Jesse Dunietz, Isaac Joseph, Mike Kling 6.005 Project 3 First Deliverables

Abstract Design:

In our initial implementation, we expect to create a system in which users primarily conduct two-way conversations with each other. A conversation is initiated by one user attempting to send a message to another user (similar to dialing a phone number). We will also allow for chat rooms, which are publicly accessible forums with arbitrary numbers of conversation participants. Unlike two-way conversations, chat rooms can be joined by anyone who knows the name of the room.

In the long run, we hope to implement buddy lists as well, though this feature will be added after a basic system is working. Buddy lists will be maintained on the client-side, and will not persist between IM sessions. Associated with buddy lists are sign-in and sign-out messages: if user X signs in or out, all users who have user X on their buddy lists will be notified.

We also hope to implement a number of extra features. In order of priority, these are:

1. Forwarding messages to cell phones
    - In a buddy list, a user may specify a cell phone number and carrier for a particular buddy. When that buddy is offline, messages will be delivered to that number via the email associated with that cell number and carrier.
2. Typing notifications
    - A user receives notification that the user on the other end of a two-way conversation has started, stopped, or paused his/her typing of a message.
3. File-sending
4. A "Hello" protocol by which the server determines whether or not each client is still actually connected.
5. User invisibility
    - An online user can choose not to appear on others' buddy lists.

To initiate a conversation, a user "dials" another user by opening a dialog box and typing the name of the person he or she wants to send message to. Each client maintains a list of conversations, while the server simply acts as a switch that forwards received messages from a sender on one channel to a recipient on another channel. The server knows nothing of conversations; the client maintains a list of past messages from a particular user or chat room as a conversation (each conversation has its own window). When the client receives a message, it displays it in the correct conversation window based on username or chat room name. If the client receives a message from a username it hasn't seen before, then it opens a new conversation window (prompting the user to accept if the sender is not in the user's buddy list).

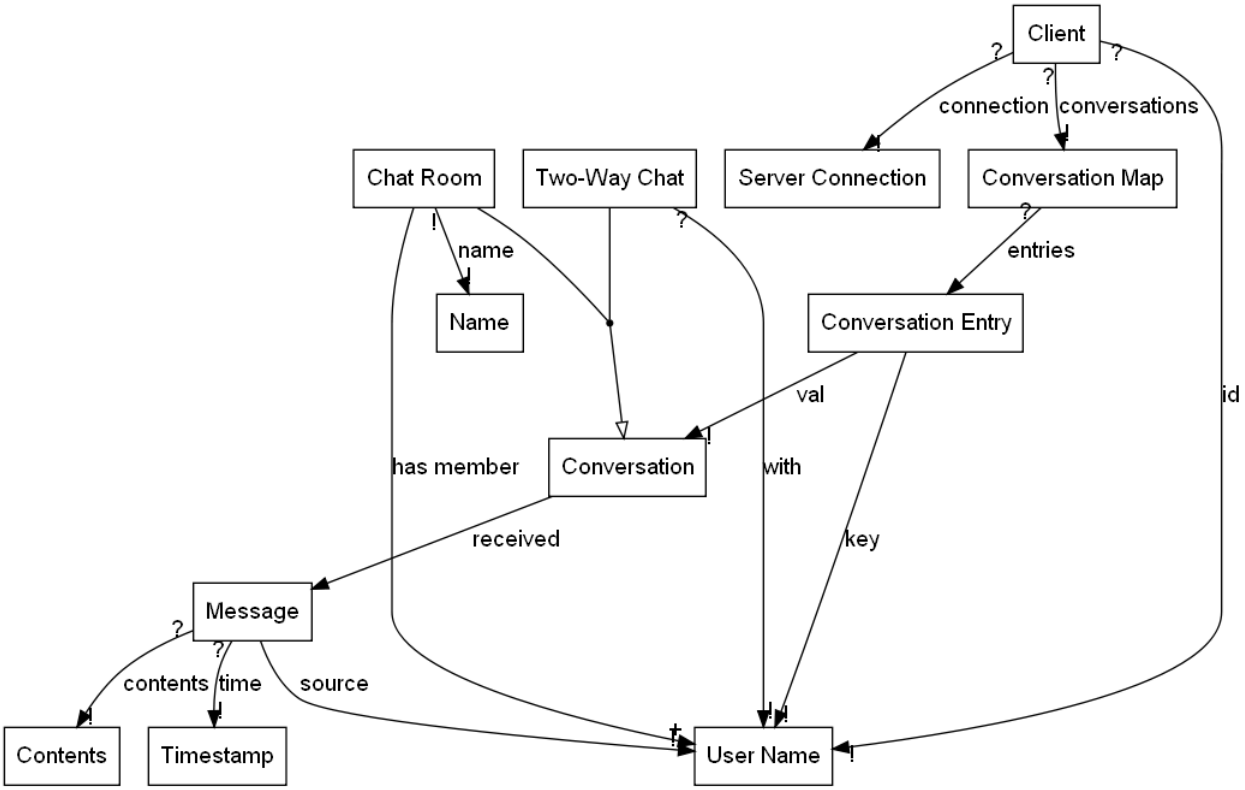
We chose this model over having the server manage conversations as pairings of users and maintain one connection for each end of the conversation. For the latter model, the server would need to maintain much more state about the currently connected users. This would mean more more threads, since each socket needs its own thread listening for incoming data, and

