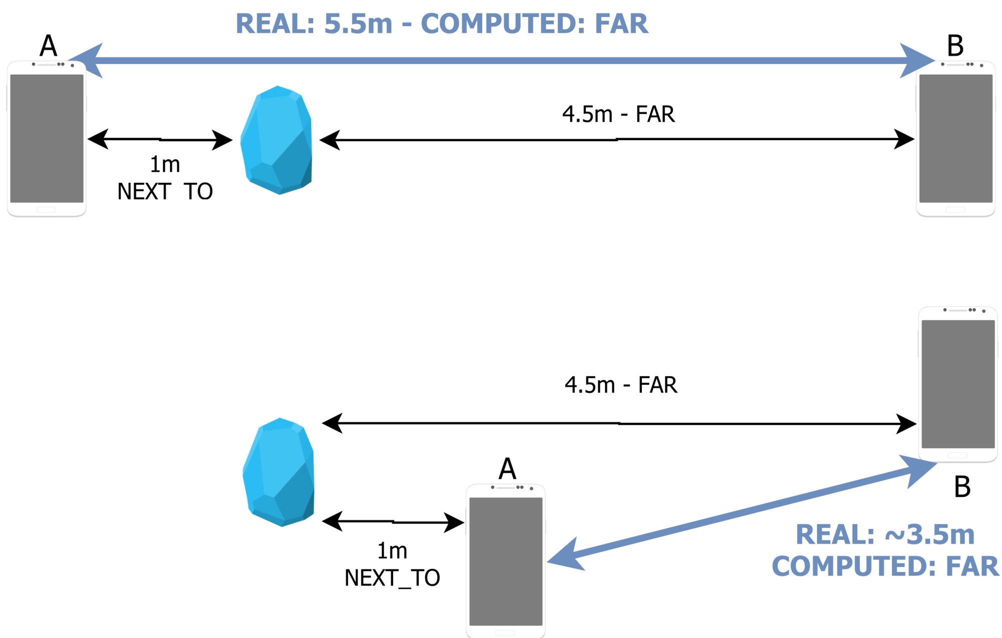

Figure 4.5: Topology example: devices "near" to beacon

Starting from these assumptions, the method examines every common beacon and computes the best distance, i.e. the smallest `DistanceRange` obtained. In the end, it sends a `PROXIMITY_UPDATE` message to inform the other `Entity` that they are in proximity with the `DistanceRange` computed. This message dispatch is not immediate; let's assume that the `Entity` sending

a PROX_BEACONS message is in proximity with respect N devices: this means that all these devices are going to reply with a PROXIMITY_UPDATE, causing the first Entity to receive N messages simultaneously. For this reason, to avoid getting the device (and the network) stuck when N is a great number, we decided to introduce a random delay between 0 and 3000ms that every Entity has to wait before sending a PROXIMITY_UPDATE message, in order to distribute better messages in time.

When instead a PROXIMITY_UPDATE is received, the evaluateProximity method is called: if there is some proximity subscription matching, the corresponding event is fired.

Geofence computation

Geofence events differ from simply compute the distance as for proximity events, because we are interested in actual distance but also in the previous (if exists), in order to understand if we are in case of entry, exit, or either of them. For this reason, we have to maintain in memory a "state" of every geofence created through a geofence subscription, in particular a List of entities which are inside the geofence.
Geofence computation happens every time a PROX_BEACONS or a PROXIMITY_UPDATE message is received: in the first case GroupEntityManager calls the evaluateGeofence method, that checks if there is some matching geofence subscription involving selfEntity and one or more beacons; in the second case, instead, GroupEntityManager calls the evaluateGeofenceD2D that checks for geofence subscriptions matching with the two devices involved. Both these methods call the evaluateGeofenceEventFiring method for every matching Subscription passing it, the geofence Entity and the DistanceRange between entities: this method verifies the List of entities inside that geofence, compares the DistanceRange parameter with the one of geofence, and on the basis of that decides if a ENTITY_GEOFENCE_ENTRY or a

`ENTITY_GEOFENCE_EXIT` has to be fired.

Up to this point, two important observations has to be underlined:

- the computational effort is higher with respect to group and proximity events.
- as said before, we have to deal with memory occupation.

Starting from these two observations, we decided to set two boundaries for the programmer: a maximum number of geofence subscriptions fixed to 5, and also a maximum number of devices that can be stored in memory for every `Subscription`, which is set to 50 (see test 5 in section 5.4 for details about the choice of these values).

Synchronous request: Entities in proximity

In section 3.10 we described how is performed the synchronous request, here we explain how we implemented it at a technical level.

A synchronous request starts with a call to the corresponding API method, that:

- creates a new topic with a unique name, and subscribes selfEntity to it. This topic will be used by other entities in order to notify only to selfEntity their proximity.
- sends on topic `BROADCAST` a `SYNC_REQ` message, containing the `DistanceRange` and the `Group` requested, and the aforementioned topic.
- sets a `Timer` (we call it **sync_req_timer**), at the end of the which it unsubscribes from the topic, stopping the receiving of proximity notifications from other entities: thus, the **sync_req_timer** is very important because it represents the time an `Entity` has to wait to ideally have a complete list of all the entities in proximity. At the moment, this `Timer` is set to 10s, as explained in section 5.4 (test 7).

When a SYNC_REQ message is received, GroupEntityManager calls the handleSyncReq method, that takes as parameter all the fields of the message and on the basis of that it computes if selfEntity is in proximity with the requesting Entity, and if belongs to the Group required. If so, a SYNC_RESP message is sent on the topic passed with the SYNC_REQ one, in order to notify the requesting Entity of their proximity. Also in this case, as for PROXIMITY_UPDATE message, every device sends the SYNC_RESP after a random delay between 0 and 3000ms, in order to distribute better messages in time (see test 7 in section 5.4 for more details).

It's important to notice that we decided to store all the business logic needed to compute proximity on the devices receiving the SYNC_REQ: in this way, we reduced the number of messages (if we let the requesting Entity to compute proximity, all entities in the network had to send a message, while now only "interesting" entities send a SYNC_RESP), and also we distributed computational effort as much as possible among devices.

4.3 Technology Interface

This layer defines the technological abstraction used by GroupEntityManager and acts as manager for lower layer. The main class this layer is composed of is TechnologyManager that implements PassiveTechnologyListener interface: it contains all business logic related to lower level and knows the GroupEntityManager expected results format.

Figure 4.6 shows a simplified UML class diagram of the layer.

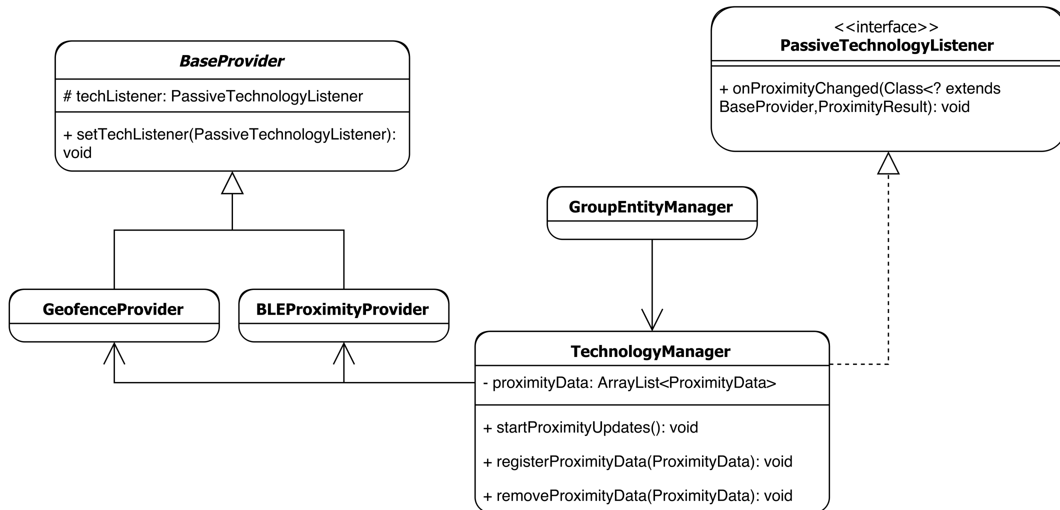

Figure 4.6: Simplified class diagram for Technology Interface layer

4.3.1 Utility classes and interfaces

This section describes classes and interfaces used to interact with `GroupEntityManager` and with Technology level.

ProximityListener

This interface defines callbacks the `GroupEntityManager` uses to be notified about proximity and POI events.
In particular, methods defined are:

- `onProximityChanged`: this method is called by `TechnologyManager` whenever a proximity information is available from providers. The `ProximityResult` object containing the technology information is passed as parameter.

- `onEnterGeofenceArea`: this method is called whenever a POI entry event is fired: the related POI is implicitly known because of the one-to-one relation between `ProximityData` and `POI` objects.

- `onExitGeofenceArea`: this method is the opposite of the previous one, thus is called when a POI exit event is fired.

ProximityData

This class defines a DAO object used by `GroupEntityManager` to instruct `TechnologyManager` about interesting `POIs` to monitor, defining all its data briefly described here:

- ID: an identifier (containing `POI` object's name) useful to notify correct `POI`-related events to client app (see section 4.3.2).
- Geofence data: `POI` center geolocation data (maintained as latitude and longitude values) and geofence radius value defined by client application.
- Geofence status: a value containing the current status of the device with respect to the specific geofence.
- Proximity technology data: the set of parameters used by lower levels to properly instruct proximity technology providers; for example "switch" values indicating if a certain provider has to be used for the specific `POI` and others providers specific data.
- Proximity listener: this field contains the callback implemented by `GroupEntityManager` to be executed on proximity events firing.

