EXPLORING WIRELESS COMMUNICATION ALTERNATIVES WITH ANDROID THROUGH THE IMPLEMENTATION OF A TICKET QUEUING SYSTEM

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Francisco Narváez

In partial fulfilment

of the requirements for the degree of

MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Juan Luis Gorricho

Barcelona, August 2014

Abstract

Android has become the most widely used operative system on mobile devices. As it evolved, it has incorporated wider support for wireless communications. Android devices could help managing every day issues, relying on their communication capabilities. The aims of this study are to explore different communication mechanisms supported by Android, through the implementation of a ticket queuing system. The proposed system consists of a server application acting as queue manager, and a client application capable of requesting tickets. Clients remain informed about the ticket waiting time and get alerts when they need to head to the server's physical location. In this study, a queuing mechanism has been developed, capable of running on top of different wireless communication mechanisms. It successfully implements communication through the Android's Network Service Discovery mechanism, in combination with TCP and UDP communication. This system can incorporate Wi-Fi Direct or any other communication protocols in the future.

To my parents, for all their support, and for trusting me with their cell phones.

Acknowledgements

I would like to express my deepest gratitude to my Advisor, Juan Luis Gorricho, whose directions and ideas have helped me throughout the project. His clear guidelines allowed us to come up with the project idea, and helped me to pinpoint the areas on which my project should focus. Without his guidance, this project would not have been possible.

Revision history and approval record

Revision	Date	Purpose
0	01/08/2014	Document creation
1	20/08/2014	Document revision
2	31/08/2014	Document revision

Written by:		Reviewed and approved by:	
Date	31/08/2014	Date	
Name	Francisco Narváez	Name	Juan Luis Gorricho
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

1. Introduction

Only 3 years after the first Android-based phone was released in 2008, [1], it became the mobile operative system with the largest installed base worldwide [2]. And in 2012, this reflected on the global market share, becoming Android devices the most selling around the world [3]. Android's market domination has grown so large that they have doubled the iOS market share, the second largest one in the world when it comes to mobile devices. In Spain, Android's leading position is even clearer, with over 84% of market share by June 2014 [4]. Given that there were over 55 million active mobile lines in Spain in 2013[5], we can infer that several million users have an Android-based mobile device.

Android OS has been changing drastically over the years, since Android's first release (Api level 1) until the latest one (Api level 19) [6]. Among all the improvements, the networking capabilities have multiplied, and the devices' hardware have allowed newer communication techniques. In the wireless context, besides regular TCP/UDP communication, connections through Bluetooth, NFC and Wi-Fi P2P can be stablished [7][8].

In our everyday lives, we face situations where we have to stand in long queues waiting for a service to be granted. Situations such as waiting for customer service in a store, writing our name on a waiting list for an empty table at a restaurant, or getting a procedure done on a government agency, require a long wait of several minutes or even hours; and we have to physically remain at the service location until the service staff tells us that it is our time to be served. However, a very large amount of the customers waiting in line do have mobile devices that run Android, capable of different types of wireless communications. By exploding these characteristics, a better service could be provided, where the customers will have more control over their activities during the waiting time, not having to physically remain at the same location, and therefore are likely to have a better experience.

There are several Android applications for handling queues. We chose to inform ourselves about three of the most successful ones on the market:

- WaitKnowMore[9]: This app has been rewarded in the Startup Weekend Cebu, in Philippines[10]. It offers a queue management system from the cloud, where the businesses can register via web. The customers can download an app where they get a feed of their queue status and waiting time.

- Qless[11]: Another awarded queuing system, which runs on POS systems or mobile devices. It offers a very complete service to the business, generating analytics that can even forecast the future customers. Customers are able to communicate and request tickets on their web browsers, through an app, or via SMS.

- Qminder[12]: This solution allows the business to use a tablet app for issuing queues, besides regular paper tickets. The customers are able to request a ticket through an app connecting to a centralized system. They select the queue from a list of businesses in a certain geographical area. The app attempts to give an exact waiting time to the user, depending on their location.

All these apps rely on an external server for queue handling. While it gives the ubiquity for the client to request a ticket from anywhere with an internet connection, they are almost non-functional without internet. Besides Qless support of SMS registration, no other method for interconnecting devices wirelessly is exploited.

1.1. Proposal requirements and specifications

We propose a solution for handling queues without relying on an external server, taking advantage of different types of communication offered by Android, thus providing a better service. We want to address problems such as the lack of network coverage in certain areas, keeping a low battery usage but profiting from many of the characteristics of Android devices beyond the regular internet connection. We propose a queue management system capable of handling different network and transport protocols, adapting to the environment, network status, available hardware, battery level, and message relevance.

We will target devices with Android Jelly Bean installed (Api 16, 17 and 18), since this version is installed in over 50% of all Android devices[13]. Therefore, our app will aim devices with Api 16 (version 4.1) as minimum Api version. The hardware of our target devices should handle Wi-Fi connections, GPS location, and should have vibration capabilities.

For our development and testing processes, we will use three Samsung Galaxy GT-S6310N devices, running Android 4.1.2 (API level 17).

1.1.1. Server application

The proposed system requires an Android application acting as a server, for hosting the queue management functions. The Android device running this server application must be physically located at the installations of the service-offering entity. The device needs to be located on an area under Wi-Fi coverage, with a router capable of assigning IP addresses to the visiting devices, but internet access is not required. The server application will act as the ticket provider, and it will issue tickets to the connecting client applications.

1.1.2. Client application

In order to connect to the queue management system and request a ticket, a client application is required. This application will be developed using Android, and it should connect to the server application easily by running under the same Wi-Fi network. Once a ticket has been granted, the client may physically leave the network location and will be alerted by the application when it is time to head to the physical location of the ticket provider.

1.2. Objectives

This thesis focuses on the following aspects and objectives:

1.2.1. Application Structure

The system should follow the basic Android guidelines and design philosophy, such as the use of fragments as the interface component of the activities [14]. The project should be modularized and reusable, and through the use of interfaces it should handle the implementation of different communication protocols, without altering the basic core system functionalities. Methods for specific purposes should be encapsulated, allowing their reuse by different components.

1.2.2. Queuing System

The developed queuing system should handle several queues from a single provider, with a different maximum size for each queue. It should alert the client when it is time to head toward the server location, so they arrive before their ticket number is called. The estimated waiting time for each queue should adapt and get more precise over time.

1.2.3. Data Management

The data to be collected by the system should be stored in the most appropriated way [15]. The system architecture should be able to split the data management in layers, separating the business logic from the functionalities that handle directly the stored data.

1.2.4. Communication Protocol

The application should be able to choose the appropriated low-level data transport protocol for data transmission. For the client application to discover the servers around, it should implement the discovery mechanisms provided by Android[16].

1.2.5. Data Exchange

Information must be shared wirelessly between client and server. The messages for carrying this data should be designed in a simple manner, keeping them in small size and allowing their transformation to and from a serializable stream of bytes.

1.2.6. Prediction

The application should predict user behaviour in order to offer a better service. The server application should be able to provide every time more accurate results to the waiting time estimated for each user.

1.3. Methods and procedures

This project is not a continuation of any other projects. It does rely on other libraries, frameworks and pieces of code concerning general Android development. It does not, however, use any software or code created for queuing, routing among different communication protocols, or any other aspect closely related to the objectives of this thesis.

1.4. Work plan

The different tasks for this thesis are planned based on an average of 30 work hours per week, and taking into account the holidays spread throughout the year.

1.4.1. Tasks

The following list contains all tasks required for the development of the thesis project:

1.4.1.1. Background Research

This research comprehends:

- Research of the existing applications with similar functionalities
- Research on the existing studies handling this matter
- Research of the available technologies.
- Tutorials and tests of the available technologies.

1.4.1.2. Development planning. Requirements and use cases.

Constructing a working plan for both client and server applications, based on the required functionalities and the focusing aspects of this thesis. Use cases and a data model will be designed for describing the activities performed by the applications.

1.4.1.3. Development of server application.

This task comprehends the following subtasks:

- Generation the initial server application code.
- Development of the initial graphic user interface.
- Incorporation and testing of frameworks and external libraries.
- Implementation of a graphic message logger, thus making easier the debugging process.
- Implementation of the database structure, based on the drafted data model.

1.4.1.4. Implementation of basic queue management function.

Incorporation of testing data to the database. Development of objects and functionalities for mocking a client connection to the server. Having the mocked client perform the activities of ticket request, and developing on the server the methods for ticket issuing, ticket serving and sending update messages to the client.

1.4.1.5. Development of client application.

This task comprehends the generation the initial client application code, including an initial graphic user interface. The client application should also implement a graphic message logger, for logging purposes.

1.4.1.6. Implementation of basic queue request functionalities.

This task incorporates to the client application the required objects and functionalities for basic ticket request. It should mock a server connection, since communication between both applications is not available at this point.

1.4.1.7. Implementation of Wi-Fi Network Service Discovery

Development of a Network Service Discovery (NSD) publisher on the server application, as well as including service discovery functionalities on the client application. The NSD functions should be integrated with the rest of the already implemented classes and methods, as well as connecting to the graphic user interface.

1.4.1.8. Implementation of TCP communication.

Development of both a TCP server waiting for client connections on the server application, and the corresponding client socket method on the client application, along with the required methods for making server requests and processing the responses.

1.4.1.9. Development of communication message.

Planning and development of the different calls in the client-server communication. Structuring a message capable of handling the data exchanged on each call.

1.4.1.10. Exchange of hello messages

Development of the message sent by the client through TCP once a service is discovered, with the client's basic information. Development of the message handling done by the server and the corresponding actions to be performed when a new client connects.

1.4.1.11. Exchange of hello reply messages

Development and implementation of a message sent by the server application as a reply of the HELLO messages. This message should include the relevant information for a client to make a ticket request. Development of the necessary actions as a result of this message, such as the corresponding database queries on the server application, and the relevant information display on the client application.

1.4.1.12. Exchange of ticket request messages

Programming of the client message in charge of requesting a ticket from a specific queue to the server. Development of the necessary methods in order for this message to trigger the ticket issuing mechanism on the server side.

1.4.1.13. Exchange of reply messages to the ticket request

Implementation of the message sent by the server, with the issued ticket information. Programming of the methods to handle the response in an appropriated manner, by adjusting the client application interface to the received message, reflecting the granted ticket.

1.4.1.14. Implementation of methods for handling hardware components

Development of functionalities for requesting information and triggering actions on the following hardware components:

- Battery information retrieving methods, in order to get the battery level at any given moment.
- Vibration enabling methods, for making the device vibrate as a way to alert the user that their ticket is about to be called.
- Location methods for handling the aspects related to the device physical location. This information should be fetched from the appropriated location provider (network or GPS), taking into account factors such as the battery level and accuracy of the last measure.

1.4.1.15. Exchange of ticket status request messages

Development of a message sent by the client, requesting the status of their open ticket and the estimated waiting time left.

1.4.1.16. Exchange of ticket status messages

Implementation of messages sent by the server application when it considers necessary to inform about the status of a ticket. This message should include relevant information such as whether the ticket has already been called, or the waiting time it still has.

1.4.1.17. Implementation of waiting time prediction algorithm

Development of an algorithm that would allow the client to know when they should head to the server application physical location, based on the location of both the client and the server. Implementing a way to make the estimated measures improve themselves over time.

1.4.1.18. Implementation of UDP communication

Programming a way for both the client and the server to be able to send and receive asynchronous messages via UDP. Establishing a way of sending the connection information (port, host) to each other.

1.4.1.19. Message transmission through an specific transport protocol

Managing the message transmission in order to use TCP or UDP, according to the message, connection information, and any other relevant condition.

1.4.1.20. Implementation of automatic status updates

Development of a structure capable of requesting open tickets information asynchronously, periodically and after any relevant events happening to the client or the server.

1.4.1.21. Implementation of Wi-Fi Direct Network Service Discovery

Implementation of the necessary mechanisms in order to access the Wi-Fi Direct protocol. Development of the NSD publisher on the server side, and the service discovery functionalities on the client side. Integration with the regular Wi-Fi NSD, and the rest of the system.

1.4.2. Milestones

The milestones identified for the software development are:

- **Project planning finished.**

  At this point the research phase is terminated, and the software development process may begin.

- **Offline ticket manager application developed.**

  The basic server app is created and can be installed on mobile devices. It also is capable of executing ticket management functionalities by mocking client connections.

- **Offline ticket client application developed.**

  The basic client app is finished and can be properly installed. It features the user interface and has the inherent ticket request functionalities. However, it is not capable of network communication.

- **Client-Server communication stablished.**

  Client and server are able to communicate wirelessly, in a basic manner. The client can find the server information published by the NSD services.

- **Server ticket issuing by client request**

The clients is able to request tickets to the server, and the server can provide them. End users are able to see these functionalities on the graphic interface.

- **Client alert prior to ticket serving**

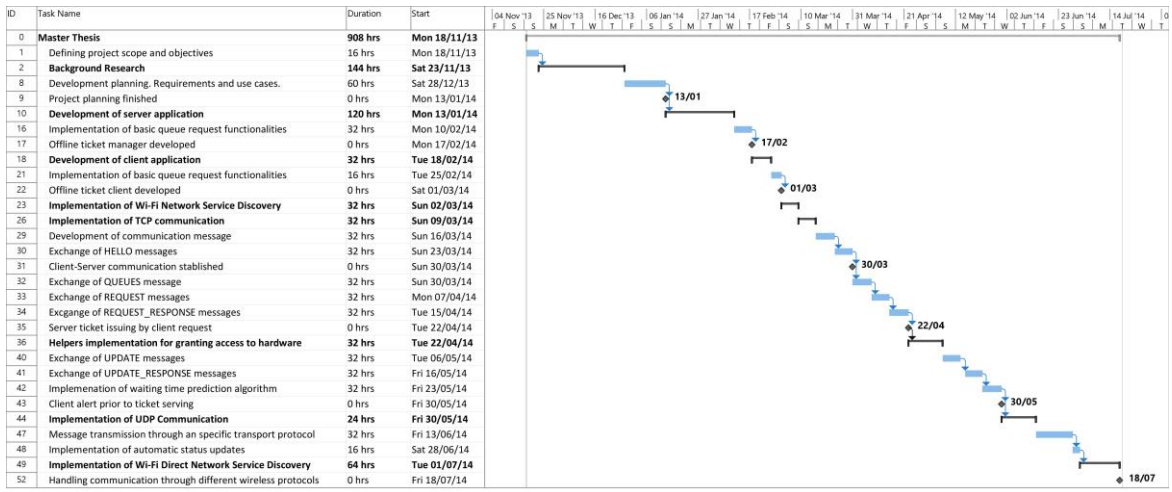
  The server alerts the client when it has to head to the server location, by predicting how long it would take, and how long it has before the client's ticket is called.

- **Handling communication through different wireless communication protocols.**

  Client and server are able to communicate through more than one protocol, thus profiting the communication capabilities of Android devices.

1.4.3. Gantt Diagram

ID	Task Name	Duration	Start
0	**Master Thesis**	**908 hrs**	**Mon 18/11/13**
1	Defining project scope and objectives	16 hrs	Mon 18/11/13
2	**Background Research**	**144 hrs**	**Sat 23/11/13**
8	Development planning. Requirements and use cases.	60 hrs	Sat 28/12/13
9	Project planning finished	0 hrs	Mon 13/01/14
10	**Development of server application**	**120 hrs**	**Mon 13/01/14**
16	Implementation of basic queue request functionalities	32 hrs	Mon 10/02/14
17	Offline ticket manager developed	0 hrs	Mon 17/02/14
18	**Development of client application**	**32 hrs**	**Tue 18/02/14**
21	Implementation of basic queue request functionalities	16 hrs	Tue 25/02/14
22	Offline ticket client developed	0 hrs	Sat 01/03/14
23	**Implementation of Wi-Fi Network Service Discovery**	**32 hrs**	**Sun 02/03/14**
26	**Implementation of TCP communication**	**32 hrs**	**Sun 09/03/14**
29	Development of communication message	32 hrs	Sun 16/03/14
30	Exchange of HELLO messages	32 hrs	Sun 23/03/14
31	Client-Server communication stablished	0 hrs	Sun 30/03/14
32	Exchange of QUEUES message	32 hrs	Sun 30/03/14
33	Exchange of REQUEST messages	32 hrs	Mon 07/04/14
34	Excgange of REQUEST_RESPONSE messages	32 hrs	Tue 15/04/14
35	Server ticket issuing by client request	0 hrs	Tue 22/04/14
36	**Helpers implementation for granting access to hardware**	**32 hrs**	**Tue 22/04/14**
40	Exchange of UPDATE messages	32 hrs	Tue 06/05/14
41	Exchange of UPDATE_RESPONSE messages	32 hrs	Fri 16/05/14
42	Implemenation of waiting time prediction algorithm	32 hrs	Fri 23/05/14
43	Client alert prior to ticket serving	0 hrs	Fri 30/05/14
44	**Implementation of UDP Communication**	**24 hrs**	**Fri 30/05/14**
47	Message transmission through an specific transport protocol	32 hrs	Fri 13/06/14
48	Implementation of automatic status updates	16 hrs	Sat 28/06/14
49	**Implementation of Wi-Fi Direct Network Service Discovery**	**64 hrs**	**Tue 01/07/14**
52	Handling communication through different wireless protocols	0 hrs	Fri 18/07/14



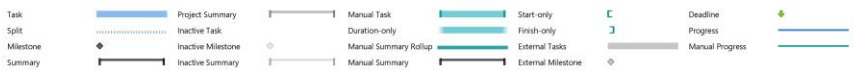


Figure 1: Gantt Diagram

1.4.4. Deviations

During the project development process we faced some difficulties that forced us to spend more time on certain tasks, which had an impact on the final project result. The most relevant difficulties were:

- The UI implementation was originally planned to benefit from the touch screen capabilities, allowing the server to see a grid of connected clients, and the tickets would be assigned by dragging them to the client icon. This helped us mocking the wireless client connection before implementing it, so we could to develop the ticket issuing functionalities on the server.

