Design Milestone 1

Gary Huang (jiarui)
Henry Nassif (hnassif)
Sresht Rengesh (sresht)

PART I: Conversation design. Define a precise notion of *conversation* in your IM system. See the hints on how to do this. Specifically, name the Java classes you will create to implementing conversations, give specs of their public methods, and give a brief description of how they will interact. Include a snapshot diagram of a conversation in action.

Login

Before we define a conversation, we will first describe the method of logging in. Upon running the program, the user will be prompted to enter an IP address, port number, and a username and port in a welcome screen to the program. We use the stipulation that this username is alphanumeric (capital letters acceptable), and repeat usernames are not allowed. The IP address must be valid (four eight bit numbers or "localhost"), and the port must be a valid port number (a number between 0 and 65535). Once all of these conditions are met, the user can log into the chat client, where his or her username will be visible to all other users who specified the same port and IP address. We assume that there are no availabilities (available, busy, idle, invisible, etc), and all people show up as online once they enter the port. If given time, we would also consider implementing a dynamic search feature in which contacts are filtered when a query is searched into the search bar.

Initializing a Conversation

We will be defining a conversation as a text interaction between two users. A user can click on any other username that shows up as online and he will initiate a chat conversation with the other user. This list of online users will be updated by the server, which will store an arrayList that is modified every time a user logs in or out of the port. Time-permitting, we will extend the definition of a conversation to a text interaction between many users. In this type of conversation, every multi-person chatroom will have a randomly generated conversationID which will be stored in an arrayList. If there is a match in conversationIDs, the user will enter the conversation. If there is no

match (i.e. if the conversationID is not already in that arrayList), there will be an error message printed and the user will be prompted to enter a different conversationID or start a new conversation with another participant. We will also hold an arrayList of participants in each conversation in case we want to display this information to the participants in that conversation or use that information in some other way. This arrayList would be needed in case we later extend our allowed number of participants per conversation to more than two.

Participating In and Exiting a Conversation (Walkthrough)

Each conversation will have two participants. Users will have the option to either press return on their keyboard while in the input field or to press a "send" button to rally messages. A history will be stored of the current conversation, and this history will be displayed to each user in the conversation. The conversation ends when one of the participants leaves (i.e., the user who didn't leave will no longer be able to send messages). The user who didn't leave is still able to see the history of that conversation.

Classes, Methods, and Specs of the Conversation

Below is a summary of the classes, methods, and specs that we will use for the Conversation class. This is neither absolutely inclusive nor is it final; it's simply a list of the methods and classes that we think we will need. We will probably be adding to this list at some point in the future, depending on functional necessity.

```
 Conversation Class :

private int conversationID
private ArrayList<msg> messages
private ArrayList<Participants> currentParticipants

Constructor:
Conversation (Participant participant1, Participant participant2)//
creates a conversation between participant1 and participant2

private int getConversationID() // returns the unique ID of the
conversation
public ArrayList<Participants> getCurrentParticipants() // returns
all the participants in a conversation
public ArrayList<msg> getMessages() // returns all the history of the
```

conversation between two participants.
public void addMessage(msg message) // adds a msg object to the List of messages

//potential methods needed for conversations with more than 2 participants
public void connectParticipant(Participant user) // adds a participant to the participant's list
public void disconnectParticipant(Participant User) // removes a specified participant from the participants list. If there is only one participant, it will also remove the conversation from the list of active conversations.

Message class

private String msgContent // content of the message
private String senderID // ID associated with the sender

Constructor:
msg(String content, Participant user)// creates a msg on behalf of user

Methods:
public String getContent() Returns the content of the message
public User getSenderID() // Returns the ID of the sender
public String printMessage() // Returns representation of message to be printed by clients

We will also look into Java documentation to generate time stamps for the messages. The timestamp for each message would be stored in a private field in the Message class

Participant class

private String username
private ArrayList<int> activeConversations //stores the integer ID's of the conversations the participants is involved in

Constructors:
Participant(username)

