Chat, Bank System through the Java RMI (Final Project CS5523)

Richard Garcia Lebron*[1] and Tanvir Irfan Chowdhury†[1]

[1]Department of Computer Science, University of Texas at San Antonio

December 9, 2014

1 Introduction

This project contain an implementation of a chat and bank system where users can interchange message, share group messages and transfer money from user to user. Both client and chat server implementation are multi-threaded where a thread pool was customized created for both the client and the server. A client GUI is provided to easel the interaction with the system. To store the user profiles, messages, groups and transactions we uses the Apache MySQL database. The remote communication between servers and client was made possible using the Java RMI implementation. The communication flow in the system is described in Figure 1. The synchronization problem were addressed by placing the code in the critical section, under thread worker in both client and server side.

1.1 Project Status

Currently the project is running, local in the same computer. We haven't been able to test it on different computers.

*richard.garcialebron@utsa.edu

†tanvir_cse072@yahoo.com

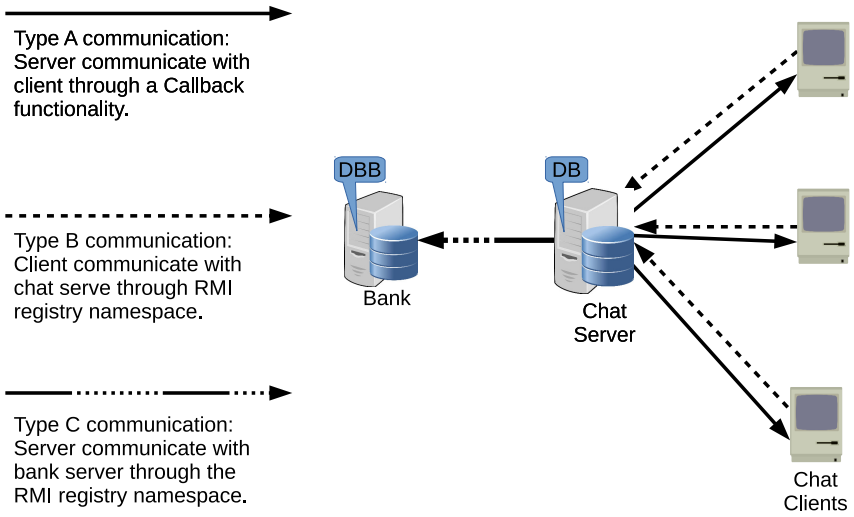

Figure 1: Diagram to describe the communication flow in the system, bank, chat, clients and its databases

.

1.1.1 Challenges

- Deadlock in the thread pools client and server side.
- Running the project as three separate projects.
- Debugging null exception, out of bound exceptions, etc.
- Working with the SQL figure out that group is actually and operation on SQL. Therefor no column can be named as group.
- Trying server and client on different computers on campus wireless network. Connection refuse exceptions.

1.2 Project Compilation

The file COMPILE.bat and COMPILE.sh contain a scrip to compile the whole project.

1.3 To run the project

Files RUN_CLIENT_GUI.bat, RUN_CLIENT_GUI.sh and RUN_SERVER.bat, RUN_SERVER.sh are the scripts to run the client and chat server respectively.

1.4 Project Task Distribution

Richard Garcia Lebron: Databases management, server bank functionality and wrote the report.

Tanvir Irfan Chowdhury: ClientCallback functionality and Chat Server also GUI and thread pool.

2 Chat Server

In this section we describe the Chat server implementation an interfaces. The interfaces and implementations correspond to the files: *ChatServerServant.java*, *UserAccount.java*, *UserGroup.java* and *ChatServerServantImpl.java*, *UserAccountImpl.java*, *UserGroupImpl.java* respectively. Both the interface and the implementation make use of the *User.java*, *Message.java*, *Group.java* and *FriendRequest.java* abstractions for the chat users, messages, groups and friend requests. The class *User.java* encapsulate:

- full name
- user name
- living city
- profession
- collage
- password
- graduation year
- company
- list of friends
- list of groups

The class *Messages.java* encapsulate:

- sender
- receiver
- group
- is a group message
- time stamp
- message
- message id
- status

In the *Group.java* the following items are encapsulated:

- group id
- group name
- list of member in the group
- owner/moderator of the group

Furthermore the *FriendRequest.java* contain the following items:

- sender
- receiver
- status
- message

Notice the friendship abstraction is on the *User.java* and *Group.java*. Our implementation of the chat server uses the thread pool implemented in the classes: *ServerSideThreadPoolManger.java* and the *ServerSideWorker.java*.

