Conversation Design

We try to emulate, for the most part, the design of the web-based Google Talk client. The major exception is that we use the chat room model for all conversations – a given pair of users can start multiple independent conversations with one another, since each conversation happens in a separate "room" to which additional users can be added at any time.

In particular, the client displays a list of online users. If a user wants to create a chat room with some set of users of size greater than or equal to one, he clicks on one of the desired users in the online users list and a chat window appears. To add any additional users to the conversation, the initiating user (or any other user currently in the conversation) simply needs type their handles into the appropriate text field in the chat window. Conversations do not have user-visible names or IDs, so individuals not currently in a conversation cannot add themselves to it – they must be added by a current communicant. Furthermore, a user cannot deny a conversation – if a message is sent to him or her in a new conversation, a new chat window automatically appears. Finally, the server stores the list of communicants for every conversation. Thus, only the conversation ID, the sender's handle, and the message contents need to be sent to the server for any given message, since the server knows all of the communicants in a conversation with given ID.

Classes:

1. Datatypes for server-to-client messages
    a. Classes that represent server-to-client messages all implement the interface SMessage with the following method
        i. accept(SMessageVisitor)
    b. The class SMessageImpls contains as static inner classes all of the classes that implement SMessage, which are (these represent each of the possibilities for SC_MESSAGE in the server-to-client grammar)
        i. OnlineUserList, a container (I use container to mean a class that simply stores data and has little additional functionality) with one variable: List<String> of handles
        ii. NormalAction, a container with the following variables (NormalAction also implements CMessage, see 2)
            1. Long id
            2. String senderHandle
            3. ActionType at; ActionType is an enum nested in NormalAction with three values: TEXT_MESSAGE, EXIT_CONV, and ADD_USER
            4. List<String> handles (may be null if not ADD_USER action)
            5. List<String> currentUsers (may be null if not ADD_USER or if this message was sent from client to server)
            6. String textMessage (may be null if not TEXT_MESSAGE action)
        iii. AvailabilityInfo

            1. String handle
            2. Status s – a nested enum with values OFFLINE and ONLINE (can easily be expanded to include other states, e.g. “busy”)
        iv. BadHandle
            1. String handle
        v. ReturnId
            1. Long id
    c. SMessageImpls also contains a factory method to convert (deserialize) a String server-to-client message into the appropriate SMessage – it leverages the deserialize method present in each class that implements SMessage
2. Datatypes for client-to-server messages
    a. The datatypes representing client-to-server messages are structured very similarly to those for server-to-client messages
        i. CMessage interface with one method: accept(CMessageVisitor)
        ii. CMessageImpls class that contains as static inner classes all of the classes that implement CMessage (other than the ones that also implement SMessage, which are found in SMessageImpls), which together cover all of the possibilities for CS_MESSAGE in the client-to-server grammar below
        iii. CMessageImpls also includes a factory method to convert a String client-to-server message into the appropriate CMessage, leveraging the deserialize method present in each class that implements CMessage
        iv. Both CMessage and SMessage (and their associated implementations) are needed because the set of client-to-server messages is different from the set of server-to-client messages and because the two types of messages are processed in completely different modules in the code
        v. The remaining details are omitted for brevity
3. Other datatypes used by both server and client
    a. SocketOutputWorker, extends Thread
        i. Constructor takes a socket whose output stream we should write to
        ii. Has a BlockingQueue in which messages to be written to the output stream are stored
        iii. run() method sits in a loop calling take() on the BlockingQueue and writes any data it pulls from the queue
        iv. Has add() method that takes a String to be written to the socket output stream, adds this String to the BlockingQueue
4. Server-specific datatypes
    a. Conversation – a container class with the following variables
        i. List<String> communicants – a list of the handles of the people in this conversation
        ii. Long id – the server-assigned ID for this conversation

