Location based Social Network

Tiago Alberto Gorgulho Gonçalves

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Professor João Paulo Baptista de Carvalho

Examination Committee

Chairperson: Professor Nuno Cavaco Gomes Horta
Supervisor: Professor João Paulo Baptista de Carvalho
Member of the Committee: Professor David Manuel Martins de Matos

April 2014
Acknowledgments

Throughout this project, there were several people who supported me and made possible the development of this thesis. Thus, I dedicate this space to all those who, in one-way or another, spent time, patience and knowledge, allowing me to complete this stage of my life.

First, I would like to thank Professor João Paulo Carvalho the opportunity to work in an area of my special interest, recognizing all his support, availability and assertiveness, which allowed me always to keep focused in an area where creativity can also be an obstacle to the ultimate goal.

A deep gratitude to all my colleagues and friends who have always been by my side throughout this stage of my life and who contributed to the completion of my course successfully.

Finally, I want to let a special acknowledgment to my parents for all the support, confidence and advices that have always given me, and have been essential in my career over the years.
Abstract

Nowadays, technology plays an increasingly important role in our society, in particular in the telecommunications world. With a smartphone and all the technology available in it, namely internet connection and location services, people can now stay connected and communicate with each other when and where they want. For those reasons, mobile applications have increased their market and are now being used for every day common tasks, like gaming, chatting, social networking, location based information, etc.

This Thesis proposes a new concept for an application, gathering three of the most common features in only one mobile app: chat, social network and location services. The challenge is to develop an application where people can talk with their contacts, like in a chat app, maintain an online profile to meet new people, like in a social network and share their geographic position to find contacts near by, using the technology of gps on their smartphones.

The main focus here is to supply a set of tools, available on a smartphone that enables users to interact and communicate with each other more efficiently.

The idea is to have a fully operational mobile version of the application, supporting all these required features in an elegant and intuitive fashion.

Keywords

iOS
Location Services
Mobility, Social Network
Chat
Mobile Application
Web Sockets
Resumo

Hoje em dia, a tecnologia desempenha um papel cada vez mais importante na nossa sociedade, em particular no mundo das telecomunicações. Com um smartphone e toda a tecnologia disponível nele, como por exemplo, ligação à Internet e serviços de localização, as pessoas podem agora estar conectadas e comunicar entre elas quando e onde quiserem. Por esses motivos, as aplicações móveis têm vindo a aumentar o seu mercado e são agora usadas para as mais diversas tarefas diárias, como jogar, conversar, navegar nas redes sociais, fornecer serviços baseados na localização, etc.

Esta tese propõe um novo conceito para uma aplicação, reunindo três das características mais comuns numa só aplicação: chat, rede social e serviços de localização. O desafio é desenvolver uma aplicação onde as pessoas podem falar com seus contatos, como um chat, manter um perfil online para conhecer novas pessoas, como numa rede social e partilhar a sua posição geográfica para encontrar contatos por perto, utilizando a tecnologia gps dos seus smartphones.

O foco principal aqui é fornecer um conjunto de ferramentas, disponíveis num smartphone, permitindo aos utilizadores interagir e comunicar uns com os outros de forma mais eficiente.

A idéia é ter uma versão móvel plenamente operacional da aplicação, fornecendo todas estas características de uma forma elegante e intuitiva.

Palavras-chave

iOS
Serviços de Localização
Mobilidade
Rede Social
Chat
Aplicação Móvel
Web Sockets
Contents

Acknowledgments ........................................................................................................... iii

Abstract ......................................................................................................................... v

Resumo ......................................................................................................................... vii

List of Figures ............................................................................................................... xi

List of Tables ............................................................................................................... xiii

List of Acronyms ......................................................................................................... xv

1. Introduction ............................................................................................................. 1
  1.1 Context and Motivation ....................................................................................... 1
  1.2 Objectives ........................................................................................................... 2
  1.3 Original Contributions ....................................................................................... 2
  1.4 Thesis Outline .................................................................................................... 3

2. Mobile Application Development ......................................................................... 5
  2.1 Mobile Platforms ............................................................................................... 5
  2.1.1 Native versus Hybrid versus Web Applications ............................................ 6
  2.2 Development Tools .......................................................................................... 7
    2.2.1 Client-Side Development Tools .................................................................... 7
      2.2.1.1 IOS SDK ................................................................................................ 7
      2.2.1.2 XCode .................................................................................................. 8
      2.2.1.3 IOS Simulator ....................................................................................... 8
    2.2.2 Server-Side Development Tools .................................................................... 9
      2.2.2.1 Node.js ................................................................................................ 9
      2.2.2.2 Socket.IO ............................................................................................ 10
      2.2.2.3 MySql .................................................................................................. 10
      2.2.2.4 Git ........................................................................................................ 10

3. Implementation Methodologies ............................................................................. 11
  3.1 Application Main Architecture .......................................................................... 11
  3.2 Web-Server Implementation .............................................................................. 12
    3.2.1 Web-Server Architecture ............................................................................ 12
    3.2.2 Database design .......................................................................................... 13
    3.2.3 Server Modules .......................................................................................... 15
      3.2.3.1 Http Server .......................................................................................... 15
      3.2.3.2 Updates Users .................................................................................... 17
List of Figures

Figure 1.1 – Viber (left-side) & WhatsApp (right-side) both running on iOS devices ....... 2
Figure 2.1 – Global Smartphone OS Market Share – 2013 [4] ........................................ 5
Figure 2.2 - Advantages and disadvantages of App development tools [5] .................. 6
Figure 2.3 - XCode 5.0.2 running on Mac OS X .......................................................... 8
Figure 2.4 - XCode running project in iOS Simulator ................................................... 9
Figure 2.5 – Client-Server Architecture model ............................................................. 9
Figure 2.6 - MySQL Workbench Home Screen .......................................................... 10
Figure 3.1 - Diagram of application main architecture .............................................. 11
Figure 3.2 - Diagram web-server architecture ........................................................... 12
Figure 3.3 - Database final design ............................................................................. 14
Figure 3.4 - Broadcast architecture schema .............................................................. 30
Figure 3.5 - Model-View-Controller design pattern used in mobile application .......... 34
Figure 3.6 - Core data model of application .............................................................. 36
Figure 3.7 – Parent-Child Multi-Context architecture [23] ..................................... 37
Figure 3.8 - Mobile main loop schematic ................................................................. 38
Figure 3.9 - Overview of mobile application structure on XCode ......................... 40
Figure 4.1 - Login views iOS7 with country code+phone number ......................... 41
Figure 4.2 - Validation view (iOS7) with account status inactive (A) and active (B), after
inserting validation code received by sms .............................................................. 42
Figure 4.3 - "Me" profile (iOS7) with all profile info and last notifications .......... 43
Figure 4.4 - “Me” profile (iOS6) with all profile info and last notifications .......... 44
Figure 4.5 – Public profile views (iOS7) from user with public info .................... 44
Figure 4.6 – Public profile view (iOS6) from user with public info .................... 45
Figure 4.7 - Contacts view (iOS7) on both normal and filtering mode .............. 46
Figure 4.8 - Contacts view (iOS6) on normal mode ............................................. 46
Figure 4.9 – Requests (A) and invites (B) views (iOS7) ........................................ 47
Figure 4.10 – Common contacts view (iOS7) ...................................................... 47
Figure 4.11 – -- find map view in both iOS7 (A) and iOS6 (B) versions ............ 48
Figure 4.12 - Find list views (iOS7) with users near by .................................... 49
Figure 4.13 - Chat view showing all ongoing conversations ............................. 50
Figure 4.14 – Conversation views ........................................................................ 51
Figure 4.15 – Settings main views ......................................................................... 52
Figure 4.16 – Settings profile info views ............................................................... 53
Figure 4.17 - Settings visibility info views ............................................................. 53
Figure 4.18 – Settings preferences views .............................................................. 54
Figure 4.19 – Settings account views .................................................................. 55
FIGURE 4.20 – SETTINGS SYSTEM STATUS VIEWS................................................................. 55
FIGURE 5.1 - PERFORMANCE RESULTS ON BOTH NODE.JS (A) AND APACHE PHP (B) SERVERS...... 57
FIGURE 5.2 - MEMORY USAGE FROM BOTH SERVERS.................................................................... 58
FIGURE 5.3 - TIME PROFILER: SYNCING CONTACTS PART 1......................................................... 58
FIGURE 5.4 - TIME PROFILER: SYNCING CONTACTS PART 2...................................................... 59
FIGURE 5.5 - TIME PROFILER: UPDATING USER’S PROFILE INFO (PROFILE IMAGE) PART 1........ 59
FIGURE 5.6 - TIME PROFILER: UPDATING USER’S PROFILE INFO (PROFILE IMAGE) PART 2......... 60
List of Tables

TABLE 3-1: HTTP SERVER PARAMETERS .............................................................. 16
TABLE 3-2: SOCKET.IO SPECIFICATIONS .......................................................... 16
TABLE 3-3: UPDATES USERS – ROUTES ............................................................... 17
TABLE 3-4: UPDATES USERS – SOCKET EVENTS IN ........................................... 18
TABLE 3-5: UPDATES USERS – SOCKET EVENTS OUT ....................................... 19
TABLE 3-6: UPDATES USERS – ERRORS THROWN .............................................. 20
TABLE 3-7: UPDATES USERS - DATABASE ROUTINES/PROCEDURES ............... 21
TABLE 3-8: UPDATES USERS - ENUMERATIONS TYPES .................................... 22
TABLE 3-9: RELATIONS USERS – EVENTS IN/OUT ............................................. 23
TABLE 3-10: RELATIONS USERS – ROUTES ....................................................... 23
TABLE 3-11: RELATIONS USERS – DATABASE ROUTINES/PROCEDURES .......... 24
TABLE 3-12: RELATIONS USERS – ENUMERATIONS TYPES ............................ 24
TABLE 3-13: RELATIONS USERS – ERRORS THROWN ...................................... 25
TABLE 3-14: CONVERSATIONS USERS – EVENTS IN ....................................... 26
TABLE 3-15: CONVERSATIONS USERS – EVENTS OUT ..................................... 27
TABLE 3-16: CONVERSATIONS USERS – EVENTS OUT ..................................... 27
TABLE 3-17: CONVERSATIONS USERS – DATABASE ROUTINES/PROCEDURES .... 28
TABLE 3-18: CONVERSATIONS USERS – ENUMERATIONS TYPES .................... 29
TABLE 3-19: CONVERSATIONS USERS – ERRORS THROWN ................................ 29
TABLE 3-21: BROADCAST USERS – INBOX CODES .......................................... 32
TABLE 3-22: LIBRARIES/DEPENDENCIES MANAGED BY COCOA PODS ......... 35
TABLE 3-23: CORE DATA SUB-MODULES ON CLIENT ....................................... 39
TABLE 3-24: CLIENT-SIDE CONTROLLERS ......................................................... 40
List of Acronyms

iOS  iPhone Operating System (Apple)
MAD  Mobile application development
JS   JavaScript (Web/HTML scripting language)
HTML Hypertext Markup Language (and file extension)
APP  Application
OS   Operating System
CSS  Cascading Style Sheet
SDK  Software Development Kit
IDE  Integrated Development Kit
T-SQL Transact-Structured Query Language
RDBMS Relational Database Management System
SCM  Source Code Management
HTTP Hypertext Transfer Protocol (World Wide Web protocol)
CRUD Create Read Update Delete
JSON JavaScript Object Notation
APN  Apple Push Notifications
UTC  Coordinated Universal Time
GMT  Greenwich Mean Time
UPS  Uninterruptible Power Supply
Chapter 1

1. Introduction

This chapter explains the overall vision for this thesis work. It describes the objectives and motivations that led me to develop this project and the main reasons why I did it. It also explains how the project is structured and organized.