Methods:
public String getUsername() // return Participant username
public void login() // asks participant for username, server IP and port and establishes
connection with server
public void logout() // terminates connection with server
public void initiateConversation(Participant friend) //creates new Conversation object with the specified participant

//potential methods needed for conversations with more than 2 participants
public void joinConversation(int conversationID) //adds the participant from the currentParticipants list
public void leaveConversation(int conversationID) //removes the participant from the currentParticipants list

Server class

private ArrayList<Conversation> conversations
private ArrayList<User> registeredUsers // used to check username uniqueness

public startNewConversation(Participant p1, Participant p2) // calls the conversation constructors and establishes connection with server

PART II: Client/server protocol. Design a set of commands the clients and server will use to communicate, allowing clients to perform the actions stipulated by the specification. Create a specification of the client/server protocol as a grammar. Also think about the state of the server, and the state of the client (if it stores any).

Below, we specify both a client-server protocol as well as a server-client protocol. This way, we have a grammar that details the communication that a client can make to the server as separate from those that the server can make to the clients. We won't be changing the states of the server or the client, but instead storing information as arrayLists and other data types (either primitive or not). We will handle errors appropriately, as we did in Problem Set 5: Jotto.

In our grammar below, we will use certain terminal symbols in both the client-server and server-client protocols. These include: **USERNAME** and **SPACE**. Both are specified only once, in the client-server protocol, but will be used in both.

Per formal grammar specification rules, we will specify strings messages using the keyword "_MSG". For example, our **CREATE_CONVERSATION** command is **CREATE_MSG SPACE USERNAME**, where **CREATE_MSG** is "create."

CLIENT-SERVER-PROTOCOL :: USERNAME SPACE COMMAND EOL
USERNAME :: [A-Za-z0-9]{6,18} *//6-18 character alphanumeric combinations are accepted*
COMMAND:: LOGIN | LOGOUT | CREATE_CONVERSATION | LEAVE_CONVERATION | SAY
LOGIN :: LOGIN_MSG SPACE USERNAME
LOGOUT :: "logout"
CREATE_CONVERSATION :: CREATE_MSG SPACE USERNAME *//create conversation with one other user*
LEAVE_CONVERSATION :: LEAVE_MSG SPACE USERNAME *//leave conversation with one other user*
SAY :: USERNAME SAY_MSG .+ *//say followed by a username followed by any message*
LOGIN_MSG :: "login"
CREATE_MSG :: "create"
LEAVE_MSG :: "leave"
SAY_MSG :: "say"
SPACE :: " "
EOL :: "\n"

SERVER-CLIENT-PROTOCOL :: MESSAGE EOL

MESSAGE:: ACCEPT_LOGIN | DUPLICATE_USERNAME | ACCEPT_LOGOUT | JOINED_CHAT| LEFT_CHAT | ONLINE_USERS | ACCEPT_MSG | ERROR

ACCEPT_LOGIN :: USERNAME SPACE WELCOME_MSG

DUPLICATE_USERNAME :: USERNAME SPACE DUPLICATE_MSG

ACCEPT_LOGOUT :: USERNAME SPACE ACCEPT_LOGOUT_MSG

JOINED_CHAT :: USERNAME SPACE JOINED_CHAT_MSG SPACE USERNAME

LEFT_CHAT :: USERNAME SPACE LEFT_CHAT_MSG USERNAME *//if A leaves chat with B, we specify "A left B"*

ONLINE_USERS:: (USERNAME NEWLINE)+

ACCEPT_MSG :: USERNAME SPACE USERNAME SPACE SAY *//if A says hi to B, the server will interpret it as "A B say hi"*

WELCOME_MSG :: "welcome, "

DUPLICATE_MSG :: "username_taken"

ACCEPT_LOGOUT_MSG :: "logout"

JOINED_CHAT_MSG :: "chat_with "

LEFT_CHAT_MSG :: "left"

NEWLINE :: "\n"

ERROR:: CONNECTION_ERROR | INVALID_CONVERSATION

CONNECTION_ERROR:: "connect_error"

INVALID_CONVERSATION:: USERNAME DNE_MSG

DNE_MSG :: "does_not_exist"