b. ServerMessageDispatcher – forwards received CMessages to the appropriate CMessageVisitor (in this case ServerInputProcessor, see d)
    i. Implements CMessageDispatcher, which has the following method
        1. add(CMessage cm)
    ii. Extends Thread
        1. Its behavior in add() is to add the message to a BlockingQueue
        2. In its run() method it takes messages one-by-one from the BlockingQueue and passes them to the CMessageVisitor (meaning that message processing on the server is effectively sequential)
    iii. The reason we need a CMessageDispatcher like ServerMessageDispatcher is that we want to support both sequential (what we do currently) and parallel processing of incoming CMessages. A CMessageDispatcher designed for a parallel processing system would just pass the CMessage directly to the CMessageVisitor in add() rather than using a BlockingQueue.
    iv. It is appropriate here to justify why we chose a sequential message processing design as opposed to a parallel one. We initially had a parallel design but realized there were innumerable places where concurrency issues could crop up (e.g. a client receives a message whose correct interpretation requires knowledge of a previous message that has not yet been received) and much work would be required to ensure deadlock-free, correct operation. Since our chat system is only designed for a small number of users anyway (there is no buddy system and every user sees all other users connected to the server), there was no point going through the trouble to create a parallel processing server.
c. One SSocketInputWorker ('S' stands for 'Server') per connected client – reads input from the socket for the client
    i. Extends abstract class SocketInputWorker, which itself extends Thread
        1. Constructor takes a socket
        2. run() method reads input stream for socket and calls the abstract method process(String) with the read data
    ii. SSocketInputWorker constructor takes both a socket and an instance of CMessageDispatcher (in this case a ServerMessageDispatcher)
        1. process() is implemented to use the CMessageImpls factory method to produce a CMessage from the input String. It then calls the add method of the CMessageDispatcher with this CMessage as the argument
d. ServerInputProcessor, implements CMessageVisitor
    i. It has the following key variables (it has a few others, but these are most important)

1. Map from Long (conversation ID) to Conversation
2. Map from String (user handle) to SocketOutputWorker (for the socket for that particular user)

ii. For each possible CMessage, it decides what (if any) SMessage to send to the clients that should be notified. It constructs this SMessage and calls toString() on it to generate the serialized message to pass to the SocketOutputWorkers for the recipient clients.

iii. CMessages are acted on to the fullest extent possible – for example, if a NormalAction tries to add a set of users to a conversation, some (greater than or equal to 0) of whom are online, some of whom are not, and some of whom are already in the conversation, the server adds the online users who are not already in the conversation to the conversation and ignores (as opposed to reporting some sort of error) the other requested users

5. Client-specific datatypes
   a. Each client has one CSocketInputWorker, which extends the aforementioned abstract class SocketInputWorker
      i. Constructor takes an instance of SMessageVisitor and process() simply uses the SMessageImpls factory method to deserialize the message and then passes it to the visitor
      ii. There is no code duplication between CSocketInputWorker and SSocketInputWorker – both are very brief classes, with most of the key logic in the shared superclass SocketInputWorker
   b. ClientInputProcessor implements SMessageVisitor and Controller
      i. Controller has the following methods, all of which return void (these methods are called by window – in this case MainWindow and ConversationWindow – event handlers)
         1. initialize(Socket s, MainListener ml) – see c for more on MainListener
         2. getId() – sends a GET_ID message to the server
         3. registerHandle(String handle) – sends a REGISTER_HANDLE message to the server
         4. getUsers() – send a GET_USERS message to the server
         5. addUsers(long id, List<String> users) – send a NORMAL_CONV_ACTION with action ADD_USER to the server (id allows the controller to identify the relevant conversation)
         6. exitConversation(long id) – send a NORMAL_CONV_ACTION with action EXIT_CONV to the server
         7. sendMessage(long id, String message) – send a NORMAL_CONV_ACTION with action TEXT_MESSAGE to the server
      ii. ClientInputProcessor has the following key variables

1. Map from Long (conversation ID) to ConversationListener (in this case the implementation is ConversationWindow, see d)
2. Map from Long (conversation ID) to ConversationLog, which is essentially the model for the client – it stores all of the messages for a particular conversation. If the client leaves a conversation and then is added back to it by a user who remained in the conversation, this data structure allows us to repopulate the chat window with the old messages.
3. MainListener ml (in this case the implementation is MainWindow, see c)
4. SocketOutputWorker sow
5. String handle – this client's handle, set once a handle is registered