1.1 Context and Motivation

The use of a smartphone for common tasks and routines is becoming a huge habit amongst people. The best examples are the communication apps, which allows interacting with everybody in just a second. These kinds of applications have become very popular in the app world. However, market leaders applications like Viber and WhatsApp (see Figure 1.1 on page 2), despite having a very robust messaging service, don’t take full advantage of all the available capabilities in a mobile device and focus mainly on the chat component. Given the limited amount of information that this sort of applications supply at the moment, it’s very tempting to develop an application where location status becomes a part in the process of meeting and starting a conversation with some contact.

Also, knowing the solid background that people have been acquiring about digital social networking for the past few years, it motivated me to re-think a new way for establishing new online connections in a more controlled environment. So, gathering both abilities of talking and locating themselves, each user can restrict their ability to send/receive new requests only to people on a limited range, defined by them. With that privacy level, each person can control their visibility and also know that each invite received is being sent from a person that is very close to him. The introduction of these factors can bring some interest amongst people, by redefining how people connect with each other in a social network.

Therefore, the inspiration for this thesis emerged from the idea of having a chat application and combine it with all the information given by a location based social network, where people can share their geographic position to find their contacts and friends, or meet new people near by with ease.

The main focus is to provide a single application where people can access to all these features gathered in a harmonic fashion without switching from app to app.

In addition to that, as I refer above, one of the main reasons I chose this theme was the opportunity to understand and work with all the technologies involved, in which I have special interest, particularly web and mobile development.
1.2 Objectives

This thesis proposes a mobile application deployed to iOS devices with a shift in the paradigm of communication. It adds to a chat module, all relevant information derived from user’s geo-location, which allows users to be notified of contacts near, enabling to talk with them, giving the opportunity to catch up with some friend you haven’t seen for a while.

In addition to that, the objective is to use the Contacts App as a base for our initial “social network” and complementing it with all the basic social skills, allowing users to find and meet new people, based on their current location. Also, assuming that exists a full synchronization between address book and application, each new contact added from application or inserted on contacts list will be automatically synced on both sides. For instance, if two people nearby each other meet and want to share their phone numbers, one can send a request to the other and automatically will be added to address book as a persistent contact.

Basically, the main goal for this thesis is to re-implement a mobile chat application in order to compete with the current market applications, recognizing the potential that these new introduced features can bring advantages over those that already exist.

1.3 Original Contributions

With all things said above, this thesis tries to solve the issue of decentralization when using different applications to collect all this information, namely mobile communication, location services and social network into a single application. In addition that, it tries to
include location services not only in the communication process (chatting with people around) but also connecting with new people based on their current location.

As I referred on objectives, this dissertation involves the development of an application for mobile devices, deployed for iOS devices, which is already available on AppStore for download.

1.4 Thesis Outline

This document is structured in chapters, each representing components that contributed to the development of this Thesis.

Starting by the first chapter (current chapter), here are presented the context and motivations behind the development of this mobile application. Also, all main objectives are described, to allow visualizing better the idea for the final product.

In chapter two are introduced all theoretical concepts and all decisions taken for the development environment, for instance, which platform to deploy, development tools, both client and server side, etc.

On the third chapter, all methodologies used for each component are explained in detail as well as the decisions behind them. Particularly, all architectures designed for client and server part including database models.

The fourth chapter is used to show all final interfaces and outputs for the iOS application deployed for both iOS6 and iOS7 versions. Also, some performance tests are executed to show how application behaves on some critical parts of it.

The last chapter shows all conclusions made from this thesis and also all work intended for the future.
Chapter 2

2. Mobile Application Development

By definition, Mobile application development describes the process by which application software is developed for mobile devices. Thanks to all available markets, these applications can be downloaded and installed on each device from various mobile software distribution platforms.

Truly effective mobile application development (MAD) delivers apps and content that stands on their own, specifically to leverage the flexibility and features of mobile devices.

This chapter not only introduces these MAD related concepts but also explains in-depth all development tools chosen for this project and the main reasons for these choices.

2.1 Mobile Platforms

There are three big mobile platforms also known as mobile Operating Systems. Those are Google’s Android [1], Apple’s IOS [2] and Microsoft’s Windows Phone OS [3] as shown on Figure 2.1.

![Global Smartphone OS Market Share - 2013 Q3](image)

Nowadays, Android holds the biggest market share and it became the most popular one, mainly because every manufacturer could build a device and make it run Android. However, the benefits of being open, customizable and independent raised some unexpected disadvantages. On the other side, iOS is only available for specific devices and isn’t licensed for installation on non-Apple hardware, which can limit users at some point.
Despite Android being currently the market leader, my personal preference goes to iOS, and so, the first version of the application will be deployed to apple devices. Obviously, depending on the evolution of the application, an Android version will be distributed as well in a near future.

2.1.1 Native versus Hybrid versus Web Applications

Mobile app development presents unique, like different operating systems and devices, display sizes, devices build-in features, etc. Achieving a beautiful and well-developed app to run flawlessly on all devices within minimum requirements can be a major headache. For those reasons, the decision making process on which tools to use can depend on a various number of factors, especially because there is no perfect answer.

There are three main alternatives: Native, Hybrid and Web Applications, all having advantages and disadvantages, depending on the purpose of the app.

![App Development Comparison](image)

**Figure 2.2 - Advantages and disadvantages of App Development Tools [5]**

With **Native Apps**, each program is developed using a programming language and interface for a specific operating system and device. This may deliver best performance and full access to device capabilities, but requires a different version for each OS, which costs more time and money.

An opposite approach consists on **Web Apps**, using all well-known web development tools to deliver mobile “look and feel”. This method supports most devices and platforms, but often cannot access mobile device features, such as a camera or geo-location, inconvenience that become a key-factor in most apps. Other aspects like performance can also be affected with this methodology as Figure 2.2 shows.
An intermediate implementation gives origin to **Hybrid Apps**, which proposes a compromise between native and Web. It basically makes use of industry-standard Web programming languages such as HTML5 and JavaScript, allowing then to package in a natively installable format for app store distribution. Mainly, it produces all features from native apps, saving costs with reused code and deploying apps to all mobile platforms. There are several cross-platform development tools for this type of solution like PhoneGap [6] and MoSync [7].

However, although this approach appears to be the perfect solution for any case, it raises more problems and challenges than it seem at first glance. For example, in terms of speed and user experience, it can be seriously affected with the increase of complexity in the application. In addition to that, since all interfaces are designed primarily using HTML and CSS, it can be challenging to reproduce the appearance of each native platform.

With all the information collected on possible approaches to the mobile solution and some tests performed, the decision was to implement the project in a native environment and, as I mentioned on section

Mobile Platforms, my preference goes to iOS platform. This choice of going native was based on the complexity of the application (includes modules for chat, social networking and geo-location) and full use of the capabilities of the phone, including location services [8] and Push Notifications both Local and Remote [9]. Also, knowing the issues of reproducing the look and feel of these platforms, by using the native frameworks and SDK’s, it will not be a problem during development.

### 2.2 Development Tools

This section describes and explains all the approaches taken and development tools used, both in client and server side.

#### 2.2.1 Client-Side Development Tools

As it was mentioned above, the decision was to implement the application a native environment, specifically for iOS devices. The following sub-topics will enumerate and briefly explain all development tools used for the client part.

##### 2.2.1.1 IOS SDK

By definition, iOS SDK is a software development kit (SDK) released by Apple that allows people to write applications for Apple devices like iPhone, iPod and iPad [10]. It includes the XCode IDE, Instruments analysis tools, iPhone Simulator and others frameworks. In order to use the SDK, developers need to register on Apple’s Member Center [11] with an Apple ID and pay an annual fee.
By having a specific development platform to create iOS applications, Apple can prevent people from hacking the operating system used by the device.

2.2.1.2 XCode

XCode [12] is a set of tools created by Apple for the purpose of developing computer/mobile applications. It is mainly an Integrated Development environment (IDE) where programs can be written, compiled and executed using C++ or Objective-C code. XCode also offers interface-builders for laying out all Application’s Interfaces and Views.

![XCode 5.0.2 running on MAC OS X](image)

**Figure 2.3 - XCode 5.0.2 running on MAC OS X**

2.2.1.3 iOS Simulator

The iOS Simulator [13] allows the developer to rapidly prototype and test builds of your app during the development process. Being installed as part of the XCode tools along with the iOS SDK, iOS Simulator replicates an iPhone or iPad environment behaving like a standard Mac app. It was developed to enable users to quickly test their apps before testing them on real devices.

It also supports all current market versions of the iOS operating system, allowing developers to test and debug all versions to deploy.
2.2.2 Server-Side Development Tools

An application with these specifications relies heavily on a client-server architecture where each client (mobile device) communicates with the server in order to interact with other clients. In this section, it will be described all tools and frameworks used to implement the server part.

