Final Report on Project 2

Alvaro Morales, Ayesha Bose, John Holliman

{alvarom, aybose, holliman}@mit.edu

Client-Server Protocol

We use Request and Response objects for server/client communication. More specifically, we convert Java objects to JSON, send the JSON objects between the client and server, and then convert the received JSON object back into a Java object. The java object is then processed using the Visitor pattern as it will implement either the Request or Response interface as is appropriate. For instance, when a client wishes to communicate with the server a Request object of a specified type (e.g. LoginRequest) is created, converted to JSON using GSON and sent to the server where it is converted back into a Java object and processed using a visitor. The same is done with Response objects for communication from the server to client.

A Request object is used in the following cases: login, logout, leave a room, join a room, get available rooms, and send a message.

A Response object is used in the following cases: login, join a room, errors, send a message, user joins or leaves a room, receiving available rooms.

Note: Request and Response are marker interfaces. The objects that implement them are processed by means of the visitor pattern after they go across the wire.

GSON (https://code.google.com/p/google-gson/)
“Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of. There are a few open-source projects that can convert Java objects to JSON. However, most of them require that you place Java annotations in your classes; something that you can not do if you do not have access to the source-code. Most also do not fully support the use of Java Generics. Gson considers both of these as very important design goals.”

Overall Functions
Server: accept connections
User: handle requests
Chat Room: broadcast messages to users
ResponseHandler: handle responses

Revised Design

`class Server`
Starts a ServerSocket on the default server port. Listens for connections, and starts a User thread for every client that connects. It keeps a blocking queue instance, and passes it to every User.

`class User implements Runnable`
The class first authenticates the connected client, by listening for a Login request and processing it. The first request must be a Login request.

Then, the User listens for any more requests. Upon receiving a request, it adds it to the blocking queue.

`class ChatRoom`
Represents a chat room. Keeps a list of Users connected to the chat room. Has methods to push a Response to connected users.

`class RequestHandler implements Runnable`
Class in charge of processing requests and emitting responses. Keeps a list of active chat rooms, and a blocking queue for processing requests. Takes a request from the blocking queue, and atomically processes it. Broadcasts a Response to all users in the chat room. Listens for more Requests.

`class Client`
Creates an instance of the GUI and sets it to be visible. At this point, the client is not connected to the server.

`class ChatGUI`
This class represents the GUI. It contains all of the GUI components and the corresponding action listeners.

`class ChatSession`
This class represents an instance of a chat session. It is created when the user attempts to login. If the server does not accept the client's login attempt the ChatSession will terminate. The ChatSession is responsible for spawning two threads: ResponseListener and ResponseHandler. Keeps a list of active ChatWindows, previously active ChatWindows, and available chat room. Also contains a blocking queue of server Response objects.

`class ChatWindow`
This class represents the client version of a conversation. It contains a name, which is unique, a list of message (only ones that were sent while the client was in the room), a list of users, and an unread message count.

`class ResponseListener implements Runnable`
This class listens to the socket for server Responses and puts them into a blocking queue in the ChatSession.

`class ResponseHandler implements Runnable`
This class passes the Response objects in the ChatSession queue to an instance of ClientVisitor.

Concurrency Strategy

Server Side

We will use a LinkedBlockingQueue implementation for the request blocking queue. This implementation is thread-safe, and allows multiple Users to put Requests onto it without running into race conditions, messing up the order of requests or overwriting requests. These methods will be atomic.

The request handler class will contain an overloaded handleRequest method (one for each type of requests, e.g. Login, SimpleMessage,etc). Each of these methods will be synchronized. The running RequestHandler thread will take from the queue, and run the corresponding handleRequest method.

Because we're using a data structure that is thread-safe, and building a thread-safe RequestHandler class that contains synchronized, atomic methods, our server design is thread-safe.

