Location based Social Network
Tiago Alberto Gorgulho Gonçalves
Department of Electrical and Computer Engineering
Instituto Superior Técnico - Technical University of Lisbon, Portugal
Email: tiago.gorgulho@ist.utl.pt

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 a in 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.

Index Terms—iOS, Location Services, Mobility, Social Network, Chat, Mobile Application, Web Sockets

I. INTRODUCTION

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 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.

A. 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.
II. MOBILE APPLICATION DEVELOPMENT

Mobile app development presents unique challenges, like different operating systems and devices, display sizes, device 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.

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 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 [2] and Push Notifications both Local and Remote [3]. 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.

App Development Comparison

![App Development Comparison Table]

- **Native**
  - Device Access: Full
  - Speed: Very fast
  - Development Cost: Expensive
  - App Store: Available
  - Approved Process: Mandatory

- **Hybrid**
  - Device Access: Partial
  - Speed: Reasonable
  - Development Cost: Reasonable
  - App Store: Available
  - Approved Process: User Overhead

- **Web**
  - Device Access: Partial
  - Speed: Fast
  - Development Cost: Reasonable
  - App Store: Partial
  - Approved Process: None

Fig. 1. Advantages and disadvantages of Application Development Tools [1]

A. 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 all development tools used for the client part:

1) **iOS SDK**
2) **XCode Framework**
3) **iOS Simulator**

B. 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.

To implement the server part of the application Node.js was the web framework used. Node.js [4] is a platform used to build fast, scalable network applications. It is a wrapper around the high-performance V8 JavaScript runtime [5] from the 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 [6].

III. IMPLEMENTATION METHODOLOGIES

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]

Fig. 2. Diagram of Application Main Architecture

As diagram on Fig. 2 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 I/O 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 [7].

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 1 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-Socket 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).

A. 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’s explained in-depth how server behaves and the main steps it takes to process different events

1) Web-Server Architecture

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

![Web-Server Architecture Diagram](image)

As illustrated in Fig. 3, 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 WebSocket connections. Thus, to implement it, Node.js provides a framework called Socket.IO [8] 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 “ErrorHandle.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.

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 [9] resources and T-SQL language, the usage of procedures and routines for data manipulation and processing, can significantly benefit performance. The next figure represents database’s final design.

![Database Final Design](image)

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 in 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 refreshes an UTC timestamp of the related user, according to the specific event and info: lastUpdateIsOnline (online status), lastUpdateTime (profile info), lastUpdateImage (profile info/image), lastUpdateHeader (profile info/header), lastUpdateStatus (profile info/status), lastUpdateConversation (conversation info).

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, 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) 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.

   a) 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 re-directed to the specific module so it can be processed.

This module is also responsible for connecting to Apple Push Notifications (APN) [10] 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.

   b) 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.).

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.

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.

c) Relations Users

This next module implements all necessary routes and events to handle all connections between users, namely relations like contacts, requests and invites.

From the server perspective, when each user forces a synchronization with their address book, all contacts, requests and invites are sent to server as an array. 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.

This enumeration “relationsUsers” defines all possible status of relations between two users. Starting from PENDING, it represents a state after sourceId 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 sourceId to targetId to become contacts. All the others values UNKNOWN, DECLINED and CONTACTS are self-explanatory.

d) 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.

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”.

The next table (TABLE 2) helps identifying and characterizing all conversations and messages status:

<table>
<thead>
<tr>
<th>Enumerations</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>conversationsUsers</td>
<td>STATUSES_INVITED = 0</td>
</tr>
<tr>
<td></td>
<td>STATUSES_IN = 1</td>
</tr>
<tr>
<td></td>
<td>STATUSES_OUT = 2</td>
</tr>
<tr>
<td></td>
<td>STATUSES_DELETE = 3</td>
</tr>
<tr>
<td>messages</td>
<td>STATUSES_INIT = 0</td>
</tr>
<tr>
<td></td>
<td>STATUSES_SENDER_ACK = 1</td>
</tr>
<tr>
<td></td>
<td>STATUSES_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>
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.

c) 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 shows its architecture and main workflow:

![Broadcast Architecture Diagram](image)

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 sourced 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.

4) Scalability and Robustness

As I mentioned on section 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.

B. Mobile Implementation

After explaining all web-server components and all main events and routes that enables all communications between server and clients, it’s 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 Fig.9) illustrates:

![Model-View-Controller design pattern used in mobile application](image)

Fig. 6. 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 topic will explain the process throughout the implementation of the first two parts: Model and Controllers, and the interfaces will be presented in next chapter – Results and Interfaces.

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, which allows a project to keep all libraries used up-to-date.

a) 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 redirected 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.

b) 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.

Starting from the setup of the model, 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 (“physical” 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 [12], the one used is presented on the next figure:
As it can be seen, it was adopted a parent-child multi-context architecture (see Fig.8). 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.

c) 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 Schematic](image)

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:

### TABLE III

<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.</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>

d) 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 one of the features as next table illustrate:

### TABLE IV

<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 like UITabBarController and UINavigationController extended for specific purposes.</td>
</tr>
</tbody>
</table>
IV. 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 shortly explained in sub-groups organized in a similar way to controllers.

A. 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. Here are some interfaces:

![Fig. 10. Login Views (iOS7) with both login and validation interfaces](image)

B. 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. Here are some views from profile group:

![Fig. 11. Profile Views (iOS7) with both “Me Profile” and Contacts Interface](image)

C. 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 and lists:

![Fig. 12. Find Views (iOS7) in both Map and List Modes](image)

D. 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. Here are some samples from this group:

![Fig. 13. Conversations Views (iOS7) with both Chat and Conversation Views](image)

E. Settings Views

This last group of views 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. Here are presented some interfaces:

![Fig. 14. Settings Main View (iOS7) and System Status](image)


V. TESTS AND PERFORMANCES

A. Server – Performance Results

To prove and justify all tools used in the implementation of server (section II.B), here are some performances tests [11] 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:

![Fig. 15. Settings Info Profile, Visibility and Preferences (iOS7 version)](image)

![Fig. 16. Performance Results on Node.js server](image)

![Fig. 17. Performance Results on Apache PHP server](image)

![Fig. 18. Memory Usage from both servers over time](image)

As all results show, node.js 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.

B. 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 a specific case, syncing contacts with
address book:

![Fig. 19. Time Profiler: Syncing Contacts part 1](image1)

![Fig. 20. Time Profiler: Syncing Contacts part 2](image2)

As both figures illustrate, the synchronization process is a very expensive process that gathers two distinct parts. The first one (Fig. 19) 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 Fig. 20) 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-context 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.

All updates and requests are done similar to this process, as explained above on Mobile Main Architecture, 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.

VI. 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. 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, a web 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