2.2.2.1 Node.js

Node.js [14] is a platform used to build fast, scalable network applications. It is a wrapper around the high-performance V8 JavaScript runtime [15] from Chrome. By using JavaScript as an event-driven language, Node takes advantage of it to produce highly scalable servers, reproducing concurrent processes without blocking the web server [16].

Another key tool of Node.js is its package manager (NPM) which is used to install all sorts of dependencies needed like Socket.IO [17] for using Web-socket [18] technology to provide real-time services.
2.2.2.2 Socket.IO

Socket.IO is a JavaScript library used to provide real-time services in both sides: client and server (node.js). Socket.IO is based on the Web Socket protocol [18] and provides lots of features including broadcasting to multiple sockets and asynchronous I/O.

2.2.2.3 MySql

MySql [19] is a relational database management system (RDBMS) popularly used in projects like web applications to provide ways to persist data and content between connections in a relation based model.

The official GUI tool to administer MySQL databases is MySQL Workbench [20], which is a free IDE that enables developers to create, design and manage databases with more ease and flexibility.

![MySQL Workbench Home Screen](image)

Figure 2.6 - MySQL Workbench Home Screen

2.2.2.4 Git

Git is a distributed revision control and source code management (SCM) that keeps track of changes of any projects. It allows a team of developers to work in the same project always in the last up-to-date version with commands like push, pull, merge, etc.

This tool is also used in the client-side to keep a version control system of it.
Chapter 3

3. Implementation Methodologies

In this chapter all the implemented methodologies are explained for each part of the application, such as server, client and architectures used. In addition to that, all database designs, both from server and mobile, are also described in detail.

3.1 Application Main Architecture

Here is explained the main architecture of the application and presented an overview of how devices and server communicate with each other.

![Main Architecture Diagram]

Figure 3.1 - Diagram of Application Main Architecture

As diagram on Figure 3.1 shows, all devices can interact with each through a series of events to and from the webserver.

Starting by the Mobile Application Main Loop (1), each device runs their own program receiving input from users. In addition to that, upon Internet connection, in order to maintain a persistent communication with the server, each device tries automatically to establish a full duplex socket connection using Web-Socket protocol.

Next, assuming that each device maintains a persistent connection with the server, each client uses a system of updates events (2) through that WebSocket channel. With that, chat messages, profile info and status updates (profile Image, position, etc.) and relation's updates can be performed without the overhead of each http request handshake.
As a consequence, given the interdependency between each device, when user one updates some content, all others related users need to receive the updated information (3), regardless being online or offline.

To finalize, the main process, that allows all said above, occurs on the server throughout all modules that will be explained in detail later (4).

Using the Web-WebSocket technology instead of http requests, brings enormous advantages given the application’s specifications. It enables a full-duplex communication by communicating over TCP port 80, by performing a handshake that is interpreted on HTTP servers as an Upgrade request. Once it has a chat module, each client wants to talk with his contacts in real-time. Secondly, knowing the amount of data that can be transferred between server-clients, it becomes clear that a low-consumption socket channel is the best solution for this type of applications.

In addition to that, considering all updates made by each client at a random time, it would not be possible for the server to broadcast in real-time all the updates, without a request from each target client or some sort of long polling technique (Comet channels).

### 3.2 Web-Server Implementation

As mentioned above, the web-server represents the main core of this entire process, handling all events and requests from clients. On this section, it is explained in-depth how server behaves and the main steps it takes to process different events.

#### 3.2.1 Web-Server Architecture

To better understand how server works and the job of each module inside it, let us observe a diagram (see Figure 3.2) that illustrates what is happening under the hood.

![Web-Server Architecture](image1)

**Figure 3.2 - Diagram Web-Server Architecture**
As illustrated in Figure 3.2, the entire process is based on the Http Server Module running at port 3000. Also, as I referred earlier, most of communications are made through Web-Socket connections. Thus, to implement it, Node.js provides a framework called Socket.IO allowing it to listen for the http server handling that real-time part of the server.

To better understand how it all works, let us go through all visible steps suggested by the schematic. Initially, a client sends a request/event, over http or web-socket, which is then received by the http server. After that, the package data (normally in JSON format – lightweight data-interchange format) is routed to one of three modules according with its content: Updates Users, Conversations Users and Relations Users: the first one handles all user info and status properties, the second updates all conversations statuses and the last one manages all relations between users. These three modules, which will be explained in detail later, are basically responsible for the entire logic and process of the data received, including all CRUD functions to database. Therefore, every valid request/event received from server only gets replied after passing through one of these modules.

After that, an output package (normally in JSON format) is generated and transferred to the Broadcast Users Module, which is then responsible for broadcasting it to all target clients. For instance, when a new text message arrives, it needs to broadcast not only this new message to receivers but also a package with confirmation to sender. Another important feature related to this Broadcast Module is the offline mode, enabling for example, people to receive text messages sent while being offline.

Knowing the importance of ensuring that each package is delivered to its target, all architectures were implemented, namely on Broadcast Users Module, to support any type of fault and broadcast error, turning the web service into a much more robust system.

Despite not being mentioned on diagram, given all validations processes and errors that may occur during any module, a module “ErrorHandler.js” is implemented to handle all errors (runtime & logic errors) and report those errors to clients, acting as a sub-module of Broadcast Users Module.

All steps and processes occurring under each module will be explained in detail on the next sections.

3.2.2 Database design

On this sort of applications, a well-designed and structured database can be a major factor for scalability and maintainable persistent storage. Additionally, taking the most on MySQL resources and T-SQL language, the usage of procedures and routines for data manipulation and processing, can significantly benefit performance. The Figure 3.3 represents database’s final design.
To better understand the proposed model, it is important to review the three major features of the application: instant messaging, location services and social networking.

Starting by the social component, tables like "Users", "Users_Status", "Accounts" and "Users_Relations" are designed to save profile info, user’s accounts and relations between users (contacts, requests, etc.).

Then, for the chat part there are tables to keep conversations history (all messages saved) and users involved on them, as “Conversations”, “Messages” and “Conversations_Users”.

To handle locations from users and supply the so called “near by” feature, tables like “Users”, “Users_Status” and “Users_Visible” are needed not only for that but also to save user’s preferences on visibility and distances ranges.
In order to keep all target users with up-to-date information, each update also refresh an UTC timestamp of the related user, according to the specific event and info: lastUpdateIsOnline (online status), lastUpdateInfo (profile info), lastUpdateImage (profile info/image), lastUpdateHeader (profile info/header), lastUpdateStatus (profile info/status), lastUpdateConversation (conversation info). All these timestamps will be described and contextualized during the explanation of each server module.

Finally, associated with Module broadcast Users is the table “Inbox” which performs a huge part on the offline mode of the application along with UTC timestamps and a code system. It basically keeps track of a list of entries with a specific code, source user, target user and a key id (see on Figure 3.3) each representing an update related to a code, performed by a source user and destined to some target that could not receive in real time (probably because target was offline, connection was lost, no ACK received from target, etc.).

3.2.3 Server Modules

As explained above, the web server is implemented in separated modules intercommunicating with each other in a controlled way. This section tries to clarify what is happening on every module. All modules are implemented in Node’s platform language JavaScript.

On the next modules, all WebSocket events and http routes will be enumerated in their category (Updates Users, Conversations Users, Relations Users) and discriminated all data, In and Out, database’s routines/procedures involved, possible errors, Http Status Codes, including brief descriptions.

3.2.3.1 Http Server

Http Server module is responsible to run server’s main program running both the http and “Socket.IO” server at a static IP on port 3000. All routes and events made to the server are received and started here. From this point, each data package received is properly redirected to the specific module so it can be processed.

This module is also responsible for connecting to Apple Push Notifications (APN) [21] server to handle remote notifications to all registered clients. To receive remote notifications from server, each client needs to be registered on APN and transmit their device token to server “HiThere” through socket event, so they can be identified. After that, server can inform a device about new text messages or be notified of a nearby contact, while the application is in background.

The next tables shows all server main configurations:
Table 3-1: Http Server parameters

<table>
<thead>
<tr>
<th>Http Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static IP/Hostname</strong></td>
</tr>
<tr>
<td><strong>Port</strong></td>
</tr>
<tr>
<td><strong>Http Codes</strong></td>
</tr>
</tbody>
</table>

Table 3-2: Socket.IO specifications

<table>
<thead>
<tr>
<th>Socket.IO specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heartbeat interval</strong></td>
</tr>
<tr>
<td><strong>Heartbeat timeout</strong></td>
</tr>
<tr>
<td><strong>Close timeout</strong></td>
</tr>
</tbody>
</table>

Currently, http server is hosted with static IP/Hostname hithere.l2f.inesc-id.pt listening on port 3000 (see on Table 3-1), however these values obviously may vary over time. To support this migration of one ip to other, each server is ready to activate re-direction (Http Code 302) to route /updateHostConfigs and update server new configurations on client through a custom event. In addition to this, last row represents all Http Status Codes used throughout the entire program: 200 (OK!), 201 (CREATED), 202 (ACCEPTED), 302 (FOUND), 400 (BAD REQUEST), 422 (UNPROCESSABLE ENTITY), 500 (INTERNAL SERVER ERROR) for all various requests and updates.

About Socket.IO specifications, the most important values are related to each socket connection timeouts between server and client. Thus, the heartbeat interval represents the time for the server to send a new heartbeat to the client (30 seconds) to keep connection alive. The second one describes a time for the client to receive a heartbeat from the server within that time (40 seconds). If one of these timeouts expires, then the channel is considered disconnected for a period equals to “Close timeout”, and assuming that it's not re-established within that time, the connection is closed.
### 3.2.3.2 Updates Users

This module handles all routes and events to User’s Info and Status. User’s Info represents mostly profile information such as public name, profile image, header image and some other features like distance ranges and timestamps for controlling up-to-date content.

User’s Status keeps primarily properties for representing user’s connection status and all properties related to geo-location including coordinates and visibility preferences (visibilityOnline, longitude, latitude, isUpdatingLocation, etc.).