Client Side

The client concurrency strategy is similar to that of the server. When the client connects to the server three things are created: a ChatSession, responsible for keeping track of the ChatWindows the client has open and maintaining a blocking queue of Response objects, a ResponseHandler, responsible for handling the Response objects, and a ResponseListener, responsible for listening for Responses for the server and adding them to the blocking queue in ChatSession.

As with the RequestHandler on the server side, the ResponseHandler will contain a Visitor class with overloaded visit methods. The ResponseHandler thread takes Responses from the blocking queue in the ChatSession and routes them to the correct visit method.

Most of the Response objects will require that we mutate the ArrayList of messages in a specific ChatWindow and write to the corresponding JTextArea in the GUI. This is all handled by

synchronized methods in the ChatWindow class. All modifications to the GUI will be done by a few appropriately synchronized methods.

When the user elects to terminate a ChatWindow, we send a Request to notify the server, remove the window from the list of active ChatWindows, and move it to the list of previously active ChatWindows. When the ChatSession is terminated everything is lost.

GUI Side

The GUI currency strategy revolves being careful of which methods an update the GUI. There are two parts of our GUI that are updated with regularity: The chatTextArea, the area messages appear in, and the JTableModel which keeps track of the status of different rooms the client is a part of. There is only one method in our whole system that can append text to the chatTextArea and that method, writeToWindow, is synchronized. Similarly to modify the JTableModel one must obtain a lock.

Testing Report

- Connect to Chat Server
- Create a username
- Create a new room
- Type message into room
- Leave room

This test showed the message displayed next to the specified username, the new chat room was in the list of available chat rooms. The user was then able to leave the room.

- Connect to Chat Server
- Create a username
- Create a new room
- Type message into room
- Leaves room
- Other user connects to chat server
- Other user creates a username
- Other user creates a new room

- Other user types message into new room

This test showed the message displayed next to the first specified username, the new chat room was in the list of available chat rooms, the other user was able to create a new username and login to the chat server and not see the other user's chat room immediately, the new chat room that the second user created joined the list of available chat rooms, the message the second user types was displayed in the second user's chat room next to the second user's name.

- Connect to Chat Server
- Create a username
- Create a new room
- Type message into room
- Other user connects to chat server
- Other user creates a username
- Other user joins existing room
- Other user types message into new room
- First user types message into room
- First user leaves room
- Second user leaves room

This test showed the message displayed next to the first specified username, the new chat room was in the list of available chat rooms, the other user was able to create a new username and login to the chat server and see the other user's chat room in the list. The second user was able to join the first user's room. The first user was notified that the second user joined. The second user was not able to see the first user's previous message. The second user was able to type in a message, which was displayed next to their username. The first user saw this new message. The first user was able to type in a message, which was displayed next to their username. The second user saw this new message. The second user was notified that the first user left. The second user was able to leave.

Other tests:

- One user, multiple chat rooms
- Multiple users, one chat room
- Multiple users, multiple chat rooms

All these tests behaved as expected. One limitation that we have is based off our definition of a conversation. If all users leave a room, it is no longer in the list of rooms for users to join. Any user that was previously part of the room, however, has the room in their list and can recreate the room by using the right panel.

Amendment tests:

1. Viewing an open conversation's history
2. Viewing a closed conversation's history (without re-entering)
3. Reopening a closed conversation brought up the previous history

Our history is limited to what the user was present in the chatroom for, to protect privacy.

We tested our client through the GUI tests detailed above.

In addition, we added some JUnit tests to test our client/server protocol and our server response handler. In order to do so, we had to build some infrastructure. We built a RequestTester class, tha runs a thread that sends (possibly delayed) requests to the server and listens for responses. We can compare the response list to the expected responses. Our server tests extend an abstract class called ServerTest, which contain methods to start and correctly stop a server inside a JUnit test.