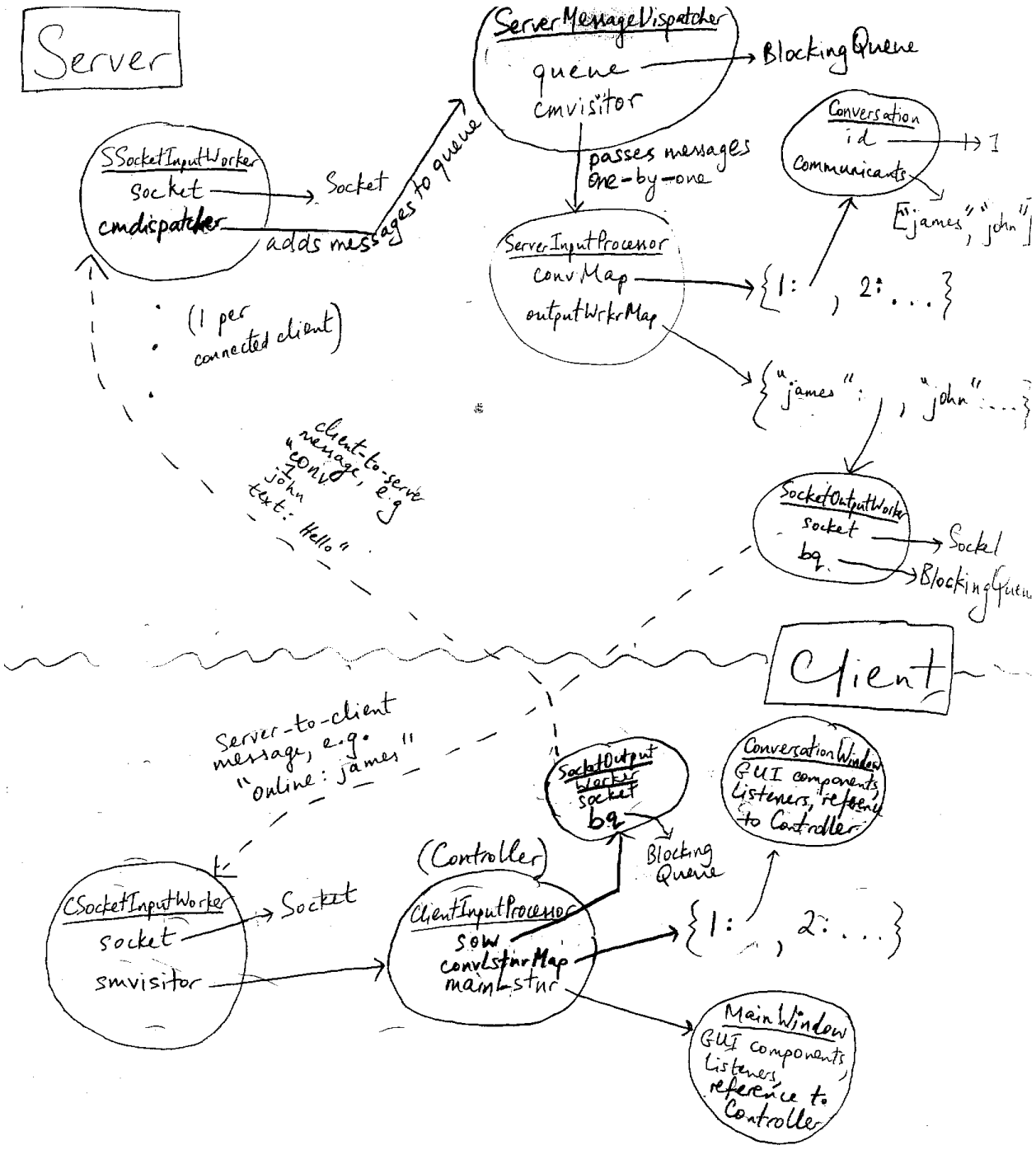
iii. At a high level, ClientInputProcessor not only forwards GUI-triggered messages to the server in the methods inherited from Controller, but it also processes messages from the server and forwards them to the GUI (ConversationListeners and MainListener) in the methods inherited from SMessageVisitor

c. MainWindow, extends JFrame, implements MainListener (can be thought of as the view for the buddy list part of the GUI)
   i. It implements the methods in MainListener, which are
      1. addOnlineUsers(Collection<String> handles)
      2. removeOfflineUser(String handle)
      3. badHandle(String handle) – called when the server indicates that a handle that the client attempted to register was already taken or contained invalid characters
      4. handleClaimed(String handle) – called when the server indicates that a handle registration request was successful
      5. newId(long id) – called when the server returns a new conversation ID in response to a client request (made when the user tries to initiate a new conversation)
      6. ConversationListener makeConversationListener(long id) – creates a new ConversationListener (in this case ConversationWindow) with the given conversation ID
      7. setController(Controller c)
   ii. When the user is in the process of registering a handle or tries to start a conversation with a new user, the MainWindow makes the appropriate call to the controller
d. ConversationWindow, extends JFrame, implements ConversationListener, one per active conversation on the client (can be thought of as the view for an individual conversation in the GUI)
   i. It implements the methods in ConversationListener, which are
      1. addMessage(String senderHandle, String message)
      2. removeUser(String handle)