  However, this kind of interface proved to be very difficult to implement, as grids don't support drag-and-drop capabilities. After spending twice the time we planned on, we got the interface to work. Once we developed the TCP communication between the server and the client, this functionality was no longer needed, since the client could select the desired queue from its own graphic interface, and it was dropped in favour of a simpler interface.

- Implementing and maintaining the SQLite database on the server was more complicated and time consuming than we originally planned. Even though we were aware of certain libraries to help dealing with the database management with Android, we decided to implement it ourselves, since the server-side database only consisted of two tables and Android does have several libraries in order to manage data. However, adding queries to the database helper class got very complicated because of the amount of code and detail that it requires. After spending more time than planned struggling with a growing database helper and the required cleanup functions, we moved on to the ORMLite framework. It simplified our work greatly, and we would have saved time if we had used it from the beginning.

- The implementation of the Wi-Fi NSD communication also had a higher implementation cost, related to the testing time. The NSD process can easily lead to silent errors because of cached information, where the only workaround is uninstalling the application and rebooting the device. On the server side, the publishing mechanism would fail on a regular basis after a number of trials, without warnings, because of apparently reaching a limit on the different names the NSD can assign to a service. On the client side, old NSD names would remain cached, creating lingering services which would no longer work, but where the client nevertheless would try to connect to.

- The development of reading sockets also took longer time than expected, since our first approach of developing them on background AsyncTasks would randomly cause other threads to fail. This error was especially hard to track, since the sockets themselves would not fail, but the thread-related tasks done afterward.

This difficulty remained until we decided to change our server implementation to a thread pool approach.

- The hardware used for testing purposes (Samsung Galaxy GT-S6310N running Android 4.1.2) would reboot often after finishing the installation of the application to be tested. It lead to higher testing times on any task that could not be run in the simulator, such as the ones concerning internetworking.
- While we prepared the architecture for Wi-Fi Direct Network Service Discovery, and had the infrastructure necessary for handling the Wi-Fi Direct connections, our hardware would constantly fail in accepting the requested connection. Because of this, and because of many prior tasks having consumed a longer time than expected, we were forced to drop this feature.

2. State of the art of the technology used or applied in this thesis

2.1. Related studies

The Autonomous Network Research Group from the University of Southern California published a Multilayer Application for Multi-hop Messages, for Android devices based on Wi-Fi Direct[17]. It is based on four layers, providing a communication protocol among them. They propose a connection manager, a routing manager, and a file discover manager, in order to stablish the communication among peers by sharing and forwarding text files.

Our application could profit from this study to provide an offline routing mechanism among the clients. However, our scope on this project is mainly focused on the queuing mechanism, providing TCP and UDP connection. We considered this study when designing our application, in order to make it compatible with this or any other connection or routing mechanism that could be used.

2.2. Development Tools

There are several libraries and frameworks available for Android development. We focused our research on the following:

- The Android Kickstartr tool [18] sets up a basic empty project that includes a group of the most popular open libraries. It saves time setting up an Android project, and the user can select the libraries to be included, in order to match the needs of the application.

- The Support v4 library [19] is developed by Android, to make applications compatible with the earlier Android API levels, up to Android 1.6 (API level 4). Most of the included features aim to bring to earlier devices the graphic characteristics developed after Android 4.0, such as the improved notifications bar, swipe to refresh functionalities, and other graphic widget that have become an standard for Android applications. This library also adds the support for fragments, and other commonly used functionalities nowadays. After our research, we found that this library did not include the support we need for Network Service Discovery, thus making us unable to generate a backward compatible version of our client and server application.

- The AndroidAnnotations[20] framework is a very popular framework for Android development. It greatly simplifies the android code written, making it much easier

to program, understand and maintain. The framework provides specific behaviour to classes, functions or field variables, by marking them with an annotation. Some of the most relevant annotations are:

@**EBean:** Marking a class with this annotation turns it into a so-called “enhanced” class. Enhanced classes can use annotations on their fields and methods, and can easily be imported by other enhanced components such as classes, fragments, activities or by the application itself, only by declaring it as a class field with the @Bean annotation, no constructor or initialization needed.

@**ViewById:** View components marked with this field will be automatically fetched from the parent view layout. Works as the equivalent of view.findViewById(R.id.name).

@**Click(R.id.clickable):** Runs the method below, when the layout clickable component is clicked.

@**Background:** Indicates that the method below will run on a thread different from the main thread, hiding the AsnycTask implementation recommended for background threads. This is very useful for heavy computations.

@**UiThread:** Runs the method on the main thread. This can be very helpful, since it is only through functions on the main thread that the graphic components of the application can be updated.

- ORM Lite[21]: ORM stands for Object Relational Mapping, and acts as a middleware between a SQL database and java objects. ORM Lite provides access to the database through Database Access Objects (DAO), completely hiding the complexity of a direct database access. Each java object that needs to be persisted has a DAO, through which it can be created in the database, retrieved, updated or deleted. ORM Lite also supports “foreign objects” (equivalent to foreign table references done by foreign keys). ORM Lite also generates and executes SQL code, not only for the object handling but also for creating and dropping the databases. AndroidAnnotations provides support for ORM Lite.

2.3. Data Management

In Android there are five basic ways of storing data, depending on the amount, complexity, frequency of access, who can access it, and other application specific requirements. They are Shared Preferences, Internal Storage, External Storage, SQLite Databases, and Network Connection. For our application, we discarded the use of external servers and connections, thus eliminating the option of data storage through Network Connection. Internal and External Storage are suited for storing files, and our application doesn’t aim to handle any type of media. The remaining options are:

Shared Preferences: This mechanism allows the storage of primitive data types (booleans, floats, ints, longs and strings) in a key-value fashion. The storage is persistent, even if the application gets killed.

SQLite Databases: SQLite is a software which implements a relational, transactional SQL database engine [22]. It is self-contained and requires neither configuration nor a database server. SQLite is used worldwide, in mobile applications, web browsers, and others; and provides binding for a large amount of programming languages. Android has full support for this type of databases through a series of classes and methods. By extending the SQLiteOpenHelper class, we gain access to the SQLiteDatabase object. This object has different query methods, for simple or complex queries, and a SQLiteQueryBuilder. Each query returns a Cursor object pointing to the returned row, from which the application can read the query results.

2.4. Communication Protocol

For the communication protocol that the devices should follow, we studied first the Android guides for wireless communication. Android has a way of getting objects in the same area to communicate, which is the Network Service Discovery.

The regular Network Service Discovery supports publishing a service in the network. A device runs a server in a certain port, and publishes the open port number, along with the service name and communication protocol. Other devices acting as clients request the network for the published services. Once the client finds published services, it filters them by name and communication protocol, and once it gets the open port number, it can start sending data. Android support for TCP and UDP sockets, along with NSD, is enough to build a communication mechanism between server and clients.

The Wi-Fi Direct NSD is another method provided by Android in order to publish a service. It is based on the Wi-Fi Direct peer-to-peer communication, which enables devices to act as peers and share files among each other. Wi-Fi Direct acts in a similar fashion than Bluetooth, adding devices to peer groups, but with a longer range and offering a more stable connection. In order to connect devices among each other, they use a protocol called Wi-Fi Protected Setup, which is implemented in all Wi-Fi Direct enabled devices. Once the devices are peered and forming a group, the group manager can share data with them. Through this option, we could pass the communication information necessary to stablish a direct communication between the queue system clients and the server.

2.5. Data Exchange

For the data exchange, we considered different ways we could structure our information to pass through a data socket, such as xml, json, or designing our own data structures, going from very low to very high level.

In our research we found a library called Gson, developed by Google [23], capable of parsing java objects to and from JSON strings. Including the Gson library in our applications would benefit of JSON's simple structure and human readability. Through the use of Gson we could create a sophisticated message structure, which would be translated to a simple array of bytes for network transmission purposes. Gson implementation also simplifies the debugging process..

3. Methodology / project development

The project implementation was performed in a linear way, aiming to work on one task at a time, consequently avoiding working on a new task while others are still open. The implementation of a new functionality on the server would be followed by the corresponding functionality on the client, and vice versa. Each developed functionality was followed by an intensive testing phase on both the client and the server application.

The following diagrams will describe the implemented solution in this project.

3.1. Use Cases

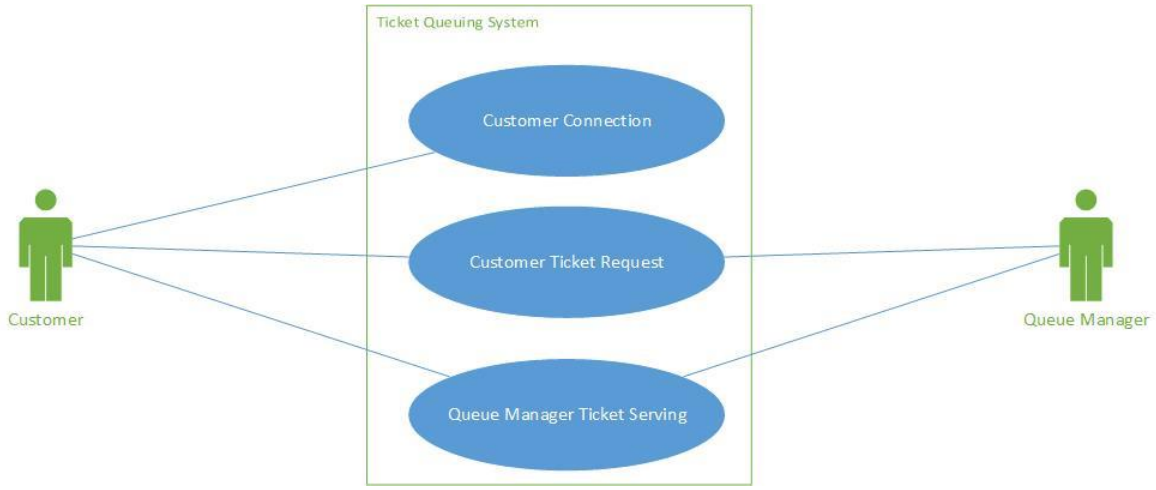


Figure 2: Use cases

3.1.1. UC1-Customer Connection

Use Case Name:	CU1-Customer Connection
Description:	The customer through the client app looks up the available queues from the nearby services.
Actors:	Customer
Preconditions:	The customer is not connected to the queuing server.
Basic Flow:	1. The customer clicks on “discover services” 2. The application looks up the available services nearby. 3. The application displays a list of the queues from the available

	services.
Alternate Flows:	A1. The customer clicks on “discover Wi-Fi Direct services” The application looks up the available services via Wi-Fi Direct.
Exception Flows:	E2. The application does not find any applications matching the expected name or protocol. The application performs a limited number of discovery retries. The application displays an error connecting to the services.
Post Conditions:	The client application has connected to a server, and is able to start exchanging messages.

Table 1: Use Case UC1 - Customer Connection

3.1.2. UC2-Customer Ticket Request

Use Case Name:	CU2-Customer Ticket Request
Description:	The customer through the client app requests a ticket to a certain queue. The queue manager, through the queue app, sees the requested ticket.
Actors:	Customer, Queue Manager
Preconditions:	The customer is connected to the queuing server and doesn’t already have an issued ticket on the desired queue.
Basic Flow:	1. The customer clicks on the desired queue. 2. The client application connects to the server application and performs the ticket request. 3. The server application provides a ticket and sends it back to the user. 4. The client application receives the issued ticket and displays it through the graphic interface. 5. The server app displays through the graphic interface that a new ticket has been issued on one of the server queues.
Exception Flows:	E3. A new ticket cannot be issued, because the queue max number has been reached. The server returns an empty ticket message to the client application. E4. A network error prevents the ticket issuing messages to be exchanged.
Post Conditions:	The client has been granted a ticket.

Table 2: Use Case UC2 - Customer Ticket Request

3.1.3. UC3-Queue Manager ticket serving

Use Case Name:	CU3-Queue Manager ticket serving
Description:	The Queue Manager, through the server application, calls the next number once the client has been served.
Actors:	Customer, Queue Manager
Preconditions:	None
Basic Flow:	1. The queue manager clicks on “call next” on the respective queue. 2. The server application closes the current open ticket from the queue, if any. 3. The server application sends an update message to all the clients waiting on that queue. 4. The client application gets an update message, and refreshes the estimated waiting time.
Alternate Flows:	A1. There were no clients waiting on the queue, nobody gets notified.
Exception Flows:	E2. The waiting client application is no longer available.
Post Conditions:	The current client has been served, and the next client on the queue is the new current client. All the clients in the queue have been notified.

Table 3: Use Case UC3 - Queue Manager Ticket serving

3.2. Activity Diagrams

The following diagrams represent a high-level description of the main activities carried by the system.

3.2.1. AC1-Client Connection Request Activity

Activity related to the use case UC1-Customer Connection. It starts with the discovery finding process on the client side, followed by the connection request once the services has been discovered. The server application replies with the list of available queues, which are then displayed on the client application's user interface.

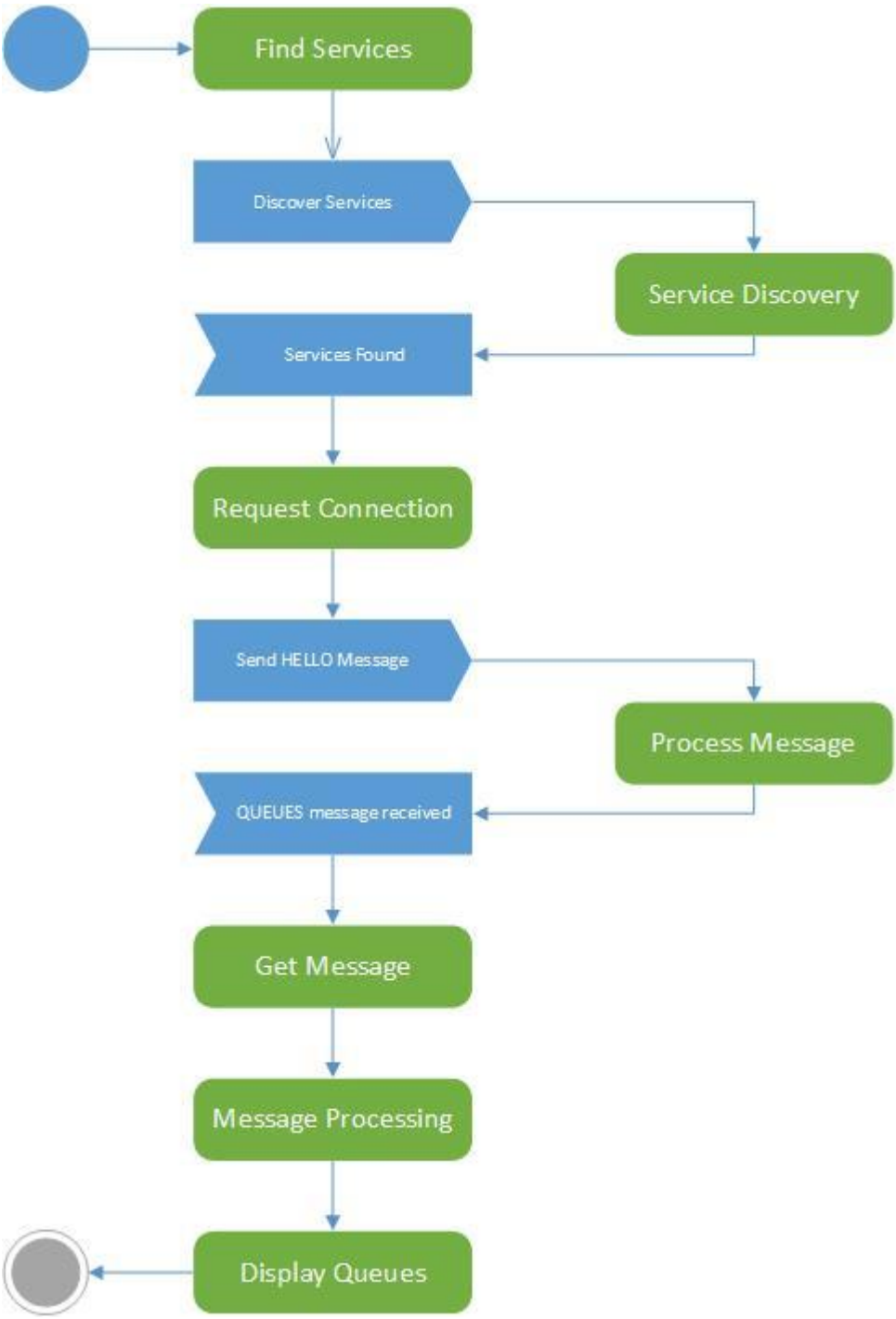


Figure 3: Activity Diagram AC1 – Client Connection Request Activity

3.2.2. AC2-Client Ticket Request Activity

Activity related to the use case UC2-Customer Ticket Request. It illustrates the process of ticket request started by the client. Once the ticket is issued, the client begins a cycle of updates over the open ticket in order to keep the waiting time updated.