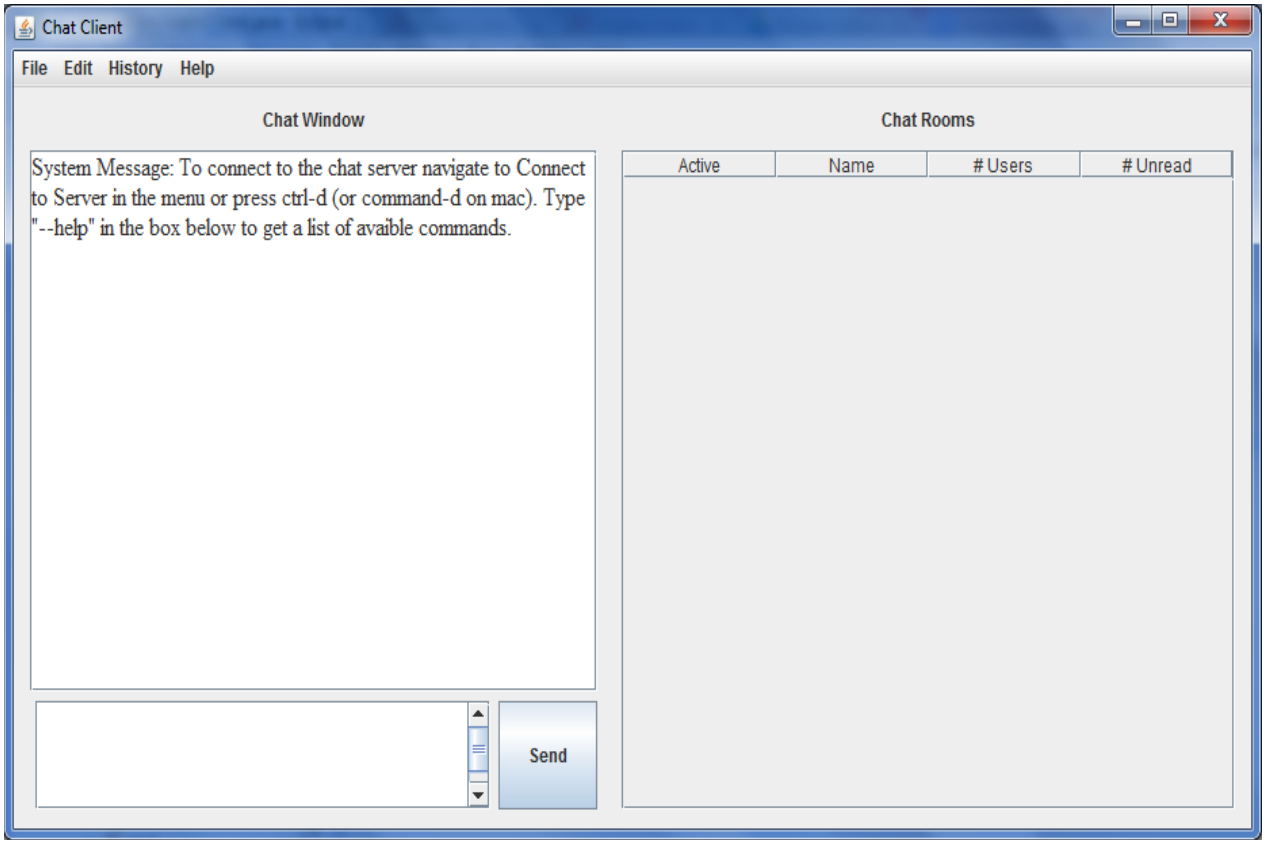
Please note that because google-gson uses reflection in its implementation, we cannot run our tests on didit. They have been flagged as `@category no_didit`. In addition, you cannot have an instance of the server running before running a test file, as a test starts a server instance.