The next tables describes all events and routes related to this module, subdivided in categories with a brief description of each:

**Table 3-3: Updates Users – Routes**

<table>
<thead>
<tr>
<th>Routes</th>
<th>Parameters</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET /login</strong></td>
<td>/login?</td>
<td>✓ Register new User’s Account with a phone number, dial Code and a country Code (If no validationCode).</td>
</tr>
<tr>
<td></td>
<td>✓ userId</td>
<td>✓ Login User with active account and an authorized validation Code.</td>
</tr>
<tr>
<td></td>
<td>✓ [validationCode]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓ dialCode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓ countryCode</td>
<td></td>
</tr>
<tr>
<td><strong>POST /validateAccount</strong></td>
<td>Body – JSON Format:</td>
<td>✓ Validate User’s Account and allows login with phone number.</td>
</tr>
<tr>
<td></td>
<td>✓ userId</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓ validationCode</td>
<td></td>
</tr>
</tbody>
</table>
Attached to this category is also a service called Twilio to support Sms Applications via a Web API, involved on validation process by sending validations codes generated to new users through Sms.

Table 3-4: Updates Users – Socket Events IN

<table>
<thead>
<tr>
<th>Events IN</th>
<th>Data In</th>
<th>Actions</th>
</tr>
</thead>
</table>
| EVENT_UPDATE_USER_LOGGED_IN_IS_ONLINE | JSON Format (Dict):
  - userId
  - isOnline | ✓ Update user Web Socket Connection (offline/online)
  ✓ Update database table “Users_Status”.
  ✓ Broadcast user connection to all target users with new lastUpdateIsOnline UTC timestamp. |
| EVENT_UPDATE_USER_LOGGED_IN_INFO | JSON Format (Dict):
  - userId
  - [profileInfo properties]
  - [profileImage]
  - [headerImage] | ✓ Update User Info properties including profile Image and header Image.
  ✓ Update database table “Users”
  ✓ Broadcast updated Info properties with refreshed lastUpdateInfo and/or lastUpdateImage and/or lastUpdateHeader timestamps.
  ✓ Broadcast updated users visible. |
| EVENT_UPDATE_USER_LOGGED_IN_STATUS | JSON Format (Dict):
  - userId
  - [profileStatus properties] | ✓ Update User Status properties like position coordinates, visibility preferences, etc.
  ✓ Update table “Users_Status” on database.
  ✓ Broadcast updated status properties with refreshed lastUpdateStatus UTC timestamp to target users.
  ✓ Broadcast updated users visible. |
| disconnected | | ✓ Called when socket disconnects.
  ✓ Reset status properties (isOnline, isUpdateLocation) on database table “Users_Status”.
  ✓ Broadcast reset properties with lastUpdateStatus. |
| EVENT_REQUEST_USER_LAST_UPDATESDATES | JSON Format (Dict):
  - targetId
  - userId | ✓ Requests last updates dates from userId.
  ✓ Broadcast all requested dates (lastUpdateIsOnline, lastUpdateInfo, lastUpdateImage, lastUpdateHeader) to target users. |

18
<table>
<thead>
<tr>
<th>Event Session</th>
<th>Description</th>
<th>JSON Format (Dict):</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_REQUEST_USER_LAST_UPDATESDATESDATES_ WITH_DATA</td>
<td>Requests dates with data from userId.</td>
<td>✓ targetId ✓ userId ✓ dates</td>
<td>✓ Broadcast all dates (lastUpdateIsOnline, lastUpdateInfo, etc.) with respective data. If targetId is equal to userId then server sends all data, else it only broadcasts public data.</td>
</tr>
<tr>
<td>EVENT_USER_LAST_UPDATESDATES</td>
<td>Broadcast all userId updated dates (lastUpdateIsOnline, lastUpdateInfo, etc.) to some target user. Normally this event is sent to userId himself after updating some property on device.</td>
<td>✓ userId ✓ dates</td>
<td>✓ Broadcast all dates (lastUpdateIsOnline, lastUpdateInfo, etc.) with respective data. If targetId is equal to userId then server sends all data, else it only broadcasts public data.</td>
</tr>
<tr>
<td>EVENT_UPDATE_USER_VISIBILITY</td>
<td>Broadcast updated users visible with new distances.</td>
<td>✓ userId ✓ distance</td>
<td>✓ Broadcast updated users visible with new distances.</td>
</tr>
</tbody>
</table>

Tables Table 3-4 and Table 3-5 explain in detail all events related to Updates to user’s info and status with all packages data IN/OUT in JSON format and all actions performed. As it can be seen, these events are divided into events from client to server (IN) and from server to client (OUT).

About info updates, it’s important to refer that, all media updates like profile or header image are encoded in base64 to simplify transportation and saved on server this way.
This module is also responsible to refresh users visible on each target user upon updates on user status and info, like for instance, coordinate positions, online status, visibility ranges and related preferences.

As described on actions, the most affected database tables are “Users”, “Users_Status” and “Users_Visible” where each update causes a refresh on a UTC timestamp lastUpdate[Info][Status].

About the broadcast, all events are re-directed to the broadcast Module and from there, sent to all targets including sender himself.

<table>
<thead>
<tr>
<th>Errors</th>
<th>Thrown by Routes/Events</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorInvalidUserId</td>
<td>✓ GET /login (400)</td>
<td>✓ Invalid userId (phone number) or non-existent.</td>
</tr>
<tr>
<td></td>
<td>✓ POST /validateAccount (422)</td>
<td></td>
</tr>
<tr>
<td>errorInvalidValidationCode</td>
<td>✓ GET /login (400)</td>
<td>✓ Invalid or non-existent validationCode for user.</td>
</tr>
<tr>
<td></td>
<td>✓ POST /validateAccount (422)</td>
<td></td>
</tr>
<tr>
<td>errorInvalidLoginParameters</td>
<td>✓ GET /login (400)</td>
<td>✓ Invalid login parameters</td>
</tr>
<tr>
<td></td>
<td>✓ All Events</td>
<td></td>
</tr>
<tr>
<td>errorInvalidAccountStatus</td>
<td>✓ GET /login (400)</td>
<td>✓ Error on updating account status.</td>
</tr>
<tr>
<td></td>
<td>✓ POST /validateAccount (422)</td>
<td></td>
</tr>
<tr>
<td>errorAccountStatusExpired</td>
<td>✓ GET /login (400)</td>
<td>✓ Inform that account as expired</td>
</tr>
<tr>
<td></td>
<td>✓ POST /validateAccount (422)</td>
<td></td>
</tr>
<tr>
<td>errorInvalidValidationParameters</td>
<td>✓ POST /validateAccount (422)</td>
<td>✓ Invalid validation parameters</td>
</tr>
<tr>
<td>errorInvalidSocketUserId</td>
<td>✓ All Events (400/422)</td>
<td>✓ Invalid socket connection. Informs client to reset socket connection.</td>
</tr>
</tbody>
</table>
To handle all validations done by server from each data in package, Table 3-6 shows all errors thrown from Module “UpdatesUsers” to Module “ErrorHandler” which then are reported to sender.

The Table 3-7 shows all routines and procedure implemented on database related to this module Updates Users. These routines/procedures are mainly related to Accounts Management and Updates on Users Visibility.

Table 3-7: Updates Users - Database Routines/Procedures

<table>
<thead>
<tr>
<th>Database Routines/Procedures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_generateNewValidationCodeForUser</td>
<td>✓ Generate new Validation Code for Users and sets account inactive, waiting for validation.</td>
</tr>
<tr>
<td>sp_validateAccountForUser</td>
<td>✓ Validates account for user with user and validation Code. Also sets account to active, if validation code is correct.</td>
</tr>
<tr>
<td>sp_createNewUserAccount</td>
<td>✓ Insert/Update New User+Account with arguments as data &amp; Reset Account (If Not Expired).</td>
</tr>
<tr>
<td>sp_requestUserAllUsersVisible</td>
<td>✓ Called to request User’s All Users Visible</td>
</tr>
<tr>
<td>sp_updateUserAllUsersVisible</td>
<td>✓ Updates User’s Users Visible of generic User.</td>
</tr>
<tr>
<td>sp_updateUserAllUsersVisibleWithProperty</td>
<td>✓ Updates user all users visible according updated Property.</td>
</tr>
</tbody>
</table>

To better express all types and data involved, during the application some enumerations were defined. Here are the ones related to this module and they're all self-explanatory:
Table 3-8: Updates Users - Enumerations Types

<table>
<thead>
<tr>
<th>Enumerations</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImagesSizes</td>
<td>✓ PROFILE_IMAGE_WIDTH = 65</td>
</tr>
<tr>
<td></td>
<td>✓ PROFILE_IMAGE_HEIGHT = 65</td>
</tr>
<tr>
<td>AccountStatus</td>
<td>✓ ACCOUNT_INACTIVE = 0</td>
</tr>
<tr>
<td></td>
<td>✓ ACCOUNT_ACTIVE = 1</td>
</tr>
<tr>
<td></td>
<td>✓ ACCOUNT_EXPIRED = 2</td>
</tr>
<tr>
<td>RangeLevels</td>
<td>✓ RANGE_NEAR = 0</td>
</tr>
<tr>
<td></td>
<td>✓ RANGE_MEDIUM = 1</td>
</tr>
<tr>
<td></td>
<td>✓ RANGE_FAR = 2</td>
</tr>
<tr>
<td>RangeLevelsKms</td>
<td>✓ RANGE_NEAR_KMS = 0.5</td>
</tr>
<tr>
<td></td>
<td>✓ RANGE_MEDIUM_KMS = 1</td>
</tr>
<tr>
<td></td>
<td>✓ RANGE_FAR_KMS = 2</td>
</tr>
<tr>
<td>VisibilityLevels</td>
<td>✓ VISIBLE_ALL = 0</td>
</tr>
<tr>
<td></td>
<td>✓ VISIBLE_CONTACTS = 1</td>
</tr>
<tr>
<td>ApnServerNotificationsTypes</td>
<td>✓ APN_SERVER_NOTIFICATION_TYPE</td>
</tr>
<tr>
<td></td>
<td>PE_NEW_MESSAGE_SMS = 1</td>
</tr>
<tr>
<td></td>
<td>✓ APN_SERVER_NOTIFICATION_TYPE</td>
</tr>
<tr>
<td></td>
<td>PE_NEW_CONTACT_VISIBLE = 2</td>
</tr>
</tbody>
</table>

As it was referred before presenting Table 3-8, all types are understandable. However, the last one, ApnServerNotificationsTypes wasn’t discussed yet. Therefore, before explain it, it’s important to remember that the http server establishes a persistent connection with Apple Push Notifications (APN) web service when it starts. Thus, for now, server offers push notifications services for two types: text messages and contacts visible. This enumeration is used to identify each notification.