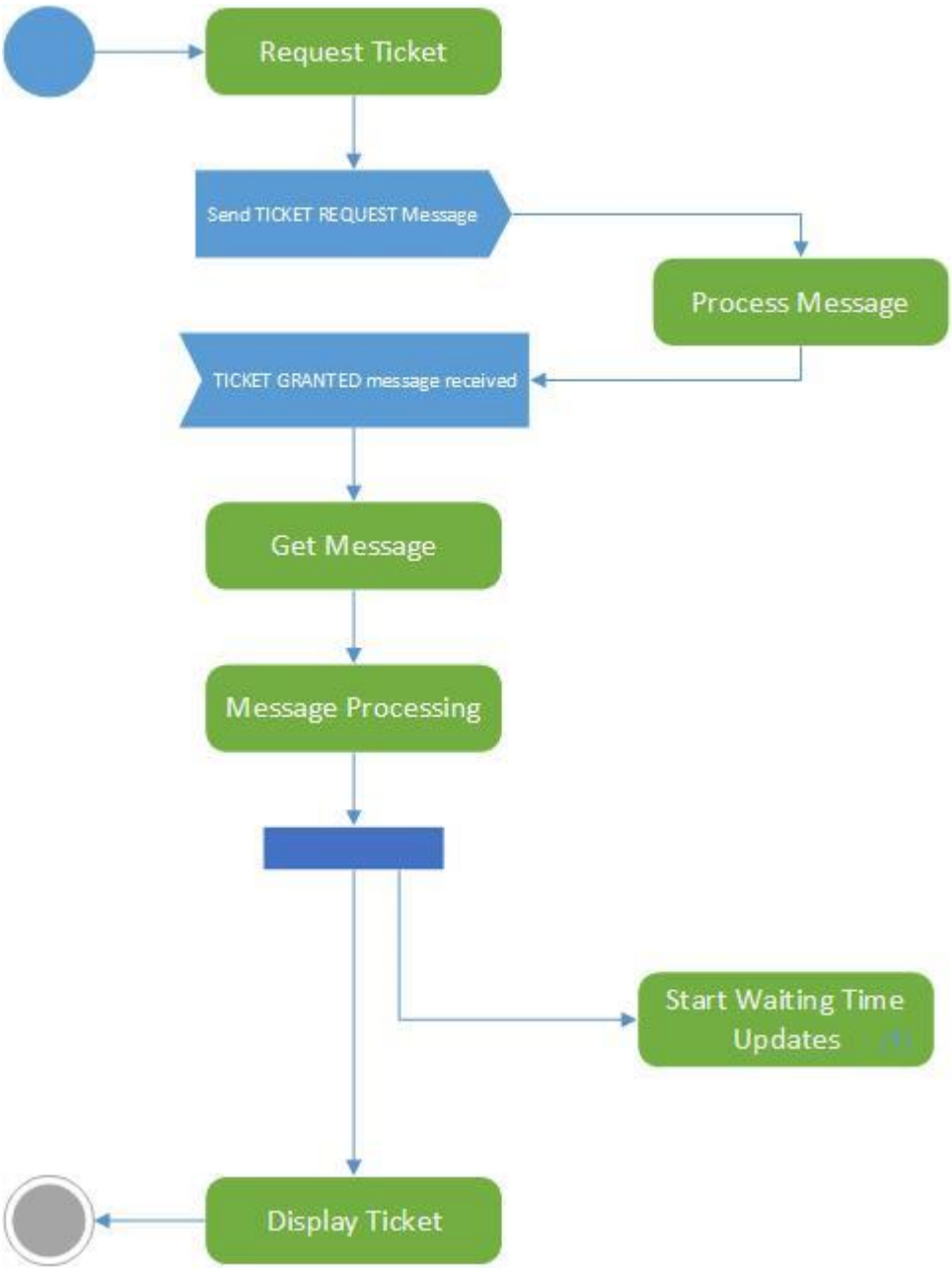


Figure 4: Activity Diagram AC2 – Client Ticket Request Activity

3.2.3. AC3-Server Ticket Serving Activity

This diagram exemplifies the actions after the server calls the next ticket on the queue. It closes the actual ticket, and performs a call to the recently closed ticket and to the newly open ticket, in order to update their status.

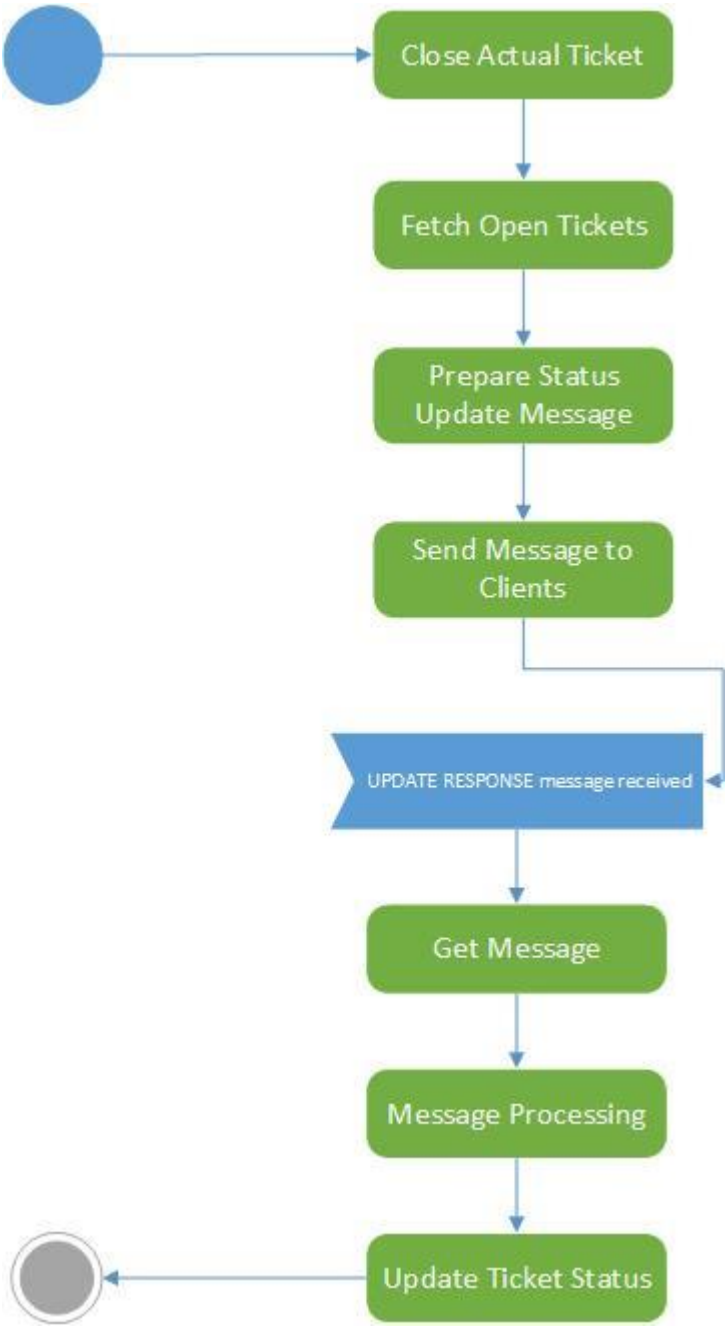


Figure 5: Activity Diagram AC3 - Server Ticket Serving Activity

3.2.4. AC4-Ticket Waiting Time Update Activity

This action is performed on all the open tickets received by the client. It estimates the remaining waiting time for the ticket to be called, and alerts the user when it is time to head to the server's physical location. The client will request an update ticket message if its current location has changed significantly. The server will also send periodically update messages to all clients with open tickets.

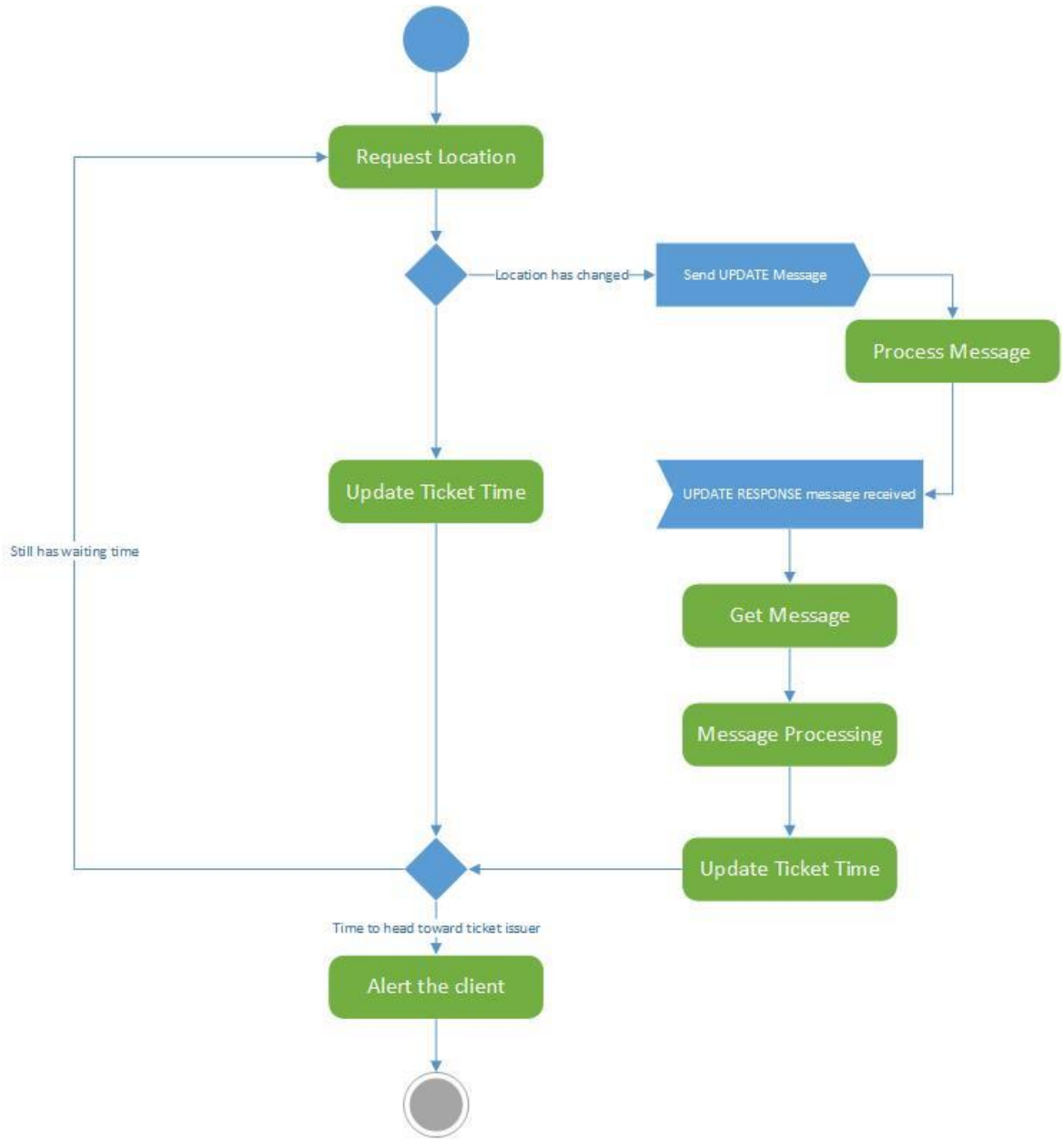


Figure 5: Activity Diagram AC4 - Ticket Waiting Time Update Activity

3.3. Components Diagram

Because of the structure implemented on the server and the client applications, it is possible to encapsulate most functionalities in modules. By the use of the components diagrams we will represent the interaction between them. In each case, the modules may interact and communicate with others, through their enclosing component.

3.3.1. CD1 - Server Application

Figure 6: Component Diagram CD1 - Server Application

On the server application, the modules are located starting on the application level. This representation shows the different interfaces from the application between components, and interfaces with the external network.

The interacting elements are:

3.3.1.1. Server Application

This is the highest level on an Android application. Usually, applications are developed starting on the activity level. Graphic elements, however, cannot always access the activity running them, but all the elements from the application do have access to the components on the application level. The components placed here are the ones related to hardware access, being the battery, location providers, and database.

3.3.1.2. Ticket Queue Helper

This class acts as an interface with the database, processing high-level requests into one or several low-level data access functions. In our case, this component includes the ORM

Lite helper class and Database Access Objects (DAO). They may receive requests from UI elements such as the request ticket button, which displays the number of open tickets on each queue. The Ticket Queue Helper is most accessed through the Client Manager, which lays on the Application and is in charge of handling the communication and logic related to the clients.

3.3.1.3. Battery Helper

This helper directly requests the system for the available battery level. It provides a single method which returns a float value with the battery percentage available.

3.3.1.4. Location Helper

The location helper class is in charge of getting the geographical position of the device, by requesting the appropriated positioning provider. It incorporates an algorithm for getting the best available location, and it access the battery helper through the Application, since location updates can consume very high power.

3.3.1.5. Main Activity

The core of our application, the activity is in charge of handling all the displayed elements and their interaction. Even though some Android applications may have more than one activity, the usual flow incorporates only one and splits the interfaces and actions through fragments, managed by the activity itself. The activity interacts freely with the Application and its components. This is, however, a one-way path; the Application cannot interact with the activity, since it may be managing more than one. We will describe our Main Activity further, and the interaction of the components inside of it.

3.3.2. CD2 - Server Main Activity

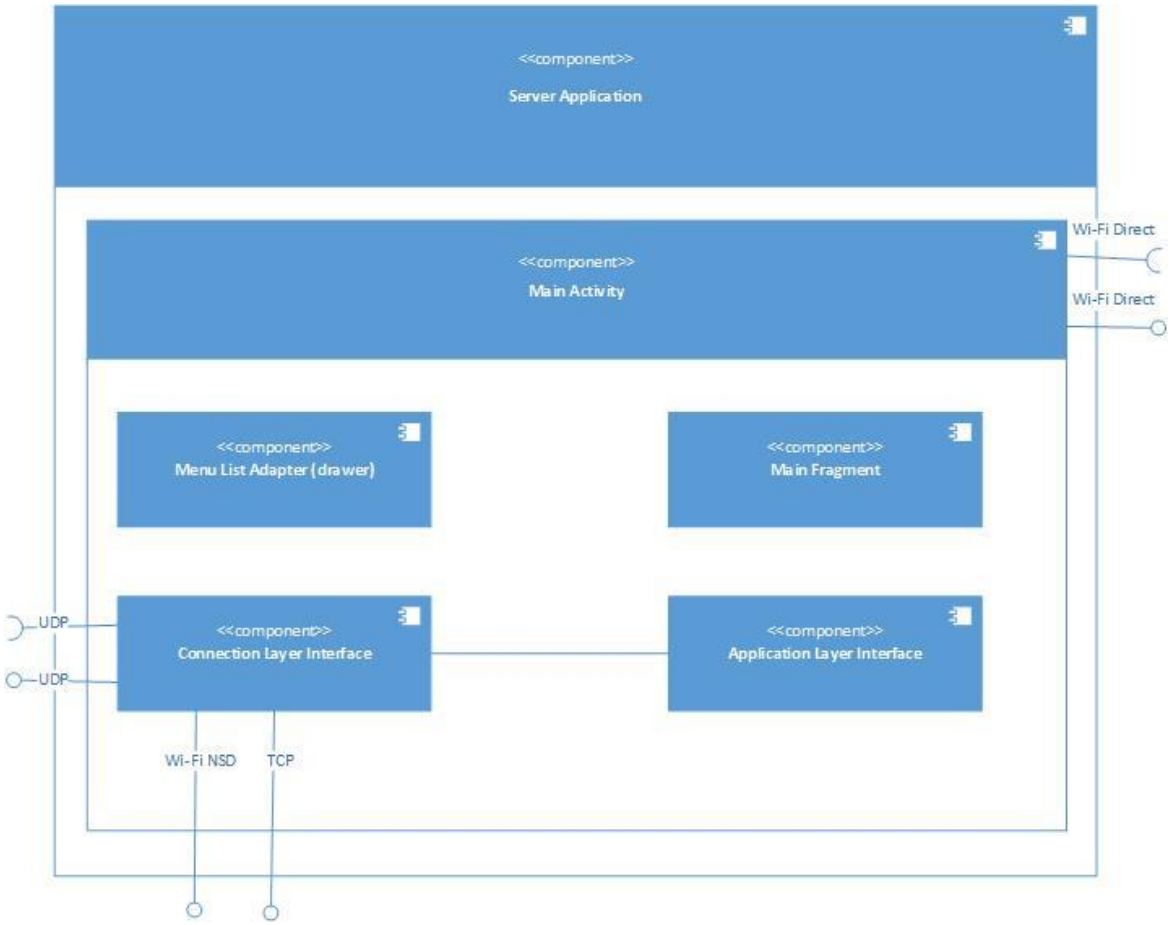


Figure 7: Component Diagram CD 2 - Server Main Activity

Going one level further, into the Application's Main Activity, we can see the different components it contains. It is worth mentioning that the Wi-Fi Direct interfaces are located directly in the activity. Android recommendation is to define the different Wi-Fi Direct intents in the activity, and have a Broadcast Receiver listen for them. Therefore, they were not encapsulated, unlike the regular Wi-Fi NSD functions.

