King's College London
Department of Informatics

Development of a chatting application

Group Project Report
7CCSMGPR

March 2017

Group:
Synomilia

Supervisors
Dr. Laurence Tratt
Dr. Elizabeth Black

Contents

1 Introduction **4**

2 Review **5**
2.1 WhatsApp 5
2.2 Google Talk and Skype 6

3 Requirements and Design **7**
3.1 Functional Requirements of the Server 7
3.2 Functional Requirements of the Client 8
3.3 Use Cases 9
3.4 Mapping Functional Requirements to Use Cases 10
3.5 General notes 10
3.5.1 Socket Programming 10
3.5.2 Database Design 10

4 Implementation **12**
4.1 Server 12
4.2 Web Application Client 14
4.3 Android Client 18
4.4 Testing 20

5 Teamwork **23**

6 Evaluation **24**
6.1 Aspects of the final project 24
6.1.1 Server 25
6.1.2 Python Client 25
6.1.3 Android Client 25
6.1.4 The final system 26
6.2 Problems and changes made 26
6.3 Project improvements and future works 27

7 Peer Assessment **28**

Appendices **29**

A Use Cases Description **30**

B Test Results **41**

C Test Unit Code **42**

Chapter 1

Introduction

This report summaries the comprehensive design, implementation and overall team collaboration that was required to meet the objectives of module 7CC-SMGPR – Software Development Group Project. The objective of this project was to develop a distributed chat system, with two separate clients developed in two separate domains, for example, desktop and mobile domains, with the clients being able to search for other people on the server and initiate a chat with them. The chat opens up a connection between the two users that by passes the server. A further requirement entailed that there are three components: a server which can tell users how to contact other people; client A which users can use; and client B which users can also use.

The report firstly highlights a review of related works (Chapter 2) before describing the functional requirements and design envisioned and showcasing the software design of the client and server in UML and detailed Use Cases (Chapter 3).

Furthermore this report details the exhaustive steps taken during the implementation phase of the project (Chapter 4), highlighting the major implementation decisions taken, as well as the testing activities embarked upon.

In conclusion, a critical evaluation of the team work (Chapter 5) and the obtained result (Chapter 6) is presented.

Chapter 2

Review

Communication has evolved. Chat programs prior to this evolution were restricted to dainty, dull and dire chat rooms. The surge of modern mobile technology (and this technology becoming widely available for GEN Y), the introduction and high demand of apps, and improvements in data structures and bandwidth has given birth to a vast array of mobile and desktop chat applications. There currently is an oversaturation of chat programs that are widely available. In this section of the report we discuss a selection of the current batch of distributed chat programs available, and what makes these forms of communication so appealing, accessible, convenient and exciting.

A distributed chat system involves the use of clients and a server (or a cluster of servers) to build a form of communication between those clients. Modern day technology allows this concept to be expanded; making chat programs the primary forms of communication. Below are examples of chat programs from which the group drew inspiration from to build *Synomilia chat.*

2.1 WhatsApp

The most popular chat program *WhatsApp* acted as the bridge between the general public (who previously used SMS text messaging as the medium for mobile communication) and distributed chat systems. The simplicity is what made *WhatsApp* accessible, and eventually what made it the most popular around the world.

A new user can simply install the application on their phone and use it instantly. The main interface consists of a list of chats the user has open and another list of all the contacts who have *WhatsApp* installed on their phone. Once the user clicks on a contact/existing chat, they are taken to the chat screen. The chat screen, like most chat screens is instantly recognisable. The user types in a message and when they send this, this message is displayed on the left of the screen, and when they receive an incoming message, this is displayed on the right. The group drew inspiration from *WhatsApp*'s simplicity. One of the main objectives of the team was to make *Synomilia chat* accessible, and this was done by following *WhatsApp*'s interface as a blueprint for the Android application.

2.2 Google Talk and Skype

The group discussed ideas about the authentication and logging in to the system. This was an essential part of *Synomilia chat*. *WhatsApp* uses a user's mobile phone number to login to the system, as the team also developed a desktop client; this login procedure was not feasible. Therefore, the team used a username authentication method which is used by *Skype* and *Google Talk*. *Google Talk*'s interface is simple to use, with minimalist designs and a lack of clutter making it a popular application among desktop users. This was crucial, as the team wanted to make an application which was easy to use on both Android and the desktop application, following a similar design structure on both clients was essential. This allowed the team to use the *Google Talk* interface as a design structure for the desktop client.

Chapter 3

Requirements and Design

This section describes the software design of the client and server entities according to the Software Engineering Paradigm Unified Modeling Language (UML) standardization.

Initially, the functional requirements for the server side (Section 3.1) and the client side (3.2) are presented. Then the Use Cases (Section 3.3) are identified and mapped to the functional requirements previously identified (Section 3.4). In conclusion, some general notes are presented (Section 3.5).

3.1 Functional Requirements of the Server

In order to have an optimal development of the project, the following functional requirements from the Server side are identified:

(A) the server is able to listen for connections from remote clients

- (A) the server starts a new thread

(B) the server is able to query a database

- (A) to check the existence of a username
- (B) to check the matching of username and password
- (C) to update the information of a user inside the database
- (D) to select information about a user
- (E) to insert a new contact-relationship between two users
- (F) to select all the contacts of a user

(C) the server is able to reply to the client's requests

- (A) to allow the registration of a new user
- (B) to allow the login of a user
- (C) to update the connection details of a user inside the database
- (D) to provide information about a second user
- (E) to pair two users as contacts
- (F) to provide the user her list of contacts
- (G) to allow the logout

3.2 Functional Requirements of the Client

In order to have an optimal development of the project, the following functional requirements from the Client side are identified:

(a) the user is able to log in the application
   - (a) the user inserts her credential
   - (b) the user accesses her personalised main page

(b) a new user is able to register herself in the application
   - (a) the user inserts her new credentials
   - (b) the user accesses a standard main page

(c) the user is able to add a contact
   - (a) the user looks for another user
   - (b) the user sees the new contact displayed in her contacts page

(d) the user is able to start a chat with another user
   - (a) the user starts the connection
   - (b) the user opens a new chat tab
     - (a) the contacted user's chat tab is opened automatically
   - (c) the user sends and receives messages

(e) the user is able to leave the application
   - (a) the user closes all her connections

(f) *the user is able to use a secure connection*

(g) *the user is able to set a status, a name and a display picture*

(h) *the user is able to delete her account*

(i) *the user is able to send emojis and pictures*

(j) *the user is able to back up her conversation*

(k) *the user is able to remove a sent message*

(l) *the user is able to customise the aspect of her interface*

The requirements from (a) to (e) were considered mandatory in the first phase of the software's design, with the only exception of the availability of a contact list for each user: these all have been implemented.