Again, just like all the others preferences, each user can configure from their mobile application whether or not they want these services through preferences settings.

3.2.3.3 Relations Users

This next module implements all necessary routes and events to handle all connections between users, namely relations like contacts, requests and invites. The next table shows all available events to perform this social component of the application.
Table 3-9: Relations Users – Events IN/OUT

<table>
<thead>
<tr>
<th>Events IN/OUT</th>
<th>Data In/Out</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_UPDATE_USERS_RELATIONS</td>
<td>JSON Format (Dict): ✓ userId ✓ data (Array): - sourceld - targetId - status</td>
<td>✓ Updates users relations with given data (sourceld, targetId, status) on table “Users_Relations”. ✓ Broadcast Updates on users relations for source and target users. ✓ Broadcast User’s Info and status to all new connections (mainly new contacts relations)</td>
</tr>
<tr>
<td>EVENT_REQUEST_COMMON_CONTACTS</td>
<td>JSON Format (Dict): ✓ Sourceld ✓ targetId</td>
<td>✓ Requests all commons contacts between sourceld and targetId. ✓ Broadcast all common relations between both users</td>
</tr>
</tbody>
</table>

Table 3-10: Relations Users – Routes

<table>
<thead>
<tr>
<th>Routes</th>
<th>Parameters</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /updateUsersRelations</td>
<td>Body – JSON Format: ✓ userId ✓ data (Array): - sourceld - targetId - status</td>
<td>✓ Updates Users Relations by Http Post request, normally used to sync all list of contacts with application. ✓ Re-directs to same procedure as EVENT_UPDATE_USERS_RELATIONS mentioned above.</td>
</tr>
</tbody>
</table>

The table Table 3-9 enumerates and describes all events used by socket connection that can be called to update relations. By definition, both of them can be used as events IN and OUT which can be easy understandable, once the server also may have to send all relations updates to all users involved.

About the second table (see Table 3-10), a route to perform an update on Users relations was also created. This route was included basically to avoid using the socket connection and blocking it while sending all user relations.
From the server perspective, when each user forces a synchronization, all contacts, requests and invites are sent to server as an array, as both tables Table 3-9 and Table 3-10 suggest. Then server inserts/updates those relations on database, returning the ones that actually changed. For example, a pending user (onClient: -1) updated to contact (onServer: 2), then user must be informed.

The next table Table 3-11 shows all auxiliary procedures and routines implemented to perform all CRUD operations on database related to users relations:

Table 3-11: Relations Users – Database Routines/Procedures

<table>
<thead>
<tr>
<th>Database Routines/Procedures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_requestUserAllContacts</td>
<td>✓ Requests all user Contacts (status = 2)</td>
</tr>
<tr>
<td>sp_requestUserAllPendingContacts</td>
<td>✓ Requests all user Pending Contacts (status = -1)</td>
</tr>
<tr>
<td>sp_requestUserAllTargetUsersOffline</td>
<td>✓ Requests all target users offline from specific user. Includes all related users (any status).</td>
</tr>
<tr>
<td>sp_requestUserAllTargetUsersOnline</td>
<td>✓ Requests all target users online from specific user. Includes all related users (any status) and all near by.</td>
</tr>
<tr>
<td>sp_updateUsersRelations</td>
<td>✓ Updates all users relations given by array on argument of procedure.</td>
</tr>
<tr>
<td>sp_requestCommonContacts</td>
<td>✓ Requests all common contacts between two users returning an array with relations.</td>
</tr>
<tr>
<td>sp_countCommonContacts</td>
<td>✓ Counts all common contacts between two users.</td>
</tr>
</tbody>
</table>

Table 3-12: Relations Users – Enumerations types

<table>
<thead>
<tr>
<th>Enumerations</th>
<th>Values</th>
</tr>
</thead>
</table>
| relationsUsers| ✓ PENDING = -1  
|              | ✓ UNKOWNS = 0  
|              | ✓ DECLINED = 0  
|              | ✓ REQUEST_INVITED = 1  
|              | ✓ CONTACTS = 2  |
This enumeration type on Table 3-12 requires a bit more explanation that the others declared for the previous module.

Therefore, relationsUsers defines all possible status of relations between two users. Starting from PENDING, it represents a state after sourceld synchronizes all contacts with address Book, waiting for targetId do the same and so become contacts in the application like in both their address books. REQUEST_INVITED represents a status of request sent by sourceld to targetId to become contacts. All the others values UNKNOWN, DECLINED and CONTACTS are self-explanatory.

About the ErrorHandler Module, the next table shows all custom errors that may be thrown by RelationsUsers Module:

Table 3-13: Relations Users – Errors thrown

<table>
<thead>
<tr>
<th>Errors</th>
<th>Thrown by Routes/Events</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorInvalidParams</td>
<td>✓ All Events (400/422)</td>
<td>✓ Invalid parameters on event</td>
</tr>
<tr>
<td>errorInvalidSocketUserId</td>
<td>✓ All Events (400/422)</td>
<td>✓ Invalid socket connection from userId</td>
</tr>
</tbody>
</table>

3.2.3.4 Conversations Users

This module implements the chat component of the application. All events are handled by Web-socket and no routes are used for this part, given the real-time aspect of the application. Among many features, it allows initialize conversations with contacts, keep tracking of conversations (messages sent, seen by receiver), to talk with them both in offline and online mode, group conversations (upcoming releases) and more.

The following tables show all events In, Out and In/Out, database procedures and enumerations created for this module to work properly. All custom errors available to this module are also shown in a table.
### Table 3-14: Conversations Users – Events IN

<table>
<thead>
<tr>
<th>Events IN</th>
<th>Data In</th>
<th>Actions</th>
</tr>
</thead>
</table>
| EVENT_REQUEST_CONVERSATION | JSON Format (Dict):  
  ✓ users (Array)  
  ✓ pendingConversationGUID  
  ✓ isGroup  
  ✓ [host]  
  ✓ [name] | ✓ Insert/Update conversation with users and specifications passed on data in.  
✓ Broadcast conversation info to users  
✓ Broadcast conversation users and respective status on it to all users involved. |
| EVENT_REQUEST_CONVERSATION_USERS | JSON Format (Dict):  
  ✓ conversationId | ✓ Requests all conversation users from conversation with id conversationId.  
✓ Broadcast conversation users and respective status. |
| EVENT_UPDATE_CONVERSATION_SEND_NEW_MESSAGE | JSON Format (Array):  
  ✓ conversationId  
  ✓ messageGUID  
  ✓ userId  
  ✓ status  
  ✓ type  
  ✓ [sentDate] [text] | ✓ Updates conversation with new message of type ‘type’ from conversationId.  
✓ Broadcast conversation info with new message.  
✓ Broadcast new message status to target users, for instance, sent or seen by all. |
### Table 3-15: Conversations Users – Events OUT

<table>
<thead>
<tr>
<th>Events OUT</th>
<th>Data Out</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_UPDATE_CONVERSATION_WITH_MESSAGE</td>
<td>JSON Format (Dict): ✓ [messageInfo]</td>
<td>✓ Broadcast message of some type with message info, including an UTC timestamp generated by server.</td>
</tr>
<tr>
<td>EVENT_UPDATE_CONVERSATION_MESSAGE_STATUS</td>
<td>JSON Format (Dict): ✓ conversationId ✓ messageId ✓ messageGUID ✓ status ✓ type</td>
<td>✓ Broadcast message status to all target users.</td>
</tr>
</tbody>
</table>

### Table 3-16: Conversations Users – Events OUT

<table>
<thead>
<tr>
<th>Events IN/OUT</th>
<th>Data In/Out</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_UPDATE_CONVERSATION_INFO</td>
<td>JSON Format (Dict): ✓ conversationId ✓ [updatedProperties]</td>
<td>✓ Updates conversation info with properties given by data in on database ✓ Broadcast updated conversation info to target users.</td>
</tr>
<tr>
<td>EVENT_UPDATE_CONVERSATION_USERS</td>
<td>JSON Format (Dict): ✓ conversationId ✓ userId ✓ status</td>
<td>✓ Update user status in conversation with conversationId on database ✓ Broadcast conversation users to all target users.</td>
</tr>
<tr>
<td>EVENT_UPDATE_CONVERSATION_IS_TYPING</td>
<td>JSON Format (Dict): ✓ conversationId ✓ userId ✓ type ✓ status</td>
<td>✓ Updates conversation with message of type isTyping with given status to know if any of the conversation users is typing new message. ✓ Broadcast isTyping status to all conversation users.</td>
</tr>
</tbody>
</table>
All three tables represents all events used on conversations module. Mainly, there are events for updating conversation info, conversation users and messages of various types (text messages, isTyping messages, sent dates messages and more) and they affect mostly tables like “Conversations”, “Conversations_Users” and “Messages”.

To finalize all elements on this module, the following tables describe all procedures and enumeration types used.

Table 3-17: Conversations Users – Database Routines/Procedures

<table>
<thead>
<tr>
<th>Database Routines/Procedures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_requestConversation</td>
<td>✓ Insert/Update Conversation with users and configurations passed on argument</td>
</tr>
<tr>
<td>sp_updateConversationSendNewMessage</td>
<td>✓ Insert new message into conversation on database</td>
</tr>
<tr>
<td>sp_updateConversationUsers</td>
<td>✓ Updates participants on a specific conversation and their status.</td>
</tr>
<tr>
<td>sp_updateConversationMessageStatus</td>
<td>✓ Update message status on conversation with statuses available (STATUS_INIT,STATUS_SENDER_ACK,…).</td>
</tr>
</tbody>
</table>

Here are presented all routines/procedures implemented to enable conversations between users. They are used as a bridge between event methods and database itself.

Moreover, as tables show, this module is implemented and designed for both peer-to-peer and group conversations. However, this feature is still not fully implemented and properly tested on client, and so, it only will be available on possible future releases.