The coexisting components of the server application's Main Activity are the following:

3.3.2.1. Menu List Adapter

This adapter refers to the drawer interface object from the standard Android design. It is located on the application level, since it acts as a menu for switching the fragments to be displayed on the application. Because of the simplicity of our user interface, which contains only one fragment, this adapter falls out of our scope.

3.3.2.2. Main Fragment

The Main Fragment contains all the interface elements of our application. It incorporates the functions onResume and onPause, which are called when the fragment is displayed and goes out of display respectively. When the application is pushed to background, or

when it is closed, the onPause method is called. Therefore, we incorporate the methods for starting up the networking components on the fragment's onResume function. And, to assure a proper cleanup of the network resources, we include the methods for tearing down the connections and stopping threads on the onResume function.

3.3.2.3. Application Layer Interface

The Main Activity includes an object of type Application Layer Interface, in charge of handling the logic of our queue system. In this case, the class implementing this interface is the Client Manager class, and communicates through this interface with the rest of the elements from our server application. As we mentioned before, the Client Manager executes most of the calls to the Ticket Queue Helper, the class that handles all database accesses.

3.3.2.4. Connection Layer Interface

Through this interface are managed all the functionalities that require network access, except for the Wi-Fi Direct functionalities. As depicted, we can see that it implements the interfaces to TCP, UDP and NSD communication. We will now analyse the components interacting inside the WiFi Connection Manager, which is the concrete class that implements the Connection Layer Interface.

3.3.3. CD3 - Server Wi-Fi Manager

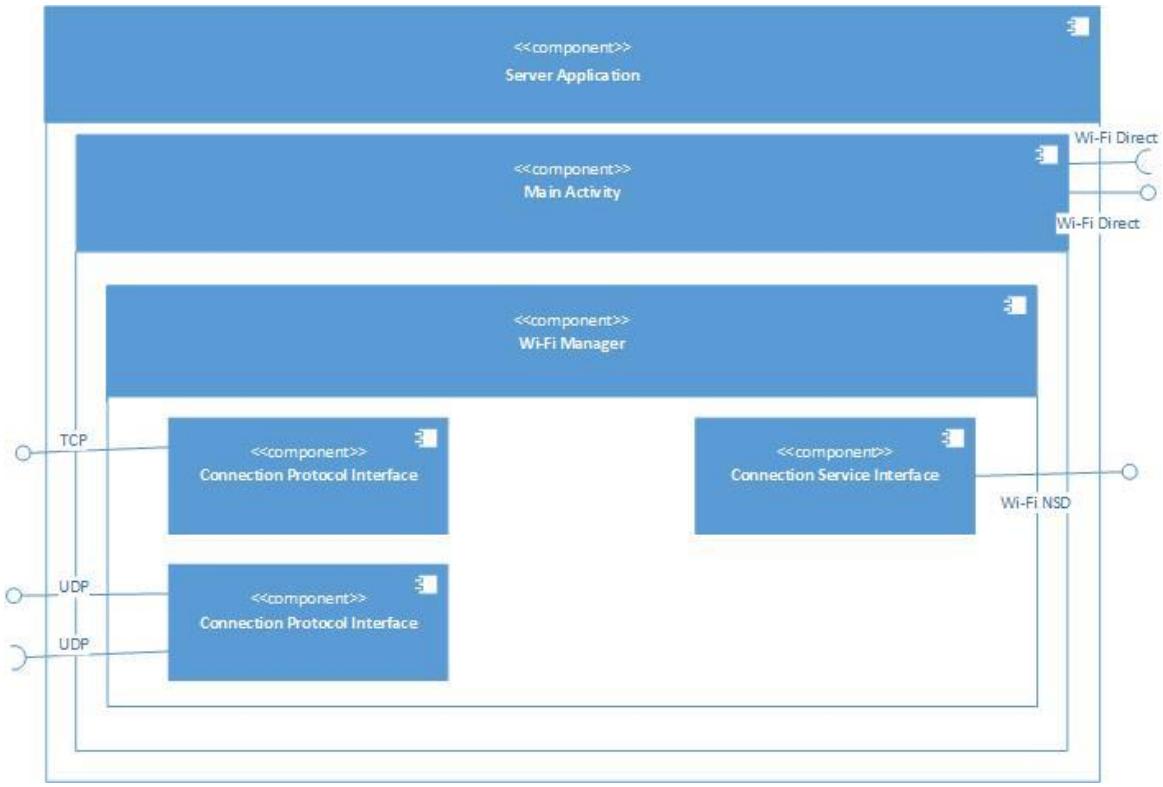


Figure 8: Component Diagram CD3 - Server Wi-Fi Manager

The Wi-Fi Manager component lays on the Main Activity, but through the Connection Layer Interface is accessed by other components, such as the Main Fragment, which commands when to start or stop all networking communication.

3.3.3.1. Connection Service Interface

This component takes care of the NSD publishing functionalities. It is modelled as an interface in order to allow the implementation of any other type of service publishing.

3.3.3.2. Connection Protocol Interface

This element takes care of receiving and transmitting messages, through any transport mechanism. In the case of our application, we incorporated one for TCP communication, and another one for UDP messages. When the Application Layer Interface wants to send a message through the Wi-Fi manager, it passes the chosen transport protocol as parameter, so the manager selects the appropriated interface.

In the case of the client application, we will begin with the structure of the Main Activity. It was not necessary to deploy any components in the Application level, since no database access were required. Accessing the components could be performed easily through the Main Activity, so we could spare one application level.

3.3.4. CD4 - Client Main Activity

Figure 9: Component Diagram CD4 - Client Main Activity

In the case of the client's Main Activity, it shares many characteristics of the server's Main Activity, but including more components to it. As in the case of the server application, the Wi-Fi Direct interfaces are located directly in the activity. The interacting components of the client application's Main Activity are the following:

3.3.4.1. Menu List Adapter

As in the case of the server application, this adapter deals with graphic interface components not implemented on this application.

3.3.4.2. Main Fragment

Similarly to the server application, the Main Fragment includes in its functions onResume and onPause the methods for starting and stopping the networking components of our application. They are accessed through the Connection Layer interface in the Main Activity.

3.3.4.3. Battery Helper

Similar to the Battery Helper defined on the server application, its accessing method returns a float with the percentage of the available battery level.

3.3.4.4. Location Helper

The Location Helper acts in a similar way from the Location Helper located in the server application. The connecting and updating parameters passed however, are different. In the case of the client application, we need to perform location requests more often, since the client is supposed to be moving, while the server should remain static. Moreover, the probabilities of a client device being charging are much lower, so there is more emphasis on preserving the battery level than on the server application's Location Helper.

A location update on the client triggers an update on the estimated waiting time, managed by the Application Layer Interface. The waiting time estimation has to take into account the distance between the server and the client's physical location.

3.3.4.5. Vibration Helper

This helper accesses the vibration mechanism of the device, and can activate it with a defined pattern. It implements the functionalities to make it start and stop vibrating, and a thread launched automatically after a certain vibration period, in order to stop the vibration.

3.3.4.6. Application Layer Interface

As in the server application, the implementation of the business logic is done through the Application Layer Interface. In this case, however, the implementing class is the Service Manager. This class handles all the actions related to the services found after the service discovery, and all the actions derived from the client-server communication.

3.3.4.7. Service Discovery Helper

This component aids managing the results of the Service Discovery. It keeps a list of the discovered services and can be accessed by any kind of discovery method we wish to implement. This component is accessed regularly by the Connection Layer Interface, since it needs the discovery results in order to handle the queues from each service.

3.3.4.8. Connection Layer Interface

Analogously, the functionalities that require network access are handled by the Connection Layer Interface, except for the Wi-Fi Direct functionalities, handled by the Main Application directly. It implements the interfaces to TCP, UDP and NSD communication, but in contrast with the server application, it only implements TCP and NSD as clients. The interacting components inside the WiFi Connection Manager, which implements the Connection Layer Interface, will be described below.

3.3.5. CD5 - Client Wi-Fi Manager

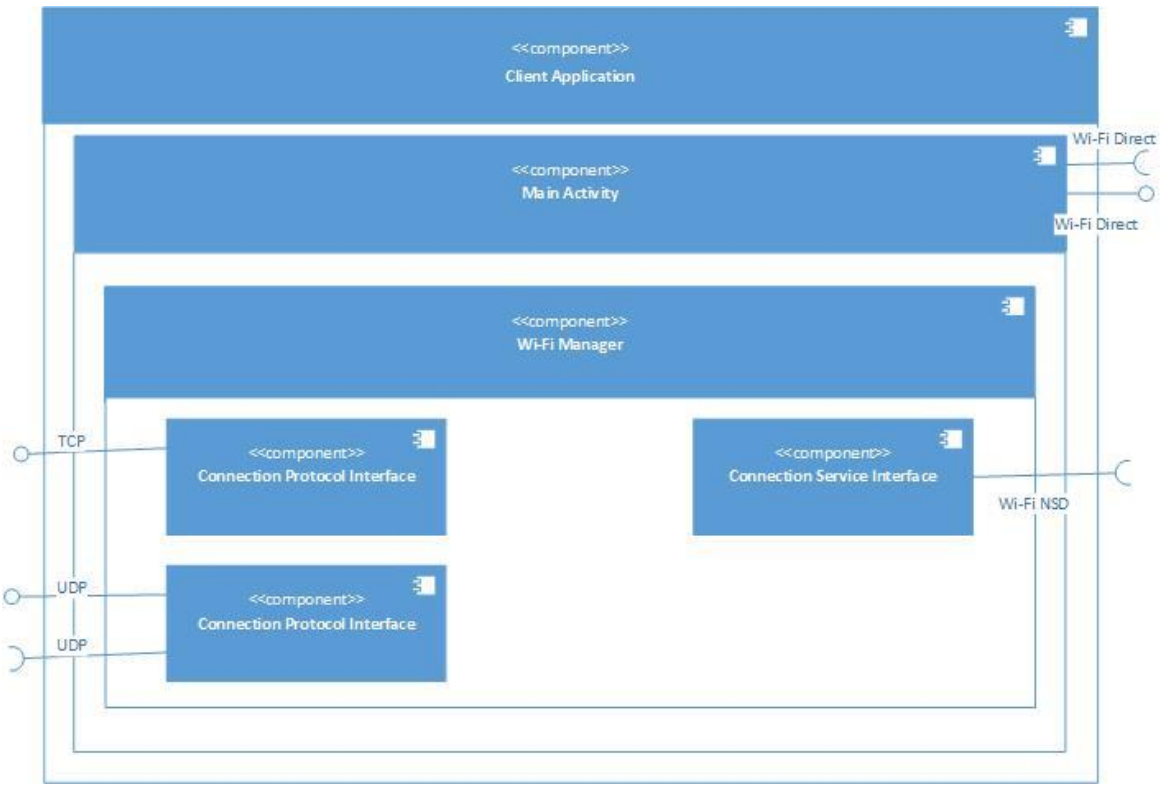


Figure 10: Component Diagram CD5 - Client Wi-Fi Manager

The Wi-Fi Manager component behaves very similarly to its analogue component in the server application. There are however, a few differences worth noticing:

3.3.5.1. Connection Service Interface

In the service application, this component is triggered by the user interface and starts the service discovery. The found services are kept in the Service Discovery Helper. In order to preserve the battery, the service discovery is run for a few seconds every time. Once the services are found, there is no need to call the service discovery again.

3.3.5.2. Connection Protocol Interface

As in the server application, it comprehends two instances, one for message transport via TCP and another one for UDP. However, the TCP instance only acts as a client, it can perform requests and read responses, but it cannot accept isolated requests.

3.4. Sequence Diagrams

The following sequence diagrams describe the different interactions performed by the application components. The components in our sequence diagrams may be interfaces, thus keeping the interactions simple.

3.4.1. SE1 – General Application Flow

This diagram displays the general application flow and interactions on a high level. More specific diagrams will follow.

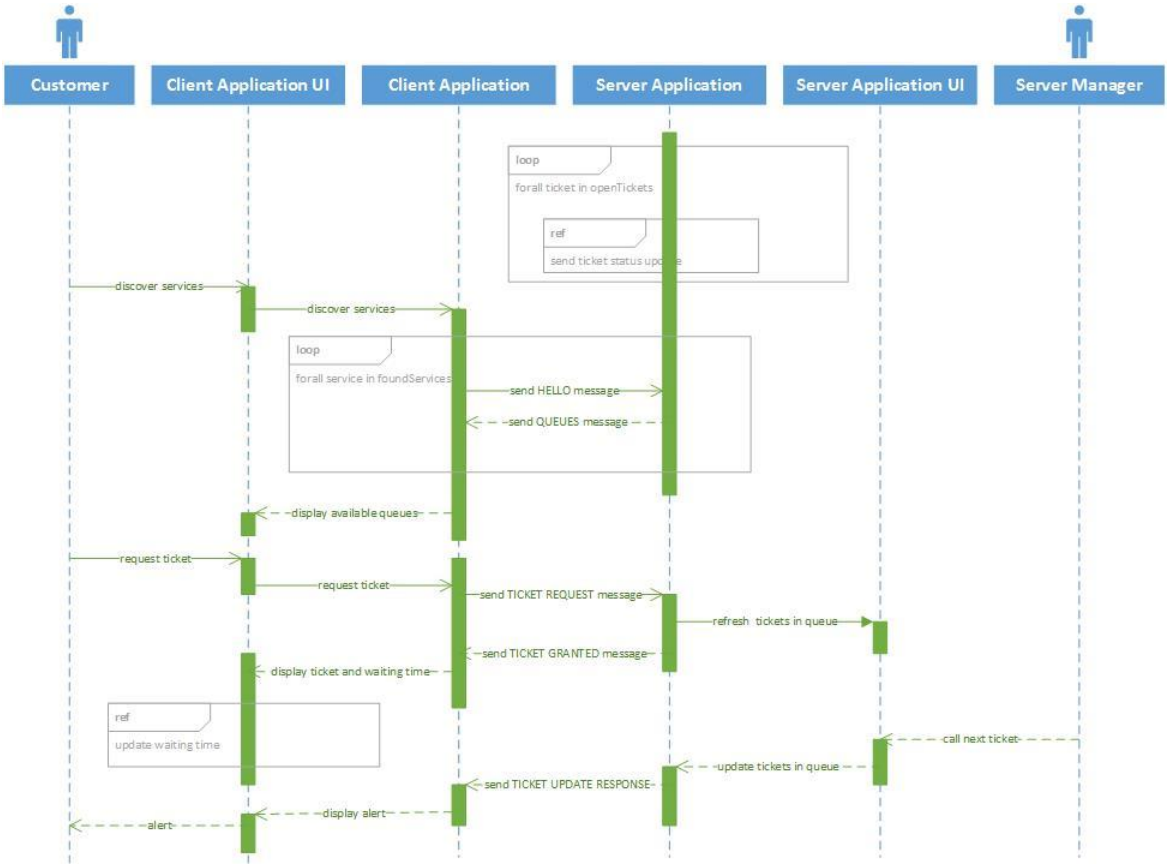


Figure 11: Sequence Diagram SE1 - General Flow

3.4.2. SE2 - Service Request (Client)

This process begins when the client performs a NSD lookup, and finished after finding the available services, when they send their queues information. The queue information is processed, and if there are open tickets corresponding to the client, a ticket status update will be requested for each open ticket, and a waiting time update task will be open.