ProximityResult

It is the opposite of `ProximityData` class because it allows `TechnologyManager` to obtain results from proximity providers. In particular each `ProximityResult` composes of the following fields:

- ID: it is the `ProximityData` identifier used by `TechnologyManager` to map results contained with the `ProximityData` they are about.
- Technology related data: set of fields containing technology specific proximity information such as "switch" to understand available data and update type, together with proximity values returned by providers.

4.3.2 TechnologyManager

As stated before `TechnologyManager` acts as "glue" between framework business logic and technology level. This subsection aims to describe its main tasks and working principles.
`TechnologyManager` manages `ProximityProvider` objects at lower layer instructing them on new requests from the `GroupEntityManager`, changing their state when needed (for example initialize, start or stop them) and listening for geofence events.
In following sections we describe operations executed in all component lifecycle phases, motivating them and describing internal "tools".

Initial setup

This phase starts when `GroupEntityManager` instantiates the `TechnologyManager`. First of all an instance of each `ProximityProvider` implementation available is created and initialized (calling its `init` method), so that they can complete their own setup phase. Secondly a "worker thread" is started (using Android `HandlerThread` facilities) and waits for tasks submission to complete on behalf of the `TechnologyManager` itself: this prevents it from being stuck in long-running operations and unable to deal with upcoming requests.
Lastly, in this phase it registers itself as `WakefulBroadcastReceiver`, so that POI events can be handled properly, even when the device is in standby.

New proximity request and results

When `GroupEntityManager` passes a new `ProximityData` object that contains information seen in section 4.3.1, the `TechnologyManager` passes it to proper proximity providers in order to let them make all operations needed (see section 4.4). This operation is executed by the aforementioned worker thread that could be busy when this request is submitted: to confirm its handling by proximity providers a completion callback is provided by the

GroupEntityManager.
Whenever a provider has a “fresh” proximity information to return, it calls the onProximityChanged method provided by TechnologyManager: the latter identifies the ProximityData interested and calls the corresponding ProximityListener callback, making data available to the GroupEntityManager.

Remove proximity request

If the GroupEntityManager wants proximity providers to stop returning their results relative to a certain request, simply asks for it to the TechnologyManager: the latter, on the basis of the ProximityData object to remove, notifies providers about the request and can perform specific operations for each of them, such as instruct it to stop or even to release all used resources.

POI events

For each ProximityData submitted, the TechnologyManager creates a corresponding geofence (see section 4.4.2) and monitors entry and exit events.
As reported in section 4.2.4, the Android Broadcast mechanism is adopted to receive and handle these events: the TechnologyManager, upon receiving the event, retrieves POI object related, identifies the corresponding ProximityData and calls the proper ProximityListener callback (depending on the event type) notifying the GroupEntityManager. Lastly it executes providers-specific operations such as, in case of POI entry event, submit ProximityData to the provider returning BLE beacons information.

Inter-layer communications

As reported in previous section, TechnologyManager implements PassiveTechnologyListener interface that defines the only method onProximityChanged: this method is called by providers (see section

4.4) and takes as parameters the specific provider's class performing the call and a `ProximityResult` object containing last proximity values computed by the provider itself.
Figure 4.7 shows a schema of `PassiveTechnologyListener` calls by different `ProximityProviders`.

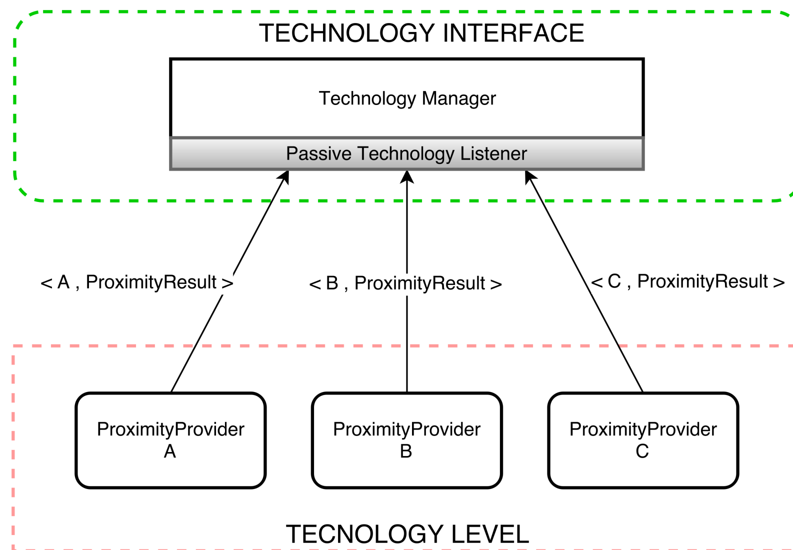

Figure 4.7: Providers passing data through `PassiveTechnologyListener`

Similarly, `TechnologyManager` passes `ProximityData` objects, when proper, to proximity providers through the `ProximityProvider` interface implemented by them. Figure 4.8 schematizes this scenario.

4.4 Technology level

This level is composed of different objects each one providing low level functionalities such as setup a geofence and provide proximity information using available sensors and solutions, together with technology specific control tasks as turn on/off used sensors when needed.
In particular these objects extend `BaseProvider` class and they have to implement the `ProximityProvider` interface (inherited by `BaseProvider`). This

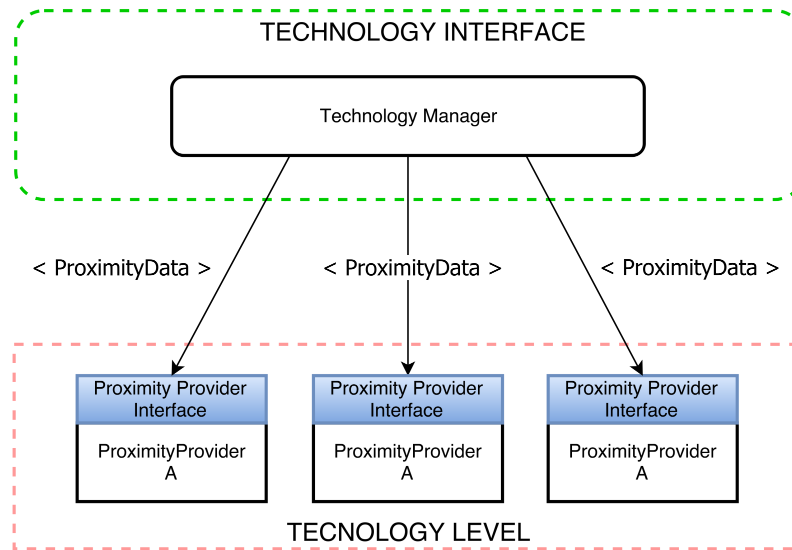

Figure 4.8: `TechnologyManager` passing requests to providers

interface defines following methods used by `TechnologyManager` as shown in Figure 4.8:

- `init`: this method is called to enable the proximity provider and to make it initialize all needed objects or services.
- `onNewProximityRequest` and `onRemoveProximityRequest`: this methods are used to submit tasks to providers in order to collect proximity data and pass results back to `TechnologyManager`, or (in case of `onRemoveProximityRequest`) to ask the provider to remove the task previously submitted.
- `start`: this method is called to make the object start executing its tasks and providing proximity values on the basis of `ProximityData` passed.
- `stop` or `destroy`: this method is called when the proximity provider has to release all used resources because its tasks are no more needed.

Generally each provider keeps track of `ProximityData` objects submitted by `TechnologyManager` in order to build properly the corresponding `ProximityResult` to pass back to it. Following subsections describes detailed

behaviour of available providers.

Figure 4.9 shows a simplified UML class diagram of the level.

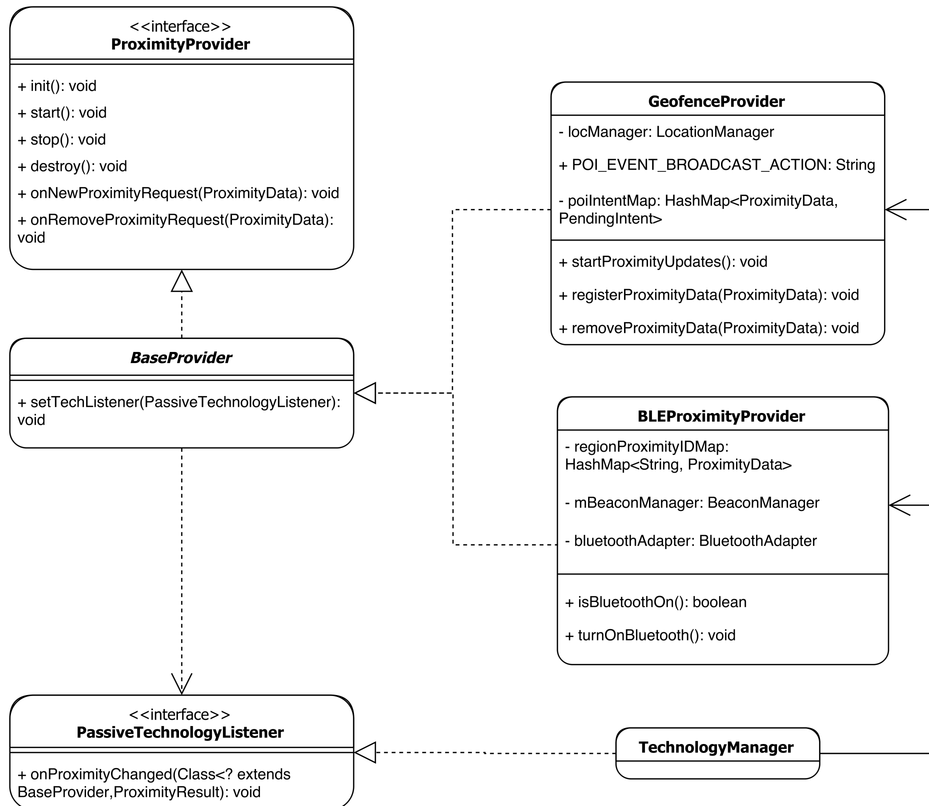

Figure 4.9: Technology level simplified class diagram

4.4.1 BLEProvider

This provider aims to return "proximity values" to `TechnologyManager` as `ProximityResult` objects suitable for upper levels on the basis of `ProximityData` requests submitted to it. `BLEProvider` builds these results using device's Bluetooth Low Energy capabilities: in fact each `ProximityResult` object returned contains the list of all BLE beacons detected in device surroundings.