concurrency would be a bigger problem to deal with. Additionally, then we would have to have the client signal the server when a conversation is over, and when that should happen is not clear. In our design, the server can function with minimal knowledge of what goes on at the endpoints, and need not worry about proliferation of running threads and open sockets.

Object models:

Client:
Descriptions:
Client - an IM client instance
Server Connection - a communication channel with the server
User Name - a string identifying some current user of the IM network
Conversation Map - a map of usernames to conversations
Conversation Entry - an entry in the conversation map
Conversation - a collection of past messages received from a particular user
Message - a record of a previously received message from another user
Contents - an actual message string
Timestamp - an indicator of message arrival time
Chat Room - a multi-way conversation
Two-Way Chat - a standard chat conversation between two people which cannot be joined by anyone else
Name - the publicly accessible name of a chat room (and the name by which it can be joined)

Relationships:
connection - the server connection for this client
conversations - the set of currently active conversations for this client
entries - the set of key-value mappings in this map
key - the key for a map entry
val - the value for a map entry
with - the username on the other end of a two-way chat
name - the name of this chat room
has member - one of the users currently in the chat room
received - a message that was received as part of this conversation
id - the username the client chose at logon
contents - the contents of a logged message
time - the timestamp of a logged message
source - the sender of a logged message



Server:

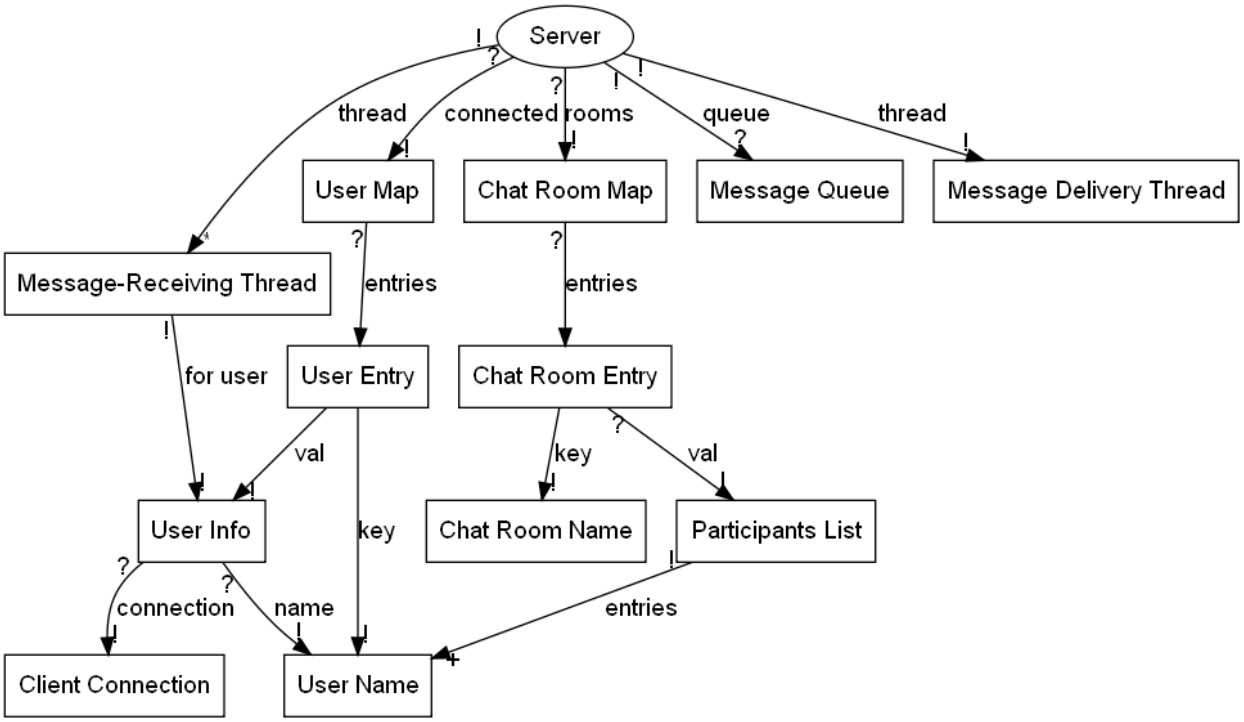
Descriptions:

Server - an IM server instance
User Map - a map of usernames to user info
Chat Room Map - a map of chat room names to lists of chat participants
Message-Receiving Thread - a thread to receive messages from a particular user
Message Delivery Thread - the one thread for this server instance that processes messages awaiting routing, and requests awaiting treatment
Message Queue - a FIFO queue of undelivered messages and unprocessed requests
User Entry - an entry in the user map
User Info - a data structure containing information about a particular user
User Name - a string identifying some current user of the IM network
Client Connection - a communication channel with a particular client
Chat Room Name - a string uniquely identifying a particular chat room
Chat Room Entry - an entry in the chat room map
Participants List - a list of current chat room participants for a particular chat room

Relationships:

thread - a thread running on the server
connected - the users who are currently connected
queue - the queue of pending messages/requests
entries - the entries of this map or list

key - the key of a map entry
val - the value of a map entry
for user - the user from whom this thread receives messages
connection - the client connection associated with a particular user
name - the name associated with a particular user



Client/Server Protocol

All communications between client and server will take place via exchanges of XML messages.

There are two types of XML snippets that a client may send to the server:

- A *message* consists of content to be sent to another user. Every message XML snippet contains a destination indicating what user it is to be sent to. There may be different types of messages - contentful messages, messages about typing status, chat room messages, and so on.
- A *request* is a command sent to the server, such as a login request or a request to join a chat room. The contents of this request XML document depend on the type of request.

The server also sends two types of XML snippets:

- A *message* is as defined above. When a server forwards a message to the appropriate destination(s), it inserts a tag indicating the user who was the source of this message.
- An *acknowledgment* (or ACK) is as it sounds - a simple indication to a client that an XML document sent by the client was received. Every ACK has a status element, which

is "OK" if the operation requested by the client succeeded, and otherwise contains a string indicating what error occurred.

When a client sends a message for which it expects an ACK, it hangs momentarily waiting for the response, and times out if it does not receive one quickly. (An error message is displayed on timeout.) No acknowledgment is given for requests to log out or to leave a chat room.

There is no request type for creating a new chat room; when someone requests to join a room by a particular name, the room is created on the server if it does not exist already. The room is destroyed when the last person leaves it.

Below are some sample client-generated XML messages.

```
<LoginRequest>
    <username>Group03</username>
</LoginRequest>

<LogoutRequest>
    <source>Group03</source>
</LogoutRequest>

<Message type="2way">
    <dest>Group04</dest>
    <contents>This is a message.</contents>
</Message>

<Message type="chat">
    <dest>chatroom2008</dest>
    <contents>This is a message.</contents>
</Message>

<Message type="starttyping">
    <dest>Group04</dest>
</Message>

<Message type="pausetyping">
    <dest>Group04</dest>
</Message>

<Message type="endtyping">
    <dest>Group04</dest>
</Message>
```

```
<JoinChatRoomRequest>
    <room>chatroom2008</room>
</JoinChatRoomRequest>

<LeaveChatRoomRequest>
    <room>chatroom2008</room>
</LeaveChatRoomRequest>
```