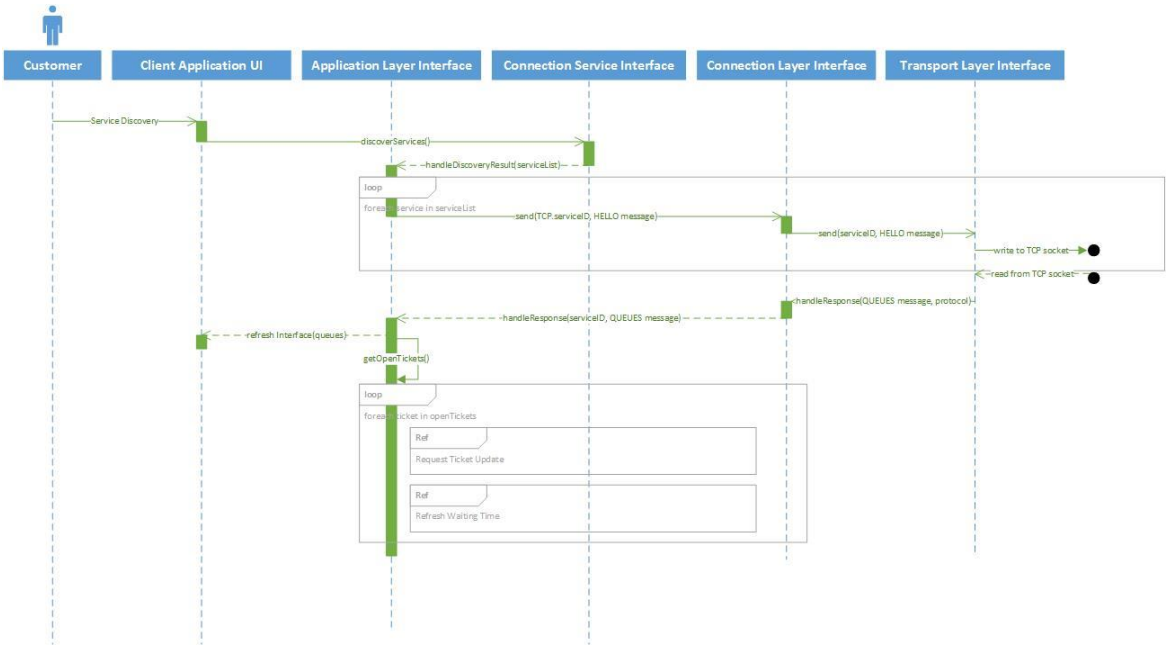


Figure 12: Sequence Diagram SE2 - Service Request

3.4.3. SE3 - Ticket Update Request (Client)

The ticket update request is fairly simple and straightforward. It is triggered by the Client Manager for each open ticket after getting the list of available queues in a service, and by the location manager after each location update. It is sent using UDP, so it does not wait for a response.

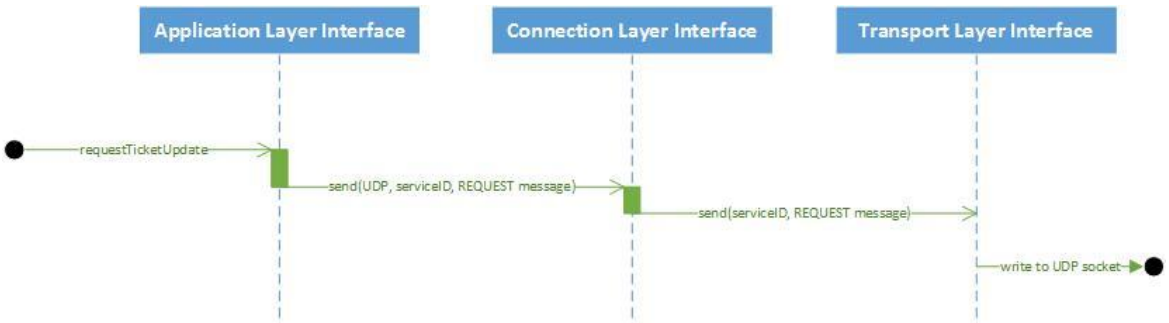


Figure 13: Sequence Diagram SE3 - Ticket Update Request

3.4.4. SE4 - Refresh Waiting Time (Client)

This process is triggered for any open tickets assigned to the client. It constantly updates the estimated waiting time and reflects it on the user interface. Additionally, every asynchronous ticket status message received from the server application carries a waiting estimation update, so this information is kept up-to-date. The update runs until an update is received that sets the waiting time to -1, which means that the ticket has been closed.

Figure 14: Sequence Diagram SE4 - Refresh Waiting Time

3.4.5. SE5 - Service Request (Server)

The following illustrates the steps performed by the server once the client has found the published service and has sent a HELLO message.

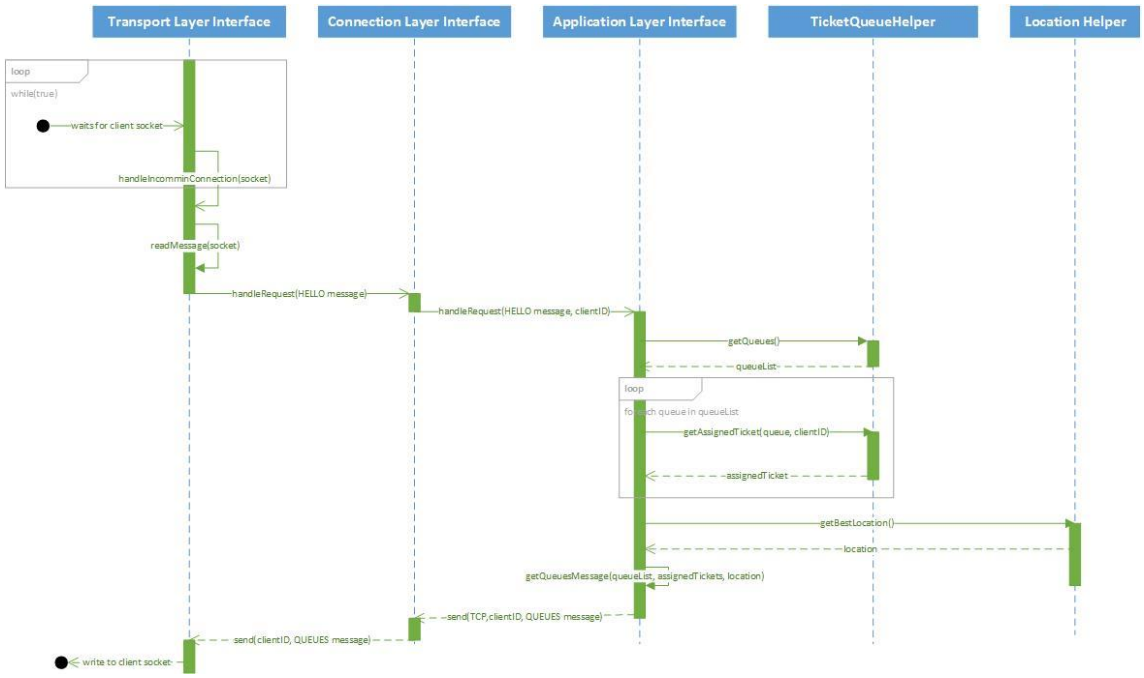


Figure 15: Sequence Diagram SE5 - Service Request

3.4.6. SE6 - Ticket Request (Client)

After receiving the list of queues from the available service, the client is able to request a ticket for any of the queues. The ticket request action triggers the following sequence of events.

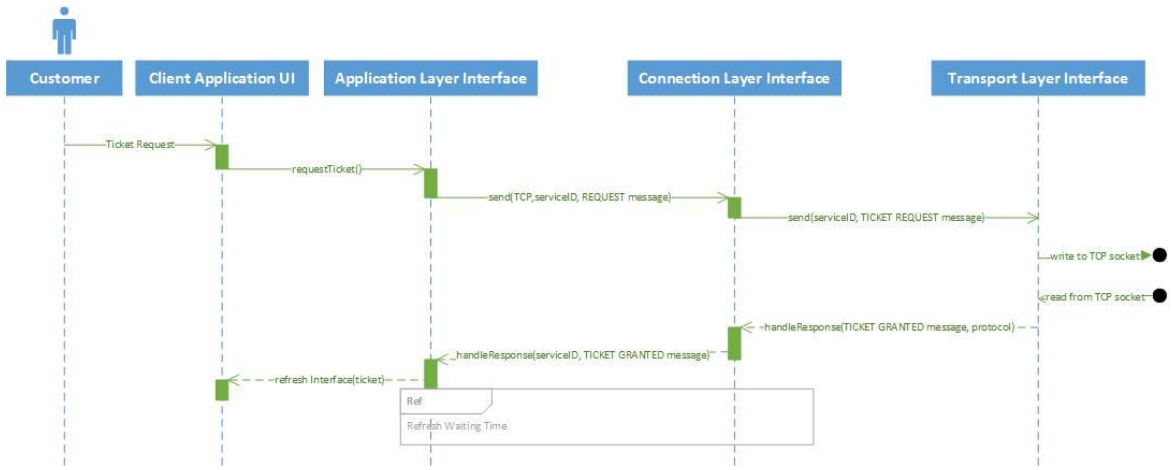


Figure 16: Sequence Diagram SE6 - Ticket Request

3.4.7. SE7 - Ticket Granted (Server)

Upon receiving a ticket request, the server application issues a ticket for the requesting client, and sends it back. The server application also updates its graphic user interface, in order to display the number of tickets that have been issued for each queue. It is worth mentioning that, if the client had already been assigned a ticket and it remains open, the requestNumber function will not generate a new ticket, and will return the old one instead.

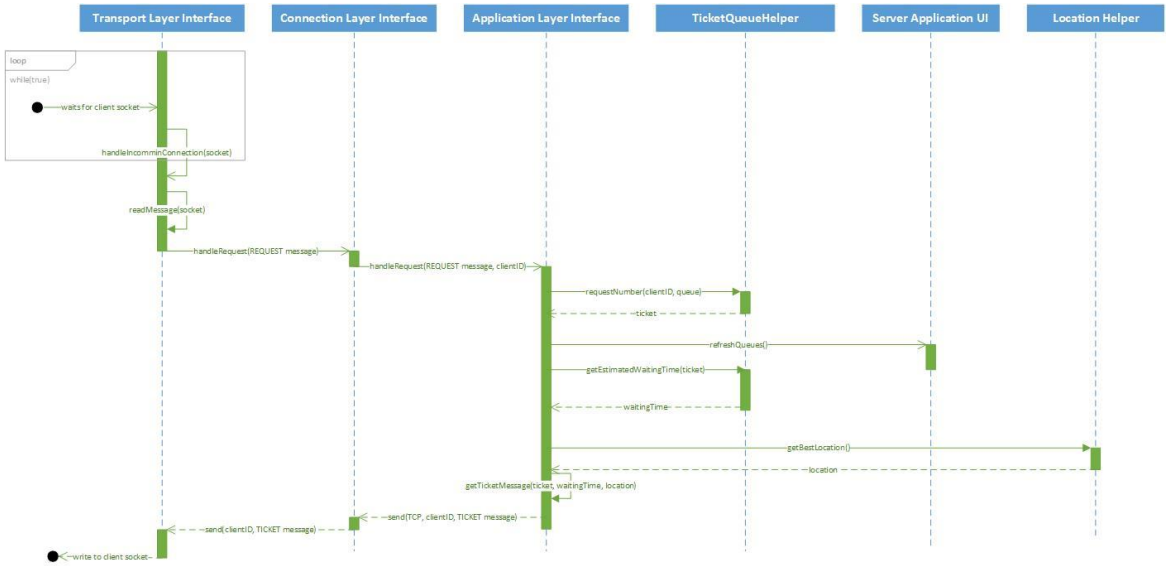


Figure 17: Sequence Diagram SE7 - Ticket Granted

3.4.8. SE8 - Ticket Update Response (Server)

Upon receiving a ticket status request message from the client (represented in the sequecence diagram SE2), the server generates a status response, carrying the open ticket information, and it is sent through the sendStatusUpdate function.

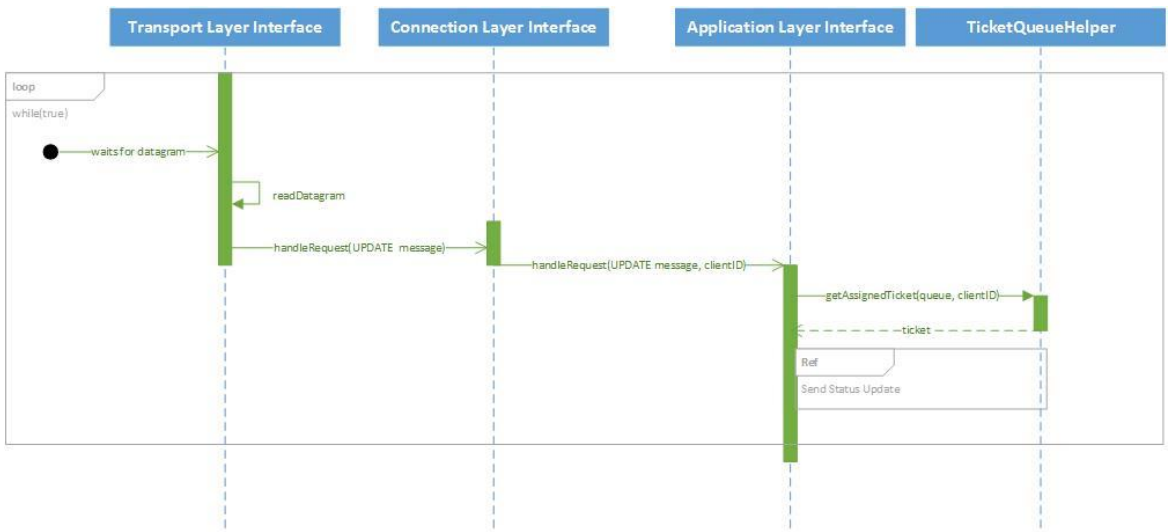


Figure 18: Sequence Diagram SE8 - Ticket Update Response

3.4.9. SE9 - Send Status Update (Server)

Sending a status update message can be triggered by a specific client request, as the mentioned in the sequence diagram SE5, or by the Location Helper after a location update. There is also a thread running on the server application, specifically on the Client Manager, which generates status messages regularly for all the clients with open tickets.

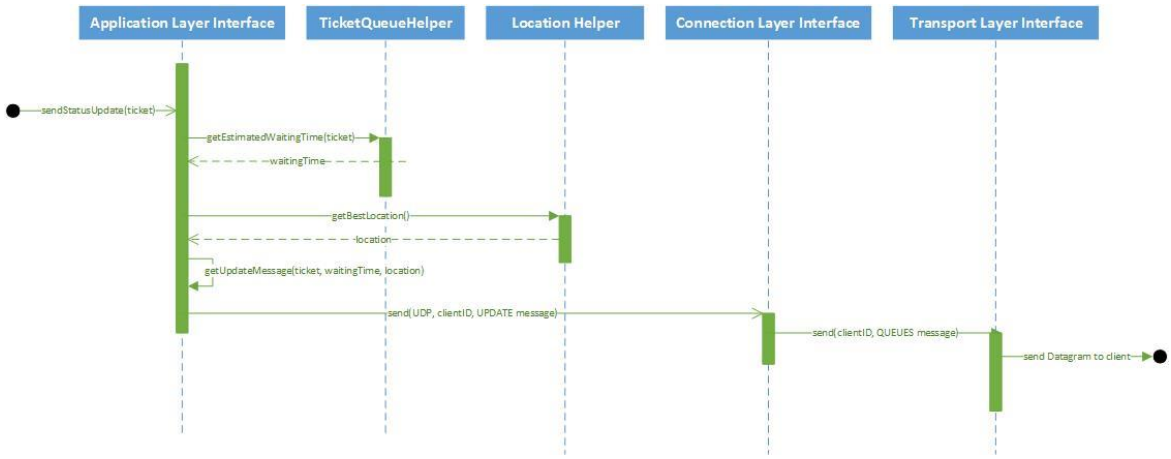


Figure 19: Sequence Diagram SE9 - Send Status Update

3.4.10. SE10 - Ticket Status Received (Client)

After receiving a ticket status response from the server, the client application handles it and updates the waiting time from the related ticket.

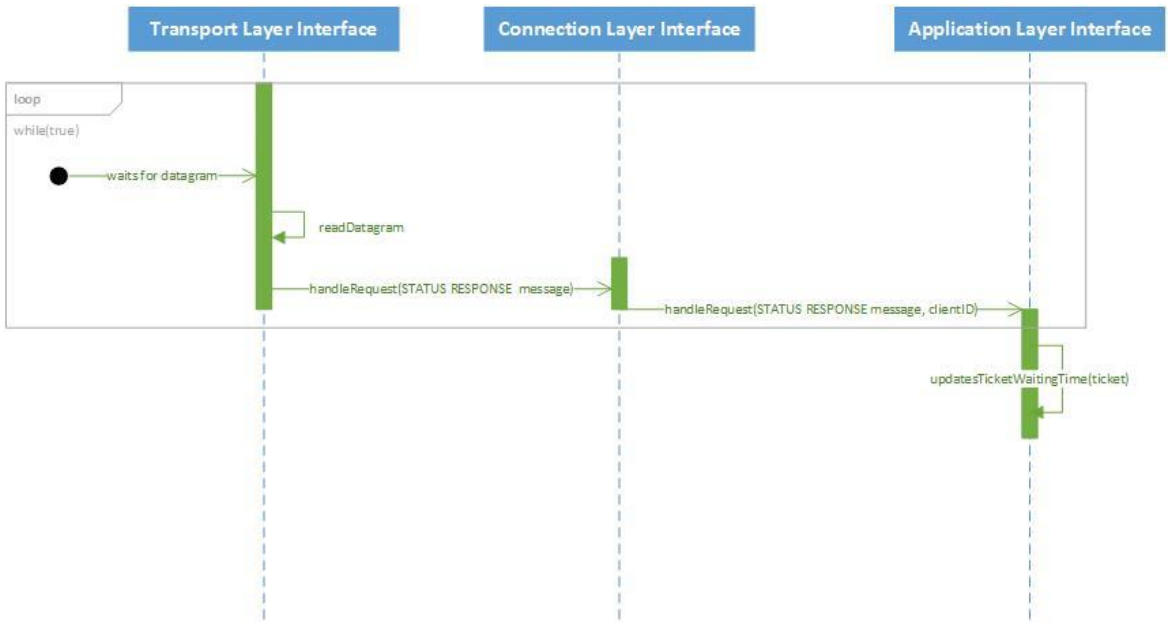


Figure 20: Sequence Diagram SE10 – Ticket Status Received

3.4.11. SE11 – Call Next Ticket (Server)

Through the user interface, the server application provides the functionality to close a ticket and call the next ticket on the queue. This process is exemplified below, and it terminates sending update messages to both the recently closed ticket and the ticket to be called.