For our server, we used JUnit tests that tested the following:

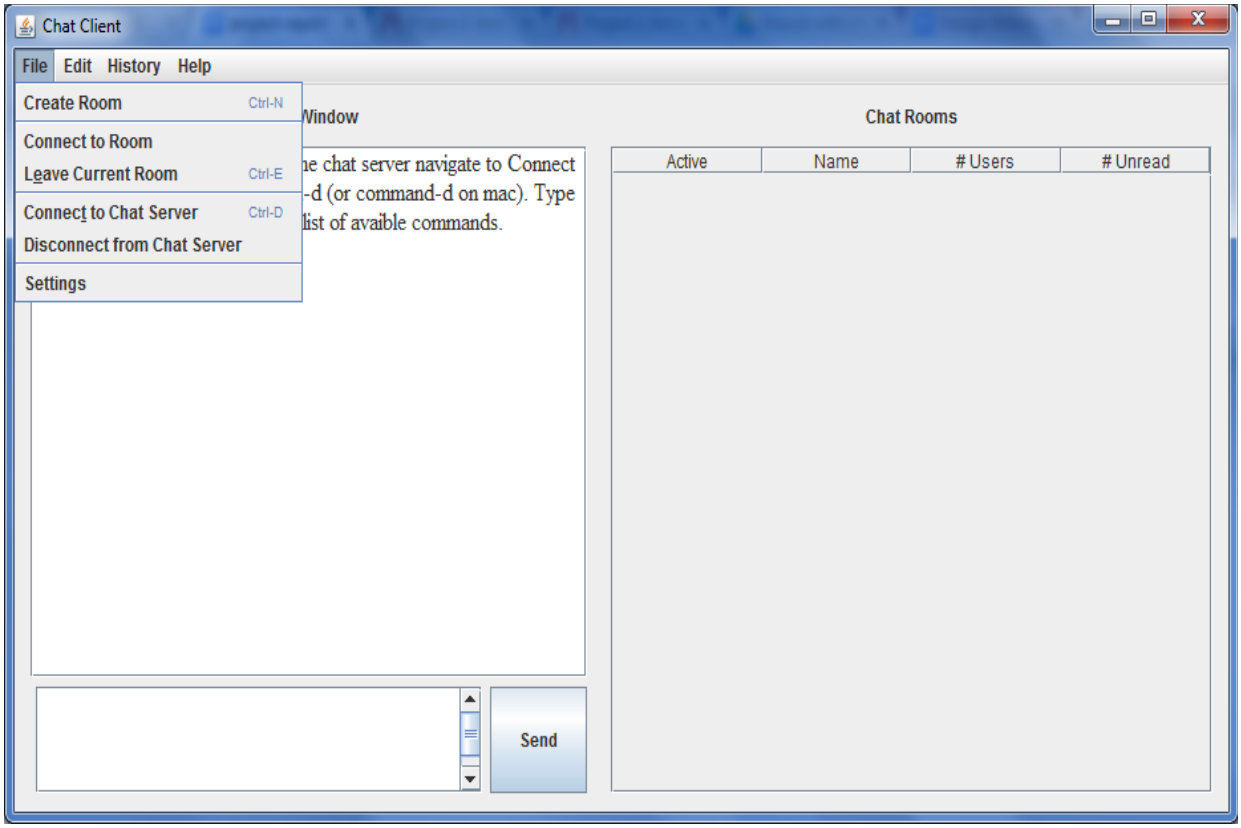
- RegistrationTest.java: Tests the login process
    - test the serialization of a login request object
    - test a user logging into the server
- RoomTest.java: Tests joining and leaving rooms
    - test a user joining a room
    - test a delayed request to join a room
    - tests the user joining an existing room
- MessageTest.java: Tests sending a message
    - test a user sending a message and receiving it back (this also tests that other users received it, because the server has to broadcast the message).

