I. Abstract

The goal of my project was to create an iPhone application capable of tracking the locations of an individual’s friends in real time. The final product that I created is an application called the Marauders App. As I went about building this application, I was able to explore many different aspects of iPhone application development. My first tasks involved designing the basic layout of the application and structuring the user network. From there, I delved into the implementation of these elements. On the user-facing side, I familiarized myself with Apple’s XCode IDE and learned how to structure and build full-scale applications for iOS. For the back-end, I set up a web server to process requests to a MySQL database, which I used to store all of the user information for the application.

II. User Interface Design

1. Basic Application-side Design

I chose to structure the application around the user’s friend groups. The first screen that the user sees after logging in is a table of his friend groups. From here, the user can proceed in three directions. Clicking directly on the cell will take the user to a map, showing the locations of all of the user’s friends in that group. If the user clicks on the “add” button on the top right, he will be taken to a screen that will allow him to create more groups. Lastly, clicking on the blue disclosure button on the right will show which of the user’s friends are included in the group.

From the table showing all of the friends in a group, clicking on the friend’s name will bring the user to his profile. Clicking instead on the “add” button on the top right will allow the user to add more friends to the group.

Below I have added a series of screen shots that illustrate the flow of the application from one screen to the other. Simply follow the matching numbers to see which buttons correspond to which screens.
These eight screens above illustrate the main functions and layout of the application.

2. Social Network Structure
I decided to make the social network structure similar to that of Facebook’s structure. In order for a user to view a friend’s location, he must first send him a friend request. Then his friend must accept his request before either party can see the other’s location.

The main difference between my network’s structure and Facebook’s structure is the fact that all friends must be placed into a group before the user can see their locations. As shown from the application screenshots above, a user can only send a friend request from within the context of one of his groups. So when his friend accepts his request, his friend is automatically added to that group. On the other side, although that friend has just accepted the first user’s request, he must place the first user in a group before he can track his location.

Each user has his own set of groups, independent from the groups of his friends, so each user can group his friends in any way that makes the most sense for him.

Another decision that I made was to allow friends to be placed into multiple groups. This allows each user another level of flexibility when organizing their friends.

3. Updating Location Information
Another major element of the iPhone application design was the management of location updates. There were a number of different schemes that I could have used to handle how often users updated their locations to the server and how often those locations were broadcast from the server to their friends.

I could have allowed continuous location broadcasting and continuous reception of location information; I could have implemented request-only reception and broadcasting; or I could have implemented continuous broadcasting with request-only reception of friend location information.

Each of these methods had unique downsides. Continuous location broadcasting would have drained battery life and consumed significant amounts of data, so its practicality was a major concern. Continuous friend location requests would have also required a lot of data. Request-only location broadcasting had the issue of being potentially slow and unreliable. If a user wished to request a friend’s location but the friend did not have service or did not have the application open, the request would likely fail.

Initially, I had hoped that I would be able to implement intermittent updates, with the application running in the background at all times. However, I learned that Apple has pretty strict policies
against this method (because of the data usage and battery drain), and that my app might be rejected from the app store if it used this method. Applications are typically allowed to run in the background if they have a short task that has not yet been completed; however, Apple labels it bad practice to have apps running in the background indefinitely.

Ultimately, I decided to only broadcast the user’s location when he opened up the map view in the application. Similarly, the user would only request location information from the server when he opened up the map view. This approach had the benefit of practicality, as all location updates and location requests would only occur exactly when a user wished to see the (last stored) locations of his friends.

III. iOS Architecture

1. Apple’s Xcode IDE and the Cocoa Touch API

One of the early challenges with the project was familiarizing myself with the iOS development environment. The first step was getting to understand the Model-View-Controller framework- a design pattern common to many user interfaces. From there, I learned how to use the Xcode interface builder to design my own views and how to pass information from the view to the controller. The first several weeks of the semester involved creating dozens of sample applications and learning how to use these tools and others in Apple’s Xcode IDE.

2. Navigation Controller and Tab Controller

The Marauders App application centers around the capabilities of the objective-C view objects: the navigation controller and tab controller. The tab controller allows the user to switch between the three different portions of the application: the groups and maps views, the pending requests view, and the settings view. The navigation controller allows the user to navigate between the different pages in the groups and maps views.

The navigation controller is implemented as a stack, so all views are simply pushed onto or popped off of the stack. Views can also pop directly to the root controller, which in the case of my application is the Groups View. The following is a diagram of the navigational flow of the application stack (including the tab controller layout):
3. Controllers and Objects

The following is a list of all controllers and objects in my project:

- GroupsController
- AddGroupsController
- IndividualsController
- AddIndividualsController
- IndividualProfilesController
- LoginController
- NewUserController
- MapController
- PendingRequestsController
- SettingsController
- LocationController
- IndividualLocation
- PhoneNumber
- BackgroundUpdate

The 10 controllers (except LocationController) each correspond to one of the views in the application. The 4 objects (including LocationController), each store different types of data, necessary for the functions of the application.