2.1 The ChatServerServant interface

This interface extend the Remote class, also uses the auxiliary classes *User.java*, *Group.java* and the interface CallbackClientInterface presented on section §. A short summary of the ChatServerServant.java interface is show in the next table, for a complete description see the java code.

- User ⇐ createUserAccount: Create a user and inserted in the *USERS* table, and return the object for the user.

- User ⇐ loginAccount: Verify if the given user name and password match if so return the object for the user.

- User ⇐ getUserAccount: Verify if the user exist in the *USERS* table and return the object for the user is exist or null if not.

- Group ⇐ findGroup: Return the group with given pair (groupName, moderatorUserName) if exist in the *GROUPS* table, null if not exist.

- ArrayList<Group> ⇐ findAllGroups*: Return al list with all the groups in the *GROUPS* table that match the give group name.

- boolean ⇐ connect: Insert to the hash the client user name and client CallbackClientInterfase, if success return true, false otherwise.

- boolean ⇐ disConnect: Remove from the hash the pair client user name and CallbackClientInterfase, if success return true, false otherwise.

- Group ⇐ createGroup: Create a group and insert it to the *GROUP* table, return the created group.

2.2 The User Accounts interface

This interface also extend the Remote class and use the same auxiliary classes used by the ChatServerServant the major functionalities are summarized bellow (for a more detail please se the java code):

- User ⇐ getUser: Verify if the user exist in the *USERS* table and return the object for the user is exist or null if not.

- boolean ⇐ updateProfile: Update the user profile but can't change the user name, return *true* if the update success *false* if not success.

- Group ⇐ getGroup: Return the group with given groupId or pair (groupName,moderatorUserName) if exist in the *GROUPS* table, null if not exist.

- boolean ⇐ sendMessage: store the given message in the *MSG* table, return true if messages is inserted successfully or false if inserting the message fail.

- ArrayList<Message> ⇐ getAllMessageFromUser: query in the *MSG* table all the messages send by a given user after the messages are retrieved, those messages are
- deleted from the table, return a list of Messages or null if there is no messages.

- Group ⇐ joinGroup: Join a given user the given group, return updated Group or null if the join fail.

- boolean ⇐ leaveGroup: delete given group from the *GROUP* table, return true if success or false if not.

- boolean ⇐ inviteFriend: send a friend request to a given user, return true if request successfully saved in the *FriendRequest* table or false if not.

- boolean ⇐ acceptFriend: send a friend request to a given user, return true if request successfully saved in the *FriendRequest* table or false if not.

- User ⇐ viewProfile: return the user object.

- ArrayList<Messages> ⇐ readAllOfflineMessages: Read all the messages in the message table that match a given sender and receiver. Recall this function will delete the retrieved messages from table.

2.3 The User Group Interface

As known also this interface extend the Remote class. This interface use the *UserAccount.java* interface to provide the functionalities addUser to add a user into a group and the removeUser to remove a user from the group.

2.4 The ChatServerServant Implementation

This implementation uses the auxiliary classes: *Group.java*, *User.java* also uses the *DatabaseManager.java* and the *CallbackClientInterface.java*. Similar to the other implementation classes extend the UnicastRemoteObject class. For a complete description of the implementation see the respective java code. In summary this implementation uses mostly all the *DatabaseManager.java* member functions to implements its functionalities.

2.5 The User Account Implementation

Similar with the ChatServerImplementation this implementation extend the UnicastRemoteObject class. Also uses all the auxiliary function describe in §, the thread pool implementation to handle sending messages, also get and instance of the char server driver *RMIChatServer.java*. For complete detail

of the implementation see the java code. Summary the core of this implementation is the use of the member function in the *DatabaseManager.java*

2.6 The User Group Implementation

As required extend the UnicastRemoteObject class. Different from the previous implementation here the auxiliary classes uses are *UserAccount.java* and the *UserGroup.java* to implement the add user and delete user functionalities.