The framework relies on AltBeacon[23] library to deal with beacons broadcasted data and BLE scanning tasks, instead of build all from scratch.

In the initial phase several tasks are completed: first of all, if needed, the

Bluetooth is turned on, then the AltBeacon service is started and configured to scan for iBeacon advertised packets, `BLEProvider` binds itself as a consumer for library results and scanning intervals are set. In particular a result is returned to `TechnologyManager` approximately every 10 seconds, splitted in 7 seconds of idle state and 2.5 seconds of BLE scanning state: this time division led to good outcomes in terms of framework reactivity and battery consumption.
Whenever a `ProximityData` is passed to `BLEProvider`, it checks for the requested iBeacon UUID to monitor and properly instruct AltBeacon library to monitor beacons advertising that UUID in device surroundings. When a "matching beacon" is detected the provider switches to ranging mode, making the library scan for beacons and return a distance estimate for each of them: depending on this (highly unreliable) estimate a framework-defined distance is computed in order to mitigate proximity errors (see section 3.5.3).
Obviously in case of a `ProximityData` remove request, the provider instructs AltBeacon library to stop scanning for beacons matching the related UUID and, if requested, to stop related service and unbind from it. In order to keep a behaviour as safe as possible, when upper level asks the service to stop, the Bluetooth is not turned off: this choice was made to allow more than one framework instances and other Bluetooth-powered services run simultaneously on a device.

4.4.2 GeofenceProvider

The aim of this `ProximityProvider` is to setup geofences identifying `POIs`. Currently it exploits Android `LocationService` proximity alerts for this purpose: this choice was made to limit external libraries used by the framework, keeping it as lightweight as possible and providing to the programmer a "well-known" behaviour.
When `TechnologyManager` passes it a new request (a `ProximityData` object) it sets the related geofence (keeping a reference to it) and the system starts

its monitoring; similarly when the geofence is no more needed it is removed from the system too, both in case of stop command received or a particular `POI` monitoring stop.
As explained in section 4.3.2 the `TechnologyManager` handles system events fired in case of entry or exit events.

4.5 Security layer

This layer controls information sent out through framework messages on to the network. Its aim is to limit, on behalf of client application, the proximity and group information received by peers on the network about the device itself. Furthermore the client app can use SSL encryption in order to avoid information sniffing over the network.
The user can define wanted security policies through a `SecurityManager` object, which is the object used to perform inter-layer communication between the Security layer and other components (`GroupEntityManager`, application using ADPF, and Network Layer); the possible choices are:

- `useSSL`: this option forces the framework to encrypt outgoing messages and allows it to decrypt messages received from peers; in this way only users authenticated through the broker can receive network messages correctly.

- `groupAdvertiseEnabled`: this option, if set, allows the framework to send out group related data (i.e. `PROPERTIES_UPDATE` messages) to peers in order to make them aware of group events.

- `proximityAdvertiseEnabled`: this option, if set, allows the framework to send out proximity related messages such as `PROX_BEACONS` and `PROXIMITY_UPDATE` messages and make peers aware of proximity information related to the device.

The `SecurityManager` object mentioned above has to be passed as argument during `GroupEntityManager` object creation. This seemed us as a limitation because in some scenarios a device could need to be sometimes "invisible" to peers: this consideration motivated us to include an invisible mode called Ghost Mode, which high-level description is reported in section 3.10. From a technical point of view, enabling Ghost Mode means sending a `CHECK_OUT` message and setting to false both options `groupAdvertiseEnabled` and `proximityAdvertiseEnabled`, while disabling it means obviously sending a `CHECK_IN` message and restoring the previous state, i.e. `SecurityConfiguration` starting settings.
Notice that every time user switches from visible to Ghost Mode and vice versa, a reconnection is performed in order to set/unset the will message: in fact, in case Ghost Mode is enabled and a network error occurs, the will message has not to be sent in order to preserve consistency (the `CHECK_OUT` message was already sent when Ghost Mode was enabled), while when Ghost Mode is disabled the will message has to be restored.

Figure 4.10 shows the UML class diagram of the Security Layer.

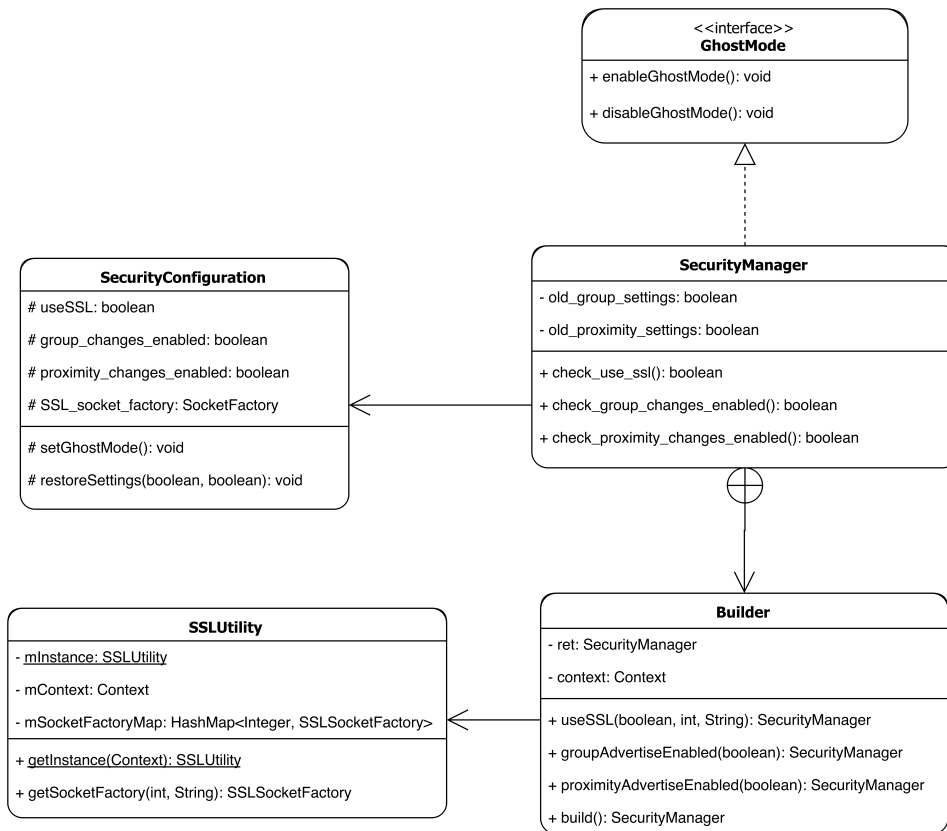

Figure 4.10: Security Layer class diagram

4.6 Network layer

This layer aims to hide to the `GroupEntityManager` the network details such as send or receive messages or connection process.
To fulfill requirements reported in sections 3.3 and 3.4 we adopted a communication protocol as lightweight as possible and with publish-subscribe functionalities: this helped us to reduce the overhead of the distributed architecture chosen, limiting the number of messages each peer has to deal with and the overall network overhead.
In the released version of the framework we used MQTT protocol to exchange messages between peers and the Paho library[24] as MQTT client. Nevertheless an ad-hoc object have been used as "interface object" between `GroupEntityManager` and the Paho client provided by the library: this choice

gave framework developers the opportunity to easily change underneath network protocol with another one implementing MQTT similar functionalities (first of all the publish-subscribe one) decreasing ADPF-powered apps deployment constraints.
In the released version the main object belonging to this layer is the `MQTTPahoClient` object that performs MQTT protocol operations on behalf of `GroupEntityManager`. Other classes and interfaces this layer defines are:

- `MessageCallback`: this interface is implemented by an ad-hoc object contained in the `GroupEntityManager` and contains the only `onMessageReceived` method, called by `MQTTPahoClient` when a message is received by whatever peer.
- `MessageTopic`: this class is an `enum` class containing topics used in network communications.
- `ConnectionStateCallback`: this interface can be implemented by the client application in order to be notified when the connection to peers network is completed or when a connection lost occurs.