The LocationController object manages the update timing for the user. This object is called whenever the user wishes to update his current location. As the application is set up, this occurs whenever the user enters the Map view. The IndividualLocation object stores the latitude and longitude coordinates of each of the user’s friends to be displayed in the Map view. The PhoneNumber object stores the name and phone number of the user once he logs into the application. It seems odd to have to input your own phone number when the entire application is based around users on mobile devices; however, Apple discourages direct usage of device phone
data in applications on their App store. So, I needed to have users fill out their phone number themselves as part of their login. The BackgroundUpdate object was meant to monitor settings for allowing or disallowing background updates; however, as I discussed in the Updating Location Information section above, I chose to disallow background updates for this application.

Each of the controllers above translates the user inputs on their respective views into operations on the model. These user inputs trigger all of the queries to the MySQL database- for data requests and for saving user network data.

For instance, loading the Groups view triggers a request to the database server to display all of the user’s groups. Adding a new group sends a query to the database to save the new group. Similarly, loading the Friends view triggers a request to the database server to display all of the user’s friends in that particular group. Loading the Find Friends view triggers multiple requests to the server to find active friends, pending requests, and all users. Clicking on these one of these sections triggers a request to the server to: add a friend to the group, accept a pending request, or send a friend request to another user.

### IV. Back-end Architecture

#### 1. Overview

To manage the network of users and their groups and relationships, I created a MySQL database on the zoo database server. To give each app user read and write access to the MySQL database server, I also set up a number of PHP-enabled web pages on the zoo web server.

Each time a user wants to see his groups or his friends, a request is sent to the web server to pull the relevant data from the database. Each time a user wants to add a group or a friend or accept a pending friend request, new data is sent to the web server to be written to the database. Each time a user wishes to see his friends’ locations, their location information is also pulled from the database via the web server.

#### 2. PHP-Enabled Web Server

The way that I designed the app to communicate with the MySQL database was through a PHP-enabled web server. This web server was set up on the zoo servers by the computer science department’s system administrators.

Whenever a database query needs to be made, the iPhone application sends a POST request to a specific PHP page hosted on the server. This page accepts the POST data and passes it on to the database. In cases where data is requested, not just inserted, the PHP page then returns the
database data as a simple string, delimited by commas and semicolons. This string is then passed to the application, where it is parsed and the relevant information is displayed on the screen.

The following is a list of every PHP page that I created- each handling a different type of query to the database server:

- acceptPendingRequest.php
- addFriend.php
- addGroup.php
- addPendingRequest.php
- addUser.php
- checkLogin.php
- checkNewUser.php
- deleteFriend.php
- deleteGroup.php
- getFriends.php
- getGroups.php
- getLocations.php
- getPendingRequests.php
- getUsers.php
- updateLocation.php

3. MySQL Database Design

My database contains four tables, named:

   a) main
   b) friends
   c) groups
   d) pendingRequests

These four tables each handle one major aspect of the user network. The main table stores each user’s profile information, friends stores the active friendships between all users, groups stores every user’s unique organization of his friends, and pendingRequests stores all of the un-resolved friend requests between users in the system.

a) main

My main table stores the profile information for every user in the system. Here is the table’s format:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
<tr>
<td>Number</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
</tbody>
</table>
The fields Name, Number, and Password make up the primary key and can never be null. Latitude and Longitude are initialized to NULL before the application updates the user’s location for the first time. Last_Updated stores the date and time of the most recent location update. If there has not yet been a location update, Last_Updated stores the date and time that the profile was created.

b) friends

My friends table stores the active friendships between all users. Here is the table’s format:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>varchar(45)</td>
<td>No</td>
<td>--</td>
</tr>
<tr>
<td>Number</td>
<td>varchar(45)</td>
<td>No</td>
<td>--</td>
</tr>
<tr>
<td>Friend_Name</td>
<td>varchar(45)</td>
<td>No</td>
<td>--</td>
</tr>
<tr>
<td>Friend_Number</td>
<td>varchar(45)</td>
<td>No</td>
<td>--</td>
</tr>
<tr>
<td>Group_Name</td>
<td>varchar(255)</td>
<td>Yes</td>
<td>--</td>
</tr>
</tbody>
</table>

Every friendship is stored at least twice in this table. For instance, if User 1 accepts a friend request from User 2, the data is stored in the table as:

<User 1>  <User 1’s Number>  <User 2>  <User 2’s Number>  <NULL>
<User 2>  <User 2’s Number>  <User 1>  <User 1’s Number>  <User 2’s Group>

Remember that all friend requests must stem from within a group. User 2’s friend request must have included a group name, so after User 1 accepted his friend request, he was automatically added to that group. User 1 has not yet added User 2 to a group, so the group name for his relationship is still NULL.

If User 1 chooses to add User 2 to one of his friend groups, the database simply replaces the NULL group name with the new group name. Afterwards, if he chooses to add User 2 to a second group, a whole new row is inserted into the table:

<User 1>  <User 1’s Number>  <User 2>  <User 2’s Number>  <User 1’s 2nd Group>

The friends table stores a unique entry for each user’s relationships and group names. Therefore, if User 1 has placed User 2 in 5 groups, and User 2 has placed User 1 in 3 groups, there will be a total of 8 rows in the table describing their relationships.
Initially, I had imagined that I would only store 1 row for each relationship. The row would store the two friends, their numbers, and the groups that each one had placed the other one in. However, I realized that with only 1 row (instead of at least 2 rows) for each relationship, the query to find all of User 1’s friends would become more complicated. Several queries involve finding a user’s current friends, and each of these queries would have doubled in complexity. With a small expected user base, I decided that the reduced complexity of queries was probably worth an additional row in the database for each relationship.