3. addUsers(List<String> handles)
4. setController(Controller c)

ii. When the user enters a message for this conversation, adds a user to it, or tries to close the ConversationWindow, the ConversationWindow makes the appropriate call to the controller

Revised Snapshot Diagram:



Client/Server Protocol

Client-to-server grammar:
CS_MESSAGE := GET_ID | INITIAL_CONV_MESSAGE |
NORMAL_CONV_ACTION | REGISTER_HANDLE | GET_USERS | DISCONNECT
/* to obtain a new ID from the server for a conversation */
GET_ID := "getid" SEP HANDLE /* client's handle */
NORMAL_CONV_ACTION := "conv" SEP CONV_ID SEP HANDLE /* sender */ SEP
ACTION
USER_LIST := HANDLE ("," HANDLE)*
HANDLE := [a-zA-Z][a-zA-Z0-9]*
CONV_ID := [0-9]+
ACTION := TEXT_MESSAGE | EXIT_CONV | ADD_USER
TEXT_MESSAGE := "text: " TEXT
TEXT := [^\n]+
EXIT_CONV:= "exit"
ADD_USER := ("current: " USER_LIST SEP)? /* the list of users currently in the conversation, included only in the copies of this NORMAL_CONV_ACTION the server forwards to the added users */ "add: " USER_LIST
REGISTER_HANDLE := "handle: " HANDLE
/* want list of all online users */
GET_USERS := "getusers" SEP HANDLE /* client's handle */
DISCONNECT := "disconnect: " HANDLE /* this message is always faked by the server when it detects that the client has disconnected; it is never sent by the client itself */
SEP := "/" /* field separator */

Server-to-client grammar (uses productions defined above):
SC_MESSAGE := RETURN_ID | NORMAL_CONV_ACTION | USER_ONLINE |
USER_OFFLINE | BAD_HANDLE | HANDLE_CLAIMED | ONLINE_USER_LIST
/* return a conversation ID to a user who requested it */
RETURN_ID := "id: " NUM
USER_ONLINE := "online: " HANDLE
USER_OFFLINE := "offline: " HANDLE
BAD_HANDLE := "unavailable: " HANDLE
HANDLE_CLAIMED := "claimed: " HANDLE
ONLINE_USER_LIST := "users:" USER_LIST

State stored by client and server:
This is covered extensively in the conversation design part of this document. In summary, the server stores an ID for each conversation and the communicants in each conversation. It does not store the messages in the conversation. The client stores the ID for each conversation, and the client's view stores the list of online users, the communicants in each conversation, and the messages in each conversation. Furthermore, in order to implement the amendment requirement, a model data structure stores copies of the messages in each conversation.

Concurrency Strategy

Client:

To prove that our code is free of race conditions, we simply need to consider the variables touched by multiple threads and show that they are accessed in a thread-safe manner. For reference, there are three threads in the client (besides the main thread, which does no important work) – the event dispatching thread, the socket input processing thread (CSocketInputWorker), and the socket output processing thread (SocketOutputWorker).

The BlockingQueue in the SocketOutputWorker class is accessed by both the SocketOutputWorker thread and the CSocketInputWorker thread, but the Java API specification for BlockingQueue guarantees that all accesses to it will be thread-safe. The ConversationListener map in the ClientInputProcessor class is accessed by both the CSocketInputWorker thread and the event dispatch thread, so we make it a thread-safe map using the Collections.synchronizedMap() method. We do no iteration over the map contents, so we need no external synchronization. The threads that access the ConversationListener map also call methods on the ConversationListeners in the map and the single MainListener in ClientInputProcessor, but these methods modify object state – in this case, all of the GUI elements – only on the event dispatching thread using SwingWorkers or SwingUtilities.invokeLater(). There is another map in ClientInputProcessor that stores conversation logs, but it is only accessed by the CSocketInputWorker thread and therefore needs no sychronization. This covers all important client-side state.

There can be no deadlocks in the client because there is no synchronization or locking whatsoever in the code. (The only synchronization is performed internally by Java API data structures, which we can assume to be free of deadlocks.)

Server:

We use the same approach as above to demonstrate the absence of race conditions in the client. For reference, there are 2N + 2 threads on the server, where N is the number of connected clients – a socket input processing thread for each client (SSocketInputWorker), a socket output processing thread for each client (SocketOutputWorker), the server's main thread, which waits for incoming connections, and the CMessage dispatch thread (ServerMessageDispatcher), which sends CMessages to the ServerInputProcessor.