The item (f) was classified as mandatory, but it has not been implemented, as have not the items from (g) to (l) in italics.

3.3 Use Cases

The Use Cases explain how the actors can interact with the system. In this project the actors are different instances of *Clients*.

Below all the modelled Use Cases (Figure 3.1) are presented:

1. Login
    1.1 Get contacts
2. Registration
3. Add a contact
    3.1 Search a contact
4. Start a chat
    4.1 Start a connection
    4.2 Conversation
5. Close the application
    5.1 Logout

Further details about the Use Cases are available in the Appendix A.

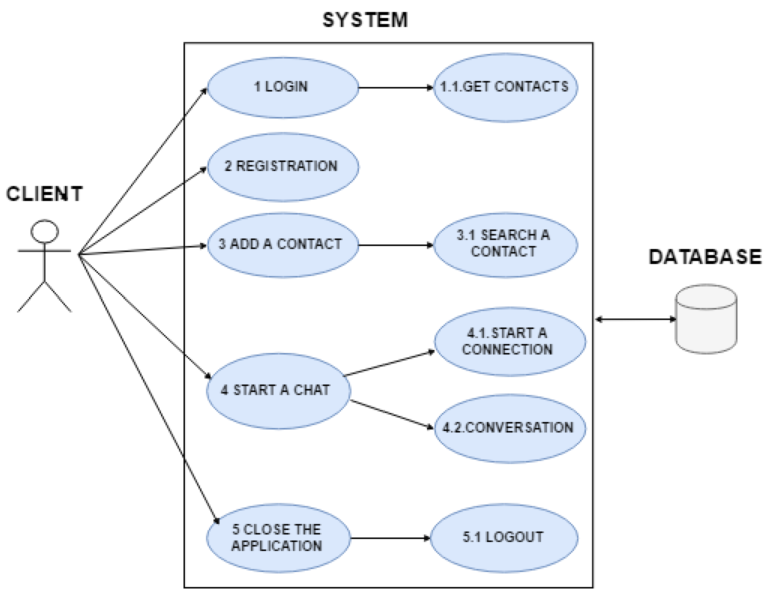


Figure 3.1: Use Cases UML Diagram

3.4 Mapping Functional Requirements to Use Cases

Tables 3.1 and 3.2 show the relation between the Use Cases and the Functional Requirements, respectively, of the Server and the Client. The optional Requirements identified in the first phase of the project's design have not been mapped to any Use Case and have not been implemented.

	1	1.1	**2**	**3**	3.1	**4**	4.1	4.2	**5**	5.1
A	x	x								
A.A	x	x								
B	x	x	x	x	x	x	x		x	x
B.A			x							
B.B	x									
B.C	x								x	x
B.D				x	x	x	x			
B.E				x						
B.F		x								
C	x	x	x	x	x	x	x		x	x
C.A			x							
C.B	x									
C.C	x									
C.D				x	x	x	x			
C.E				x						
C.F		x								
C.G										x

Table 3.1: Mapping between the Functional Requirements of the Server and the Use Cases.

3.5 General notes

3.5.1 Socket Programming

It was ultimately decided to employ sockets to develop the application, since the other methods employ several layers of abstraction. Socket programming allows the project team to implement network communication as close as possible to the operating system without rewriting the network stack. The effect of designing the application using sockets eventually required the team to implement a multi-threaded implementation.

3.5.2 Database Design

As mentioned earlier, the database design was modified as with the local copy of the database it was not possible for any other client to login from a remote machine.This was due to the fact that the remote clients contacts were stored locally.This issue was solved by storing the database on the server side.By doing this we also made the application more efficient. There are two tables in the

	1	1.1	**2**	**3**	3.1	**4**	4.1	4.2	**5**	5.1
a	x	x								
a.a	x									
a.b	x	x								
b			x							
b.a			x							
b.b			x							
c				x	x					
c.a				x	x					
c.b				x						
d						x	x	x		
d.a						x	x			
d.b						x	x			
d.b.a						x	x			
d.c						x		x		
e									x	x
e.a										x

Table 3.2: Mapping between the Functional Requirements of the Cllient and the Use Cases.

chat server database-’users’ and ’contacts’. The users table has the following fields; username,password,connection and date with username as the primary key. The contacts table has the following fields; username and contacts where both are primary keys.

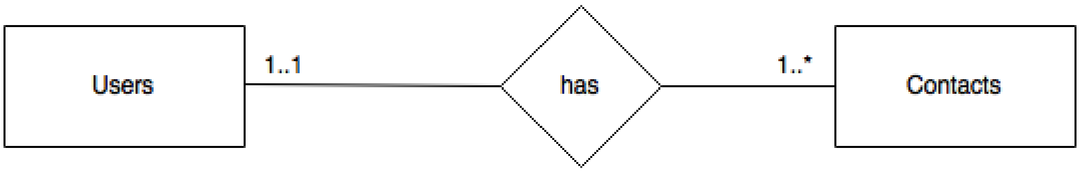


Figure 3.2: Entity-Relation Diagram for the Database

Chapter 4

Implementation

The main and more relevant aspects of the implementation of the server, the web application client and the android client are respectively presented in the Sections 4.1, 4.2 and 4.3.

Section 4.4 describes the testing plan that allowed the team to come up with the comprehensive test suite, whose results are available in Appendix **??**.

4.1 Server

This Section describes the general implementation of the server. Figure 4.1 shows an overview on the interconnections between the implemented methods.

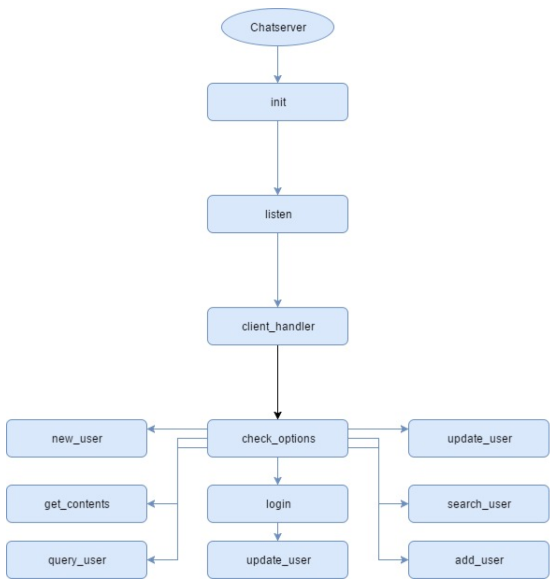


Figure 4.1: Diagram of the connections between the Servers methods

The implementation of the functional requirements began with the implementation of a TCP server as this was the fundamental component on which other requirements and tests were based on. The server was implemented in python with IPv4 TCP sockets. The sockets were implemented with the use of python's socket library. Primarily through python's `socket()` function, which allowed the project team to implement the various socket system calls.