Another decision that I made was to store each additional group in a different row. At first, my plan was to store all groups that included the same friend in the same row—in the Group_Name field. That is why the Group_Name field data type is varchar(255) instead of varchar(45). I had planned to store all of the groups in that field. Again, this type of structure would have added to the complexity of the database queries. I again made the decision to add redundant rows of data for the sake of simplicity.

c) groups

My groups table stores the groups created by each user in the system. Here is the table’s format:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type:</th>
<th>Null:</th>
<th>Key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
<tr>
<td>Number</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
<tr>
<td>Group_Name</td>
<td>varchar(255)</td>
<td>No</td>
<td>Primary</td>
</tr>
</tbody>
</table>

Similar to the Group_Name field in my friends table, I had originally planned for each row to store all of the groups created by that user. However, I quickly found that it would be simpler to store each user-group pair in a different row.

All additions to this table are made from the Add Groups screen in the application. Deletions come only from the Groups screen. All of the fields are included as part of the primary key—this is for the purpose of using the MySQL “Replace” command in my queries, which checks the new values against the primary keys in the table.

d) pendingRequests

My pendingRequests table stores all of the unresolved pending friend requests in the system. Here is the table’s format:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type:</th>
<th>Null:</th>
<th>Key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender_Name</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
<tr>
<td>Sender_Number</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
<tr>
<td>Receiver_Name</td>
<td>varchar(45)</td>
<td>No</td>
<td>Primary</td>
</tr>
</tbody>
</table>
The format of the `pendingRequests` table is fairly straight-forward. When User 1 sends a friend request to User 2, a row is entered into the table of this format:

```
<User 1>  <User 1's Number>  <User 2>  <User 2's Number>  <User 1's Group>
```

Again, all of the fields are included as part of the primary key for the purpose of using the MySQL “Replace” command in my queries. The `pendingRequests` table is then queried to display the user’s pending friend requests on the `Pending Requests` screen as well as the `Add Friends` screen.

4. MySQL Database Integrity

When building the queries to this database, one important aspect was protecting the correctness of the database. I needed to ensure that database protected against duplicate insertions. The biggest concerns were regarding insertions of new users, friends, groups, or friend requests. Each request required a check of the database to ensure that errant or duplicate data was not being added. Removing friends from groups also required integrity checks of the database data. For instance, if a user’s friend belonged to multiple groups, removing him from one would require a query to the database to delete the row in `friends` storing that relationship. However, if the user’s friend only belonged to one group, removing him from that group should not remove any rows from the database. It should simply change the `Group_Name` field to NULL. These types of checks needed to be made in almost all queries to ensure accurate data storage.

The second concern with the database was ensuring that even errant requests wouldn’t result in database errors. With an entire network of users, I knew that it might be possible for the timing of database updates to cause errors with different actions or queries. One example is the handling of friend requests. Ideally, after User 1 has requested to be friends with User 2, User 2 would be unable to send a friend request to User 1 - he would only be able to accept User 1’s request. However, if User 2 is on the `Add Friends` screen when User 1 sends his friend request, it is possible that User 2 could also send a friend request back to User 1. In specific cases like this one, I added multiple constraints preventing database errors when the network timing resulted in unexpected queries.

V. Application Deployment

After finishing designing the application, I chose to submit it to the Apple app store for distribution. I would have liked to have actual users testing my application before writing up my final report for the project; however, the app store has not yet finished reviewing my application.
The application requirements were fairly stringent as well. To distribute an application on the app store, the app must have a development certificate from the app store. It must also have a registered provisioning profile and application ID.

The first time I submitted my app, and it was actually rejected the first time for not having a sufficiently in-depth description attached. Hopefully the app will be accepted this second time around, and it will be available for download within the next week. It should found using the key words: Yale, Marauder Map.

VI. Conclusions

After finished the project, I have now successfully completed all stages of the application development process. I have covered all of the basic elements of a geo-location and social networking application. Using objective-C, PHP, and SQL, I have developed every aspect of the platform for the Marauders App application.

From this project, I have developed a much stronger understanding, not only of the languages and tools used in the project but also of the entire development process as a whole. In my opinion, the lessons that I learned with regards to structuring the application, iterating through different versions of the architecture, and modifying the overall design are just as valuable as the technical lessons learned.

This project also pushed me to learn about different elements of application design that I hadn’t previously considered at all. I was forced to think about network encryption and security for the user information on the database (even though I ultimately did not have the time to implement a secure transfer protocol). I also learned how to design my database with scalability and reliability in mind.

Ultimately, I am very satisfied with the experience that I gained from creating the Marauders App as my final computer science project.

VII. Source Code

The source code for my application can be found on the CPSC 490 website.