4.6.1 MQTT

The network protocol used in our first release is MQ Telemetry Transport (MQTT)[25] in its version 3.1.1. This protocol has been developed for IoT world, so its focus is to be as lightweight as possible in terms of memory footprint and very power efficient in message exchange tasks. It offers a publish/subscribe messaging pattern with reliability features and some degree of assurance of delivery through messages acknowledgements: in particular a quality of service (QoS) can be set for each message. Possible QoS are:

1. *QoS 0 - at most once*
   Using this QoS a message won't be acknowledged by the receiver or stored and redelivered by the sender, providing a kind of "fire and forget" message delivery.

2. *QoS 1 - at least once*
   In this case MQTT guarantees that the message will be received at least one time by each receiver, but it could be delivered more than once: the sender stores the sent message until it received related acknowledgement from the broker.

3. *QoS 2 - exactly once*
   The last case ensures message delivery exactly once for each client exploiting a sort of double acknowledgement from the broker. Obviously in this case there's a non-neglectable protocol overhead respect to previous QoS levels.

An important feature provided by MQTT protocol is the **Last Will and Testament** (or **will message**): this feature is set at connection time, when the MQTT client C can instruct the MQTT broker to send a defined message on a certain topic (MQTT offers a pub-sub pattern) upon the broker itself detects C's unwanted disconnection (for example due to a network failure or C's system crash). ADPF exploits this feature to send proper `CHECK_OUT` messages on unwanted clients disconnection, keeping a consistent framework network state (4.2.7). This feature is achieved through a proper "keep alive" mechanism at protocol level.
We decided to adopt this protocol for communication purposes because of aforementioned features matching our requirements (suitable for mobile scenarios faced by our framework), lots of open source clients are implemented in most important programming languages and several ready to use open source brokers are available: this characteristics contribute to reduce deployment costs and constraints in whatever environment, as stated in chapter 3.

Chapter 5

Results and evaluation

In this section we discuss the experimental part of the work, that means the tests we realized and the obtained results.
The main purposes of evaluation were fundamentally two: first of all the analysis of the library performance, in terms of network availability, latency and battery consumption, increasing the network utilisation and the computational effort of the framework; secondly, strictly tied to the first, the parameters tuning that permits a good operational behaviour of the system also under stressed and bad conditions.

Increasing the network utilisation basically means increase the number of exchanged messages, or in other words, augment the number of entities (in particular, Android devices) involved. To do that, we implemented a Java application that permits to create virtual devices and virtual beacons; using this application we were able to simulate the behaviour of a real environment with a large number of devices, and so a huge number of exchanged messages. Increasing the library computational effort, instead, can be easily realised augmenting the number of subscriptions: in this way, the framework has to inspect a greater number of messages, and in the same time executes a greater number of control cycles (basically, one iteration for each subscription).

To realise the tests, first of all we needed to collect and permanently store data, in order to analyse them later. For this reason, we decided to log the information we need during the execution of a test application that uses our library, store these log messages in a database and, afterwards, perform queries on collected data in order to get application-specific statistics we want. In particular, we decided to log:

- received messages
- sent messages
- triggered events

5.1 Test and tuning parameters

In this section, we define the parameters used for testing, and the tuned ones clarifying their meaning.

5.1.1 Test parameters

- **routing time**, that is the time interval between a message is sent and received. Thanks to this parameter, we could analyse the network performance increasing the number of messages.
- **dispatching time**, namely the time between a message is received and the corresponding event is fired. With this parameter, we could observe the library performance increasing the computational effort (i.e. increasing the number of subscriptions).

5.1.2 Tuned parameters

- **check_out_timer**, that is the time a device who received a `CHECK_OUT false` message waits before firing an `ENTITY_CHECK_OUT` event. The event is fired because the device does not receive a `PROPERTIES_UPDATE`

message, that means the sender failed to reconnect after updating its properties.

- **sync_req_timer**, namely the time a device waits for `SYNC_RESP` messages after it performed a `SYNC_REQ`.
- **geofence_subscription_limit**, which is the maximum number of geofence subscriptions a user can register.
- **geofence_stored_entities_limit**, that represents the maximum number of entities the library can store in its main memory for each geofence subscription.

5.2 Experimental setup

In order to collect data for the analysis, we used the following tools (see section 5.3 for an in depth review of each of them):

- a LoggerService, which is an Android IntentService that logs all the information on a file in the device memory, and once the execution is terminated, it flushes the file to a TcpServer.
- a TcpServer, that receives the information from the LoggerService and store them in a MySQL database.
- a MySQL database to store and analyse collected data.
- a NtpServer to synchronize all the devices before the execution.
- a Java application ("TestClient") which simulates an environment with a number of entities (devices and beacons) at will.

5.2.1 Log messages

As written before, we chose to log in three different situations:

- when a message is received
- when a message is sent
- when an event is fired

Every log message consists of a JSON message: in this way it could be easily parsed and stored in the MySQL database.
Also, we stated that we needed a binding between a received message and the relative event fired, in order to compute the dispatching time, and between message sent and received, to calculate the routing time. For this reason, we introduced a new field, named `log_id`, in messages (and obviously also in log messages) which is added by the sender, and it is a combination of senderID and sending timestamp. This `log_id` is used as an identifier by the receiver which extracts the field from the received message and uses it to log the reception and eventually the linked fired event. Let's clarify with a toy example:

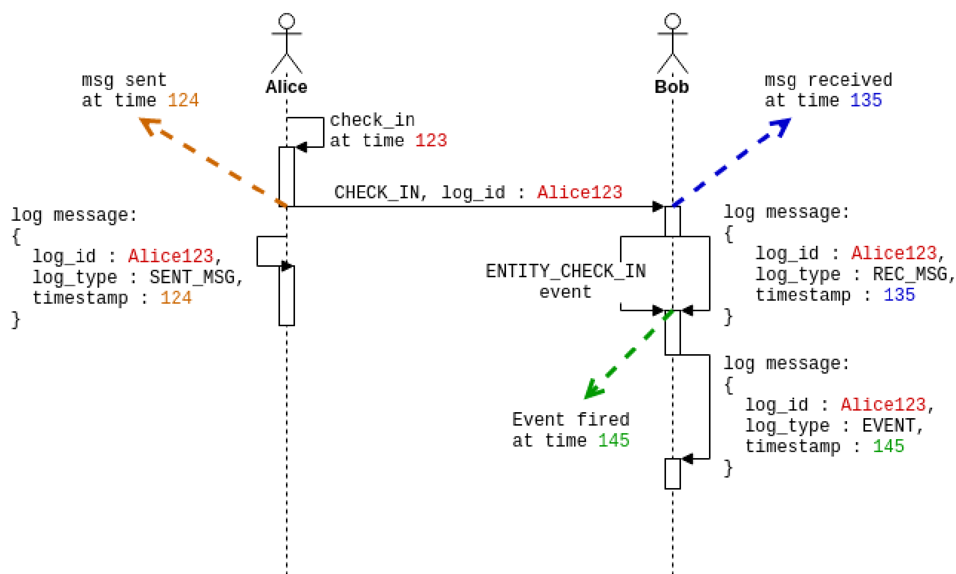

Figure 5.1: Toy example - Logging

- Alice sends a `CHECK_IN` message at timestamp `123`, adding the field `log_id : Alice123`, and logs a new log message with the same `log_id`;

- Bob receives Alice's message: he retrieves from it the `log_id`, and logs a new log message with that `log_id`;
- Bob was subscribed to Alice group: when the `ENTITY_CHECK_IN` event is fired, Bob logs a new log message with the same `log_id`.

In this way, log messages are bound by this `log_id`, so routing time and dispatching time can be easily computed.

REC_MSG

This is the log message type that represents received messages. Almost every REC_MSG log has this structure:

```
{
  "log_id":"logId",
  "sender":"senderID",
  "entity_type":"DEVICE",
  "log_type":"REC_MSG",
  "event_type":"CHECK_IN",
  "timestamp":144123123
}
```

Listing 5.1: REC_MSG log

SENT_MSG

This is the log message type that represents sent messages. Almost every `SENT_MSG` log has this structure:

```
{
  "log_id":"logId",
  "sender":"senderID",
  "entity_type":"DEVICE",
  "log_type":"SENT_MSG",
  "event_type":"PROXIMITY_UPDATE",
  "timestamp":144123123
```

```
}
```

Listing 5.2: SENT_MSG log

EVENT

This is the log message type that represents fired events. Almost every `EVENT` log has this structure:

```
{
  "log_id" : "logId",
  "sender": "senderId",
  "entity_type" : "DEVICE",
  "log_type": "EVENT",
  "event_type": "ENTITY_GROUP_JOIN",
  "status": {
    "proximity_sub_count" : 5,
    "group_sub_count" : 3,
    "geofence_sub_count" : 0
  },
  "timestamp": 144123123
}
```

Listing 5.3: EVENT log

Special cases

1. **SYNC_REQ and SYNC_RESP**

   When a `SYNC_REQ` message is sent, in the associated log message is added a field `topic : responseTopic`, and the same thing is done every time a `SYNC_RESP` is sent: this field is used as identifier, opposite to others where the identifier was the `log_id`. In this way, we were able to identify the last `SYNC_RESP` message received after a `SYNC_REQ` request, and we could compute the maximum waiting time between the request and the response from all the actors involved.