For example, as is seen in the source code:

```
self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #Create TCP socket
self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
self.sock.bind((self.host, self.port))
```

The first python socket system call – `socket.socket()` - creates the socket which is used for bidirectional communication at the network layer or layer 3. With the socket created, a socket option was applied to aid in better management of the socket connection. This was achieved with the `setsockopt()` system call to set the option at the socket level specified by `socket.SOL_SOCKET` option. The `socket.SO_REUSEADDR` flag was needed to prevent an operating system error known as "*OSError: [Errno 98] Address already is use*" from occurring, as a result of running the program over and over in quick succession, especially during testing. This is because the previous execution left the socket in a `TIME_WAIT` state, and could not be immediately reused. The `SO_REUSEADDR` flag signals to the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

The second socket system call – `bind()` - was used to tie the socket to the system's IPv4 address. For simplicity, the system's default address is used and it is assumed that there is not more than one network interfaces. In that case the socket will then listen on all interfaces. The socket was also bound to port 5001 to listen for incoming TCP connections. Port 5001 is above 1024, or the reserved ports, and it is a rarely used port by other applications.

As noted from the team's *GitHub* history, this server was initially realized via a functional implementation. This allowed the team to firstly understand socket operations and socket programming before converting the server to an object oriented design and implementation.

The object oriented implementation of the server consists of a single class and eleven class methods that helped to achieve the functional requirements. One of the methods to note is the `listen` method. This method listens and accepts incoming TCP connections via python's `listen()` and `accept()` socket methods. The 's listen method was designed and implemented to spawn a new thread when a new TCP connection was accepted by the *chatserver*'s `listen` method. Threading was implemented using python's `thread` module. By implementing threading, the project team was able to have several tasks or threads running at the same time on the server, i.e. allowing the server to listen and accept multiple connections as well as sending and receiving data from multiple clients that connected. The thread method from python's `thread` module as noted in the `listen()` method of the server and highlighted below:

```
thread.start_new_thread(self.client_handler,(connected_client,
    client_address))
```

it starts a new thread by calling the `client_handler()` method upon each new connection. The `client_handler()` was implemented to communicate (outside the main thread of the server) with connected clients. One of the caveats of socket communication that was discovered is that data sent via sockets needed to be in a specific encoding, usually as bytes, in order to be sent and received. To address this requirement, JSON encoding from pythons `json` module was used in the `client_handler()` method. The following lines of code from the `client_handler()` method illustrates how the project team implemented json decoding and encoding.

```
received_dict = json.loads(received_data.decode('utf-8'))
result = json.dumps(result).encode('utf-8')
```

The main purpose of the server is to facilitate queries from users/clients which would allow the users/clients to find other users and communicate with them. In order to achieve this, the server needed to allow users/clients to establish a connection and be able to exchange data. As described above the project team implemented these requirements with the `listen()` and `client_handler()` methods.

For the server and clients to effectively communicate, the project team devised a message exchange schema to allow clients to make specific requests to the server. The schema was implemented in the form "*key : list of values*". More specifically, it was implemented as a python dictionary, with the key representing what function each list of values represent. In addition, the list of values was also implemented as a dictionary. For example, `'query': \{'OPTION': 'QUERY_USER','USER':username\}` represents the query function. In this example the server will query the database for the user specified by the username variable. The server was implemented to always check the `'OPTION'` field via its `check_options()` method which it uses to determine what is required by the client and which of its methods to call. This schema proved particularly useful later in the project, when additional options were required. It was extremely easy to add options and methods as additional requirements were introduced.

As noted earlier, with respect to the server's main purpose, in order to keep track of users who are online and their connection details such as IP addresses, a datastore was needed. The `sqlite` database was chosen, primarily because of its ease of use and implementation: it is very lightweight and does not require a separate server process. The `sqlite` database was implemented via method calls to python's `sqlite3` module. Eight of the eleven *chatserver* methods were implemented to perform CRUD functionalities on the sqlite database named *chatserver.db*. The required CRUD functionality depended on the `'OPTION'` specified by the client.

4.2 Web Application Client

This Section describes the general implementation of the Web Application client. Figure **??** shows an overview on the interconnections between the implemented methods.

One of the functional requirement of the desktop client application is that users should be able to communicate with each other directly. This effectively meant that the application is a peer to peer application. The development of the application began with simultaneous implementation of the GUI, using python's `tkinter` module, and the implementation of the socket/network functionality: the latter allows the client application to listen and accept connection from

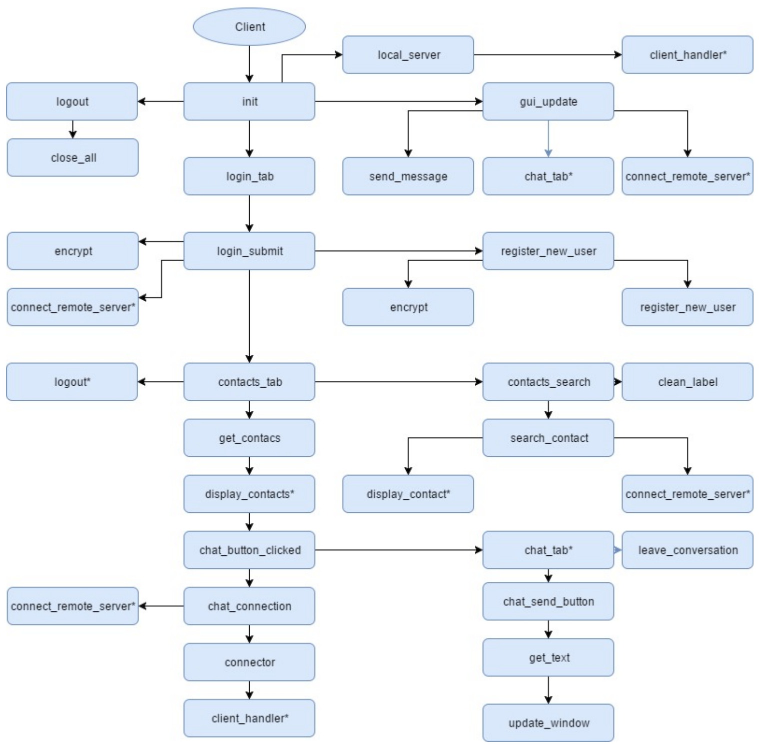


Figure 4.2: Diagram of the connections between the Servers methods

other clients, which is fundamentally the same requirement for the server. Since the desktop client was being implemented in python as well, the source code for the *chatserver* formed the base code for the desktop client. Hence, the project team utilised the same `listen()` method renamed `local_server()`, which is now started as a separate thread when the application starts, and `client_handler()` method that were described above.