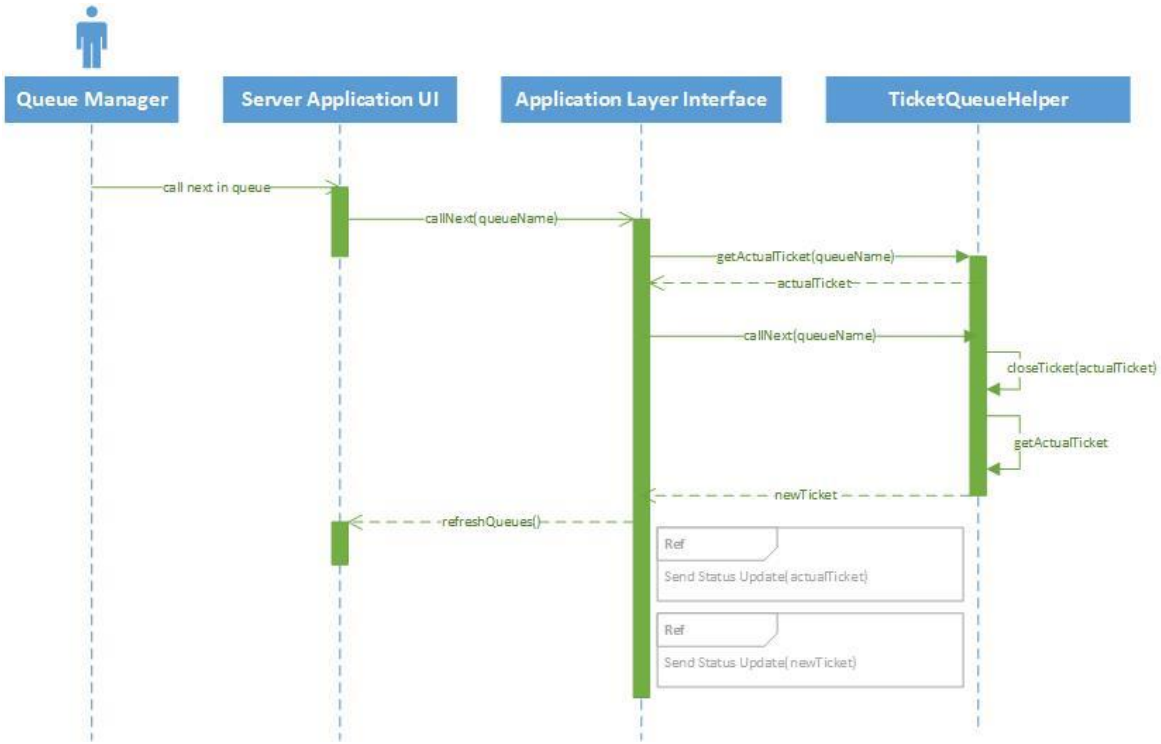


Figure 21: Sequence Diagram SE11 - Call Next Ticket

3.5. Class Diagrams

For further description of our system, we include the class diagrams of the interfaces developed for the system internal communication, and the diagrams of the transport objects developed, which store the information to be passed among classes.

3.5.1. Interfaces

3.5.1.1. Transport Layer Interface

This interface is common for both the server and the client application. It is implemented by all the supported transport protocols. Any other transport protocols could be incorporated to the system as long as they implement this interface.

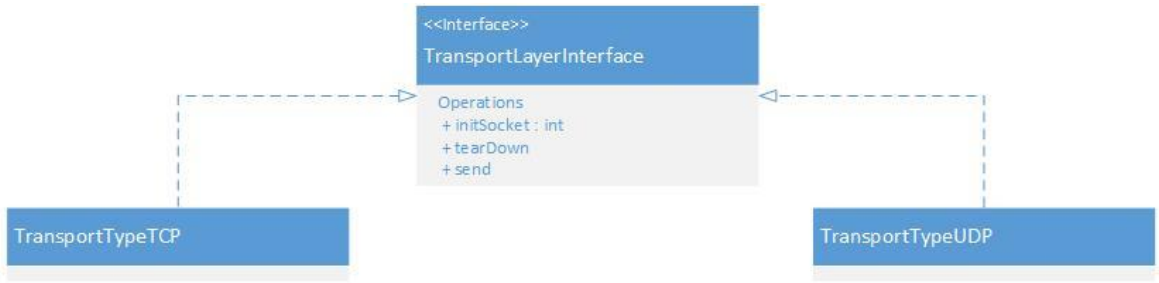


Figure 22: Class Diagram - Transport Layer Interface

3.5.1.2. Connection Layer Interface (Server)

This is the interface that encapsulates the networking functionalities. Even though it is implemented in both the server and the client application, there are some differences on their methods. The interface describes basically the send and receive functions, acting as an intermediary between the TransportLayerInterface and the ApplicationLayerInterface. Besides the init and teardown methods, some functions return the ServiceConnectionData object, which will be explained accordingly. Since the NSD service may change the requested service name, the getWiFiServiceName method returns the final name assigned name.

Figure 23: Class Diagram - Connection Layer Interface (Server)

3.5.1.3. Connection Layer Interface (Client)

This interface differs from the server application interface on the last two methods, since they relate to handle the server responses, while on the server side they handle the client requests.

Figure 24: Class Diagram - Connection Layer Interface (Client)

3.5.1.4. Application Layer Interface (Server)

On both the server and client applications, this interface encapsulates the logic from our queuing system. The setConnectionLayer is a setter for the concrete object implementing

our Connection Layer Interface. On the server side it handles the request messages from the client, and prepares the appropriated responses.

3.5.1.5. Application Layer Interface (Client)

On the client application, besides the server response handler, it incorporates methods for starting and stopping the service discovery, and to process the discovery results.

3.5.1.6. Connection Service Interface (Server)

This interface is implemented by the publishing mechanism used in the service discovery process. Besides the init and teardown methods, it incorporates methods for starting and stopping the service publishing mechanism.

Figure 25: Class Diagram - Connection Service Interface (Server)

3.5.1.7. Connection Service Interface (Client)

On the client application, this interface defines the methods for the service discovery process, and for setting the helper in charge of handling the discovery results.

Figure 26: Class Diagram - Connection Service Interface (Client)

3.5.2. Transport Objects

3.5.2.1. Service Connection Data

This object is used to store and share connection data from both the server and the client applications. Each application uses it for its own connection data, which is incorporated to the sent messages, and for the connection data of other parties.

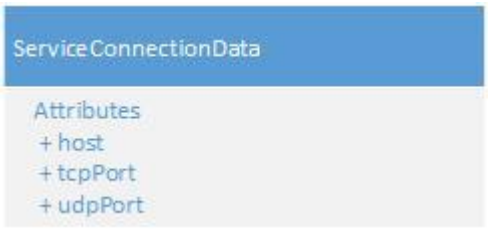


Figure 27: Class Diagram - Service Connection Data

3.5.2.2. Communication Message

This message is the core of our communication mechanism between the client and the server application. There are four types of shared messages, ad kajshdf kajs dfkahs dkfasd. The series of attributes defined on the message may vary depending on the message type. The message attributes are:

- Client ID: Unique ID from the communicating client.
- Service ID: Unique ID that differentiates the different available services.
- Service Name: The name chosen by the service to be shown on the applications' user interfaces.
- Type: The message type can have the following values:
    - TYPE_HELLO: Sent by the client to the found services after performing the service discovery.
    - TYPE_QUEUES: Sent by the server in reply to the HELLO message, it includes information about the available queues, and whether or not the client already had tickets assigned to any of it.
    - TYPE_TICKET_REQUEST: Sent by the client in order to request a ticket on a specific queue.
    - TYPE_TICKET_GRANTED: Sent by the server with the granted ticket information, as response to the TICKET_REQUEST. The ticket may be a new ticket, or the already assigned ticket, if any.
    - TYPE_TICKET_UPDATE: Sent by the client for requesting the current status of a determined ticket.
    - TYPE_TICKET_UPDATE_RESPONSE: Sent by the server, either asynchronously as a regular update, or as response to a TICKET_UPDATE message.
- Queue Names: The list of queues available on the service.
- Queue Tickets: The ticket numbers available for each queue ticket on the queue names list. The queue names and tickets are not sent as a hashmap but in separated queues because of the limitations of Gson, the tool used for translating the object to a JSON string.
- Queue ID: The identifier of the specific queue, to which the message relates.

- Waiting Seconds: The estimated seconds for the assigned ticket to be called. This ticket is related to the message destination client, and the queue identified on the queue ID.
- Lat and lng: Location information from the sending party.
- Host Address, TCP and UPD port: Host and connection ports from the sending party.

Figure 28: Class Diagram - Communication Message

3.5.2.3. Queue Object (Server)

This object is one of the elements persisted on the database by ORM Lite. It includes the relevant queue information, the list of assigned tickets (also ORM Lite persisted objects) and a private id attribute related to the database row id.

Figure 29: Class Diagram - Queue Object (Server)

3.5.2.4. Queue Object (Client)

On the client application, the relevant queue information includes the queue name, the ticket number assigned on that queue, if any; the estimated time for the client head toward the server application's location in order to arrive before the ticket is called, and an alert flag to enable or disable the alerting mechanisms, through the user interface or the vibration mechanisms.

Figure 30: Class Diagram - Queue Object (Client)

3.5.2.5. Ticket (Server)

This object is also persisted on the database using ORM Lite. As well as the Queue Object, includes the respective database row ID. It also contains the related queue to this ticket, the ID of the client to which it was assigned, the ticket number, creation and closing dates, and the status. This status can be TICKET_OPEN or TICKET_CLOSED.

Figure 31: Class Diagram - Ticket (Server)

3.5.2.6. Client Object (Server)

The object for sharing the client information is very simple, it incorporates only its unique ID and its geographical location.

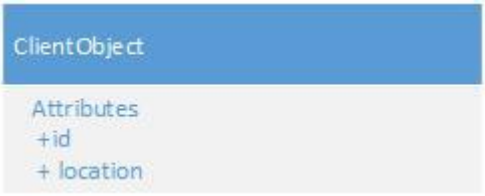


Figure 32: Class Diagram - Client Object (Server)

3.5.2.7. Service Object (Client)

On the client application, this object includes the name that the service has chosen to be displayed on the user interface, the unique service ID, and a list of the queues available for that service.

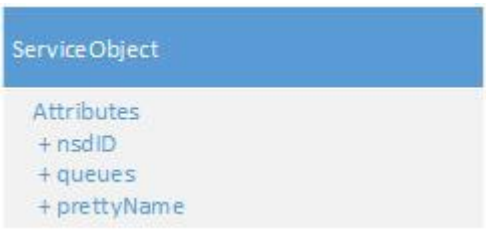


Figure 33: Class Diagram - Service Object (Client)

3.6. Database Model

The following scheme represents the database structure on the server side.

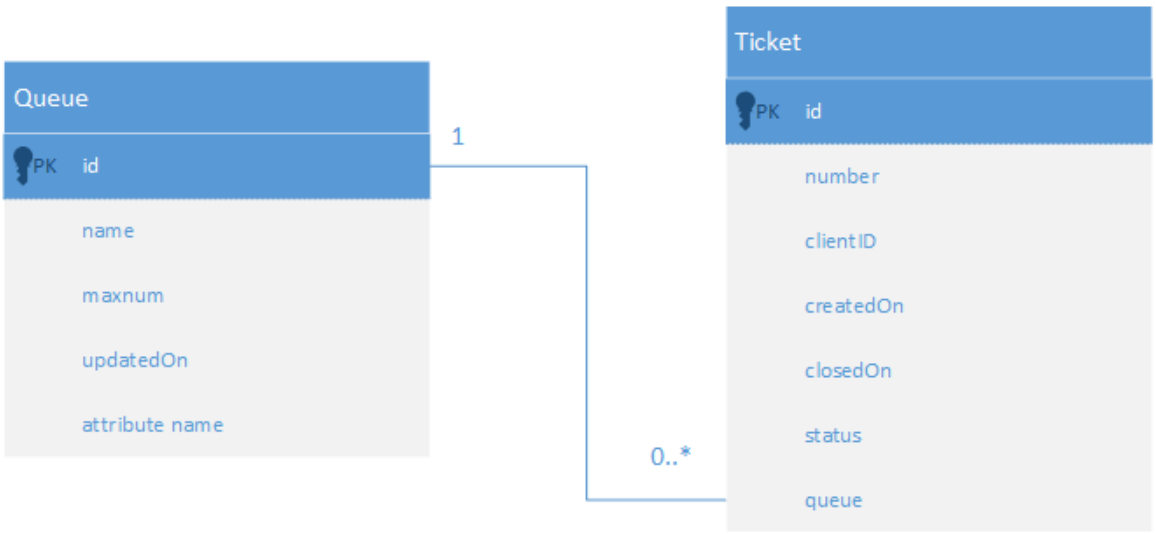


Figure 34: Database Model

On the client side, the client unique ID was generated (if it was not present before), and kept using SharedPreferences.

4. Results

We were able to implement successfully a queuing mechanism consisting of a server application interacting with several client applications. Through this we reached our primary goal, which was providing a communication structure for ticket handling in a queue system. The following aspects are worth emphasizing, they are implementation results, decisions, and encountered problems.

4.1. Queuing System

The system was successfully developed, including the required functionalities. Through these screenshots we will display the final result.

Here we can see the server application, with the two available queues and the number of tickets requested on each queue. By clicking on the queue name button, the next number from the queue will be called. Below is the logging information.

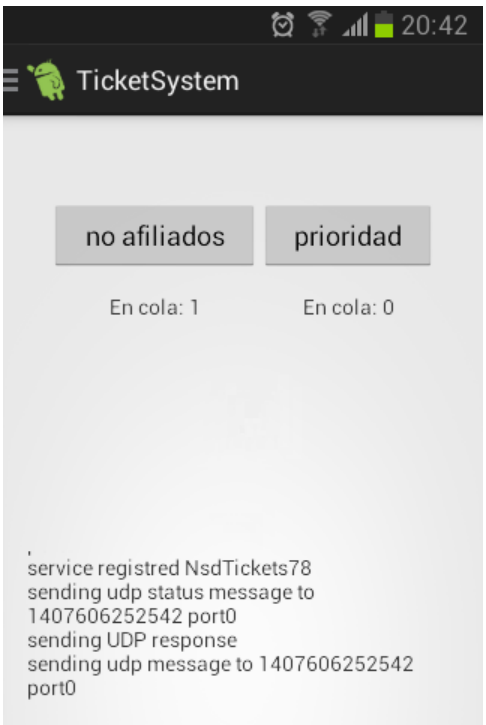


Figure 35: Server Application Screenshot

This image represents the client application initial screen, before running the service discovery.

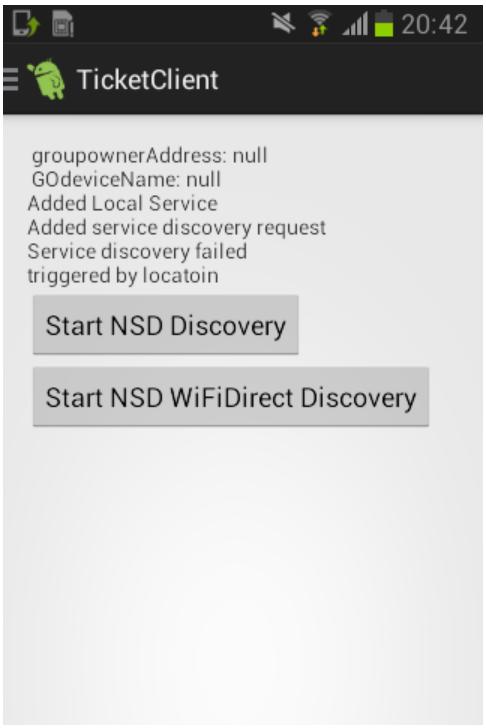


Figure 36: Client Application - Service Discovery Screenshot

In this screenshot are displayed the two available queues from the found services. The log is displaying part of the received message, where the queues information was sent.

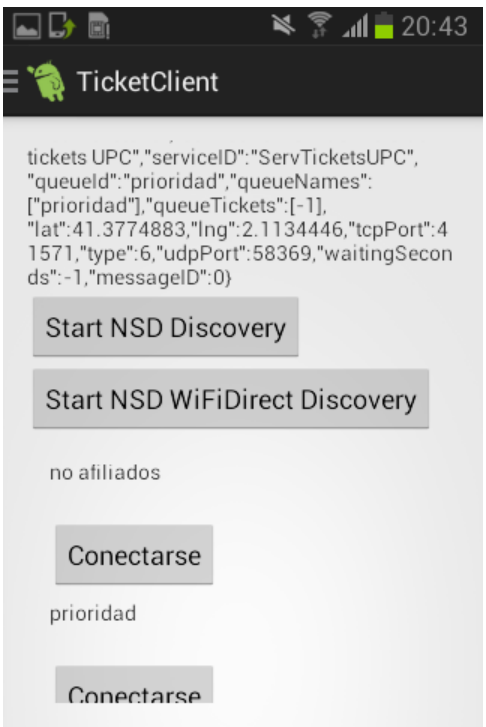


Figure 37: Client Application - Available Queues Screenshot

After requesting a ticket, the user interface would change in order to display the ticket number assigned and the waiting time.