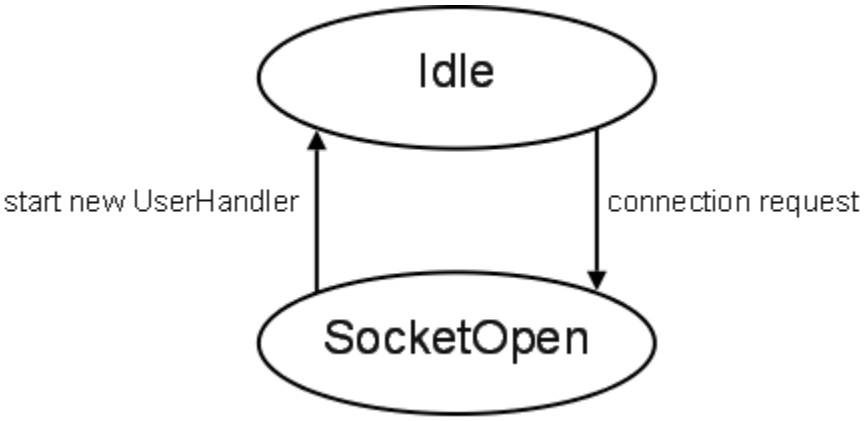
Some sample server-generated XML messages.

```
<Message type="2way">
    <source>Group03</source>
    <contents>This is a message.</contents>
</Message>

<Message type="chat">
    <source>Group03</source>
    <dest>chatroom2008</dest>
    <contents>This is a message.</contents>
</Message>

<LoginAck>
    <status>OK</status>
</LoginAck>

<LoginAck>
    <status>TAKEN</status>
</LoginAck>

<MessageAck>
    <status>OK</status>
</MessageAck>

<MessageAck>
    <status>OFFLINE</status>
</MessageAck>

<MessageAck>
    <status>MALFORMATTED</status>
</MessageAck>

<JoinChatAck>
```

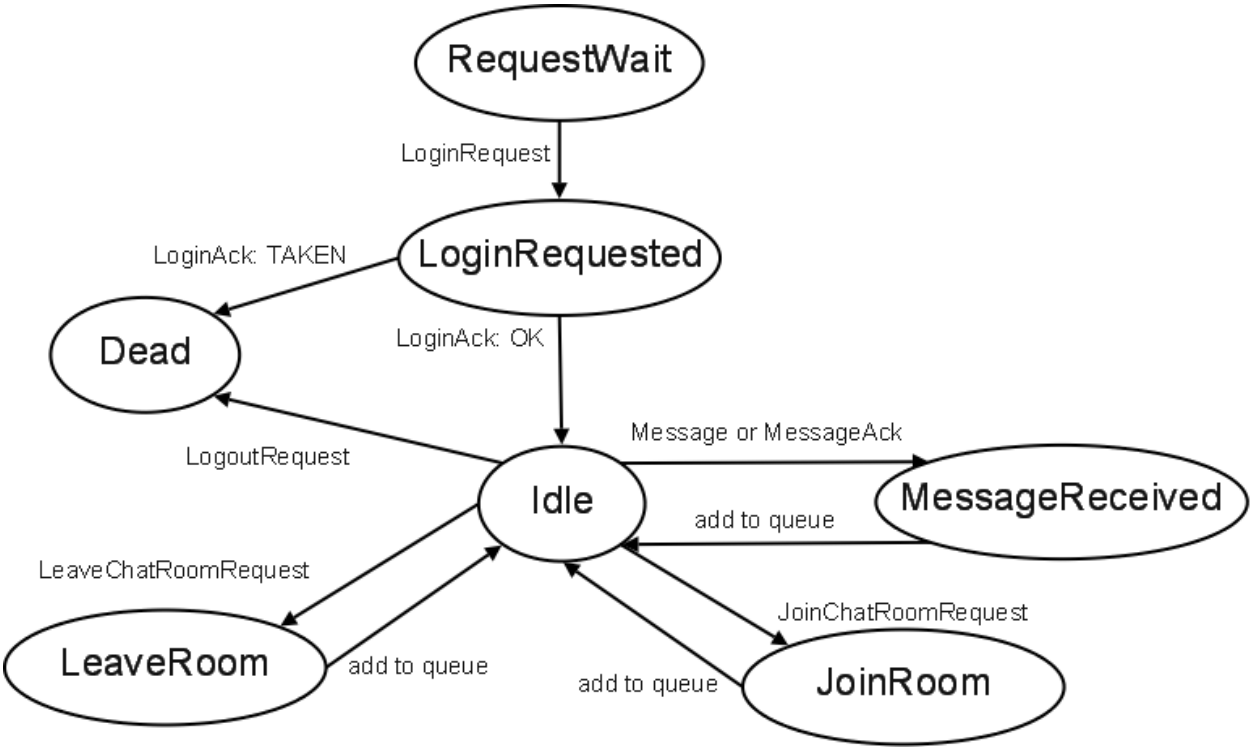
```
    <status>OK</status>
</JoinChatAck>

<JoinChatAck>
    <status>CREATEFAILED</status>
</JoinChatAck>
```