The `client_handler()` method was additionally modified to store all incoming messages into a FIFO queue. This means all received messages from users that the client is communicating with are entered into a queue, called `received_messages`. The received messages queue was implemented as an object of python's `queue` class from its `queue` module. The `queue` module implements multi-producer, multi-consumer queues. The following line in the modified `client_handler()` method highlights how the project team achieved putting messages in the queue, with the `put_nowait()` method of the queue object.

```
self.received_messages.put_nowait(received_data)
```

The `put_nowait()` method was used instead of the `put()` to allow entering items into the queue without blocking. It was believed this was advantageous because of the small scale of the application.

As will be seen later on, the use of a queue is particularly useful for the spawned threads to exchange information themselves and the `tkinter`-based GUI. The graphical user interface for the desktop application was implemented using python's `tkinter` windowing toolkit. With the help of `tkinter`, the user interface was designed as a notebook with different tabs representing each action page of the chat application. The main tabs of the application are the `login_tab` for logging in, the `contacts_tab` for viewing the personal contact list and the `chat_tabs` for having one to one conversations with contacts. The `login_tab` shows two buttons associated to two different options - 'login' and 'register'. The client registration can be performed by selecting the register option that open up a registration window for submitting the username and password. The client encrypts the entered password and sends it along with the entered username to the remote server for registration. The password encryption functionality was implemented using the DES encryption standard from python's `Crypto.Cipher` module which uses 8 bytes long block ciphers to generate encrypted password strings as illustrated below:

```
des = DES.new('synomili', DES.MODE_ECB)
```

The `tkinter`'s notebook widget was utilized to achieve a tabbed window.

```
self.tab_controller = Notebook(self.demoPanel,
    name='notebook',width=420, height=550)
```

The above line of code illustrates how the project team implemented the tab window widget. One of the main reasons for implemented the notebook widget is that the user can communicate with several other users from the same window, just using different tabs. This was particularly significant from an implementation point of view, as the project team discovered that the behaviour of `tkinter` windows became unreliable when updated from another thread. This is what led the team to implement queues for the exchange of messages. The design that followed was that all messages received were entered into the `received_messages` queue as described above. Each message as part of the message exchange schema has a field for `'USER'` which identifies who the message is from. This allows the application to know which tab to insert each message that is received, as each tab represents a different user with which the client is chatting.

This naturally leads us to a very significant method called `gui_update()`. Show cased in the following code snippet, the application was implemented to extract from the `received_messages` queue the message and the user from the message is. The program then proceeds to check if there is a tab already opened for that particular user, or proceeds to open one if none exists, and simply updates the existing tab for that user. In order to keep track of which tab belongs to which user, the project team utilises a dictionary (`user_tabs_list`) to maintain a map between remote users and the name of the tabs when they are created in the `chat_tab()` method of the program. This is particularly important as without this mechanism the `tkinter` window will not know which tab to update.

```
try:
    received_message = self.received_messages.get_nowait()
    user = received_message['USER']
    msg = received_message['MSG']

    #Check if tab exists for user or not
    #Create new tab or update existing tab
    self.user_tabs_list
    if user not in self.user_tabs_list.keys():
        self.chat_tab(user)
        self.update_window(user,msg,False)
    else:
        self.update_window(user,msg,False)
except:
    #Its ok if theres no data in the queue'
    #Will check again later'
     pass
```

The second part of the `gui_update()` method was implemented to do the opposite of the first. Each message that is sent by the local user is placed in another queue used for outgoing messages called `sent_messages`. The `gui_update()` method retrieves each message from the queue and then extracts the name of the intended recipient. A lookup is then performed by the `user_connection` dictionary which is used to maintain a mapping of each remote user and their sockets[1]. The message to be sent and their socket information is passed to the `send_message` method. This method was designed and implemented to connect to remote clients and send the messages/data to them.

```
try:
    sent_message = self.sent_messages.get_nowait()
    #Check to see if the user is in the user_connection list
    for key in self.user_connection.keys():
        if sent_message['USER'] == str(key):
            #Get the connection string
            connected_client = self.user_connection.get(key)
            #encode the message as json and send
            self.send_message(sent_message, connected_client)
except:
    #Its ok if theres no data in the queue
    #Will check again later
    pass
```

One of the most significant elements of the `gui_update()` method implementation is the very last line:

```
self.demoPanel.after(1000, self.gui_update)
```

The python `after()` function was used to allow the `gui_update()` method to call itself after a period of time. In this case, it is every 1000 milliseconds. This essentially means every second the `gui_update()` method calls itself, checks both incoming and outgoing queues for messages and updates the tabs in the window if any new messages has arrived. It may not have been possible to implement a seemingly real time application in `tkinter` without the after function.

[1]This detail is added to the `user_connection` dictionary when the user connects or a connection is initiated to the user

4.3 Android Client

This Section describes the general implementation of the Android client. Figure 4.3 shows an overview on the interconnections between the implemented methods.

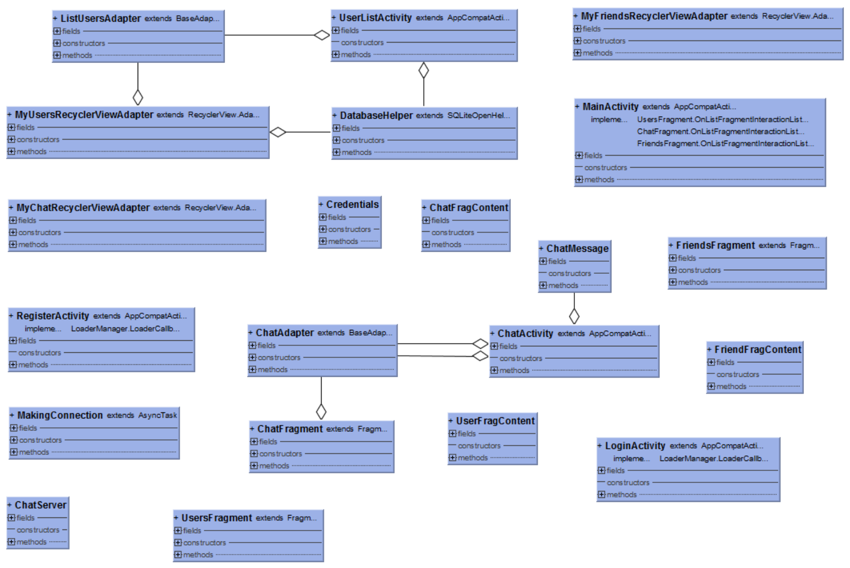

ListUsersAdapter extends BaseAdap...
UserListActivity extends AppCompatActi...
MyFriendsRecyclerViewAdapter extends RecyclerView.Ada...
MyUsersRecyclerViewAdapter extends RecyclerView.Ada...
DatabaseHelper extends SQLiteOpenHel...
MainActivity extends AppCompatActi... impleme... UsersFragment.OnListFragmentInteractionList... ChatFragment.OnListFragmentInteractionList... FriendsFragment.OnListFragmentInteractionList...
MyChatRecyclerViewAdapter extends RecyclerView.Ada...
Credentials
ChatFragContent
ChatMessage
FriendsFragment extends Fragm...
RegisterActivity extends AppCompatActi... impleme... LoaderManager.LoaderCallb...
ChatAdapter extends BaseAdap...
ChatActivity extends AppCompatActi...
FriendFragContent
MakingConnection extends AsyncTask
ChatFragment extends Fragm...
UserFragContent
LoginActivity extends AppCompatActi... impleme... LoaderManager.LoaderCallb...
ChatServer
UsersFragment extends Fragm...
fields
constructors
methods