All enumerations types declared are used to identify states on conversations. To handle participants’s status on conversations, enumeration “conversationsUsers” is used. In order to control each message status, messages enumerations types are used to inform sender on message broadcasting, for instance to know if message has been received by targets (STATUS_RECEIVERS_ACK). Finally, messagesTypes identify all sort of messages transmitted between server and clients. Obviously, the most common ones will be MESSAGE_TYPE_SMS, which represent all text messages sent on each conversation.
### Table 3-18: Conversations Users – Enumerations types

<table>
<thead>
<tr>
<th>Enumerations</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>conversationsUsers</td>
<td>✓ STATUS_INVITED = 0</td>
</tr>
<tr>
<td></td>
<td>✓ STATUS_IN = 1</td>
</tr>
<tr>
<td></td>
<td>✓ STATUS_OUT = 2</td>
</tr>
<tr>
<td></td>
<td>✓ STATUS_DELETE = 3</td>
</tr>
<tr>
<td>messages</td>
<td>✓ STATUS_INIT = 0</td>
</tr>
<tr>
<td></td>
<td>✓ STATUS_SENDER_ACK = 1</td>
</tr>
<tr>
<td></td>
<td>✓ STATUS_RECEIVERS_ACK = 2</td>
</tr>
<tr>
<td>messagesTypes</td>
<td>✓ MESSAGE_TYPE_SMS = 1</td>
</tr>
<tr>
<td></td>
<td>✓ MESSAGE_TYPE_DATE = 2</td>
</tr>
<tr>
<td></td>
<td>✓ MESSAGE_TYPE_CONVERSATION_INFO = 3</td>
</tr>
<tr>
<td></td>
<td>✓ MESSAGE_TYPE_CONVERSATION_USER_STATUS = 4</td>
</tr>
<tr>
<td></td>
<td>✓ MESSAGE_TYPE_RELATIONS_USERS = 5</td>
</tr>
<tr>
<td></td>
<td>✓ MESSAGE_TYPE_IS_TYPING = 6</td>
</tr>
</tbody>
</table>

### Table 3-19: Conversations Users – Errors thrown

<table>
<thead>
<tr>
<th>Errors</th>
<th>Thrown by Routes/Events</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorRequestConversation</td>
<td>✓ EVENT_REQUEST_CONVERSATION(400)</td>
<td>✓ Invalid parameters to request a conversation</td>
</tr>
<tr>
<td>errorInvalidParameters</td>
<td>✓ All Events (400/422)</td>
<td>✓ Invalid/non-existent parameters to perform event</td>
</tr>
<tr>
<td>errorInvalidSocketUserld</td>
<td>✓ All Events (400/422)</td>
<td>✓ Invalid socket connection of userld. Informs to reset connection on client.</td>
</tr>
</tbody>
</table>
3.2.3.5 Broadcast Users

As I referred in the beginning of this chapter and have been mentioning throughout all previous modules, Broadcast Users is the module responsible for sending and handling all communications and transferring data between clients. The next diagram (see Figure 3.4) shows its architecture and main workflow:

Figure 3.4 - Broadcast Architecture Schema

As diagram illustrates, its main job is to broadcast events passed from the others three modules (1), specifically Updates Users, Relations Users and Conversations Users, by using all events OUT available from each one already presented on tables above. After sending a package, Broadcast Module sets timeouts to each pair event-target that it transmits (2). After this, for each package received by a target, he needs to confirm delivery with an ACK back to server (3). Then, if server receives ACK from client, the timeout is deleted and transfer process is completed successfully. However if, for some reason (connection lost or package damaged), ACK does not arrive to server, the timeout will expire and the event will be postponed to database at Table “Inbox” with a specific code and users involved being saved, specifically sourceId and targetId.

This component is also the one that ensures the so-called offline mode of the application, by postponing as soon as possible, all updates events to targets that are not connected (online).
Another important aspect is that this module is divided into three sub-modules: BroadcastSelfToSelf, BroadcastSelfToAll and BroadcastAllToSelf.

The first one is used to broadcast content to the user himself that performs the update (to send new timestamps, confirmations) or request (server sends fetched data).

The second sub-module is responsible to transmit those updates to all target users (all users that must receive the updated information).

The last one, BroadcastAllToSelf, is called whenever a user interacts with server and it broadcasts all events that couldn’t be delivered in real-time to user himself, for example, a message sent while user was offline or a package that wasn’t confirmed by user through an acknowledge. These tasks are supported by a system of Inbox messages ("Inbox" Table) with specific codes that registers all undelivered data to target users. Note: All key words refer to each field on Inbox Table.

All these aspects will be explained shortly with the auxiliary of all implemented procedures/routines and enumeration types declared.

<table>
<thead>
<tr>
<th>Database Routines/Procedures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_broadcastUsers</td>
<td>✓ Evaluate all targets of specific event passed by argument and prepare broadcast to users online and postpone delivery to all users offline, with specific code into Inbox Table. Note: All codes available are on the table with enumerations types for this module.</td>
</tr>
<tr>
<td>sp_updateInboxWithCode</td>
<td>✓ Postpones updates events by inserting them into Inbox Table with a specific code, sourceId, targetId and a keyId</td>
</tr>
<tr>
<td>Enumerations</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Inbox</strong></td>
<td><strong>CODE_UPDATES_USERS_101 = 101</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about online status update of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_102 = 102</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about User Info/Info update of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_103 = 103</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about User Info/Image update of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_104 = 104</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about User Info/Header update of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_105 = 105</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about User Status update of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_106 = 106</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about “lastUpdatesDates” (update) of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_107 = 107</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about “lastUpdatesDates” (request) of sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_108 = 108</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about “lastUpdatesDates”+[public] data (update) of</td>
</tr>
<tr>
<td></td>
<td>sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_UPDATES_USERS_109 = 109</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld that needs to update his/her users visible.</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_RELATIONS_USERS_204 = 204</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld that relation between him and sourceld did update.</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_RELATIONS_USERS_205 = 205</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about common contacts between him and sourceld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_CONVERSATIONS_USERS_300 = 300</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about requested Conversation Info with pendingGUID</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_CONVERSATIONS_USERS_301 = 301</strong></td>
</tr>
<tr>
<td></td>
<td>• Inform targetld about update on Conversation Info with conversationld</td>
</tr>
<tr>
<td></td>
<td><strong>CODE_CONVERSATIONS_USERS_302 = 302</strong></td>
</tr>
</tbody>
</table>
• Inform targetId about all conversation users from conversationId as keyId
   ✓ CODE_CONVERSATIONS_USERS_303 = 303
• Inform targetId about conversation user update of sourceId with conversationId as keyId
   ✓ CODE_CONVERSATIONS_USERS_306 = 306
• Inform targetId about update on message Info with messageId as keyId.
   ✓ CODE_CONVERSATIONS_USERS_307 = 307
• Inform targetId about update on message status with messageId as keyId.

As table above shows (see Table 3-21) all codes are organized given their module. Thus, codes for Updates Users module are formatted with 1xx and each relates to updates on Users Info and/or Status that failed broadcast to target Id. For the Relations Users, they all obey to format 2xx and represent mostly updates on users relations that weren't notified on targetId. Finally, Conversations codes use format 3xx and represents all events related to conversations, conversations users and messages that failed broadcast when sent to targets.

Again, it is important to enhance that all entries at Inbox table happen, whether because target Id was offline at the time, broadcast failed or receiver did not send back an ACK confirming delivery of the package.

3.2.4 Scalability and Robustness

As I mentioned on section 2.2.2 Server-Side Development Tools, server is implemented using web framework Node.js, which is a wrapper around the high-performance V8 JavaScript engine from Google Chrome browser. It uses an event-driven non-blocking programming paradigm to build highly scalable servers.

What makes Node.js so powerful and scalable is that everything it does is non-blocking, and so, the time between an event being emitted and Node being able to act on that event is very short because it is not waiting on things such as disk I/O. Instead, it uses a system of callbacks that are called by each working process after they finish their time consuming tasks. Because of this, Node brought huge advantages over typical PHP and many others web platforms, which by default handle each “order” one at a time and only start the other, after finishing the last one.

Despite being “single-threaded” by standard, Node.js also offers a module “cluster” that allows you to delegate work to child processes and make the most out of all machine cores.
Although this multi-threading seems perfect to any sort of web service, given the nature of my server and all reliability on socket connections to perform real-time features, the support for MPI (message process interface) between master-child processes, does not seem suited to this type of implementations yet. However, knowing that this technology is available, it gives the opportunity to scale up in proportional way, if it justifies in a short-medium term.

In terms of robustness and fault tolerance, all sort of measures were taken and thought, starting by all architectures explained above to improve viability (Inbox system, Broadcast timeouts, errors Handlers through try catch implementations, etc.), to all hardware minimum requirements, for instance, server protected by UPS avoiding crashes and injuries on service, caused by disruptions on power supply. Also, by choosing a MySql database, it allows performing database mirroring and redundancy if necessary.

### 3.3 Mobile Implementation

After explaining all web-server components and all main events and routes that enables all communications between server and clients, it is important to describe how client-side is structured. Therefore, the entire mobile application is based on the Model-View-Controller (MVC) design pattern, which can be described into three different parts: Model, View and Controller as the figure below (see Figure 3.5) illustrates.

![Model-View-Controller design pattern](image)

**Figure 3.5** - Model-View-Controller design pattern used in mobile application

Basically, the model part defines and handles the entire process and logic computation of all data, including core data model and the entire persistent storage of the application. The Controller module acts as an intermediary between views and model and is responsible for updating both views and model objects. Finally, views represent all objects in the application that the users can see (interfaces) and their main purpose is to display data from the application’s model objects and respond to user actions. This design pattern allows an application to be more reusable, scalable and easy to maintain.

This section will explain the process throughout the implementation of the first two parts: Model and Controllers, and the interfaces will be presented in Chapter 4 – Results and Interfaces.
3.3.1 Client Modules

In order to interact with server, two major classes were created on client: Web-Server and Core Data Modules. The first one handles all network connections through http requests and web-socket events to/from the web-server. The second module describes all objects used in these processes of requests and updates refreshing all data model related to the main components of application, chat, relations and locations services already referred above.