3 Client Callback

In this section we describe the client Callback interface functionality and its implementations. Moreover we describe the *RMIChatClient.java* that hold the driver for the client. Our client design relay in the thread pool classes *ClientSideThreadPoolManager.java* and *ClientSideWorker.java*, Pool manager and worker class respectively. The interface and implementation of the client callback user the auxiliary classes: *FriendRequest.java*, *Group.java*, *Messages.java* and *User.java* described in section §.

3.1 Callback Client Interface

Different from the chat server the client offer only one interface (and implementation), this is the interface: *CallbackClientInterface.java* this class extend Remote as well as the other interfaces presented. The functionalities are:

alertUserMesssage: This alert the client about the arrival of a given message. Also open the chat window of the GUI with given message.

deliveryReport: This tell the client that the given message have been delivered.

alertOfflineMessage: When user login this alert the client there some unread messages.

alertFriendRequest: To alert the client about some other user asking to be friends.

3.2 Callback Client Implementation

Similar with the other implementation this implementation extend the UnicastRemoteObject. This implementation display and interact with the GUI using the auxiliary classes. See code for more details.

3.3 The RMIChatClient

This class contain the driver for the chat client, making uses of the GUI and the thread pool classes. The *RMIChatClien.java* contain the function and other functions such setter and getter for the ChatServerServant, ClientSideThreadPoolManger. Also include function to manage the current client:

- login
- logout
- view profile
- get/set current client
- create user

To manage the group functionalities the following function are introduced:

- create group
- find all groups
- join group
- get group

The next group of functions were introduced to manage the friend relationships:

- invite friend
- accept friend

Messages between client are managed with the functions:

- get all messages from
- get all off-line messages

The interaction with the GUI is managed with the functions:

- add new chat windows
- get chat windows
- close all chat windows

For more details about this function see the java code. In the main function the only operation is to create a new RMIChatClient.

4 Databases Integration

We use a MySQL database system by Apache to easy store users, groups and messages information and make the object persistent. This allow all the object User, Groups, Messages, Friend Relations and Friend Requests been persistence. Because the data is persistence, data can been accessible after a server restart, preventing the lost of chat and bank users and even some of the pending friend request, unread messages and groups and money transaction history. Furthermore this will reduce the memory (RAM) used since we only need on memory some properties and data for only the on-line users. In this section we will describe what is the database design and relational diagram.

4.1 Chat Server Databases design

The server for the chat contains a database we call it **DB**. This database contain five tables: GROUPS to store the groups, MSG to store the unread messages, FRIENDS to store the friendship relationship, and status, FR to store the pending friend requests and USERS to store the users profiles and login data.

4.1.1 The Group table

This table store the groups created by the users, the group identifier are the groupId and the pair (groupName, moderatorUserName). The columns in this table are:

groupId	groupName	memberUserName	moderatorUserName

groupId: Integer to store the id of the group.

groupName: String to store the name of the group (limit 15 character).

memeberUserName: String to store the user name a member in the group.

moderatorUserName" String that store the user name of the owner/creator/moderator of the group (limit 30 character).

4.1.2 The Message table

This table store only the unread messages, ones a message is retrieved from the table this one is deleted, server does not store history of message communication. The columns in this table are:

mId	sender	receiver	message	theTimeStamp	isGroupMessage

mId: Integer to store the message id.

sender: String to store the user name of the user who send the message (limit 15 character).

receiver: String that store the user name of whom message is addressed (limit 15 character).

message: String to store the actual message (limit 5000 characters).

theTimeStamp: A timestamp object contain the date and time when message was send.

isGroupMessage: Integer to store a groupId if the message is send to a group or zero if not send to a group.

4.1.3 The Friends table

This table store the $user \Longrightarrow friend$ relation ship, since our implementations is in both direction both $user \Longrightarrow friend$ and $friend \Longrightarrow user$ are stored. The columns in this table are:

userName	userFriend	status

userName: String to store the user name of the user (limit 15 character).

userFriend: String to store the user name of the friend (limit 15 character).

status: Boolean to store the status of the friendship relationship *FALSE* when user block friend (only one direction) *TRUE* otherwise.

4.1.4 The Friends Requests table

This table store the pending friend request. The columns in this table are:

sender	receiver	status	message

sender: String to store the user name of the user who send the friend request (limit 15 character).

receiver: String that store the user name of whom friend request is addressed (limit 15 character).

status: Integer to store the status of the friend request accepted, declined, unanswered.

message: The string to store the message send with the friend request (limit 5000 characters).