Figure 4.3: Diagram of the connections between the Android client methods

The different classes for Android were implemented using *Android Studio*. The implementation of the functionalities began by building the main activities for each aspect of the platform. An activity in Android is needed for each page of the Android model that needs to be implemented. The following activities are the ones that were developed for the Android client:

- LoginActivity
- RegisterActivity
- MainActivity
- UserListActivity
- ChatActivity

The activities are responsible for the functionalities for each page. The build of the UI .xml files is generated for each activity, which develops the interface; the .java files associated with the activities contain the classes and methods needed for each activity. The `onCreate()` methods are implemented in all the activities, this method initialises the UI components and creates the functionalities for them, such as buttons for transitioning to other screens.

`LoginActivity` is the first activity that is launched when the application is run. The `attemptLogin()` method inside `LoginActivity` validates the user login details and then calls the LoginOperator subclass inside LoginActivity. The LoginOperator subclass creates a socket connection and initialises printwrite and bufferedreader. The printwrite variable is used to write details to the server and the bufferedreader variable is used to read the input from the user, therefore the LoginOperator reads the login credentials and sends these details to the server, which checks if these credentials exist in the database. If they exist, the user is allowed to transition to the next activity, MainActivity.

However, if a user does not already have their credentials stored on the database, they will have to register. This is done using RegisterActivity. The attemptLogin() method within RegisterActivity validates the password and username entered by the user in the client (the user must enter their password twice and it must be greater than two characters in length). This method calls the RegisterOperator subclass which opens a stream of communication with the server. The doInBackground() method within this subclass which connects to the socket on the particular user's chat IP address and port number. Next, printwrite and bufferedreader are initialised, much like in LoginActivity, therefore the RegisterOperator reads the login credentials and sends these details to the server, which adds the new login credentials to the database.

Once a user has logged in, they will transition to the next activity, MainActivity. MainActivity contains three fragments, ChatFragment, UserFragment and FriendFragment. Each of these fragments have a class; ChatFragContent, FriendFragContent and UserFragContent as seen in the above class diagram.

The initial conceptualisation of the implementation for the Android client was to have these fragments, and each fragment contains a list of the users for that associated fragment. For example, the FriendFragment would contain a list of only the user's friends. However, due to implementation issues and timing constraints these fragments could not be implemented for the final system, as detailed in the evaluation part of the report.

The final Android client allows users to see a global list of all the users in the database. This is implemented using the UserListActivity. UserListActivity contains a method, copyDatabase(), which copies all the users on the database and then displays them in the UI. The ListUsersAdapter contains a constructor for the users, ListUsersAdapter. This class then contains the get methods which retrieve all the users in the database. Once users click on a user from the list, they are taken to the next and final activity, ChatActivity. The ChatActivity class contains the methods and subclasses needed to send and receive messages from another client. The ChatOperator method subclass opens a stream of communications to and from the server. The Send and Receive subclasses within ChatActivity then allow a user to send and receive by connecting to the socket on the particular user's (the person with whom the user is chatting) IP address and port number.

The other classes seen in the Figure 4.3 are needed for the Android application to perform other tasks. The MakingConnection class connects to the socket, taking in the IP address and port number from the parameters initialised. The DatabaseHelper class returns the list of user details that are stored in the database on the server. The getAllRows() method within DatabaseHelper class retrieves all the rows of the database.

4.4 Testing

For testing the software, both unit testing and functional testing was performed. Unit testing involved testing both the backend features and the UI features. It helped in catching couple of bugs quite early such as displaying appropriate messages on the application's user interface every time a use case fails. It also helped in identifying a few design glitches. For example, the 'logout' feature initially only navigated the display to the 'login' page without actually disconnecting the client(closing the socket) from the server. This caused the online user to open the chat window for a client which was not logged in without displaying an appropriate message about the status of the offline client on the user interface. This bug was identified by unit testing and was fixed in the later versions. Additionally, unit testing was performed incrementally for subsequent versions of the software. The functional testing was performed at later stages to test the connection and communication between multiple clients. Rigorous testing of the software helped in improving the quality and robustness of the software.

Unit Testing

The unit testing involved testing the features of the client and the server for a number of use cases. Both positive and negative test cases were run for evaluating the successful use cases and the unsuccessful ones respectively. The unit testing was performed manually as well as using automated testing scripts for testing the backend and UI for both the web and android client and the server backend. The python module `unittest` was used for generating the automated test suite. The test cases were categorized according to the different use cases mentioned in the previous sections. The results of unit testing and functional testing are included in the appendix. The lists of test cases run for each use case are mentioned below.

Login

- Log in with a registered username and password
- On successfully login, view the personal contacts displayed on the personalized main page
- Log in with an empty string as username
- Log in with an empty string as password
- Log in with an empty string as username and password
- Log in with an unregistered username
- Log in with a registered name and an incorrect password

Register

- Register a new user with a valid username and password
- On successful registration, view the standard main page without any contacts displayed

- Register with an empty string as username
- Register with an empty string as password
- Register with an empty string as username and password
- Register with a valid username but different password
- Register with an already registered user
- Closing registration window without entering username and password

Start a chat

- Click on a contact button of an online contact to open a chat window
- Start a chat with an online contact
- Click on a contact button of an offline contact

Chat conversation

- Write a text message to a chat window
- Read a text message from a chat window

Search and add a contact

- Search for a registered user and add it to the contact list
- View the added contact in the list of contacts on the personalized main page
- Search for an empty string as username
- Search for a non registered username
- Search for a existing user

Logout

- Log out from the chat application using logout button
- Log out from the chat application by clicking on the close window button

UI

- Navigation between Login activity and Register activity
- Navigation between Login activity and contacts list display
- Navigation between contact list display and Chat tab
- Logout button activity
- Close button activity

- Text messages display on chat tab
- Notification for username and password mismatch
- Notification for adding a non existing username as contact
- Notification for adding yourself as contact
- Notification for adding an empty string as contact
- Notification when the contact is offline

Functional Testing

After performing unit testing for a satisfactory number of versions, functional test was performed for a relatively more stable version(with as few bugs as possible). This helped in testing the overall functionality of the chat application. Few of the functional test cases are mentioned below.