All dependencies necessary to enable all sorts of communications with server are managed by Cocoa Pods, a dependency manager for Objective-C projects that allows a project to keep all used libraries up-to-date. For this particular application, all libraries used are presented on next table:

Table 3-22: Libraries/Dependencies managed by Cocoa Pods

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket.IO</td>
<td>✓ A Web Socket library to communicate with Node.js server through Web-Sockets</td>
</tr>
<tr>
<td>Base64</td>
<td>✓ Base64 is a set of categories and methods to encode and decode data as a base-64 string</td>
</tr>
<tr>
<td>GKImagePicker</td>
<td>✓ Image Picker with support for custom crop areas.</td>
</tr>
<tr>
<td>MSCellAccessory</td>
<td>✓ An UITableViewCell accessoryType Class that can easily customize the colors and positions. Supported iOS7 Flat Design.</td>
</tr>
<tr>
<td>Reachability</td>
<td>✓ Library that monitors the network state on iOS and OS X devices.</td>
</tr>
<tr>
<td>SocketRocket</td>
<td>✓ A Web Socket library for client, particularly to support Socket.IO framework.</td>
</tr>
<tr>
<td>UIImage-resize</td>
<td>✓ Category to add some resizing methods to the UIImage class, to resize it to a given CGRect or fit in a CGRect keeping aspect ratio.</td>
</tr>
</tbody>
</table>

3.3.1.1 Web-Server Module

This module is responsible for all connections with server including web-socket connections (keep-alive connection), socket events and http requests (GET's and POST's).
It handles all events sent and received over the socket channel and then, all data packages are re-directed to Core Data Module to be properly processed and updated on all necessary core data objects, possibility refreshing views contents. Notice that all events and routes mentioned here, including data in/out and actions, were already described on the Web Server implementation and so, it is useless to repeat them here again.

Libraries like Socket.IO, Reachability and SocketRocket have an important roll on this module.

### 3.3.1.2 Core Data Module

The core data module handles not only the design and process of core data model using SQLite software (self-contained, transactional SQL database engine) but also to all sub-modules that support this class.

![Core Data Model Diagram](image)

**Figure 3.6 - Core Data model of Application**

Starting from the setup of the model (see Figure 3.6), its design is basically a mirror from the server database model with small changes.

The first difference that pops up immediately is that Table “Users_Status” from server does not exist and all its properties are merged with “Users” into a single entity. Actually, what makes sense is a single entity. However, as I explained earlier, table “Users” on server handles all info of each client, being a more “persistent” data and “Users_Status” holds all properties related to connection status, visibility aspects turning it into a more dynamic
content. Thus, to achieve better performances on accessing both tables, it was decided to separate them on the server-side.

The other difference that may not be visible at first, is that Core Data is an Apple’s ORM Framework and so, each element is an entity Class instead of a Table and each entry is represented as an object. Obviously, as it can be expected, it shifts the entire paradigm of design and data manipulation. For instance, instead of representing relationships with foreign keys, all relations are actually defined by the objects themselves.

After designing the model, a persistent store (“physic” database) was created based on this model. To perform changes on this persistent store, Core Data introduces the concept of contexts, each representing a single “object space” that manage a collection of model objects organized and consistent with the persistent store itself.

Given the dimension of the database and the amount of traffic generated on database accesses (requests, updates, inserts) for this particular application, a multi-contexts architecture was adopted, to allow performing I/O tasks in background threads, without blocking the interface.

Thus, after a carefully study of pros and cons of many approaches [23], the one used is presented on the next figure:

![Stack #2](image)

Figure 3.7 – Parent-Child Multi-Context Architecture [23]

As it can be seen, it was adopted a parent-child multi-context architecture (see Figure 3.7). With that, unlike the standard architecture, after each context saving, all their updated content is passed to the next layer (parent context), instead of saving directly on the
persistent store (physical database). Thus, when Worker Context updates an object and saves it, that change is passed to the next context, in this case Main Context and so on, until it reaches to the persistent store coordinator (database file). In addition to that, as figure illustrates, both worker and master contexts run on background threads, which enables the Main Context, the context responsible to feed views on main thread, working without blocking and thus, not affecting user experience.

### 3.3.1.3 Mobile Main Architecture

This section shows a diagram that explains the client’s main architecture and how mobile application communicates with web-service through requests and updates events.

![Mobile Main Loop Diagram](image)

**Figure 3.8 - Mobile Main Loop Schematic**

With this setup, each update made by user is done through the Worker Context on a background thread (1), and from there, a data package is passed to Web-Server Module (2), which will be responsible to send the event to Web-Server (3). After that, server responds to client with a UTC timestamp to confirm update (4), which is then updated on WOC (5). Upon receiving timestamp, WOC can commit changes (6) and user’s interfaces are refreshed with new content (7). From this point, both Main and Master contexts will commit their changes until they are saved in Persistent Store (8) (9), and once Master is running on a background thread, the read/write process does not affect performance nor user experience.

Although this configuration brings all these advantages referred above, the idea of having the Main Context to save his uncommitted changes to its parent context, can in some situations, affect the user experience depending on the amount of data that needs to be
merged at a time. To solve this problem, the application needs to ensure that data is saved in small chunks, especially on Main Context to avoid long blockings on user interface.

It is also important to refer that Core Data Module Class is sub-divided into sub-modules, each responsible for a component, namely, Updates Users, Relations Users, Conversations Users, just like on server as I explained above. Broadcast Users is not defined on client because, as I referred, Web-Server Module Class is the one that handles all communications with server. The next Table (see Table 3-23) presents all this sub-modules with a brief description:

Table 3-23: Core Data Sub-Modules on client

<table>
<thead>
<tr>
<th>Core Data Sub-Modules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UpdatesUsers</td>
<td>✓ This class implements all updates on User's Info and Status, including user preferences.</td>
</tr>
<tr>
<td>RelationsUsers</td>
<td>✓ This class gathers all methods that update objects from Relations Info’s.</td>
</tr>
<tr>
<td>ConversationsUsers</td>
<td>✓ This class represents all actions to perform about conversations on core data objects.</td>
</tr>
<tr>
<td>CoreDataClass</td>
<td>✓ This class act like a low level custom API implemented to help all CRUD operations requested from the each sub-module over database on client.</td>
</tr>
</tbody>
</table>

3.3.1.4 Controllers

All controllers were implemented and structured to support the connections with both model and views working as a bridge between them. With all application’s main components present, each controller was specially designed to contribute at least to one of the features as next figure and table illustrate:
Figure 3.9 - Overview of mobile application structure on XCode

Table 3-24: Client-Side Controllers

<table>
<thead>
<tr>
<th>Controllers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login_Controllers</td>
<td>✓ Controllers responsible for Account Management including Login and Validations.</td>
</tr>
<tr>
<td>Profile_Controllers</td>
<td>✓ Group of controllers that takes care on Profile Pages both from User Logged In and all his related users.</td>
</tr>
<tr>
<td>Contacts_Controllers</td>
<td>✓ Controllers responsible for every type of relation possible on application, namely Contacts, Invites and Requests.</td>
</tr>
<tr>
<td>Search_Controllers</td>
<td>✓ This group gathers all controllers used for searching new contacts, finding people near by User Logged In, etc.</td>
</tr>
<tr>
<td>Chat_Controllers</td>
<td>✓ Collects all necessary controllers to handle each part of chat component.</td>
</tr>
<tr>
<td>Settings_Controllers</td>
<td>✓ Controllers responsible for presenting all settings and configures available on application, from Profile Settings to Visibility preferences.</td>
</tr>
<tr>
<td>Others_Controllers</td>
<td>✓ This group gathers mainly subclasses from main components of visual part likeUITabController and UINavigationController extended for specific purposes.</td>
</tr>
</tbody>
</table>
Chapter 4

4. Results and Interfaces

This last chapter will be used to present all interfaces and results for all parts of the application. They represent the output of all that was explained on this thesis. All views will be explained in sub-groups organized in a similar way to controllers.

As referred above, application is available for both iOS6 and iOS7 versions, and so, although it is optimized and designed to iOS7, some interfaces will have both versions to see the differences.

4.1 Login Views

Login Views are the initial interfaces visible to user, since they are used to perform account management, including login with phone number and accounts validations.

4.1.1 Login Account

This module allows a user to login using their phone number. By using the phone number with international codes in format dialCode+countryCode+phoneNumber, all users will guarantee a unique id to apply and syncing with contacts from address book. The next figure shows the login View.

![Login Views iOS7 with country code+phone number](image_url)

Figure 4.1 - Login Views iOS7 with country code+phone number
As you can see, when a user starts application for the first time, a login is required including the country code, selected by the picker element, and a phone number. Only after filling both attributes, user can validate account.

4.1.2 Validate Account

On this View, user needs to validate the account with a unique code that server sends by sms to phone number logged in. With this procedure, server certificates that the phone number introduced corresponds to the account logged in the device. This login process is already used by applications like WhatsApp and Viber (with phone number).

![Validation View](image)

(a) Account Inactive  
(b) Account Active

Figure 4.2 - Validation View (iOS7) with account status inactive (a) and active (b), after inserting validation code received by sms

Here are presented all steps throughout validation. After logging in, user account is inactive as illustrates Figure 4.2-(a). After that, a sms is sent to phone number with a validation code that user can use to activate account (see Figure 4.2-(b)). With validation done, user can press button “Done” on navigation bar.

From here, follows a sequence of interfaces like a tutorial that helps user to configure profiles, preferences and all basic information.

In order to provide this sms service, the server uses a Web API supplied by Twilio, enabling it to send messages to any valid phone number anywhere in the world.
4.2 Profile Views

All profile views are aggregated to the social component of the app and represent the interfaces of profiles for both user logged in and all others users, namely contacts. Each profile can configure a profile image and header image so new people can meet them better. However, as you can expect, layouts will be different given that user logged in will have extra info, like pending requests, invites, etc. On the other hand, on public profile two extra fields are added to indicate the number of common contacts between a specific user and user logged in the device and distance from user logged.

4.2.1 Me Profile

Starting on “Me” profile, as said above, it is used to show all data and info related to user logged in.

Figure 4.3 - “Me” Profile (iOS7) with all profile info and last notifications

These last screenshots illustrates the “Me” Profile for both iOS6 and iOS7 and all the info referred above including a list of last notifications. Also on Figure 4.3, that group of notifications shows an invite sent by other user to become contact, enabling the user to accept or decline the invitation.

Also important to refer that version iOS6 (see Figure 4.4) maintains the same structure and disposition of components, despite the changing of look on the majority of the elements.
4.2.2 Public Profiles

About public profiles, they are used to represent the “public” info of other user in application. As mentioned earlier, the available info and actions on it are different, because we want to be able to talk with them, localize them on map, perform a relation action (request a person for contact). The next figures will demonstrate all the differences and main purposes to those changes.

Figure 4.4 - “Me” Profile (iOS6) with all profile info and last notifications

Figure 4.5 – Public Profile Views (iOS7) from user with public info
Figure 4.6 – Public Profile View (iOS6) from user with public info

All three views represent public views from users that are contacts, sent an invite or were request by user logged in to become contacts.