4.2 The Users table

In this table we store the users profiles. The columns in this table are:

fullName	userName	password	profession	livingCity	company	collageName	graduationYear

fullName: String to store the user name plus last name (limit 30 characters).

userName: String that store the user name/nick name of the user (limit 15 character).

password: String to store the user login key (limit 8 characters).

profession: String, store the user profession (limit 30 characters).

livingCity: The string to store the living city of te user (limit 30 characters).

company: String, store the user company (limit 30 characters).

collageName: String to store the user collage name (limit 30 characters).

graduationYear: Integer, store the graduation year.

Figure 3 shows the databases relational diagram.

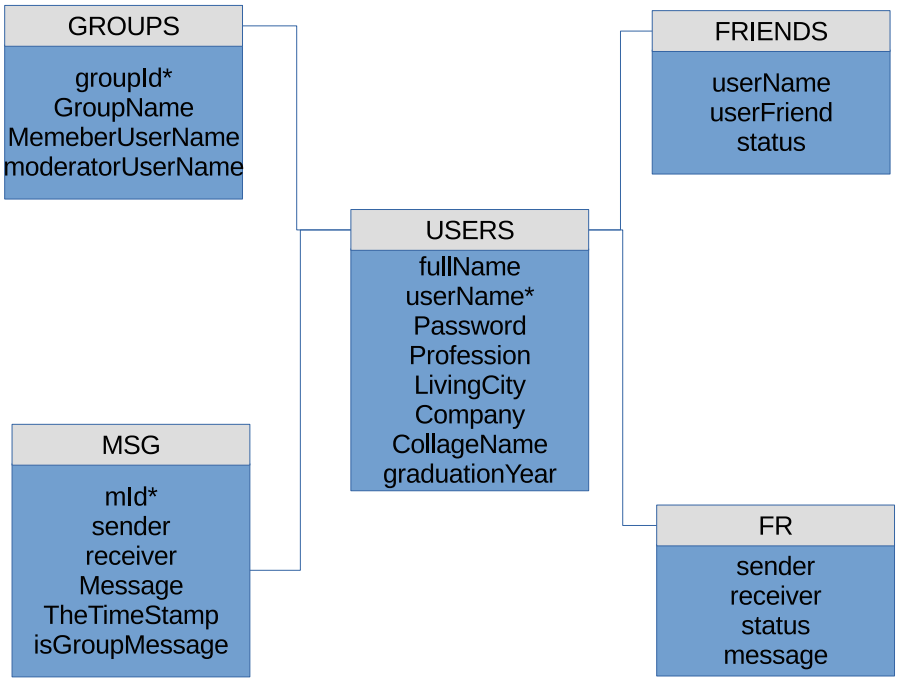

Figure 2: Server Database **DB** relational diagram.

4.3 Chat Server Databases interaction

As shown in previous section §some of the object abstractions used by the server includes: *Group.java*, *Message.java*, *User.java*, *FriendRequest.java* and *DatabaseManager.java*. The first tree classers are the abstraction for the Groups, Messages, Friend Requests and Users respectively. The DatabaseManager.java under *.rmi.server.database include the code to interact with the table in the database **DB** that store the data encapsulated on the four previous mentioned classes. Each of the five previously present table have a java class to hold connection to the database, table creation and queries. The classes are *GroupTable.java*, *MessageTable.java*, *FriendsTable.java*, *FriendRequestTable.java* and *UserTable.java*. The connection connect to connect in the database is (example for USERS table for the other table only change the content of the table variable):

```
host = "localhost";
database = "DB";
scheme = "APP";
dbType = "jdbc:mysql";
dbURL = dbType + "://" + host + "/" + database
+ ";create=true;user="+scheme;
table = "USERS";
try {
Class.forName(driver).newInstance();
conn = DriverManager.getConnection(dbURL);
//System.out.println("Connection Success");
}catch(Exception except) {
except.printStackTrace();
}
```

for details about the queries supported in each table see java code for each of the classes.

4.3.1 The Chat Server DatabaseManager

The DatabaseManager.java class make use of all the *Table.java* classes, the following functionalities are provided on the DatabaseManager (more details are available in the DatabaseManagement.java code documentation):

- User $\Leftarrow$ createUserAccount: Create a user and inserted in the *USERS* table, and return the object for the user.