As noted and justified in section 4b of the design, the server is completely sequential, since the ServerMessageDispatcher takes CMessages put on its BlockingQueue by the SSocketInputWorkers and sends them to the ServerInputProcessor one-by-one. Thus, all data structures on the server, including conversation map and the SocketOutputWorker map (section 4di of the design), are accessed only by the ServerMessageDispatcher thread. Furthermore, communication between the ServerMessageDispatcher thread and the SSocketInputWorkers and between the ServerMessageDispather thread and the SocketOutputWorkers happens via thread-safe BlockingQueues.

There can be no deadlocks for the same reason described in the above section.

Testing Strategy

General Testing Strategy:
Our first goal is to perform extensive manual tests on all user-facing parts of the client (see below for examples). Our design also allows for extensive automated testing. Our design enables easy automated testing of all message datatypes and message serialization and deserialization, which constitutes the bulk of both client and server logic. We can perform end-to-end tests of the server by either launching the server in regular form and using a mock client (telnet) to send text messages to the server and analyze the responses, or by launching the server with mocked Sockets that have some preloaded data in their input streams and whose output streams we can read. We can perform some automated end-to-end testing of the client by mocking up ConversationListeners and MainListeners (rather than using ConversationWindow and MainWindow, which have many GUI components that are difficult to automatically examine).

Possible Manual Tests:

- **Client.** *"The client is a program that opens a network connection with the IM server at a specified IP address and port number. The client should have a way of specifying the server IP, port, and a username. Once the connection is open, the client program presents a graphical user interface for performing the interactions listed below."*
    - Opening a program should give choices of Server IP, port, and username
    - Once those are filled out and submitted, Chat list GUI should appear
- **Server.** *"The server is a program that accepts connections from clients. A server should be able to maintain a large number of open client connections (limited only by the number of free ports), and clients should be able to connect and disconnect as they please. The server also has to verify that client usernames are unique and handle collisions gracefully. The server is responsible for managing the state of both clients and conversations."*
    - Check whether a user's online/offline status changes when closing the client and opening it back up again (and claiming the same handle as before).
    - Check what happens when client tries to claim an unavailable handle
- **Conversations.** *"A conversation is an interactive text-exchange session between some number of clients, and is the ultimate purpose of the IM system. The exact nature of a conversation is not specified (although the hints section details a couple of possibilities), except to say that it allows clients to send text messages to each other. Messaging in a conversation should be instantaneous, in the sense that incoming messages should be displayed immediately, not held until the recipient requests them. You should visually separate messages of different conversations (e.g., into distinct windows, tabs, panes, etc)."*
    - Multiple conversations on one screen visibly separated
    - Ability to add people to conversation
    - Check that message sent appears on both sender's/receiver's screen
    - Check that message sends to all people in multi chat

    - Check that when multiple conversations present on the server, the server forwards messages to the correct conversations
    - Check when that multiple clients opened on the same computer, the server sends messages to the right client
- **Client/server interaction.** *"A client and server interact by exchanging messages in a protocol of your devising — the protocol is not specified. Using this protocol, the user interface presented by the client should: Provide a facility for seeing which users are currently logged in; Provide a facility for creating, joining and leaving conversations; Allow the user to participate in multiple conversations simultaneously; Provide a history of all the messages within a conversation for as long as the client is in that conversation."*
    - Check that everyone who is supposed to be online is online
    - Check that when a person leaves a conversation, the other people in the conversation are notified.
- **No authentication.** *"In a production system, logging in as a client would require some form of password authentication. For simplicity, this IM system will not use authentication, meaning that anyone can log in as a client and claim any username they choose."*
    - Check that server allows clients to choose arbitrary usernames as long as they have not been claimed by another client

Project Amendment

It is helpful to consider the two paragraphs at the beginning of this document, which note that we use the chat room model for conversations. Thus, a user sees chat history only when they reenter a room they were previously in. More specifically, since, as noted in the initial two paragraphs, a user cannot add himself to a room, a user only sees chat history when he is added back to a conversation he has left by a communicant who remained in the conversation. In particular, this design means that if a user exits a chat A with a particular user and then starts a new chat B with this same user, he will not see the messages from A in the new window for B, since he is now in a new chat room. Put yet another way, two conversations with the same list of communicants are never consolidated.

UI Drawings

Please see the file ui-sketches.pdf in the docs folder of our repository.