GUI Screenshots

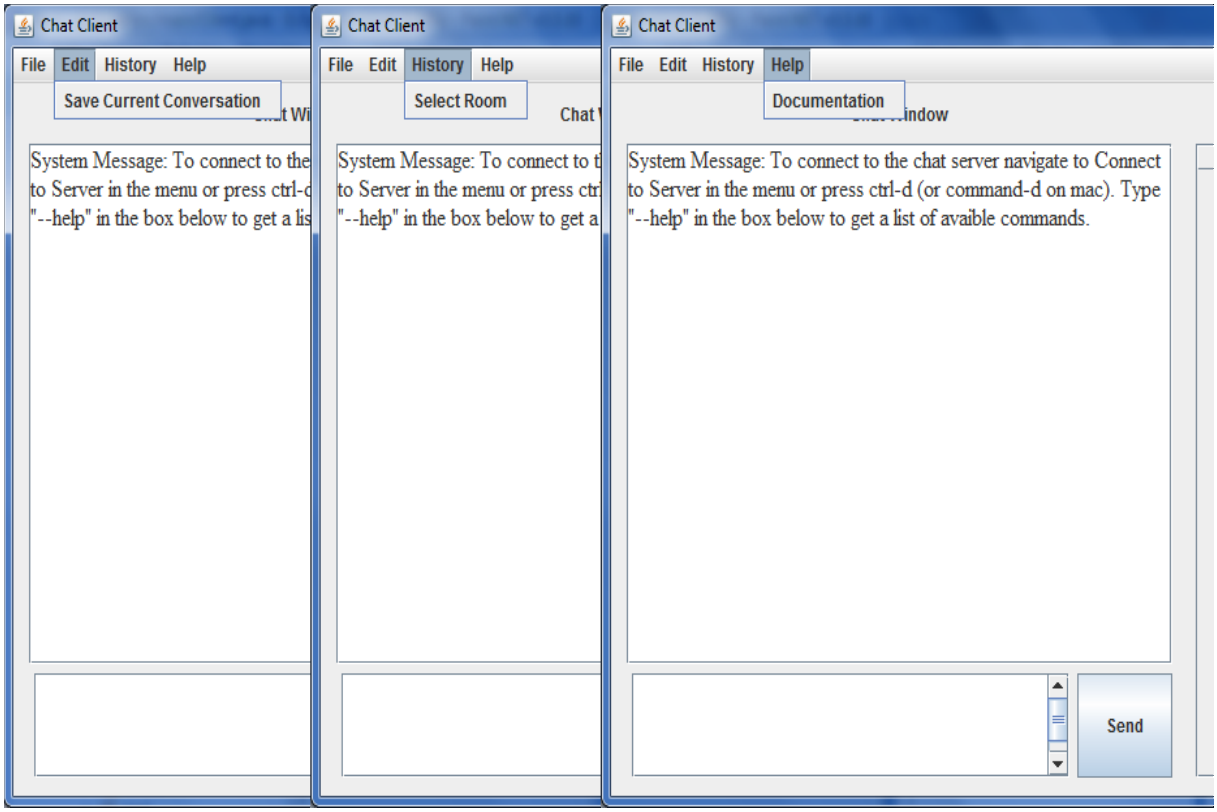
Main page:


Chat Client
File Edit History Help
Chat Window
Chat Rooms
System Message: To connect to the chat server navigate to Connect to Server in the menu or press ctrl-d (or command-d on mac). Type "--help" in the box below to get a list of avaible commands.
Active
Name
Users
Unread
Send

File menu:

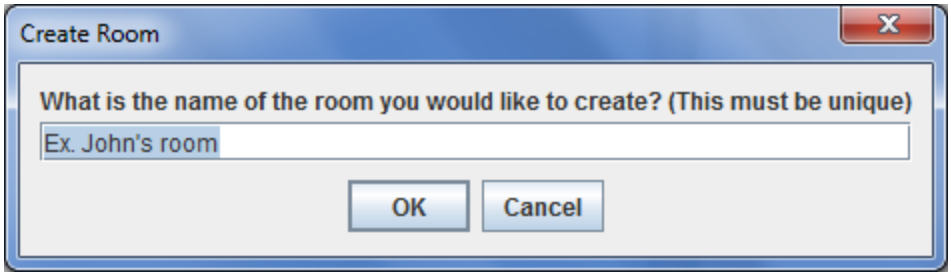

Chat Client
File
Edit
History
Help
Create Room
Ctrl-N
Connect to Room
Leave Current Room
Ctrl-E
Connect to Chat Server
Ctrl-D
Disconnect from Chat Server
Settings
Window
Chat Rooms
Active
Name
Users
Unread
Send

Edit and Help Menus:

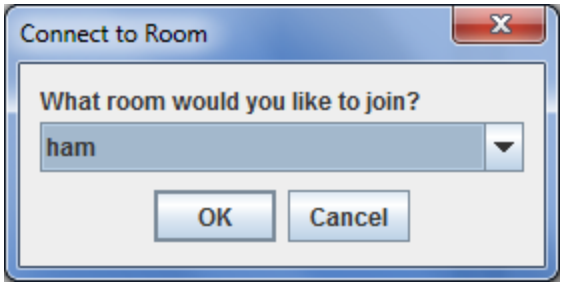


Connect to server:

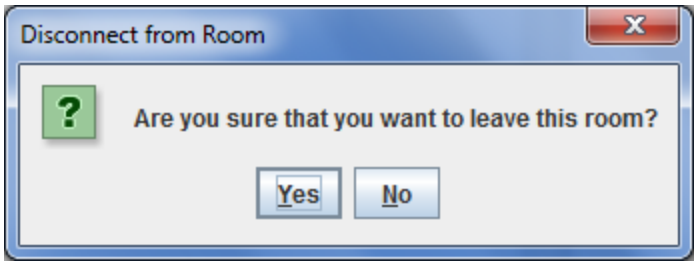
Chat room:



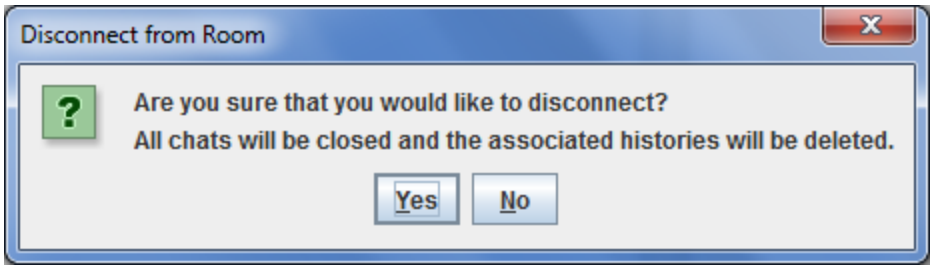
Connect to a room:



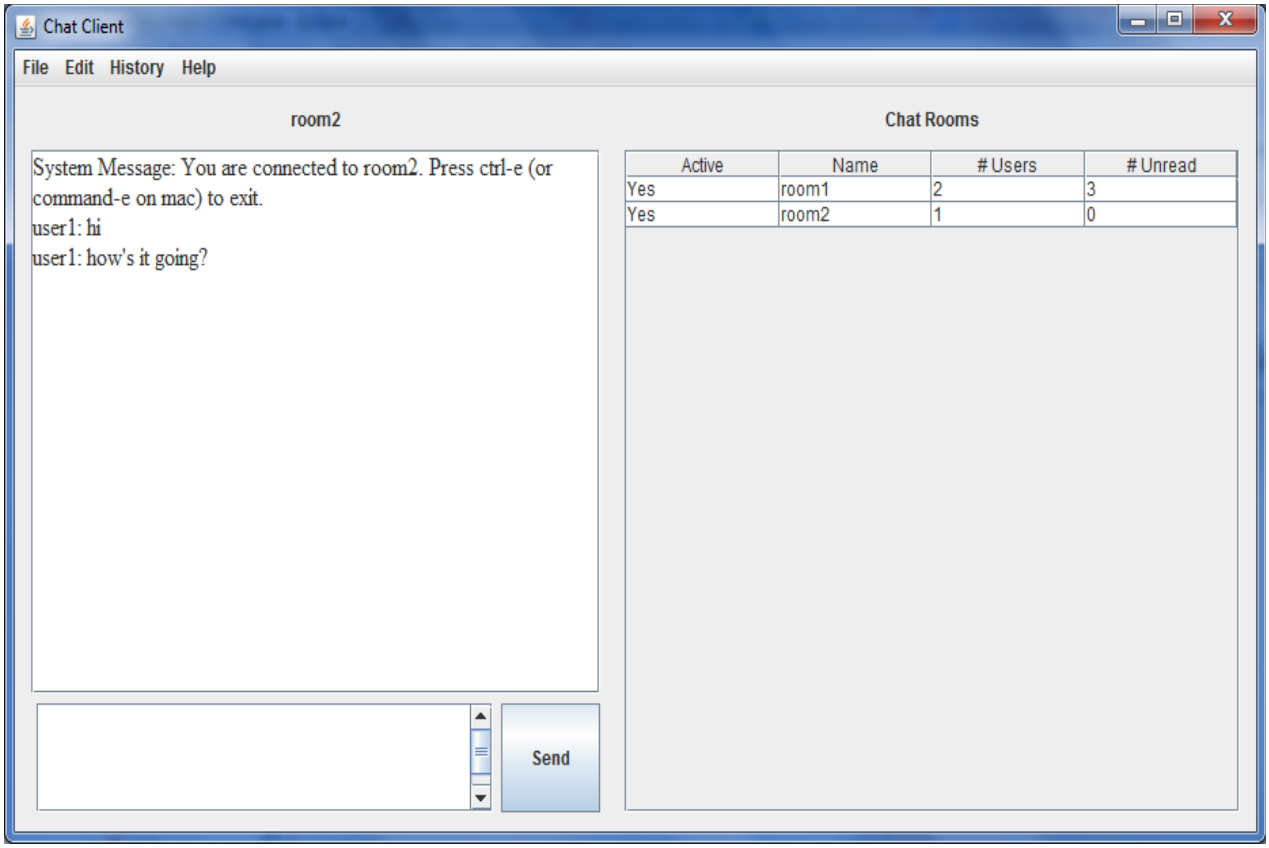
Leave a room:



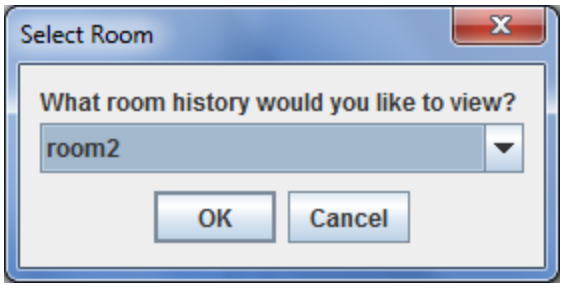
Disconnect from the server:



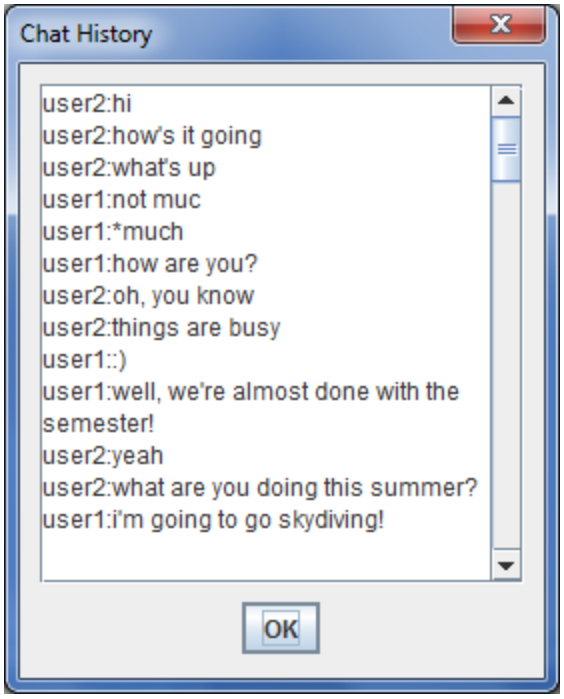
View with multiple chatrooms:



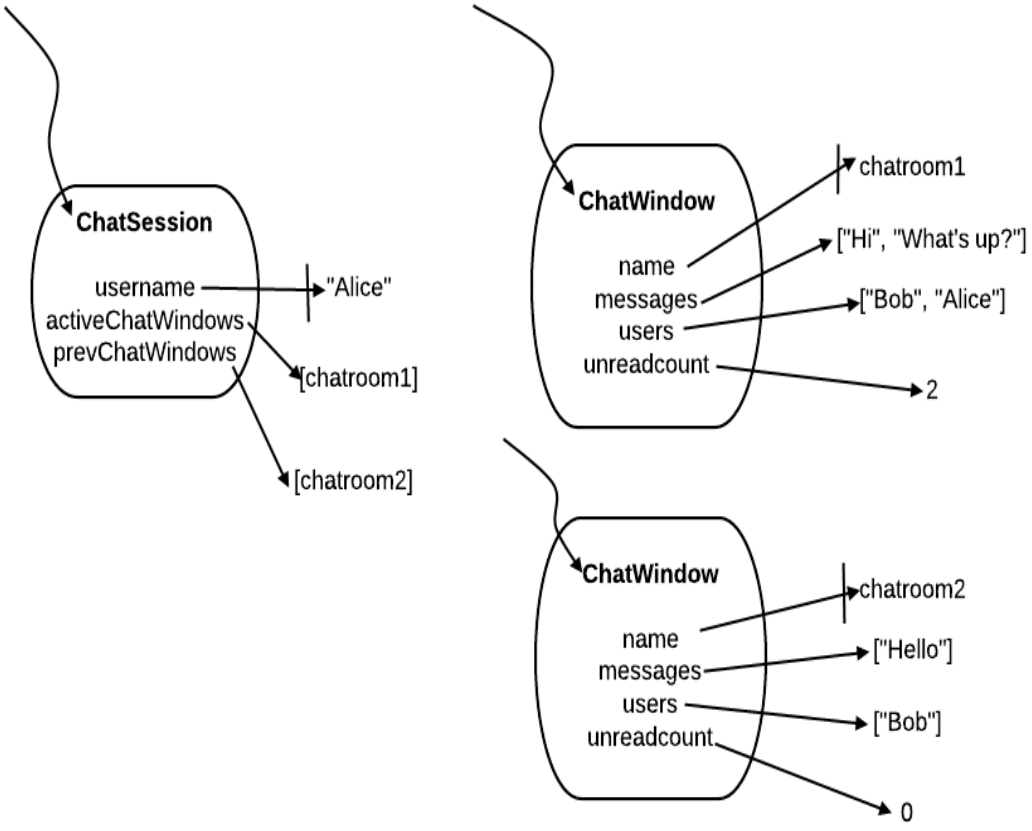
Room History Selection:



History:



Snapshot Diagram of Conversation


ChatSession
username
activeChatWindows
prevChatWindows
"Alice"
[chatroom1]
[chatroom2]
ChatWindow
name
messages
users
unreadcount
chatroom1
["Hi", "What's up?"]
["Bob", "Alice"]
2
ChatWindow
name
messages
users
unreadcount
chatroom2
["Hello"]
["Bob"]
0