- User ⇐ loginAccount: Verify if the given user name and password match if so return the object for the user.

- User ⇐ getUser: Verify if the user exist in the *USERS* table and return the object for the user is exist or null if not.

- boolean ⇐ updateProfile: Update the user profile but can't change the user name, return *true* if the update success *false* if not success.

- Group ⇐ getGroup: Return the group with given groupId or pair (groupName,moderatorUserName) if exist in the *GROUPS* table, null if not exist.

- ArrayList<Group> ⇐ getAllGroups: Return al list with all the groups in the *GROUPS* table.

- boolean ⇐ saveMessage: store the given message in the *MSG* table, return true if messages is inserted successfully or false if inserting the message fail.

- Map<String,CallbackClientInterface> ⇐ getListOfClients: Return a hash of the client user name and its CallbackClientInterfase for the online clients.

- CallbackClientInterfase ⇐ getOnlineClient: Return the CallbackClientInterfase for the given client user name.

- boolean ⇐ connect: Insert to the hash the client user name and client CallbackClientInterfase, if success return true, false otherwise.

- boolean ⇐ disConnect: Remove from the hash the pair client user name and CallbackClientInterfase, if success return true, false otherwise.

- ArrayList<Message> ⇐ getMessageForUser: query in the *MSG* table all the messages send to a given user after the messages are retrieved, those messages are deleted from the table, return a list of Messages or null if there is no messages.

- Group ⇐ joinGroup: Join a given user the given group, return updated Group or null if the join fail.

- Group ⇐ createGroup: Create a group and insert it to the *GROUP* table, return the created group.

- boolean ⇐ DeleteGroup: delete given group from the *GROUP* table, return true if success or false if not.

- boolean ⇐ inviteFriend: send a friend request to a given user, return true if request successfully saved in the *FriendRequest* table or false if not.

- void ⇐ makeThemMutualFriend: if the user accept the invitation the friendship is inserted in the *Friends* table with status *true*.

4.4 Bank Server Databases design

The server for the bank contains a database we call it **DBB**. This database contain two tables: USERS to store the bank members, TRANS to store the members transactions.

4.4.1 The Users table

This table store the users profile on the bank side (when a user is created on the chat is also created in the bank). The columns in this table are:

fullName	userName	password	livingCity	acNum	balance

fullName String to store the user name plus last name (limit 30 characters).

userName String that store the user name/nick name of the user (limit 15 character).

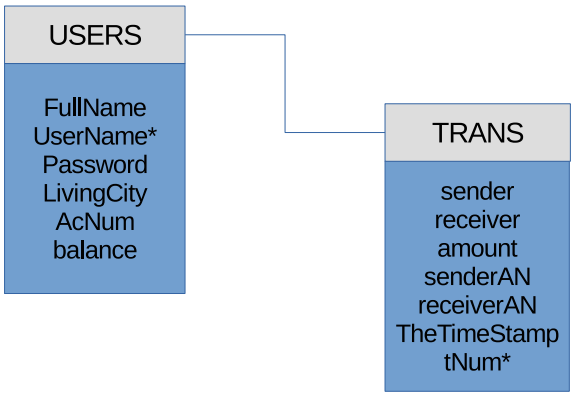

Figure 3: Bank Database **DBB** relational diagram.

password String to store the user login key (limit 8 characters).

livingCity The string to store the living city of te user (limit 30 characters).

acNum Integer to store the account number.

balance Double to store the user account balance in the bank.

4.4.2 The Transaction table

This table store only the unread messages, ones a message is retrieved from the table this one is deleted, server does not store history of message communication. The columns in this table are:

sender	receiver	amount	senderAN	receiverAN	theTimeStamp	tnum

Figure 3 shows the databases relational diagram.

sender String to store the user name of the user who send the money (limit 15 character).

receiver String that store the user name of whom money is send (limit 15 character).

ammount Double to store the amount of money send.

senderAN The sender account number.

receiverAN The receiver account number.

theTimeStamp A timestamp object contain the date and time when transaction was send.

tnum Integer to store the transaction reference number.

4.5 Chat Server Databases interaction

As shown in previous section §some of the object abstractions used by the bank server includes: *Transaction.java* and *User.java*. This two classes are the abstraction for the Users and user Transaction in the bank. The DatabaseManager.java under *.rmi.bank.database include the code to interact with the table in the database **DBB** that store the data encapsulated on the two previous mentioned classes. Each of the two previously present table have a java class to hold connection to the database, table creation and queries. The classes are *UserTable.java* and *TransactionTable.java*. The connection is similar to the char server database connection (database and table variable changes).