2. **CHECK_OUT not valid and PROPERTIES_UPDATE**
   Since there is no way to bind with `log_id` a `CHECK_OUT false` and the consequent `PROPERTIES_UPDATE` (this is due to the fact that the former is sent before disconnection, while the latter is sent after reconnection, so the sender cannot maintain the state, and in particular the `log_id`), in this case the `log_id` is not a combination of senderID and timestamp, but only the senderID. In fact, an important restriction of properties update is that it is not possible to change the entityId, but only the properties. At the same time, it is impossible to receive two consequents `CHECK_OUT false` from the same entity, without a `PROPERTIES_UPDATE` between them. In this way, we were able to bind `CHECK_OUT false` and `PROPERTIES_UPDATE` messages at log level, then we could calculate time interval between the reception of them, and we could estimate the value of the waiting timer before firing the `ENTITY_CHECK_OUT` event.

5.3 Testing tools

5.3.1 LoggerService

The LoggerService is an IntentService we used for logging information in a stable way, since the Android Log class only permits runtime logging making it is useful for debugging but not for the evaluation. We opted for a service offered by the Labgoo project[26] "android-logstash-logger", that can acts in two modes: active or silent. In active mode, it flushes every log directly to a server, while in silent mode it stores all the logs in a file in local memory, and flushes this file to the server only when the mode is changed to active. We decided to start the service in silent mode, and switch to active in the end of the data collection in order to avoid impacts on device network performance.

Logstash logger flushes the results to server using UDP protocol: since we realised that we were loosing a non-negligible number of packets, we modi-

fied the logger in a way such that the flush being executed through a TCP connection exploiting its connection-oriented features.

5.3.2 TcpServer

We implemented a TcpServer to receive log files from the LoggerService, and store log messages in a MySQL database. TcpServer, on creation, opens a connection with the database and a ServerSocket to accept socket connections, and waits for incoming TCP connection requests. Every time the LoggerService opens a new Socket and establishes a new TCP connection with the TcpServer, the latter starts a new Thread, in which it receives log messages, and stores them in the database. The storing is done upon the parsing of the message, through which the TcpServer can understand the message type and then the right MySQL database table for it.

5.3.3 MySQL database

To permanently store log messages, we had to use a database. We opted for MySQL, so for a RDBMS database, because collected log messages had to be "merged" to obtain desired results, so we needed to apply Join operations to data. As join operations were not on primary key, and the size of every table was in the order of thousands of entries, we had to use indexes to improve performances.

We identified three "starting" tables, one for each type of log message, that are: `REC_MSG` (for received message logs), `SENT_MSG` (for sent message logs) and `EVENT` (for fired event logs). Starting from them, we applied queries and created new tables containing the results. In particular, we obtained:

- `REC_EVENTS`, containing a join between `REC_MSG` and `EVENT`, used to compute the dispatching time of every couple:
  <`received_message, firing_event`>.

- `REC_SENT`, containing a join between `REC_MSG` and `SENT_MSG`, used to compute the routing time of every message.
- `PROP_UPD_TIMER`, containing the interval time between the reception of a `CHECK_OUT false` and the corresponding `PROPERTIES_UPDATE`. These results helped us to tune the **check_out_timer** parameter.
- `SYNC_REQ_TIMER`, containing the waiting time between a sent `SYNC_REQ` and the last corresponding `SYNC_RESP` received. In this way, we were able to calculate the **sync_req_timer** parameter.
- `AVG_ROUTING_TIME_COLLECTION`, `AVG_DISPATCHING_TIME_COLLECTION` and `AVG_MESSAGE_COUNT_COLLECTION`, containing respectively the average routing time varying the device number, the average dispatching time for every type (and number) of subscription, and the average count of messages changing the devices number.

5.3.4 Ntp Server

In order to synchronize devices used for testing purposes, so that we could compute routing time more precisely, we decided to use NTP (Network Time Protocol). To do that, we used a NtpServer running on the same host where Mosquitto broker was located: every time a client connects to the network, it calculates its delay with respect to NtpServer; then, every timestamp is logged adding this delay.

5.3.5 TestClient

In order to simulate virtual devices and beacons, we developed a Java program, called "TestClient" in the following, able to create multiple virtual device instances. In particular it's composed by a SWING GUI, where the user can add virtual beacons or virtual devices, and a BackgroundLogic that contains part of devices common business logic from the messages point of view.

As you can see in figure 5.2, TestClient is a simple Java SWING window with a toolbar, a menu and an empty pane, called ENV_PANE: when new devices or beacons are created they are deployed into this virtual 2D space which is a model for the real environment, where beacons and devices can move and change proximity between each others.

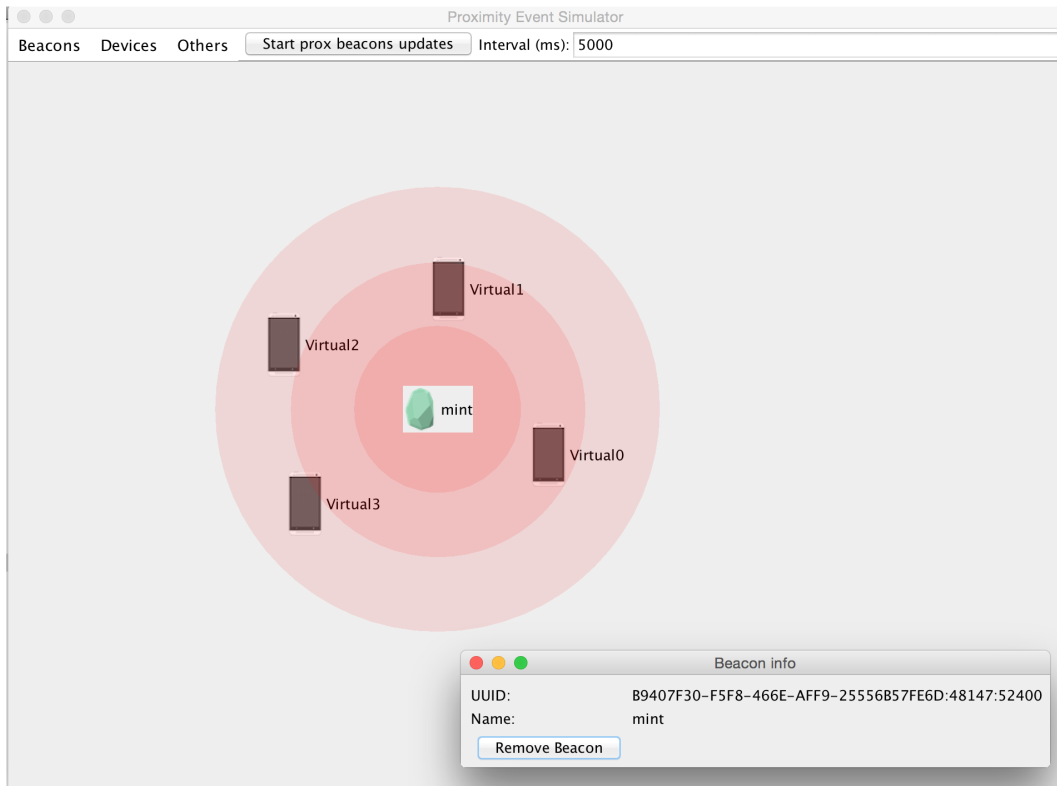

Figure 5.2: TestClient window

Main client's functionalities are:

- add a virtual beacon to the ENV_PANE. It is important to notice that a virtual beacon added with the same identifiers (UUID, major, minor) as a real beacon, simulates exactly the real one: in this way, we were able to have virtual and real devices "seeing" the same beacon.

- add a virtual device to the ENV_PANE. Once a device is deployed, it starts reacting to received messages (for example, if it receives a PROX_BEACONS message from another device, either virtual or real, it checks beacons surrounding itself and if needed sends a PROXIMITY_UPDATE message).

- move a device (or a beacon) on the `ENV_PANE` in order to change the proximity data broadcasted by devices.
- show virtual device information.
- show virtual beacons information.
- make devices broadcast `PROX_BEACONS` messages.

Business Logic

Device

A device instance is composed of all GUI stuffs we saw in previous section (icon, nearby beacons, etc.) and an MQTT client: this is an important aspect of the simulation because it would be possible simulate several beacons with the same MQTT client instance believing as a single client from the network point of view and as multiple clients by a framework point of view, potentially decreasing network impact; with our solution instead we have a single MQTT client instance for each device, making the simulation more similar to reality. Whenever a device is moved by the user on the `ENV_PANE` it checks for nearby beacons in order to refresh the surrounding beacons list it holds and (if requested by the user) broadcast periodically the `PROX_BEACONS` messages with the updated seen beacons: the periodic broadcast is managed by a Timer the device set on behalf of the background logic. The `PROX_BEACONS` messages sending interval is set by the user on the program's main window. In order to avoid all virtual devices sending `PROX_BEACONS` messages at the same time (altering simulation data due to message sending peaks) each device starts the update timer after a random delay (between 0 and 60 seconds): in this way, messages are almost uniformly distributed over a minute.

Background Logic

This module contains the code common to all devices that allows them to react to received messages performing necessary actions. The logic is similar to the real devices one: in particular a virtual device logs all received messages (as a real one) and reacts to some of them. A virtual device reacts to:

- `PROX_BEACONS` messages: as the real ones, it checks its `last_seen_beacons` and eventually sends a `PROXIMITY_UPDATE`.
- `SYNC_REQ` messages: it reacts exactly as a real device, furthermore in this case the background logic makes one deployed device to send all virtual beacons data wrapped in `SYNC_RESP` application messages: in this way the test Android app can easily subscribe to beacon entities update through their UUID.