- Initiate a two-way chat conversation with each client hosted on a different IP address and server hosted on a third IP address
- Initiate multiple chats among three clients each hosted on a different IP address and server hosted on a fourth IP address

Chapter 5

Teamwork

The team developed the project by splitting up into sub-teams: as mentioned above, one team was focussing on the Android client, the other was working on the Python server and client.

A number of different forms of communication have been used, according to what was outlined in the initial report. *WhatsApp* and *Skype* were the main platforms used for communication, allowing team members to stay in touch and update them with individual project progress.

The team met weekly to discuss the project, and development meetings were held where the team would work together to tackle specific aspects of the project, especially during the initial and the last phases of the work.

Git was a tool used for the development of the application. *GitHub*'s feature branches were used to allow team members to work on different parts of the system. The following branches were made in addition to the default, the master one:

- **android:** development of the Android application
- **develop:** development of the Python application
- **backend:** development of the Python server and Python client
- **ui:** implementation of the Python GUI
- **deliverables:** deliverables of the project, the initial report and the final report.

When changes had to be made, a procedure was in place which was followed by the team members. This involved members having to make pull requests on the branches they were working on and then having these requests approved by the Git coordinator.

Chapter 6

Evaluation

The evaluation section of this report will focus on several aspects of the final project.

The mandatory features of the chat system as outlined in the initial report were as follows:

- Send text based messages to one or more users
- Receive text based messages from one or more users
- Secure communication between users
- Search for other users on the server
- User Registration (New User)
- User authentication (Log in / Log out)
- Chat client for mobile
- Chat client for desktop

The final system is split into two aspects, the Android and the Python clients. As discussed in the Chapter **??**, during the implementation and the development of the two clients the team splits the project in two parts, with the sub-teams working in parallel on the python client and on the Android client. Because of this, the two platforms have differing final functionalities.

To evaluate the final system, it is essential to discuss the mandatory features outlined above, and how the final system compares. Below is the evaluation of the final system (Section 6.1), discussing what works and what does not, changes that were made to the initial plan, the strengths and weaknesses of the members and how they worked as part of a team (Section 6.2). In conclusion (Section 6.3) an outlining future work that could be done on the project is presented.

6.1 Aspects of the final project

Below are the three main aspects of the project, detailing how they behave as part of the final system.

6.1.1 Server

The initial purpose for creating the server was for the server to act as an intermediary between the clients. The server acts as the medium for the clients to form a handshake and the final server does exactly this.

The server accesses the database and is able to perform all the queries required from the client side, in order to log in, log out, register a new user, look for another user, add a contact and get the contacts of a user.

The server needs to be running and listening for open connections before users have attempted to login; once they have logged in, the server provides to the clients all the information they need to open connections with whichever user is online and whose address is stored on the database.

6.1.2 Python Client

The Python client allows users to register and login, and then open chats with other existing users in the database stored on the server, once they have been added to the user's contacts.

The registration procedure requires a user to enter their desired username, and then their password twice to ensure the user has not mistyped the password. After a user has registered, they are logged straight in to the system.

The login procedure requires a user to enter their existing username and password. Once a user has successfully logged in, a user is able to search for other users that exist on the database, and then add them as their contact.

After adding a contact, a user is able to initiate conversations with a number of other contacts, providing those contacts are online.

The user also has the ability of logging out of the system.

The initial design of the desktop client in `tkinter` incorporated a multimodal window design, each running in a separate thread. However, this proved to be an inefficient design as the `tkinter` framework encounters issues when its windows are accessed or updated from outside threads. To overcome this, the use of `mtkinter` framework was introduced to the development with the expectation that it would have solved the issues experienced. However, this was abandoned in favour of the improved design utilising queues detailed in the implementation.

6.1.3 Android Client

The login and registration procedures for the two clients are similar. On opening the application, the users are asked to login; if they do not have existing credentials, they are able to create new ones by clicking on the *register* button.

Once they have created a new user by registering (again, by entering a new username and their password twice), they can submit these details and then this takes the user back to the login page, where they are asked to login with these new credentials.

After logging in, the user is shown a list which displays the global user list of all the existing users on the database. The user is then able to click on the users in this list. Once they do this, a new window, which takes the user to the chat screen with that user, opens on the app.

There were aspects of the planning that could not be fully implemented. As per the design of the system, fragments were the initial idea for the Android application, and these were added however the functionality could not be implemented. Due to the complexity imposed by Android and the lack of time, it was difficult for the Android sub-team to implement these fragment classes. Initially, three fragments were added to the GUI; one for the list of friends, another for the list of users from the global list extracted from the server and a third for the list of existing chats open by that user in that session.

6.1.4 The final system

With regards to a final, overall view of the system, it is safe to assume that most of the initial requirements laid out at the beginning of the project have been met, even though secure communication has not been implemented. It is possible to send and receive text-based messages to one or more users, search for other users on the server, register new users, authenticate users and this has all been done using chat clients developed for both mobile and desktop.

The Python client is able to send and receive messages, enabling two-way communication between clients, however the same cannot be said for the Android client. Due to a lack of time near the end of the project, it was not possible to meet the this requirement for the Android client. The Android client is able to send messages however it does not receive any, making the connection one-way. The reason for this is the complexity of the queues and threading which could not be fully implemented in the final Android system.

6.2 Problems and changes made

The original plan for the team was to develop the desktop client in Java. This is because the Android client is Java-based, and the code written for the Java client could then eventually be adapted to be used for the Android client. However, at the mid-point of the project, progress for the desktop client was significantly hindered due to a lack of technical expertise in the problem domain within the team in Java. As a result of this, a decision was made by the team to develop the desktop application in Python because a larger number of members of the team was more au fait equipped in this language. This allowed more resources and more people to be involved in the process, and this accelerated the development of the application, making up for the time lost.

An improvement was made to the project regarding the physical location of the database: it was initially planned to save it locally for each client; however, due to security reasons and mobility concerns, it was later decided that the database should be stored on the server, allowing the clients to query it through the server.

Time management and the allocation of resources was an issue the team faced near the end of the project timeline. Due to the team members having tight schedules during the month of March, it was becoming increasingly difficult for the team to work on the project. As a result of this, the team was constrained and the team members could not focus their efforts on the project.

6.3 Project improvements and future works

It would be essential to improve the user interface of the clients. Currently, they are not intuitive enough to use and not user friendly.

If more time was given, the Android client would be improved and, as the initial requirements have been already implemented, the optional features outlined would have been added:

- Set status
- Notifications (banners on Android)
- Send images or files
- Set name/profile picture
- Delete user account
- Contact list
- Emojis
- Backup chat/chat log
- Remove messages
- Chat time stamps
- Customised chat background
- Adding multiple servers for stability and robustness

Chapter 7

Peer Assessment

Below is the distribution of the points to the team members.