4.5.1 The Bank Server DatabaseManager

The DatabaseManager.java class make use of all the *Table.java* classes, the following functionalities are provided on the DatabaseManager (more details are available in the DatabaseManagement.java code documentation):

- User ⇐ createUserAccount: Create a user and inserted in the *USERS* table, and return the object for the user.

- User ⇐ loginAccount: Verify if the given user name and password match if so return the object for the user.

- User ⇐ getUser: Verify if the user exist in the *USERS* table and return the object for the user if exist or null if not.

- boolean ⇐ updateProfile: Update the user profile only modify fullName and livingCity, return *true* if the update success *false* if not success.

- boolean ⇐ sendTransaction: Store transaction on the transaction table if success modify the users balances accordingly, return true if success and false if not.

- double ⇐ getBalance: return the balance of a given user.

- boolean ⇐ DepositTo: deposit and amount of money to a given user. This only allow the user to deposit itself, return true is deposit success, false if not.

5 Client GUI Interface

The GUI consist of eight class each of one implement a view in the GUI, the classes are:

- *ChatWindows.java*
- *CreateAccountSreen.java*
- *CreateGroupScreen.java*
- *FriendRequestScreen.java*
- *HomePaga.java*
- *HomePageFriendAndGroupList.java*
- *ListOfOfflineMessagesScreen.java*
- *LoginScreen.java*
- *SearchFriendOrGroupScreen.java*

A better illustration of the GUI shown in the following Figure **??**:

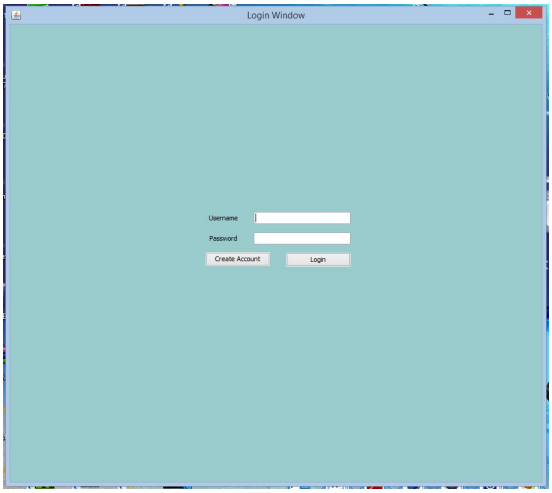

Figure 4: The user/client logins screen.

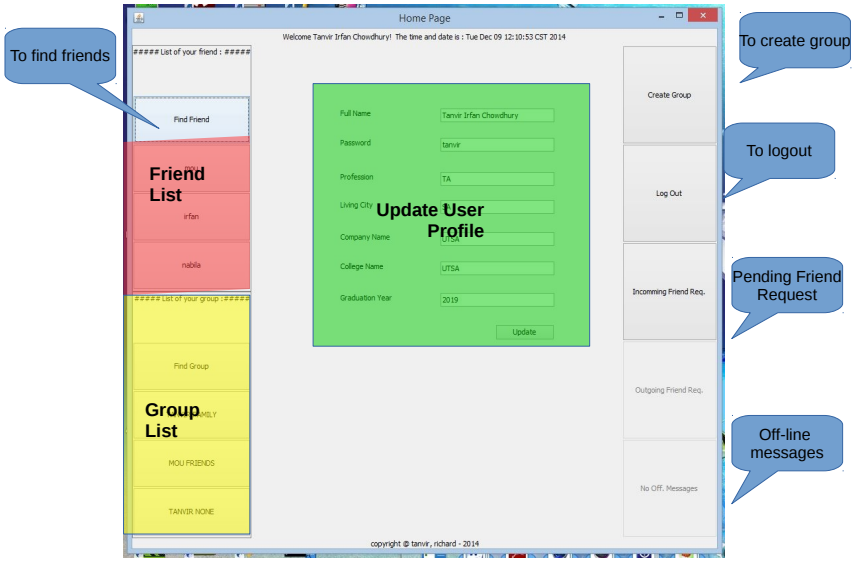

Figure 5: The home screen contain, the list of users, list of group, user profile update. Also the search users and friend request windows. Ones a user or group is selected the option to send appears.