Performance limits

During tests we needed to stress the network in order to find out the behaviour of the library. In these cases we asked the TestClient to simulate even around 100 devices at the same time: it means the TestClient had to keep opened the same number of connections simultaneously (see section 5.3.5) with often a consequent message loss.
This fact motivated us to use this value as “maximum devices number” reference during test phase.

5.4 Tests and results

In this section we describe tests performed and corresponding outcomes. For each test we report its target, application and environment setup (e.g. number of devices, subscriptions count etc.), operating methods and obtained results with a brief comment.

Test 1 - First stress test

Target

In this first experiment we wanted to observe the library beahaviour with a greater number of devices, as all the previous functional tests were executed with only 2-3 devices (the real ones we had).

Setup

- devices number: 20 (virtual: 18)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 1 (proximity subscription)
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 10s (one for each BLE scan)
- QoS: 2

Operating methods

We simply started real and virtual devices and left them exchange proximity messages (`PROX_BEACONS` and `PROXIMITY_UPDATE`) as in a real scenario. We make them detecting the same beacon: in this way, every device was in proximity with each others.

Results and explanations

In a short time we noticed that real devices, which had a proximity subscription, started lagging and dropping messages, making the test app impossible to use very soon.
We investigated possible causes on this performance degradation, and we stated that the main reason was the unsustainable number of messages each device had to deal with, since it had to manage every `PROXIMITY_UPDATE`

message sent by every device in the network. In fact, if N devices are in proximity and everyone sends a PROX_BEACONS message, every device has to deal with $N-1$ PROX_BEACONS messages and generates $N-1$ corresponding PROXIMITY_UPDATE. So, a single device receives $N*(N-1)$ PROXIMITY_UPDATE in a shot, approximately every 10 seconds. This means that a device has to manage about N^2 PROXIMITY_UPDATE, making the whole system unable to scale.

To overcome this issue, we identified a couple of solutions: first of all we tried to increase the PROX_BEACONS message sending interval, but the problem was only mitigated, because increasing a little the devices number over 20 made the app still impossible to use. Then, we thought to reduce the number of PROXIMITY_UPDATE messages limiting the flexibility of the library: in fact, in the first version (not the released one) the user could subscribe to a proximity group-to-group, meaning that he could receive proximity updates between entities belonging to two whatever defined groups (e.g. Alice could subscribe to proximity between Bob and Mark). Instead, the released version permits only a proximity subscription of type entity-to-group, where the entity can be only a beacon or the self entity (see section 3.5.2). In this way, a device can receive only PROXIMITY_UPDATE involving it, decreasing the number of these messages from about N^2 to about N.

In order to reduce battery consumption, we decided to mix the two aforementioned solutions: so, from this point on, each device receives only meaningful PROXIMITY_UPDATE messages, and sends PROX_BEACONS messages approximately once a minute (every 6 BLE scans).

Test 2 - Average dispatching time

Target

Inspect the dispatching time deviations varying the proximity subscriptions count.

Setup

- devices number: 2 (virtual: 0)
- beacons number: 1
- subscriptions number: 1, 2, 4, 8, 10 (proximity subscription)
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 2

Operating methods

For this test we used only two real devices: Alice and Bob. Alice subscribes to proximity updates between herself and Bob. Every 3 minutes, Alice increases the proximity subscription number, in a way that the matching one is kept as last.

Results and explanations

Subscription count	Average dispatching time [ms]
1	24
2	33
4	32
8	28
10	15

Table 5.1: Dispatching time table

As shown in table 5.1, increasing the subscription number has no effect on the dispatching time (for a limited number of subscription). Because of the expressiveness of filters, we can assume that in most of scenarios 10 subscriptions for each type grants to cover all user's needs.
Starting from this assumption, we can state that the dispatching time is almost constant and independent from the number of subscriptions.

Test 3 - Average routing time

Target

Measure the routing time deviations due to devices number increase, that means a greater number of messages in flight in the network.

Setup

- devices number: 20 (virtual: 18), 50 (virtual: 48), 80 (virtual 78), 100 (virtual: 98)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 0
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 2

Operating methods

We started real and virtual devices and left them exchange proximity messages (`PROX_BEACONS` and `PROXIMITY_UPDATE`) as in a real scenario; we make them detecting the same beacon: in this way, every device was in proximity with each others.
Every 3 minutes, we added more virtual devices to increase the number of

messages. It is important to notice that we used them to increase the network traffic, but we computed routing time only using data from real devices. This choice was motivated by the fact that virtual devices were running in the simulator on a Desktop computer, with performance generally greater than real ones, distorting the routing time values.

Results and explanations

Device number	Average routing time [ms]
20	522
50	592
80	844
100	3152

Table 5.2: Routing time table

As shown in table 5.2, the routing time raises smoothly with the number of devices (as expected) until this number is lower than 80. With greater devices number we noticed an evident performance worsening, due to two reasons: the network overhead caused by QoS equals to 2 (exactly once 4.6.1) and the TestClient bottleneck (see section 5.3.5).

Test 4 - Average dispatching time for group events

Target

In order to measure general application performance and fix it in case of problems, we investigated the average dispatching time needed for group events in real devices.

Setup

- devices number: 2 (virtual: 0)
- beacons number: 1

- subscriptions number: 1, 2, 4, 8, 10 (group subscription)
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 2

Operating methods

Having different number of subscriptions we repeated a simple test several times on both real devices available: we subscribed both of them to group events with a group filter matching all devices in the network. Then we make an increasing number of virtual devices entry and exit from the network: corresponding `ENTITY_CHECK_IN` and `ENTITY_CHECK_OUT` events where then fired on real devices and we inspected their behaviour.

Results and explanations

Subscriptions number	Average dispatching time [ms]
1	14
2	23
4	19
8	16
10	26

Table 5.3: Routing time table

Table 5.3 reports obtained results: as for average dispatching time in proximity events (see test 2 in this section) we noticed that the library behaviour was not influenced by the subscriptions number. This result was more than expected due to the fact that group messages are less than proximity messages and the lower computational effort needed to manage them.

Test 5 - Average dispatching time for geofence events and parameters tuning

Target

This test had a couple of targets: first of all measure the average dispatching time for `ENTITY_GEOFENCE_ENTRY` and `ENTITY_GEOFENCE_EXIT` events with a raising number of devices in the network and then choose meaningful values for **geofence_stored_entities_limit** and **geofence_subscription_limit** parameters (see section 5.1.2).

Setup

- devices number: 10 (virtual: 8), 20 (virtual: 18), 30 (virtual: 28), 50 (virtual: 48), 80 (virtual 78)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 1 (geofence subscription)
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 2

Operating methods

Having N devices in the network, we subscribed real devices to a geofence event centered on the only beacon in the environment such that subscriptions filters match all other devices: in this way we wanted to stress real device in order to analyze their behaviour with a large number of geofence events (either entry or exit) and with lots of entities stored in memory.
Initially we placed all virtual devices around the beacon in order to have a standard initial configuration. After a while (at time t_1) the beacon was moved as far as possible from all devices in the network and as a consequence all

PROX_BEACONS messages sent after t_1 generated a geofence exit event on real devices.
Then the beacon was moved back to its initial position (where all virtual devices could see it as "IMMEDIATE") and, as expected, one geofence entry event was generated on real devices for each virtual one.
The "one shot mode" we described here is needed in order to force applications running on real devices store all virtual device entities and stress application memory usage: in fact this happens in this case (and other few) only.
This sequence of events was repeated several times for a raising values of N parameter.

Results and explanations

Device number	Average dispatching time [ms]
10	34
20	12
30	15
50	21
80	10

Table 5.4: Average dispatching time for geofence events

Dispatching time values reported in table 5.4 evidence good performance also with 80 devices in the network all notifying the same event (either ENTITY_GEOFENCE_ENTRY and ENTITY_GEOFENCE_EXIT depending on virtual environment state), thanks also to stored entities deletion once each real device receives their ENTITY_GEOFENCE_EXIT event.
About memory usage we tried to infer Entity object's footprint in memory and guess values for **geofence_stored_entities_limit** and **geofence_subscription_limit** parameters bounded by memory-performance trade-off.

Our approximated calculations are reported in table 5.5:

Data	Memory Size [byte]
Entity object	16
String entityID	60
Enum DistanceRange	4
Enum EntityType	4
JSONObject properties	8000
Total size	**8084**

Table 5.5: Entity object memory schema

To make further calculation easier we decided to consider 8100 bytes as total size.
On the basis of this result we stated that good parameter values are 50 for **geofence_stored_entities_limit** and 5 for **geofence_subscription_limit**: in fact, due to the good expressiveness granted by filters, we think that most of real application scenarios can be covered, maintaining a reasonable memory footprint. In particular, with chosen values we have:

$$8100 * 50 * 5 = 2.025.000 bytes \sim 2MB$$

we think is an acceptable value for most Android smartphones.

N.B. The 2MB memory obtained before can be reached only in few and rare real scenarios, because of the "cleanup" of entities out of the geofence.

Test 6 - Properties update parameter tuning

Target

In this test we wanted to choose a value for the **check_out_timer** parameter: it is important because from this value may depend unwanted `ENTITY_CHECK_OUT` events because of a too short timeout.