	Percentage
Tharuni Avula	16.67
Pryanka Bhati	16.66
Shweta Bhatt	16.66
Sagar Mohan	16.66
Francesca Mosca	16.66
Marc Smith	16.66

Table 7.1: Team Peer Assessment

Appendices

Appendix A

Use Cases Description

Use Case 1	**Login**
Type:	primary
Actors:	Client
Description:	the Client inserts its credentials and enters in the main personalized page
Flow of events:	1. the Client gets the username and the password 2. the Client starts a connection with the Server 3. the Server queries the database for checking the credentials 4. the Server accepts the connection 5. the Client displays the main page of the application
Exceptions:	- the username and the password are empty or mismatching - the server is offline - connection problems
Exit condition:	the Server returns True or False to allow or deny the login
Preconditions:	the Client must be already registered
Postconditions:	the Client can look for contacts and starts new connections
Nested use cases:	1.1 Get contacts

Use Case 1.1	**Get Contacts**
Type:	secondary
Actors:	Client
Description:	the Client receives the list of the already added contacts and sees them displayed in its main page
Flow of events:	1. the Clients queries the Server for getting its contacts 2. the Server gets the contact list from the database 3. the Server returns the contact list 4. the Client displays the contact list in its main page
Exceptions:	- the server is offline - connection problems
Exit condition:	the Client displays the contacts list
Preconditions:	the Client must be connected to the Server
Postconditions:	the Client can add new contacts or starts new connections
Nested use cases:	-

Use Case 2	**Registration**
Type:	primary
Actors:	Client
Description:	the Client inserts its credentials and enters in the main personalized page
Flow of events:	1. the username and the password (twice) are inserted 2. the connection with the Server starts 3. the Server queries the database for checking the uniqueness of the username 4. the Server accepts the connection 5. the main page of the application is displayed
Exceptions:	- the username and the password mismatch - the username already exists - the server is offline - connection problems
Exit condition:	the Server returns True or False to allow or deny the registration
Preconditions:	-
Postconditions:	the Client can look for new contacts and starts new connections
Nested use cases:	-

Use Case 3	**Add a contact**
Type:	primary
Actors:	Client
Description:	the Client inserts another Client's username and gets displayed the new contact; the Server saves such relationship into the database
Flow of events:	1. the other Client's username is inserted 2. the connection with the Server starts 3. the Server inserts in the database the new association 4. the Server returns True or False according to the success of the operation 5. the new contact is displayed in the Client's main page
Exceptions:	- the other Client is already a contact - the username is missing or not registered - the server is offline - connection problems
Exit condition:	the Client receives True or False and displays the contact consequently
Preconditions:	the Client must be logged in
Postconditions:	the Client can look for other contacts or starts new connections
Nested use cases:	3.1 Search a contact

Use Case 3.1	**Search a contact**
Type:	secondary
Actors:	Client
Description:	the Client checks if the desired new contact exists into the database
Flow of events:	1. the Clients asks the Server about the existence of another Client given its username 2. the Server queries the database for checking the existence of that username 3. the Server returns True or False according to the success of the operation
Exceptions:	- the server is offline - connection problems
Exit condition:	the Client receives True or False and add the contact consequently
Preconditions:	the Client must be logged in
Postconditions:	the Client can add the contact
Nested use cases:	–

Use Case 4	**Chat**
Type:	primary
Actors:	Client1, Client2
Description:	the Client1 starts a new connection and send/receive messages to/from Client2
Flow of events:	1. Client1 queries the Server for getting the information about Client2's connection 2. the Server returns to Client1 the connection details of the Client2 3. Client1 opens a connection with Client2 4. a new tab is open in both Clients' applications 5. Client1 starts the conversation
Exceptions:	- the Client2 is offline - the server is offline - connection problems
Exit condition:	Client1 opens the new connection
Preconditions:	- Client1 must be logged in - Client1 must know the Client2's username
Postconditions:	Client1 sends and receives messages to/from Client2
Nested use cases:	4.1 Start a connection 4.2 Conversation

Use Case 4.1	**Start a connection**
Type:	secondary
Actors:	Client1, Client2
Description:	Client1 starts a new connection with Client2
Flow of events:	1. Client1 selects Client2's username 2. Client1 asks the Server the information about Client2's connection 3. the Server queries the database for checking the Client2's status 4. the Server returns to Client1 the connection details of Client2 5. Client1 initiates a connection with Client2 6. Client2 accepts the connection from Client1 7. Client2 creates a new thread for receiving data from Client1 8. a new tab is opened in both Clients' applications
Exceptions:	- Client2 is offline - the server is offline - connection problems
Exit condition:	the connection between the two Clients has started
Preconditions:	- both Clients must be logged in - Client2 must be in the Client1's contacts list
Postconditions:	a Client starts the conversation
Nested use cases:	-

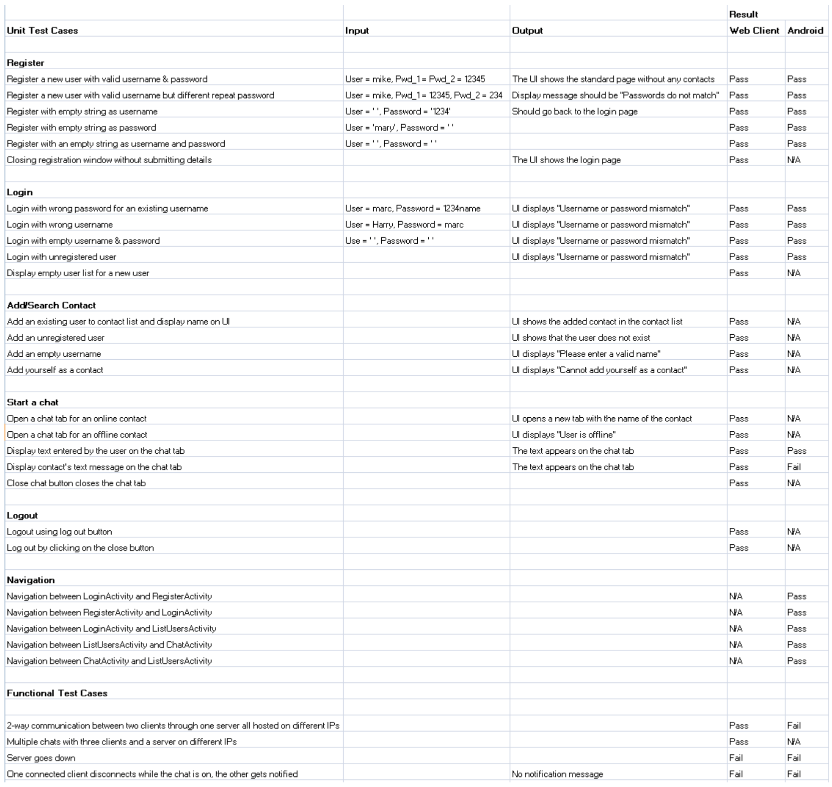
Use Case 4.2	**Conversation**
Type:	secondary
Actors:	Client1, Client2
Description:	Client1 sends/receives messages to/from Client2
Flow of events:	1. Client1 writes one or more messages 2. Client1 sends its message to Client2 3. Client2 receives the message from Client1 4. Client2 writes one or more messages 5. Client2 sends its message to Client1 6. Client1 receives the message from Client2
Exceptions:	- a Client disconnects - the server is offline - connection problems
Exit condition:	a Client logs out
Preconditions:	- both Clients must be logged in - there must be an open connection between the two Clients
Postconditions:	-
Nested use cases:	-