Testing Report

Manual Testing:

- IP/Port
    - Check error made when given invalid id + port
    - Check error made when given valid ip and invalid port
    - Check for error regarding spacing between colon and port

    - Check that only space ignored when before the port and after the colon
    - Check proceed to handle creation when proper ip and port are provided
    - Does not accept empty input
- Username submission
    - Accepts alphanumeric characters properly, moves on to show online user list GUI
    - Does not allow input with non alpha numeric characters or spaces
    - Does not accept empty input
    - If spaces are included at the end, the spaces are cut off and the input is accepted
    - Does not allow to reuse a handle already used
    - When person with said already used handle logs out, I can now use that handle again
    - There is no authentication required to select an available user name
    - There is no max length for a username
- Chatroom list
    - When I am logged in, shows my username first on the log in list
    - When someone logs in, the list auto updates
    - When someone signs out, they disappear from the online list
    - When user exits out of the online user list, he or she is logged off
    - When user exits online user list, all of his other chat windows are closed and chat windows are updated for other users
    - Resizing the chatroom list will allow for wider or taller screens, but there is always a minimum of three users wide for the screen
    - If a users name is too long for the default area, the rest of the name is cut off with "…"
- Chatroom window
    - Can create chat room with myself, displays my message only once unlike iMessage due to the chat room implementation we selected rather than the two person chat model of iMessage
    - Users are notified when others enter or exit the conversation
    - When I create a chatroom with someone else online, chat screen shows up on both of our windows
    - When user exits out of the chat room, chat room list auto updates
    - When more than two users are present in a conversation, messages show up to all users
    - Users can type in "/" in any situation and it shall show up in the chat screen even though we use it as a separator
    - When someone adds a user who was previously included in the conversation, the conversation history for the user added will show up
        - This chat history will not include messages sent while the user was in the chat

        - This is the only way to see your chat history, every conversation started with a person creates a new chatroom
        - Once both people exit a conversation, the conversation history cannot be recovered
    - One user can successfully create multiple chat rooms with the same user (John can start three separate chatrooms with Larry)
        - Said rooms are thread safe, the messages sent by John will only be seen in the corresponding window of Larry
        - This serves as our concurrency test – John's message sent in one chat window will only be sent to windows with the same corresponding conversation id, not every other user John has a conversation with
    - Adding Users
        - Can successfully add multiple users if comma separated
        - Can successfully add a single user
        - If user tries to add a user that is not online, nothing happens
        - If user tries to add someone already in the conversation, nothing happens
        - If user tries to add someone who logged off, and is hence no longer online, nothing happens
        - If the user tries to add himself of herself, nothing happens
        - If user tries to add multiple usernames, one of which is a proper name, the proper users are added to the conversation
            - If the proper user is already in the conversation, nothing happens
            - Once again, if spaces are extra and included before or after a username, the input is accepted stripped of the spaces
- Exiting
    - Closing the online user list quits the program
    - Closing from the mac menu quits the program
- Server
    - To test the server in isolation, we utilized telnet to send messages as given in the grammar to the server and we connected multiple clients as well

Automated Testing

- Messages
    - We tested the serialization and deserialization of CMessages and SMessages (CMessageImplsTest and SMessageImplsTest)
    - We tested toString() methods for the serialization component
    - We tested the CMessageImpls.deserialize() and the SMessageImpls.deserialize() for the deserialization component
    - CMessages

        - Tested the serialization and deserialization for the following actions
            - GetId
            - RegisterHandle
            - GetUsers
    - SMessages
        - Tested serialization and deserialization for the following actions
            - NormalAction: Text Message
            - NormalAction: Exit Conv
            - NormalAction: Add User
            - AvailabilityInfo: Online
            - AvailabilityInfo: Offline
            - HandleClaimed
            - BadHandle
            - ReturnID
- Clients (ClientInputProcessorTest)
    - We tested that the online user list would update when a user is added
    - We tested that the the returnId() method returns the proper id
    - We tested that a Normal Action of ADD_USER added the specific user
    - We tested that a Normal Action of TEXT_MESSAGE sent the proper message, including all possible characters on the keyboard (included both slashes and alphanumeric keys)
    - We tested that a EXIT_CONV NormalAction exits a conversation appropriately
    - We tested that badHandle(), AvailabilityInfo(), and ClaimedHandle() worked properly
    - We did not need any tests to check that the system throws the appropriate errors because the system seeks to gracefully handle errors from client input by either stripping spaces or not accepting the input and ignoring it