Setup

- devices number: 20 (virtual: 18), 50 (virtual: 48), 80 (virtual 78)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 1 (group subscription)
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 2

Operating methods

This test was performed several times with a raising number of devices (as reported in above setup).
We started virtual devices and let them exchange `PROX_BEACONS` messages to generate network traffic. Then for each real device we measured time needed for the other real device to complete the properties update "actions sequence": in particular for each properties update event we measured time interval between `CHECK_OUT false` message and the corresponding `PROPERTIES_UPDATE` message.
We make each real device subscribe to group events relative to the other, such that each of them registers the corresponding check out timer upon `CHECK_OUT false` message is received and force it to log all necessary information.

Results and explanations

Device number	Time interval [ms]
20	1749
50	2269
80	1228

Table 5.6: Time intervals between `CHECK_OUT false` and `PROPERTIES_UPDATE`

For synthesis we reported in table 5.6 only the worst time intervals obtained for each number of devices.
Table results show the global worst case has an approximated 2.3 seconds time interval. This led us to state that a good value for **check_out_timer** parameter could be 5 seconds, but a safer choice of 10 seconds is recommended in case of unwanted side effects caused by aforementioned false positives.

Test 7 - sync_req_timer parameter tuning

Target

This test aims to choose a value for the **sync_req_timer** parameter: in fact this value has to be chosen such that the process is as fast as possible without any `SYNC_RESP` message skip.

Setup

- devices number: 20 (virtual: 18), 50 (virtual: 48), 80 (virtual 78)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 0
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 2

Operating methods

As in previous test we started virtual devices and let them generate network traffic through `PROX_BEACONS` and `PROXIMITY_UPDATE` messages. The test was repeated several times for each number of devices in the network to check the impact of network traffic over measured parameters.
Both two real devices we used performed several sync requests, using a filter

matching all devices in the network, and wait for all SYNC_RESP messages from real and virtual devices: for each of these requests we measured the time interval between the SYNC_REQ message and the last corresponding SYNC_RESP message received.

Results and explanations

During tests we noticed frequent messages loss due to the high rate SYNC_RESP messages were sent by all devices in the network upon a SYNC_RESP message was received.

Device number	Time interval [ms]	Avg nr. SYNC_RESP received
20	1187	19
50	3405	49
80	3023	73.2

Table 5.7: Time intervals between SYNC_REQ and SYNC_RESP - before fix

In table 5.7 we reported the average number of SYNC_RESP messages received in the “bad case” above described. As we can see with less than 50 devices all messages were received as exptected, while increasing the number of devices in the network led to some messages loss.
As a fix we tried to make devices send SYNC_RESP messages after a random delay in the range [0:3000] ms: results obtained with this fix are shown in table 5.8.

Device number	Time interval [ms]	Avg nr. SYNC_RESP received
20	1187	19
50	3405	49
80	5041	78.4

Table 5.8: Time intervals between SYNC_REQ and SYNC_RESP - after fix

As you can see we obtained a great improvement for the average message loss count and we guessed that the few `SYNC_RESP` messages still lost are due to TestClient performance limits (see section 5.3.5). We think that the remarkable improvement by the functional point of view motivates the worsening in time interval needed for request completion.
On the basis of obtained results we suggest to use a 10 seconds value for **sync_req_timer** parameter in case of high number of devices expected in the network: obviously the programmer could wait less than 10 seconds (e.g. 5 seconds could be a good value in case of small number of devices) but he has to be aware of a possible `SYNC_RESP` messages loss.

Test 8 - Average message count

Target

Measure the number of messages exchanged varying the devices number, and the `PROX_BEACONS` message sending interval.

Setup

- devices number: 20 (virtual: 18), 50 (virtual: 48), 80 (virtual 78), 100 (virtual: 98)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 0
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 30s, 60s, 90s
- QoS: 2

Operating methods

We started this test with the PROX_BEACONS message sending interval fixed to 60s. Then we began with 20 devices detecting the same beacon, and we let them exchange messages for about 3 minutes; we repeated the same experiment increasing the number of devices every 3 minutes.
Once we reached the limit of 100 devices, we restarted the test varying the PROX_BEACONS message sending interval, firstly to 30s (in this case with a maximum number of devices of 80, due to TestClient limits) and lastly to 90s.

Results and explanations

Device number	PROX_BEACONS message sending interval [s]	Msg_per_second
20	60	12
50	60	87
80	60	221
100	60	415
20	30	26
50	30	192
80	30	433
20	90	10
50	90	59
80	90	165
100	90	254

Table 5.9: Message count table

As shown in table 5.9, the message count is directly proportional to the PROX_BEACONS message sending interval (fixing the devices number), while the number of messages raises in a quadratic way with the number of devices (fixing the PROX_BEACONS message sending interval).
We tried to find a formula that describes this growth: assuming to have N devices in proximity, when a device sends a PROX_BEACONS message, this

message is received by $N-1$ devices, and each of them replies with a PROXIMITY_UPDATE. So, from the point of view of a single device, we have: 1 (PROX_BEACONS sent) + $(N-1)$ (PROX_BEACONS received by other devices) + $(N-1)$ (corresponding PROXIMITY_UPDATE messages) = $2N-1$ messages. Then, if we consider N devices, we are going to have $N*(2N-1)$ messages approximately every minute, assuming a PROX_BEACONS message sending interval of 60s.

At this point, we tried to compare the graph obtained by the formula (dividing it for 60, in a way to obtain messages per second) and the graph obtained by test results for PROX_BEACONS message sending interval of 60s. This is what we got:

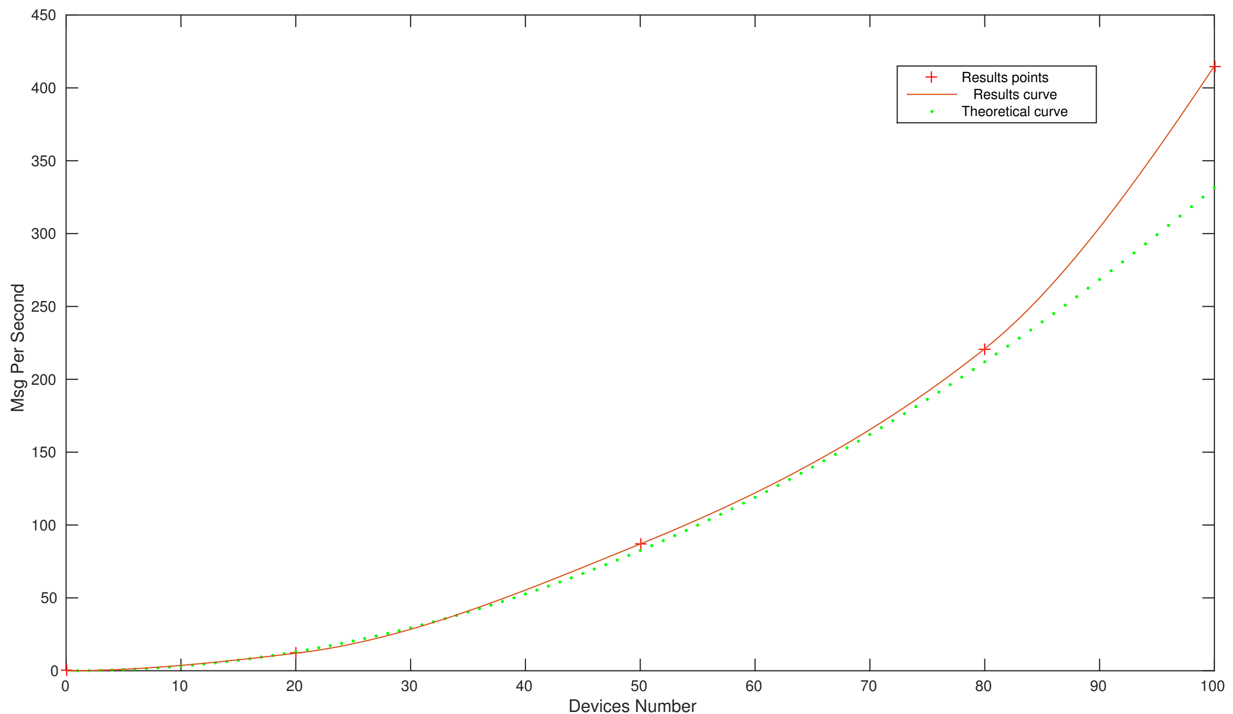

Figure 5.3: Average message count

As we can see in figure 5.3, the results respected the theoretical expectations. The deviations that increase with devices number growing are due to

the fact that in the theoretical formula we are not taking into account other messages, such as `CHECK_IN`, `CHECK_OUT` and group messages.
Eventually, from obtained results we could set `PROX_BEACONS` message sending interval at 60s, being a good trade off between scalability and performance.

Test 9 - Lost messages with QoS 0

Target

Observe how many packets are being lost with QoS equal to 0.

Setup

- devices number: 20 (virtual: 18), 50 (virtual: 48), 80 (virtual 78), 100 (virtual: 98)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 0
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 0

Operating methods

We started real and virtual devices and left them exchange proximity messages (`PROX_BEACONS` and `PROXIMITY_UPDATE`) as in a real scenario, using a QoS equal to 0; we make them detecting the same beacon: in this way, every device was in proximity with each others.
Every 3 minutes, we added more virtual devices to increase the number of messages, until the maximum number was reached (100 devices).

Results and explanations

Device number	Packet loss percentage
20	0 %
50	0 %
80	2 %
100	5 %

Table 5.10: Packet loss rate table

As shown in the table 5.10, with a small number of devices we had not messages lost, while increasing this number we started to loose some messages. Anyway, we could assert that the number of messages lost was acceptable.