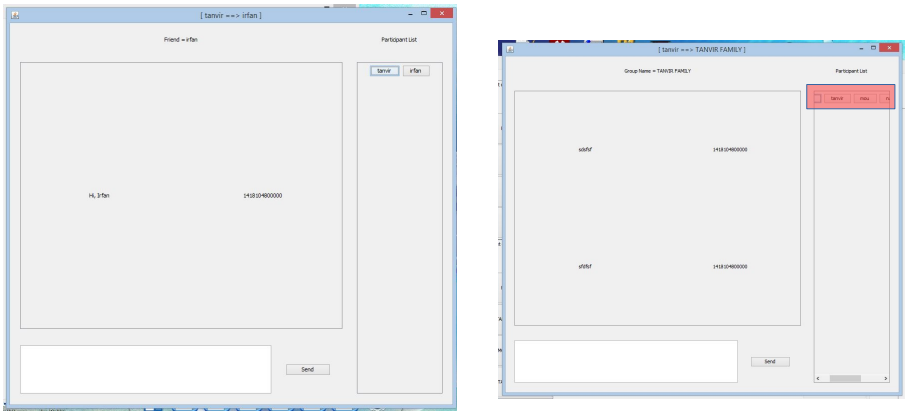

Figure 6: In the windows can see the messages conversation between users (on the left) and between user and group (on the right), the message and the time stamp are displayed. Also include the functionality to replay the message. Observer the red area in the group message contain the list of all users in the group

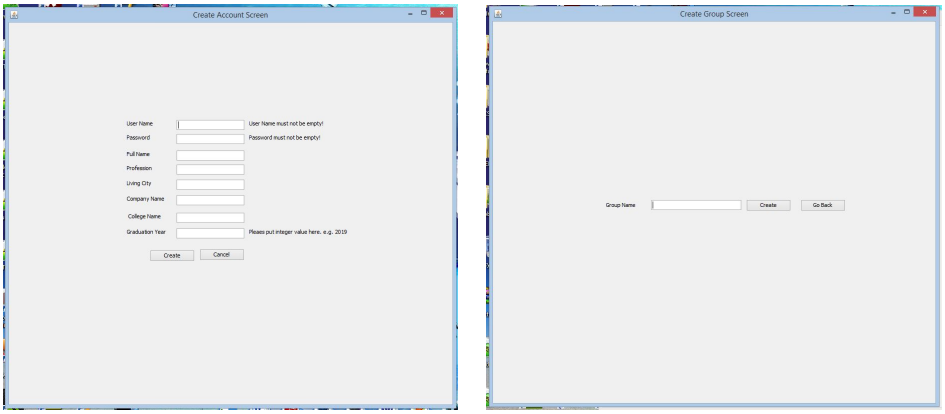

Figure 7: Windows to create a user (on the left) and window to create group (on the right).

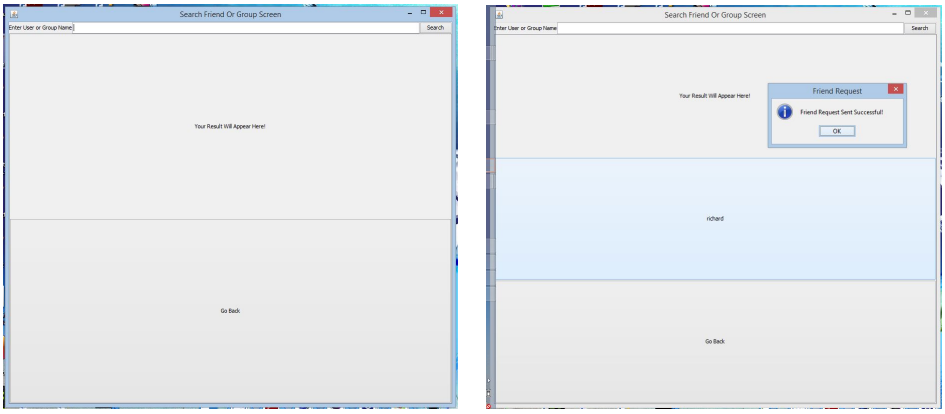

Figure 8: Windows to search for user, ones a user is selected a friend request can be send.

Contents