Use Case 5	**Close the application**
Type:	primary
Actors:	Client
Description:	the Client close the application interrupting all the active connections
Flow of events:	1. the Client wants to leave the application 2. the Client logs out 3. the Client close all the opened windows
Exceptions:	- the log out operation is not authorized by the Server - connection problems
Exit condition:	all the windows are closed
Preconditions:	the Client is logged in
Postconditions:	-
Nested use cases:	5.1 Logout

Use Case 5.1	**Logout**
Type:	secondary
Actors:	Client
Description:	the Client disconnects from the Server and from the other Clients and leaves the application
Flow of events:	1. the Client wants to log out 2. the Client asks the permission to the Server 3. the Server queries the database for updating the connection information 4. the Server allows the Client to disconnect 5. the Client closes all the active connections with the other Clients
Exceptions:	- the server is offline - connection problems
Exit condition:	connection interrupted
Preconditions:	the Client is logged in
Postconditions:	the Client is disconnected to the Server and to any other Client
Nested use cases:	-

Appendix B

Test Results



			Result	
Unit Test Cases	**Input**	**Output**	**Web Client**	**Android**
Register				
Register a new user with valid username & password	User = mike, Pwd_1 = Pwd_2 = 12345	The UI shows the standard page without any contacts	Pass	Pass
Register a new user with valid username but different repeat password	User = mike, Pwd_1 = 12345, Pwd_2 = 234	Display message should be "Passwords do not match"	Pass	Pass
Register with empty string as username	User = ' ', Password = '1234'	Should go back to the login page	Pass	Pass
Register with empty string as password	User = 'mary', Password = ' '		Pass	Pass
Register with an empty string as username and password	User = ' ', Password = ' '		Pass	Pass
Closing registration window without submitting details		The UI shows the login page	Pass	N/A
Login				
Login with wrong password for an existing username	User = marc, Password = 1234name	UI displays "Username or password mismatch"	Pass	Pass
Login with wrong username	User = Harry, Password = marc	UI displays "Username or password mismatch"	Pass	Pass
Login with empty username & password	Use = ' ', Password = ' '	UI displays "Username or password mismatch"	Pass	Pass
Login with unregistered user		UI displays "Username or password mismatch"	Pass	Pass
Display empty user list for a new user			Pass	N/A
Add/Search Contact				
Add an existing user to contact list and display name on UI		UI shows the added contact in the contact list	Pass	N/A
Add an unregistered user		UI shows that the user does not exist	Pass	N/A
Add an empty username		UI displays "Please enter a valid name"	Pass	N/A
Add yourself as a contact		UI displays "Cannot add yourself as a contact"	Pass	N/A
Start a chat				
Open a chat tab for an online contact		UI opens a new tab with the name of the contact	Pass	N/A
Open a chat tab for an offline contact		UI displays "User is offline"	Pass	N/A
Display text entered by the user on the chat tab		The text appears on the chat tab	Pass	Pass
Display contact's text message on the chat tab		The text appears on the chat tab	Pass	Fail
Close chat button closes the chat tab			Pass	N/A
Logout				
Logout using log out button			Pass	N/A
Log out by clicking on the close button			Pass	N/A
Navigation				
Navigation between LoginActivity and RegisterActivity			N/A	Pass
Navigation between RegisterActivity and LoginActivity			N/A	Pass
Navigation between LoginActivity and ListUsersActivity			N/A	Pass
Navigation between ListUsersActivity and ChatActivity			N/A	Pass
Navigation between ChatActivity and ListUsersActivity			N/A	Pass
Functional Test Cases				
2-way communication between two clients through one server all hosted on different IPs			Pass	Fail
Multiple chats with three clients and a server on different IPs			Pass	N/A
Server goes down			Fail	Fail
One connected client disconnects while the chat is on, the other gets notified		No notification message	Fail	Fail

Figure B.1: Unit Testing results

Appendix C

Test Unit Code

```
import unittest
import mock
from mock import patch
import chatserver_functions
from chatserver_functions import *
import chatserver
from chatserver import *
import socket
import threading
import thread
import json
import sqlite3
import datetime

class MyTest(unittest.TestCase):


    def test_check_options (self):
        self.assertEqual(check_options("", ""), 'INVALID COMMAND')
        self.assertEqual(check_options('SEARCH_USER', )

    def test_login(self):
        self.assertEqual(login({'USER':"marc",
            "PASSWORD":"pass"}), False)
        self.assertEqual(login({'USER':"marc", "PASSWORD":""}),
            False)

    def test_search_user(self):
        self.assertEqual(search_user({'USER':"mary"}),False)

    def test_add_user(self):
        self.assertEqual(add_user({'USER':'marc',
            'CONTACT':'shweta'}), True)
    def test_get_contacts(self):
        self.assertEqual(get_contacts({'USER':'checca'}),
            ['marc', 'sagar', 'test'])

    def test_query_user(self):
        self.assertEqual(query_user({'USER': 'checca'}), {'USER':
            u'checca', 'PASSWORD':
            u"'\\xd0\\xafLY\\xd6\\xb7\\xcd\\x8e\\x15zp\\xc9\\x83N␣
            \\xaco\\x19I8\\xecj\\xdaK\\xd0\\xafLY\\xd6\\xb7\\xcd\\x8e\\x15zp\\xc9\\x83N␣
            \\xaco\\x19I8\\xecj\\xdaK'", 'CONNECTION':
            u'127.0.0.1:53151', 'DATE' : u'2017-03-26
            21:17:53.057000'})

    def test_new_user(self):
        self.assertEqual(new_user({'USER': 'emma',
            'PASSWORD':'emma', 'CONNECTION': '123'}, True))



suite = unittest.TestLoader().loadTestsFromTestCase(MyTest)
unittest.TextTestRunner(verbosity=2).run(suite)


'''if __name__ == '__main__':
    unittest.main()'''
```