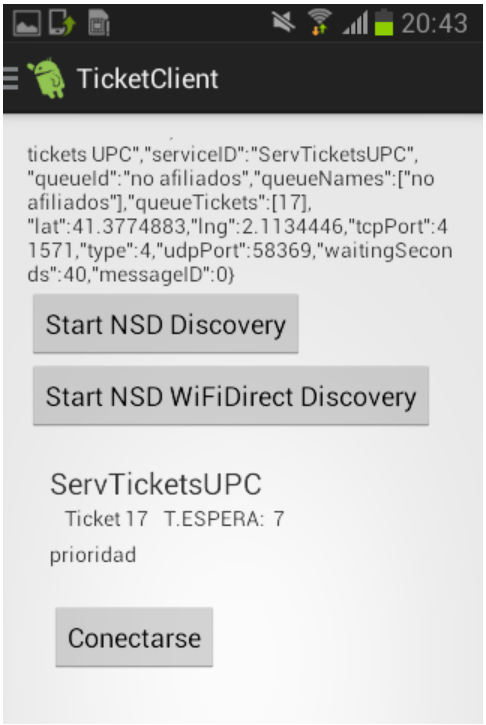


Figure 38: Client Application - Ticket Assigned Screenshot

Once the waiting time goes to 0, the device starts vibrating alerting the user. The following image displays the alert moment, where the user has the option of disabling the vibration.

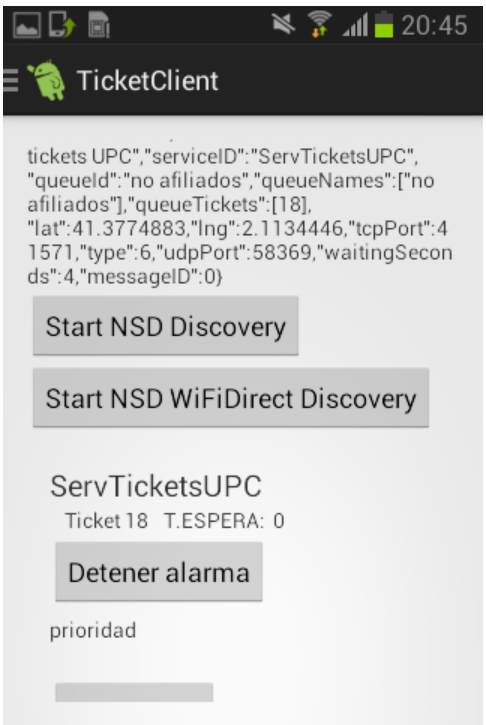


Figure 39Client Application - Alarm Screenshot

4.2. Application Structure

The implementation of Android Annotations, and the design of the different components in a modularized way, allowed us to have independent services performing specific actions. For the hardware-related functionalities we developed the VibrationHelper, LocationHelper, and BatteryHelper, which run independently from other components. Following their architecture, other components can be developed and incorporated easily.

The following is the BatteryHelper class, where we can see the simplicity and high mantainability.

```
@EBean
public class BatteryHelper {

	@RootContext
	Context mContext;

	private Intent batteryStatus;

	@AfterInject
	protected void init(){
		if(batteryStatus == null){
			IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
			batteryStatus = mContext.registerReceiver(null, ifilter);
		}
	}

	public float getBatteryStatus(){
		if (batteryStatus == null){
			init();
		}
		int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
		int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
		float batteryPct = level / (float)scale;

		return batteryPct;
	}
}
```

We can appreciate that not even a constructor is needed. After being injected to the class incorporating it, the init() function is called. However, if another component tries to access the battery helper before being injected, the init() function will make sure that the resources are available.

As we have discussed in the component's model, the BatteryHelper lays, on the server application, inside the application class. Here we display how the services are injected into the class.

```
@EApplication
public class MyApplication extends Application {

	@Bean
	public TicketQueueHelper ticketHelper;

	@Bean
	public BatteryHelper batteryHelper;
```

```
        @Bean
        public LocationHelper locationHelper;

}
```

This is the only code required to incorporate those enhanced classes, or @Bean.

Analogously, for grouping specific functionalities that did not necessarily have to do with hardware access, we implemented the use of interfaces. They can be instantiated from the code, and gives us the desired flexibility to interchange or add new components. This is the case of the transport protocol implemented, under the TransportLayer interface. Having the transport layer encapsulated allows the queuing system to work on top of any transport protocol implementing those interfaces. The application also has the capability of adapting the message transmission to a certain transport protocol in runtime, depending on the network conditions or the message importance.

This is partial code of the Wi-Fi Manager, which includes the classes in charge of TCP and UDP communication:

```
public class WifiConnectionManager implements ConnectionLayerInterface {

        @RootContext
        Context mContext;

        @RootContext
        MainActivity activity;

        private ConnectionServiceInterface wifiService;

//      private ConnectionService wifiP2PService;

        private TransportLayerInterface tcpConnection;

        private TransportLayerInterface udpConnection;

        private ServiceConnectionData ownConnData;

        private HashMap<String, ServiceConnectionData> destConnData;

        private ApplicationLayerInterface appLayer;



        /****************** STARTUP ***********************/
        @AfterInject
        public void init(){
                ownConnData = new ServiceConnectionData();
                destConnData = new HashMap<String, ServiceConnectionData>();

                tcpConnection = ConnectionTypeTCP_.getInstance_(mContext);
                udpConnection = ConnectionTypeUDP_.getInstance_(mContext);
                wifiService = NSDWifi_.getInstance_(activity);
//              wifiP2PService = NSDWifiP2P_.getInstance_(activity);
        }

...
```

This way we provide an architecture which allows alternative components to be included, or new components to be added, thus letting our application grow in a maintainable fashion.

The implementation of different transport protocols running in parallel also allowed us to find an Android limitation when it comes to parallel threading. This thesis can also act as a reference for other projects that need to deal with parallel threading and multiple services.

4.3. Data Management

For storing the required data by the application, on the client side we chose to store the client ID using SharedPreferences, since it is a simple, constant field. When the client ID was not present, a new ID was created using the following algorithm:

```
private String generateUniqueID(SharedPreferences sharedPref){
        SharedPreferences.Editor editor = sharedPref.edit();

        Long tsLong = System.currentTimeMillis()/1000;
        Random r = new Random();
        int random = r.nextInt(10000);
        String myUniqueID = tsLong+""+random;
        editor.putString(storage_uniqueid, myUniqueID);
        editor.commit();
        return myUniqueID;

}
```

By concatenating the current time millis with a random number up to 10.000, we guarantee the uniqueness of the ID among clients.

On the server side we had the need to use a relational database, since the stored data was more complex and could grow large. We chose to keep only the information related to queues and tickets, and not clients, since they were the objects created by the server side. The only relevant information from the client is the ID, so we implemented it as a field on the ticket object.

While using Android SQLite, even though we used interfaces to communicate with the Android SQLiteOpenHelper, the class extending it grew very large, and got more complicated to maintain over time. Here is an example of the code required to query for a ticket assigned to a certain client, on a certain queue:

```
private Ticket getAssignedTicket(QueueObject queue, ClientObject client) {
        SQLiteDatabase db = getReadableDatabase();

        // Define a projection that specifies which columns from the database
        // you will actually use after this query.
        String[] projection = {
                        TicketEntry._ID,
                        TicketEntry.COLUMN_NAME_ASSIGNED_NUMBER,
        };

        // How you want the results sorted in the resulting Cursor
```

```
        String sortOrder = TicketEntry.COLUMN_NAME_ASSIGNED_ON + DBUtils.DESC;

        String selection = TicketEntry.COLUMN_NAME_QUEUE + DBUtils.EQ_VAR
        + DBUtils.AND + TicketEntry.COLUMN_NAME_CLIENT + DBUtils.EQ_VAR
        + DBUtils.AND + DBUtils.OPEN_PRNT
        + TicketEntry.COLUMN_NAME_STATUS + DBUtils.EQ + DBUtils.QUOTE + TicketEntry.STATUS_OPEN + DBUtils.QUOTE
        + DBUtils.OR + TicketEntry.COLUMN_NAME_STATUS + DBUtils.EQ + DBUtils.QUOTE + TicketEntry.STATUS_ALERT +
DBUtils.QUOTE
        + DBUtils.CLOSE_PRNT;;

        String[] selectionArgs = {
                queue.getId()+"",
                client.getId()
        };

        Cursor c = db.query(
                TicketEntry.TABLE_NAME,             // The table to query
                projection,                               // The columns to return
                selection,                                // The columns for the WHERE clause
                selectionArgs,                            // The values for the WHERE clause
                null,                                     // don't group the rows
                null,                                     // don't filter by row groups
                sortOrder                                 // The sort order
        );

        try {
                if (c.getCount() > 0) {
                        if (c.moveToFirst()) {
                                Ticket ticket = new Ticket();
                                ticket.setId(c.getLong(0));
                                ticket.setNumber(c.getInt(1));
                                ticket.setQueue(queue);
                                ticket.setClientID(client.getId());
                                return ticket;
                        }
                }

        }
        catch (Exception e){

        }
        finally {
                c.close();
                db.close();               }

        }

        return null;

}
```

As we can see, the class is not only complex but also error prone, considering that we need to access a raw cursor by its index, which contains the query result. We also force a casting over the returned object at the index position. It is also worth noticing that the cursor and the database accessing object have to get closed by the end of the method, otherwise an exception is thrown on the next time we query the database.

Consequently, we decided to try the ORMLite framework. After little configuration and adding annotations to the objects to be persisted, the same method could be expressed with this much simpler piece of code:

```
public Ticket getAssignedTicket(String queueName, String clientName){

        try {
                QueryBuilder<QueueObject, Long> queueQb = queueDao.queryBuilder();
                SelectArg nameSelectArg = new SelectArg();
                queueQb.where().eq(TicketEntry.COLUMN_NAME_QUEUE, nameSelectArg);

                QueryBuilder<Ticket, Long>ticketQb = ticketDao.queryBuilder();
                SelectArg statusSelectArg = new SelectArg();
                SelectArg clientSelectArg = new SelectArg();
                ticketQb.where().eq(TicketEntry.COLUMN_NAME_STATUS, statusSelectArg)
                        .and().eq(TicketEntry.COLUMN_NAME_CLIENT, clientSelectArg);

                // then you set the args and run the query
```

```
                nameSelectArg.setValue(queueName);
                statusSelectArg.setValue(Ticket.STATUS_OPEN);
                clientSelectArg.setValue(clientName);
                Ticket result = ticketQb.join(queueQb).queryForFirst();
                if (result != null) {
                        return result;
                }
        } catch (SQLException e) {
                e.printStackTrace();
        }

        return null;
}
```

ORMLite requires a text file with the database structure in order to prepare the tables and perform the conversions between relational database and the java objects. The library, however, includes a generator for this file. For our persisted objects, the generated file is the following:

```
# --table-start--
dataClass=com.android.upc.object.Ticket
tableName=ticket
# --table-fields-start--
# --field-start--
fieldName=id
generatedId=true
# --field-end--
# --field-start--
fieldName=number
canBeNull=false
# --field-end--
# --field-start--
fieldName=clientID
canBeNull=false
# --field-end--
# --field-start--
fieldName=createdOn
canBeNull=false
# --field-end--
# --field-start--
fieldName=closedOn
# --field-end--
# --field-start--
fieldName=status
useGetSet=true
# --field-end--
# --field-start--
fieldName=queue
canBeNull=false
foreign=true
# --field-end--
# --table-fields-end--
# --table-end--
#################################
```

```
# --table-start--
dataClass=com.android.upc.object.QueueObject
tableName=queue
# --table-fields-start--
# --field-start--
fieldName=id
generatedId=true
# --field-end--
# --field-start--
fieldName=name
# --field-end--
# --field-start--
fieldName=maxnum
# --field-end--
# --field-start--
fieldName=updatedOn
# --field-end--
# --field-start--
fieldName=tickets
foreignCollection=true
# --field-end--
# --table-fields-end--
# --table-end--
#################################
```

4.4. Communication Protocol

4.4.1. Retransmissions

One of the most relevant aspects in wireless communication is the retransmission mechanisms, because of the high error rate. In the case of the NSD discovery procedures, the discovery time lasts 5 seconds, in order to preserve the battery. If after that time no services have been found, another discovery will be performed, with a small discovery time increase, and this will be repeated up to 5 times. There are two reasons for this: one is that we have found that the discovery network service may take longer than 5 seconds in some cases, and the other reason is because the user may be in motion, moving from a low coverage area. The service does not stop automatically after finding a service, since there may be more than one service available.

```
@Override
public void discoverServices() {
        if (!discoveryEnabled){
                appLayer.onDiscoveryStart();
                appLayer.getFoundServicesConnData().clear();
                discoveryEnabled = true;
                int stopDiscoveryTime =  discoveryMillis+discoveryStep*reconnectAttempt;
                mNsdManager.discoverServices(
                        SERVICE_TYPE_TCP, NsdManager.PROTOCOL_DNS_SD, mDiscoveryListener);
                discoveryHandler.postDelayed(stopDiscoveryRunnable, stopDiscoveryTime);
        }
}
```

```
@Override
public void onDiscoveryStopped(String serviceType) {
        if (appLayer.getFoundServicesConnData().isEmpty()){
                reconnectAttempt++;
                if (reconnectAttempt == MAX_RECONNECT_ATTEMPTS){
                        reconnectAttempt = 0;
                }
                else {
                        activity.alert("No found services. We will discover again");
                        discoverServices();
                }
        }
        else {
                reconnectAttempt = 0;
                activity.alert("Discovery stopped: " + serviceType+". Found
"+appLayer.getFoundServicesConnData().size()+" services");
                appLayer.handleDiscoveryResult(appLayer.getFoundServicesConnData().keySet());
        }
}
```

In the case of TCP communication, we also implemented a retransmission mechanism in case that the client would receive an invalid response.

```
	@Override
	public void send(String serviceNSId, CommMessage message) throws NullSocketException,
SocketIOException, MessageRetransmissionException {
		String msg = null;

		ServiceConnectionData destConnData = myConnection.getDestConnData(serviceNSId);

		if (destConnData!= null && destConnData.getHost() != null &&
destConnData.getTcpPort() != 0){
			Socket clientSocket = getSocket(destConnData.getHost(),
destConnData.getTcpPort());

			if (clientSocket != null){
				Gson gson = new Gson();
				String data = gson.toJson(message);

				try {
					OutputStream os = clientSocket.getOutputStream();
					PrintWriter writer = new PrintWriter(os);
					InputStreamReader streamReader = new
InputStreamReader(clientSocket.getInputStream());
					BufferedReader reader = new BufferedReader(streamReader);

					writer.println(data);
					writer.flush();
					msg = reader.readLine();

					writer.close();
					os.close();
					reader.close();
					streamReader.close();
				} catch (IOException e) {
					e.printStackTrace();
					throw new SocketIOException(e);
				}
			}
			else {
				throw new NullSocketException("TCP Client Socket is null");
			}
			handleTCPResponse(serviceNSId, msg, message);
		}
	}


	private void handleTCPResponse(String serviceNSId, String response, CommMessage message)
throws MessageRetransmissionException, NullSocketException, SocketIOException {
		if (response != null) {
			myConnection.handleResponse(response, TransportLayerInterface.TYPE_TCP);
			serviceRetries.put(serviceNSId, MAX_RESEND_TIMES);
		}
		else {
			// we got a null response from a lingering service
			if (!serviceRetries.containsKey(serviceNSId)){
				// ignored
			}
			else{
				int retries = serviceRetries.get(serviceNSId);
				retries--;
				if (retries == 0){
					serviceRetries.put(serviceNSId, MAX_RESEND_TIMES);
					throw new MessageRetransmissionException("Message sending
keeps getting NULL response after "+MAX_RESEND_TIMES+" tries.");
				}
				else {
					serviceRetries.put(serviceNSId, retries);
					send(serviceNSId, message);
```

```
                    }
                }
            }
        }
```

The retransmission would be peformed MAX_RESEND_TIMES, in our case was limited to five times.

On the handleTCPResponse function, there is a comment making reference to “lingering service”. We will discuss this further, since it was one of the most challenging problems faced in the development process.

4.4.2. Transport protocol implementation

While we successfully managed to implement communication through TCP and UDP, we ran into important issues during the process. The final TCP implementation is in section 4.7.2.

As for the structure used, it is important to mention that, for avoiding multithreading problems in Android, we ran our TCP server on a single thread, assigning a new thread for every incoming connection. However, in the case of UDP, since a communication is not stablished, only a single thread is created to listen and process the incoming messages. For UDP transmitting purposes, a thread using AsyncTask is created for each datagram. These threads have a very short life span, since they end after the transmission is done.