State Machines:

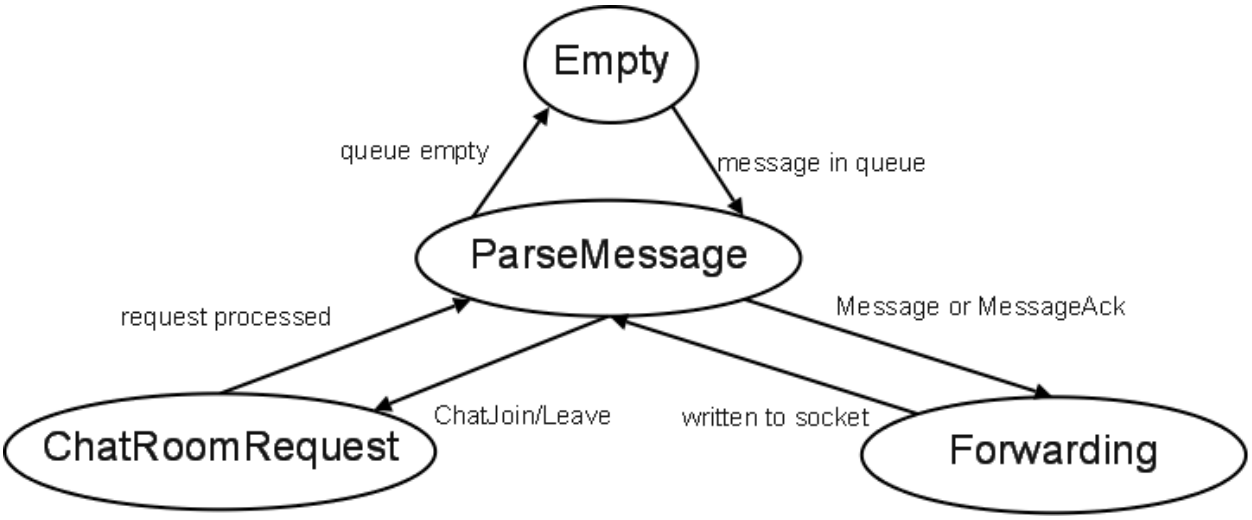
In addition to our XML text based protocol, we've created a number of state machines to help represent the client and server in our models. First, we have the state machine for the main server thread. The server starts in an idle state, awaiting connections on a given port. When a connection is received, the socket is opened, and a new UserHandler thread will be created to initialize and receive messages from this user.



Next, we created a state machine for the UserHandler thread. To start, this thread will be waiting for a login request. When this is received, if the name is taken, the connection will be determined and the thread will end. If it is not taken, this user will be added to the server, and the thread will enter an idle state, waiting to receive messages. Messages to be forwarded to other users will be added to the queue, messages to join or leave a chat room will be processed properly, and a LogoutRequest will close connections and terminate the thread.