The main bar contains info related to user status (offline/online), distance to user logged in, relation status and common contacts between both (presented in this order).

Referring to all public actions that can be performed to each user, the first one indicates the status between the two users and/or a possible status to achieve by pressing that button. When user is already a contact, “View Contact Info” will push a view with contact info from address book, given that user automatically synchronizes with it.

About the second action, it enables users to initiate/continue an ongoing conversation between them (user logged in and contact user). It is important to remember that group conversations are a feature that is included for upcoming releases.

The last available action indicates whether or not the user is visible on map, and by pressing it, it opens the map view and selects the specific user.

Again, the differences between both versions do not compromise the main structure and purpose of each element on view, which is also an advantage.

4.2.3 Contacts, Invites and Requests Interfaces

This component relates also to social component where are organized and listed all Contacts, Requests and Invites. Each interface is supported with information related to that user.
Figure 4.7 - Contacts View (iOS7) on both normal and filtering mode.

Figure 4.8 - Contacts View (iOS6) on normal mode.
These layouts illustrate almost all possible Contacts Views Interfaces that allows user logged in to see all contacts, including their connection status and distance to him (see Figure 4.11 – (a)). These examples show contacts not visible (out of range) and near by (<500 m).
meters), allowing view to turn into filtering mode (see Figure 4.11 – (b)), where user can filter by online status and visibility.

In addition to that, a button is shown at the bottom of the list to invite contacts to use “Hi There” application by sms and/or e-mail, suggesting automatically all phone numbers and emails from address book.

The next interfaces correspond to Invites and Requests Views Controllers (see Figure 4.9) where are listed all pending invites and sent requests to/from user logged in, respectively, waiting for a response. By accepting invitation/request, the user will turn into contact and synched with address book automatically.

To finalize, associated with this group of views is the one that shows all common contacts between user logged in and other selected user (see Figure 4.10). This option is only available with users that share their common contacts. This feature allows users to be better known by others for possible future relations.

### 4.3 Find Views

Find Views support the location services managing all positions and visibilities, enabling a nearby feature. All data is presented through maps and lists, so user can have an easy perception to people around him. The following figures show these views in some possible scenarios, organized by maps views and lists.

![Find Map View in both iOS7 (a) and iOS6 (b) versions](image)
Both views represent the layouts on Find Tab in map mode (see Figure 4.11). As they suggest, user logged in will be able to know who is near by and clicking on them to see more info about relation status and distance. Moreover, an info button appears to push a view controller with all public profile info of user.

Other elements like buttons on down right-hand corner enables user to track himself all the time (user is always in center) and also to activate 3D mode, although this feature is only available on iOS7. It is important to point out that all visibility parameters, for instance visible only to contacts, can be accessible and changed by clicking on “Visibility” button with ease.

These last layouts show the same content that Find Map view but in list format (see Figure 4.12). They also show a Search bar enabling users to filter by “All”, “Contacts” and “Contacts in Common” plus a name or phone number, for a more fine-grained filtering. Here is a situation where user is using the search bar to filter people visible, in this case, filtered by all contacts.

![Find List Views (iOS7) with users near by.](image)

**Figure 4.12 - Find List Views (iOS7) with users near by.**

### 4.4 Conversations Views

Conversations Views represent all interfaces responsible for the chat component. They are divided into Chat view and Conversation View. The first lists all conversations ongoing and enable starting new ones. The second one shows all conversation history of a specific
conversation Id, including all messages of type text, dates, conversation info and conversation users.

4.4.1 Chat View

As mentioned earlier, this view shows all peer-to-peer and group conversations already started by some user, where user logged in is one of the participants.

![Chat View examples](image)

(a) iOS7  
(b) iOS6

Figure 4.13 - Chat View showing all ongoing conversations.

As Figure 4.13 illustrates, all conversations are ordered by last message sent date in descending order (from recent to old). Each shows all standard info, like user’s name and profile image to which we are talking, last message sent text and date, unread messages count and some custom info, as status connection and user’s visibility.

Once again, comparing both versions, they dispose the same main structure, which again is an advantage.

4.4.2 Conversation View

After clicking on a specific conversation from chat view, a conversation view is pushed with all messages from that conversation.
These figures show the aspect of each conversation view showing all messages in it ordered by date. Also all custom information related to status of the other user is presented, including connection status and visibility from user logged in. On the bottom, there is a text view to send new messages and also an location pin that gets enabled when user is near by, and by clicking on it, locates him on map.

4.5 Settings Views

This last group of views (see Figure 4.15) allows user to configure all preferences and properties related to all three main topics already mentioned. They are divided in two different sub-settings: Settings Info and Settings System. Settings Info allows user to configure and see their profile info, visibility preferences, etc. The second shows all information related to web service status and application version.
Figure 4.15 – Settings Main Views

4.5.1 Settings Info

Settings Info gathers all sorts of configurable parameters, like profile info, visibility, account and general preferences (push notifications, etc...). The first settings view presents all profile info (Settings Profile View) that can be updated, namely, profile and header image, public name and email (see Figure 4.16).

The Settings Visibility View (see Figure 4.17) is also very important on the location-based services, especially setting ranges to all/contacts (kms/miles) and visibility filters.
The next Settings sub-views represent all preferences that can be handled and updated by user. Amongst all options available, it has Range scales, notifications, permissions,
warnings and sounds. As it can be seen on next images, all parameters are very self-explanatory. About Notifications and Permissions, user is invited to synchronize all contacts with address book once he starts running app for the first time.

(a) Syncing mode  
(b) normal mode

Figure 4.18 – Settings Preferences Views

The last layouts attached to Settings Info is the Settings Account View (see Figure 4.18) that outputs the status of current logged in account and allows user to delete it or change to another account.
4.5.2 Settings System

Settings System represents a group of views to inform user about system status, through a twitter page, and application settings, namely versions and builds.
Chapter 5

5. Tests and Performances

5.1 Server – Performance Results

To prove and justify all tools used in the implementation of server (section 2.2.2), here are some performances tests [23] to see how node.js behaves compared to Apache Servers when working as a simple web service to render a very simple web page. In order to perform these tests ApacheBench 2.3 was used.

- Total Requests: 100,000; Concurrency Level: 1,000:

(a) Node.js results:

(b) Apache-PHP results:

Figure 5.1 - Performance Results on both Node.js (a) and Apache PHP (b) servers

As all results show, node is really fast compared to the Apache results, more than 5x faster, with many more requests per second, higher transfer rate and a much smaller number of failed requests. As expected, to deploy such performance, Node.js consumes much more CPU and memory. In addition to this, given that module ‘cluster’ allows multi-threading processes on Node.js, it is expected that performance can still improve.
5.2 Mobile Application – Performance Results

In order to obtain the best performance and user experience on a mobile application, the most important factor is to ensure that all the expensive processes, like updating/requesting properties from server, syncing contacts, importing data, etc., are performed in background threads. This way, it guarantees that the main thread is as free as possible, being available to refresh all layout components and responsive to input actions from user.

The next figures were obtained from XCode’s Instruments Tools and measure the traffic in the main thread amongst all worker threads on each specific case (requests/updates, sync's, imports).

Figure 5.2 - Memory Usage from both servers

Figure 5.3 - Time Profiler: Syncing Contacts part 1
As both figures illustrate, the synchronization process is a very expensive process that gathers two distinct parts. The first one (see Figure 5.3) represents a time interval that fetches contacts from address book and sync them with server, performed mostly on a worker thread in background (48.2% of the time) while showing a progress bar with current status, on main thread (19.7% of the time).

The second figure (see Figure 5.4) represents the time interval after receiving the response from server and refreshing all interfaces with all new relations (main thread – 57.4% of the time; worker thread – 16.5% of the time). All this occupation on main thread is mostly related to the saving process that occurs in a parent-child multi-contexts stack, therefore all relations pass through to main thread context (MOC), as explained above on core data stack, without blocking the user interface. Also some of this time is spent on refreshing interfaces with new and updated relations.
These two figures above (see Figure 5.5 and Figure 5.6) show how application behaves on updating a user's property, specifically the profile image, which is one of the longest tasks that user can perform. The first figure indicates a selected area that corresponds to the process of preparing and sending image to server, including the base64 encoding, where worker thread spends 19.2% and main thread occupies 12.9% given the refresh on layout with the new image.

The second figure profiles the time on client receiving confirmation from server about updated image. This task is mainly done on main thread, about 52.5% given that interfaces are refreshed and contexts are saved, including MOC context. Despite being a high percentage on main thread, this process only takes about 0.5s (selected area) and so, interface will still be responsive.

All updates and requests are done similar to this process, as explained above on mobile application main loop (see Figure 3.8), where each task and updates on model objects are done on a worker context WOC, on a background thread. After receiving a confirmation from server about the update/request, a saving process occurs and all interfaces are refreshed on main thread.
Chapter 6

Conclusions and Future Work

All components and interfaces here presented are part of the final version of the mobile application “HiThere” for iOS devices. Apple has already approved this application and it is now available for download in App Store.

Throughout this thesis, each component was described and contextualized with the application. Amongst all technologies and tools studied for each component, the ones used were properly explained and their utilization justified, for instance, deciding going native instead of implementing a hybrid or web application and providing all major communications through web-socket channel instead of http requests. In addition to that, despite all major architectures and methodologies used were implemented from scratch by me, some of them were studied from others articles or concurrent applications and applied to this specific case, for instance, the synchronization process with address book or core data stack architecture from client model.

This application was designed and implemented to be competitor of the current market leaders apps, namely WhatsApp and Viber, recognizing that all new features introduced may overtake the robustness of these applications, mainly the introduction of user’s location as part of the process of starting a conversation or meeting new people.

Knowing the market for this kind of application, there are always new features and improvements that can be done. For example, a possible future release will definitely include group talks, which as I mentioned before, are already implemented on server but not entirely on client.

Depending on the success of the application in iOS, for instance, a thousand of registered users, a possible android version may be deployed in a near future to increase the market.

In addition to that, aweb site may be developed to support the mobile app with documentation and extra information.

Despite of achieving or not a market share amongst these applications, I enjoyed and learned a lot with each part of the entire process, which gave me a huge base and knowledge for the years to come.
References


   https://blog.surfnet.nl/?p=2918