4.5. Data Exchange

Our proposed communication protocol is based on the exchange of six different message types. It is independent of the chosen transport protocol, simple and easy to understand. The creation and handling of the messages was delegated to a single class on each application: on the server side they are created on the ClientManager, and on the client side they are generated on the ServiceManager. No other classes create or modify their structure. In order to guarantee the consistency of the messages, they are not reused in any case. Every time the application needs to send a message, a new instance is created with all the needed information depending on the message type. The reason for that strict behaviour in the creation and modification of the communication messages, is to keep the communication protocol as modularized as possible, making it transparent to all the other components.

4.6. Estimated waiting time

Every time a ticket is issued or a status update is sent, we need to estimate the waiting time for that ticket. First the serving time from closed tickets is calculated, which is the difference between closing dates of two consequent tickets. This average should get

more precise over time, as more tickets are served. This average serving time is then multiplied by the number of orders ahead of the requested ticket. The elapsed time since the ticket was requested is then subtracted from the total estimated waiting time, and sent to the client. It is worth mentioning that at this time the client location has not been taken into account. We delegate this functions to the client application, so the precise estimation does not depend on the network condition.

```
private int estimateTime(int ordersToGo, int orderWaitingTimeAvg, Date createdOn) {
        // time since the actual ticket was called
        long elapsedTimeMillis = 0;
        Date today = new Date();
        elapsedTimeMillis = today.getTime() - createdOn.getTime();
        int timePassedSinceAssigned = (int) elapsedTimeMillis/1000;

        int fullQueueWaitingTime = orderWaitingTimeAvg * (ordersToGo+1);
        int secondsToWait = fullQueueWaitingTime - timePassedSinceAssigned;
        secondsToWait = (secondsToWait < 0) ? 0 : secondsToWait;
        return secondsToWait;

}
```

On the client side, a mechanism is performed for estimating the waiting time based on the received values from the server. While a network connection is needed for the initial message exchanges, once a ticket has been issued the client application is autonomous enough to maintain an update of the waiting time depending on the device physical location. The follwing code is run every second while there are open tickets. It subtracts one second from the QueueObject waiting time, and refreshes the UI.

```
@UiThread
protected void refreshWaitingTime(final QueueObject queue){
        // Handlers need to be created on main thread
        final Handler handler = new Handler();
        Runnable timeRefreshRunnable = new Runnable() {
                @Override
                public void run() {
                        int waitingTime = queue.getWaitingTime()-1;
                        queue.setWaitingTime(waitingTime);

                        if (waitingTime > 0) {
                                handler.postDelayed(this, UPDATE_UI_SECONDS*1000);
                        }
                        else {
                                activity.vibrationHelper.alert(queue);
                                queue.setAlert(true);
                        }
                        activity.serviceHelper.refreshServicesList();
                }
        };
        handler.post(timeRefreshRunnable);

}
```

However the same object is also updated from other sources. If a ticket update message arrives, the new waiting time is updated. And when a location update is fired, the queue object is also updated. In the case of the Client Application location updates are triggered every 30 seconds for the network provider, and once a minute for the GPS provider,

every time the user moves a distance greater than 10 meters. With those metrics we are able to preserve the battery level without compromising the required location information updates.

```
private void startNWUpdates(){
		if (!locationRequestNWStarted){
			locationRequestNWStarted = true;
			// Register the listener with the Location Manager to receive location
updates
			locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,
UPDATE_FREQ_NW, UPDATE_DIST_NW, locationListener);
		}
	}

	private void startGPSUpdates(){
		if (!locationRequestGPSStarted && activity.batteryHelper.getBatteryStatus() >
MINIMUM_BATTERY_FOR_GPS) {
			locationRequestGPSStarted = true;
			// Register the listener with the Location Manager to receive location
updates
			locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
UPDATE_FREQ_GPS, UPDATE_DIST_GPS, locationListener);
		}

	}
```

Being that the walking pace is approximately 1.3 meters/second, we estimate the time that would take the client to reach to the server's location, and we substract it from the QueueObject waiting time.

```
	private int timeEstimation(float meters) {
		// walking pace = 1.3 meters/sec
		return (int) Math.ceil(meters/1.3f);

	}
```

This way, our structure not only allows a correct queue management, but also can function on areas where the network coverage is not always optimal.

4.7. Faced Problems

4.7.1. NSD lingering services

While performing tests on the Android devices for testing the NSD communication, we ran into a problem on both the server and the client application. The testing process lead to the generation of many NSD services, and in some cases the application crashed before performing the appropriated removal of the published service. This resulted into services showing up on the client application after performing the discovery process, which were not available anymore. Since the client application had no way of knowing which service was active and which one was lingering, it would try to communicate with it. We had to include a special case on the TCP send method in order not to wait for a

response in those cases. The only way to get rid of those lingering results would be by physically restarting both devices.

The server application was affected by this problem in a different way. Apparently, there is a limit on the different names that the NSD can assign to a service. If there is already a service, “nsdTicketService” for example, then another service that wants to be published with this name, will be renamed to “nsdTicketService(2)”, and so on. However, as more services remained lingering on the NSD system, the service would eventually stop being published. This error did not throw any kind of exceptions, it would simply not call the listener in charge of handling the publish results.

Since we suspect that it has to do with the service renaming process, we added a random number appended to the service name in order to avoid this problem.

```
@AfterInject
protected void init(){
        Random r = new Random();
        int random = r.nextInt(100);
        mServiceName += random;

}
```

It would not cause any incompatibilities with the client, since the clients would locate the services by a substring search, taking into account that the NSD service could also append extra information.

```
@Override
                        public void onServiceFound(NsdServiceInfo service) {
                                // A service was found!  Do something with it.
                                if (!service.getServiceType().equals(SERVICE_TYPE_TCP)) {
                                        // Service type is the string containing the protocol and
                                        // transport layer for this service.
                                } else if (service.getServiceName().contains(mServiceName)) {
                                        mNsdManager.resolveService(service, mResolveListener);
                                }
                                else {
                                        activity.alert("service discarded:
"+service.getServiceName());
                                }

                        }
```

Still, the server application would occasionally not connect to the NSD service, and the only way around this problem was to uninstall the application and perform a restart on the device.

4.7.2. Servers and threads

Since we were not using external servers in our application, we had to implement the TCP and UDP servers ourselves. One of the approaches we considered was implementing them as Android services. The downside for this approach is that these services may be killed by the system at any given time, and we needed more reliability. We then proceeded to use AsyncTasks with the help of Android Annotations, by running

the server in a function with the @Background annotation. This approach worked effectively, until our application grew more complex. There is a limitation on the number of concurrent threads that Android can run, which is very low. Background AsyncTasks may run sequentially depending on the available resources. The result of this approach was that we would have our server application stuck on an endless, silent blocking call. The server would stop at serverSocket.accept() when receiving an incoming call, but was unable to start the required thread for handling that connection. The function for handling the new connection would only be called after killing the server, thus liberating the thread.

While the AsyncTask approach is correct for short tasks, we found that for long running procedures a different approach should be taken, and that affected not only the implementation of the server, but the way we would generate new threads for the incoming calls.

The correct approach of our TCP server had to implement its own thread pool, to accept a limited amount of incoming clients. If a client would arrive after the thread pool was empty, it would be placed on hold until a thread was freed.

```
private class NetworkService implements Runnable {
        private final ServerSocket serverSocket;
        private final ExecutorService pool;
        public NetworkService(ServerSocket serverSocket, int poolSize) {
                this.serverSocket = serverSocket;
                pool = Executors.newFixedThreadPool(poolSize);
        }

        @Override
        public void run() {
                try {
                        while(keepRunning) {
                                pool.execute(new Handler(serverSocket.accept()));
                        }
                } catch (IOException ex) {
                        pool.shutdown();
                }
        }

        public void teardown(){
                shutdownAndAwaitTermination(pool);
        }

        private void shutdownAndAwaitTermination(ExecutorService pool) {
                pool.shutdown(); // Disable new tasks from being submitted
                try {
                        // Wait a while for existing tasks to terminate
                        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
                                pool.shutdownNow(); // Cancel currently executing tasks
                                // Wait a while for tasks to respond to being cancelled
                                if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
                                        System.err.println("Pool did not terminate");
                                }
                        }

                } catch (InterruptedException ie) {
                        // (Re-)Cancel if current thread also interrupted
                        pool.shutdownNow();
                        // Preserve interrupt status
                        Thread.currentThread().interrupt();
                }
        }
}
```

The NetworkService thread incorporates the ServerSocket, and waits for incoming connections. A handler would be generated from the server pool, which will call the followup methods to deal with the incoming client, in this case the function handleIncomingConnection.

```
private class Handler implements Runnable {
        private final Socket runnableSocket;

        Handler(Socket socket) {
                this.runnableSocket = socket;
        }
        @Override
        public void run() {
                handleIncomingConnection(runnableSocket);
        }
}
```

In order to start our TCP Server Socket, we would startup the NetworkService object in a thread on startup.

```
protected void startSocketReader(){
        tcpNetworkService = new NetworkService(tcpSocket, MAX_CONCURRENT_CLIENTS);
        Thread networkThread = new Thread() {
                @Override
                public void run() {
                        tcpNetworkService.run();
                }
        };
        networkThread.start();

}
```

4.7.3. Wi-Fi Direct NSD

In our application, we intended to incorporate Wi-Fi Direct NSD. We developed the required interfaces to deal with the peer connections, and the service structure necessary for publishing and discovering services through any system.

```
@Receiver(actions = WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION)
        protected void onP2PStateChangedAction(Intent intent) {
                // Determine if Wifi P2P mode is enabled or not
                int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);
                if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
                        setWifiP2pEnabled(true);
                } else {
                        setWifiP2pEnabled(false);
                }
        }

        // result of the discovery action performed before
        @Receiver(actions = WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION)
        protected void onP2PPeersChangedAction() {
                // The peer list has changed!
                serviceConnHelper.getWifiP2PService().requestPeers();
        }


        // result of the connect action performed before
        @Receiver(actions = WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION)
        protected void onP2PConnectionChangedAction(Intent intent) {
```

```
                NetworkInfo networkInfo = (NetworkInfo)
intent.getParcelableExtra(WifiP2pManager.EXTRA_NETWORK_INFO);
                if (networkInfo.isConnected()) {
                        // We are connected with the other device, request connection info to find
group owner IP
                        serviceConnHelper.getWifiP2PService().requestConnectionInfo();
                }

        }

        @Receiver(actions = WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION)
        protected void onP2PDeviceChangedAction(Intent intent) {
                WifiP2pDevice device =
intent.getParcelableExtra(WifiP2pManager.EXTRA_WIFI_P2P_DEVICE);

        }
```

However, we were unable to get our devices to connect with each other. Even though the notification for device pairing was displayed and properly accepted, they would still show an error on the pairing process. Even though we followed guidelines and multiple examples of how to get Wi-Fi Direct to properly work, we could not accomplish this task in the required time.

5. Conclusions and future development

We were able to develop the required ticket queuing system application, and we could use it as a way to explore Android development and networking functionalities. While we faced some problems with the internetworking processes, we were able to sort most of them out, thus learning from the process.

This thesis could act as a reference on queue management, it includes the required basic components and functionalities necessary, as well as data structures and persistence methods. It also acts as a reference on communication protocols between a server and multiple clients, through the use of custom messages. The internetworking is another key feature from our project, in the sense of implementing different low-level communication protocols, and providing guidelines for NSD publishing and service discovery. The way our project is structured could not only be a reference on modular architecture, but also allows further development.

We recommend the implementation of the framework AndroidAnnotations, in conjunction with ORM Lite. Through these frameworks we have been able to provide a simple, reusable code that is easy to understand and maintain.

In our experience, we found the regular NSD service over Wi-Fi easier to implement than Wi-Fi Direct NSD. Besides the aforementioned problems of making our devices more prone to restart themselves, and not getting it to work properly, we found that for a client-server infrastructure they may not be the most appropriated technology to implement.

As for future development, however, a Wi-Fi Direct structure which would be very interesting to incorporate but as a way of getting clients to communicate among them. This could allow the inclusion of projects such as Wi-Fi Direct multi-hop messaging protocols, as a way of routing messages to unreachable clients.

Other communication mechanisms provided by Android could also be implemented due to the project structure. This includes Bluetooth, NFC, and any other communication technologies.

Decisions based on the battery level could also be incorporated to our routing mechanisms, since we already have the structures for it. This information could also be passed from the clients to the server, in order to increase or reduce the update message rate. The graphic interface could also be greatly improved in order to make it appealing to the user, and some other features such as notifications could be implemented.

It would be interesting to study a service approach for this application, and allowing the necessary capabilities to run in background. However, it has to be analysed carefully in order to react when Android system kills the service.

Bibliography

[1] Marl Wilson. "T-Mobile G1: Full Details of the HTC Dream Android Phone". *Gizmodo*, 2008. [Online] Available: http://gizmodo.com/5053264/t-mobile-g1-full-details-of-the-htc-dream-android-phone. [Accessed: 10 July 2014].

[2] Charles Arthur. "Android is winning – if you're writing apps for China. Elsewhere, though…". *The Guardian*, 2012. [Online] Available: http://www.theguardian.com/technology/appsblog/2012/aug/16/android-winning-apps-china-smartphone. [Accessed: 10 July 2014].

[3] Lisa Mahapatra. "Android Vs. iOS: What's The Most Popular Mobile Operating System In Your Country?". *International Business Times*, 2013. [Online] Available: http://www.ibtimes.com/android-vs-ios-whats-most-popular-mobile-operating-system-your-country-1464892. [Accessed: 10 July 2014].

[4] ComTech. "Smartphone OS market share". *Kantar Worldpanel*, 2014. [Online] Available: http://www.kantarworldpanel.com/global/smartphone-os-market-share/about-comtech. [Accessed: 10 July 2014].

[5] ABC Madrid. "El número de líneas móviles en España cae por primera vez". *ABC*, 2013. [Online] Available: http://www.abc.es/economia/20130212/abci-telefonia-movil-perdida-lineas-201302121234.html. [Accessed: 10 July 2014].

[6] Android. "Android KitKat". *Android*, 2014. [Online] Available: https://developer.android.com/about/versions/kitkat.html. [Accessed: 10 July 2014].

[7] Android. "Connecting Devices Wirelessly". *Android*, 2013. [Online] Available: http://developer.android.com/training/connect-devices-wirelessly/index.html [Accessed: 01 December 2013].

[8] Android. "Connectivity". *Android*, 2013. [Online] Available: http://developer.android.com/guide/topics/connectivity/index.html [Accessed: 01 December 2013].

[9] WaitKnowMore. " FAQ for customers ". *WaitKnowMore*, 2013. [Online] Available: http://www.waitknowmore.com/customers [Accessed: 01 December 2013].

[10] Sunstar. " WaitKnowMore mobile app wins Startup Weekend Cebu ". *Sunstar*, 2012. [Online] Available: http://www.sunstar.com.ph/breaking-news/2012/05/14/waitknowmore-mobile-app-wins-startup-weekend-cebu-221339 [Accessed: 01 December 2013].

[11] QLess. "Why QLess". *QLess*, 2013. [Online] Available: http://qless.com/why/better-customer-service/ [Accessed: 01 December 2013].

[12] Qminder. "Mobile App". *Qminder*, 2013. [Online] Available: http://www.qminderapp.com/features/ [Accessed: 01 December 2013].

[13] Android. "Dashboards". *Android*, 2014. [Online] Available: http://developer.android.com/guide/topics/connectivity/index.html [Accessed: 01 July 2014].

[14] Android. "Design Philosophy". *Android*, 2013. [Online] Available: http://developer.android.com/guide/components/fragments.html#Design [Accessed: 01 December 2013].

[15] Android. "Storage Options". *Android*, 2013. [Online] Available: http://developer.android.com/guide/topics/data/data-storage.html [Accessed: 01 December 2013].

[16] Android. "Using Network Service Discovery". *Android*, 2014. [Online] Available: http://developer.android.com/training/connect-devices-wirelessly/nsd.html [Accessed: 01 February 2014].

[17] Wi-Fi.org. "Wi-Fi Direct". *Wi-Fi Alliance*, 2014. [Online] Available: http://www.wi-fi.org/discover-wi-fi/wi-fi-direct [Accessed: 01 February 2014].

[18] Alexandre THOMAS. "Android Kickstartr". *Github*, 2012. [Online] Available: http://androidkickstartr.com/ [Accessed: 01 February 2014].

[19] Android. "Support Library". *Android*, 2013. [Online] Available: http://developer.android.com/guide/topics/connectivity/index.html [Accessed: 01 December 2013].

[20] Pierre-Yves Ricau. "Android Annotations". *Github*, 2014. [Online] Available: https://github.com/excilys/androidannotations [Accessed: 01 February 2014].

[21] Gray Watson. "ORMLite – Lightweight Object Relational Mapping (ORM) Java Package". *ORMLite*, 2014. [Online] Available: http://ormlite.com/ [Accessed: 01 February 2014].

[22] SQLite. "Features of SQLite". *SQLite*, 2014. [Online] Available: http://www.sqlite.org/features.html [Accessed: 01 February 2014].

[23] Google. "Google-Gson". *Google code*, 2013. [Online] Available: https://code.google.com/p/google-gson/ [Accessed: 01 December 2013].

Glossary

NSD: Network Service Discovery.

FIFO: First In First Out

P2P: Peer-to-peer

UI: User Interface

TCP: Transmission Control Protocol

UDP: User Datagram Protocol