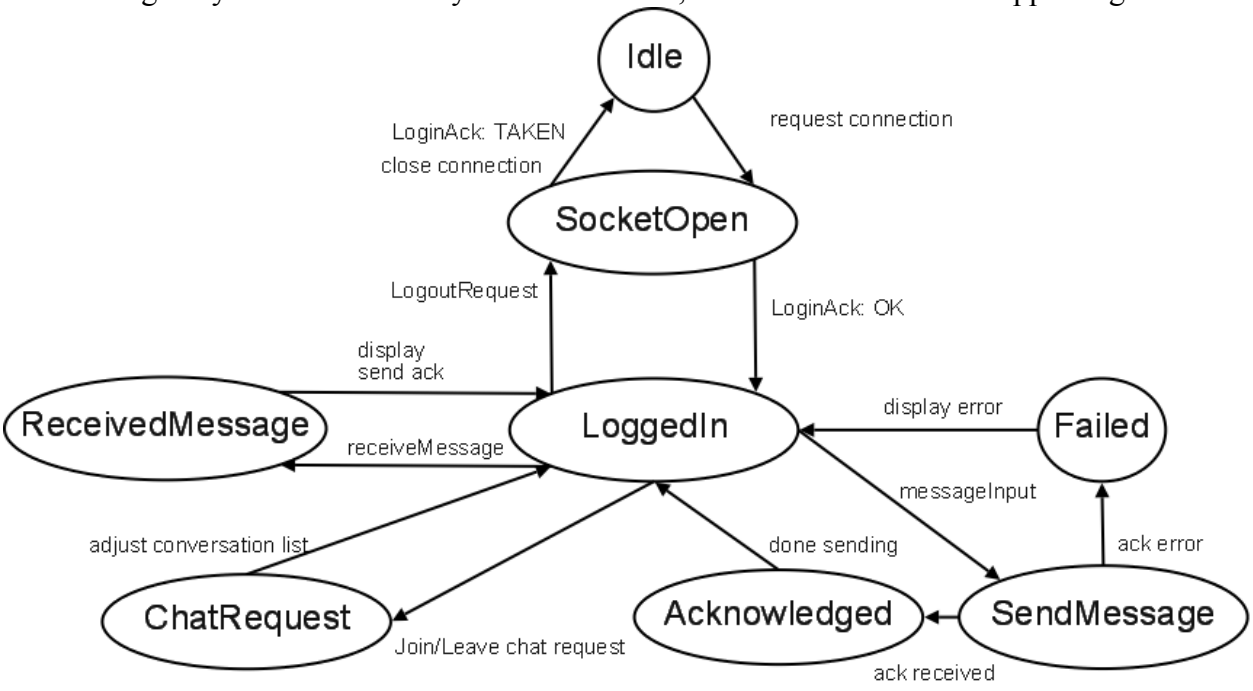


Next, we have a state machine for our message queue. This will run in a separate thread. When this thread is notified that there is a message in the queue, it will parse the message. If it needs to be forwarded, it will write to the appropriate socket, and if it is a chat room request, it will process that locally by adjusting the associated data structures.



Finally, we need a state machine for the client. The client begins in an idle state, and then attempts a login request. If acknowledged OK, the user is logged in, and if not, they are disconnected and can try again. Once logged in, they can receive messages, which need to be

properly displayed and replied to with an acknowledgement, make chat requests to the server, which will change their local list of conversations, as well as send messages, which should be acknowledged by the users that they are intended for, with an error otherwise appearing.



Usability Design:

Our user interface will begin with a basic login screen, where the user will type in a username. If this username is not taken, they will be logged in and brought to the next screen, but if it is taken, they will be disconnected and returned to the basic login screen.

The next screen will contain three options. These will be labeled as buttons, for either IM, chat, or quit. Quit will end the application. IM will initiate a 2 way chat with another logged in user. Chat room will allow the user to iniate or join a named chatroom.

Once IM or Chat is selected, another dialog will ask for the name, of either the user or chat room to connect to. For an IM, if this is selected, this window will then close, and a window titled with the name of the person being chatted with will open. This will contain a small bar for entering text on the bottom, and a large square area to view the current conversation will appear above this text entry box. As the users talk, their conversation will grow downwards in the dialog box. A small send button will also appear next to the entry area, which can be clicked on to send a line of text.

If a chat is selected, a similar window will open, containing the chatroom name, a text entry box, and a list of users in the chatroom vertically along the right hand side, next to the area where text between users is displayed. Both of these windows will have boxes which can allow the user to close them. The original window, containing the IM, chat, and quit buttons will always remain

constant. So, from there a user can initiate multiple chats at once.

The main merit of this design is its simplicity. If we have additional time, we will add buddy list visibility, as well as additional options, like text messaging abilities, and typing notifications, which will appear as a change in the way a user is highlighted in a chat room.