Test 10 - Average routing time with QoS 0

Target

Measure the routing time deviations due to devices number increase, that means a greater number of messages in flight in the network, using a QoS equal to 0. Furthermore, we want to compare results obtained for routing time with QoS 2.

Setup

- devices number: 20 (virtual: 18), 50 (virtual: 48), 80 (virtual 78), 100 (virtual: 98)
- beacons number: 1 (the same for virtual and real ones)
- subscriptions number: 0
- BLE scan time: 10s
- `PROX_BEACONS` message sending interval: 60s
- QoS: 0

Operating methods

We started real and virtual devices and left them exchange proximity messages (`PROX_BEACONS` and `PROXIMITY_UPDATE`) as in a real scenario, using a QoS equal to 0; we make them detecting the same beacon: in this way, every device was in proximity with each others.
Every 3 minutes, we added more virtual devices to increase the number of messages. It is important to notice that we used them to increase the network traffic, but we computed routing time only using data from real devices. This choice was motivated by the fact that virtual devices were running in the simulator on a Desktop computer, with performance generally greater than real ones, distorting actual routing time values.

Results and explanations

Device number	Routing time (QoS 2) [ms]	Routing time (QoS 0) [ms]
20	522	580
50	592	1111
80	844	1107
100	3152	1261

Table 5.11: Routing time comparison table

As we can see in table 5.11, there is no difference in routing time using a small number of devices, comparing QoS 2 and QoS 0. But when the devices number starts growing, we can observe that using QoS equal to 0 the routing time increases more slowly. For this reason, we suggest to use QoS 0 only when it is expected to have a huge number of devices exchanging messages.

5.5 Conclusions

Thanks to performed tests reported in previous sections, we were able to point out some boundaries of our library both from scalability and performance

points of view.

About scalability we were expecting, since development and design phases, a limit due to the distributed architecture chosen: tests revealed that this limit starts around 100 devices in the network (assuming worst cases defined in test sections). In fact the number of messages in the network does not grow linearly with the number of devices and since we have not a central node managing the large number of them, devices start losing information and consistency is no more granted.

Anyway these results were obtained using virtualized devices on an ad-hoc client with limitations described in section 5.3.5, so we can suppose that the device boundary could be greater in a real scenario. Furthermore we think that such a limit is acceptable for several real use cases.

From the performance point of view tests show good results, considering the computational effort required by each device in the network, despite of time delays sometimes needed to deal with scalability issues.

Chapter 6

Conclusion and future works

Nowadays we are living IoT birth and development, moving contents and services from being remotes to being all around us and always available. IoT evolution, with first steps made by smart objects vendors, highlights the central role of technologies used by these objects, services they can offer to users in their surroundings and new smart interaction modes: the latter aspect is today underestimated, because in these "birth phase" the focus is centered on formers aspects, mainly because of marketing reasons.
With wearable devices and context oriented technologies released in last years, the aforementioned interaction aspect have been brought back in front again: in fact a smart interaction is usually more useful and attracting than the smart service offered.
A similar change involved mobile devices applications: the initial target to build a cool and really useful app, rapidly became to invent a smart way to enjoy the application's service. So the focus in this case moved to context-aware applications.
ADPF addresses common problems faced by context-aware applications, especially in indoor-proximity field, enriching their functionalities with multi-purpose features: we called these apps "proximity-aware", recently supported by new technologies and platforms. Our work aims to differentiate itself from existing commercial solutions, seen in the state of the art section, offering

a both low cost and easy to deploy solution, using a distributed approaches where mobile devices are actors and computing nodes.
ADPF offers:

- safe and efficient resources management
- grouping functionalities for application actors, based on their properties, together with intuitive tools for programmers
- high-level abstractions for indoor proximity-aware applications, hiding technology details
- notification-based and on-demand data distribution modes

Tests performed in this work aimed to stress the whole system simulating a common scenario with a varying number of devices involved: they shown good overall performance of the framework and its limits in terms of scalability. Real Android devices behaved as expected under stress conditions and the test app built on purpose was always usable. From the scalability point of view, in medium density environments (less than 100 devices) the framework works well: unfortunately we hadn't so many real devices available to test the framework in these conditions, so we built a simulator behaving as several real devices simultaneously; as reported in test section, we make everything possible to mitigate the impact on test results and at the same time we weren't able to overcome the devices number the simulator could run. As a future test would be interesting to have the chance to perform these tests on a set of real devices, even in a hybrid fashion (half real devices and half simulated), and measure ADPF performance.
Furthermore the distributed approach choice entails as drawback a restriction of ADPF functionalities in terms of flexibility: in fact, in order to make the framework scale well, it offers proximity and geofence subscriptions only relative to a single entity and a group, where the entity is either selfEntity or a BLE beacon. In this way we are able to have a reasonable number of

messages even with a large number of devices: an improvement could be to extend ADPF such that it is possible to perform group-to-group subscriptions, as we think it may be a useful feature.
A possible solution could be to implement an hybrid architecture, with a smarter broker handling filtering tasks in order to reduce both messages number and devices computational effort.
At the moment, our framework exploits only "passive technologies" to infer proximity information: Bluetooth Low Energy and GPS belong to this category as is not required a direct user interaction to make them work. As an extension the framework could manage also "active technologies" such as NFC and QRCode, to improve scenarios and ADPF functionalities in smart proximity interactions field.
Looking at chapter 2, we believe that the growing interest in indoor-proximity field will motivate companies to develop new technologies and standards improving accuracy and overall performances: ADPF has been specifically designed to be easily expanded in this sense, improving its performance too through their integration and aggregation of proximity results provided from them.
ADPF relies on TCP protocol, an OS-independent standard almost all OSs (mobile, embedded or desktop) use for network communication. Thus, another future work could be the porting of ADPF to other mobile OSs such as iOS or Windows Mobile.

Bibliography

[1] J. Chung, M. Donahoe, C. Schmandt, I.-J. Kim, P. Razavai, and M. Wiseman, "Indoor location sensing using geo-magnetism," in *Proceedings of the 9th international conference on Mobile systems, applications, and services.* ACM, 2011, pp. 141–154.

[2] Activebat project website. [Online]. Available: http://www.cl.cam.ac.uk/research/dtg/attarchive/bat/

[3] N. B. Priyantha, "The cricket indoor location system," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[4] Navizon fact-sheet reference. [Online]. Available: https://www.navizon.com/files/Navizon_ITS_Fact_Sheet.pdf

[5] P. Bolliger, "Redpin - adaptive, zero-configuration indoor localization through user collaboration," in *Proceedings of the First ACM International Workshop on Mobile Entity Localization and Tracking in GPS-less Environments*, ser. MELT '08. New York, NY, USA: ACM, 2008, pp. 55–60. [Online]. Available: http://doi.acm.org/10.1145/1410012.1410025

[6] X. Zhu and Y. Feng, "Rssi-based algorithm for indoor localization," *Communications and Network*, vol. 5, no. 02, p. 37, 2013.

[7] W. Gardner, C.-K. Chen *et al.*, "Signal-selective time-difference-of-arrival estimation for passive location of man-made signal sources in highly corruptive environments. i. theory and method," *Signal Processing, IEEE Transactions on*, vol. 40, no. 5, pp. 1168–1184, 1992.

[8] A. Gosai and R. Raval, “Real time location based tracking using wifi signals,” *International Journal of Computer Applications*, vol. 101, no. 5, 2014.

[9] Google nearby official developer website. [Online]. Available: https://developers.google.com/nearby/

[10] J. Hallberg, M. Nilsson, and K. Synnes, “Positioning with bluetooth,” in *Telecommunications, 2003. ICT 2003. 10th International Conference on*, vol. 2. IEEE, 2003, pp. 954–958.

[11] Å. Rudström, M. Svensson, R. Cöster, and K. Höök, “Mobitip: Using bluetooth as a mediator of social context,” in *Adjunct Proc. of Ubicomp 2004*. Citeseer, 2004.

[12] L. Technologies, “Bluetooth smart and bluetooth smart ready,” August 2012.

[13] Uribeacon standard specification. [Online]. Available: https://github.com/google/uribeacon/tree/uribeacon-final

[14] Eddystone bluetooth standard specification. [Online]. Available: https://github.com/google/eddystone/blob/master/protocol-specification.md

[15] Physical web project website. [Online]. Available: http://google.github.io/physical-web/

[16] I. Qualcomm Technologies, “Lte direct always-on device-to- device proximal discovery,” Qualcomm Technologies, Inc., Tech. Rep., 2014.

[17] Navizon proximity engine features. [Online]. Available: https://support.navizon.com/navizon-proximity-engine-api-url-alerts/

[18] Allseen alliance official website. [Online]. Available: https://allseenalliance.org

[19] Allseen alljoyn location service project. [Online]. Available: https://wiki.allseenalliance.org/locationservices

[20] E. Dahlgren and H. Mahmood, "Evaluation of indoor positioning based on bluetooth smart technology," 2014, 94.

[21] Jsonpath library website. [Online]. Available: https://github.com/jayway/JsonPath

[22] Jsonsimple website library. [Online]. Available: https://code.google.com/p/json-simple/

[23] Altbeacon bluetooth library website. [Online]. Available: http://altbeacon.github.io/android-beacon-library/index.html

[24] Mqtt paho library website. [Online]. Available: http://www.eclipse.org/paho/

[25] Mqtt protocol website. [Online]. Available: http://mqtt.org/

[26] Logstash android logger project website. [Online]. Available: https://github.com/Labgoo/android-logstash-logger