guichat: Design

Note on Scale

The fact that our chat server is intended for small-scale use has several consequences in terms of design. First, we will be running the server as a single process with multiple threads. Second, **we will not protect our server and client from malicious network users**. Our product is specified to work if all users are running the client we provide with the server we provide, but we make no guarantees about the behavior of either if there is a rogue user sending bad messages around on telnet.

Conversations

Description

We define a conversation as a connection between any positive number of users who are connected to the IM server. Any user connected to the server can

- create a conversation, assigning the conversation a name which is distinct from all already-existing conversations.
- connect to a conversation by specifying its name, if he is not already in that conversation.

Initially, a conversation contains only the user who created it. Upon creation, the user may specify a unique name for the conversation or have a unique name automatically generated by the server. Any user in a given conversation can

- send a message, which will be seen by all other users in the conversation
- invite another user to the conversation
- leave the conversation.

When user A invites user B to join a conversation, user B's client will allow him to accept or reject the invitation. If he accepts, the client will join the room in the same way that user A joined the room.

Implementation

Conversations are represented by instance of Conversation. The Conversation class, located in the server package, has the following (final) instance variables:

- users
  A set of Users that are currently in the conversation.
- name
  The name of the conversation represented by this Conversation (a string). This string will be globally unique among all conversations on the server, and will be the string corresponding to the conversation in the conversations map in IMServer.

Conversation contains the following methods, all of which are called by the associated methods in IMServer. For more detailed descriptions on usage, see the corresponding descriptions of methods in the Server section below.

- sendMessage(User u, String m, int messageId)
  Sends message m to each User in this.users with sender name u and ID messageId.
- add(User u)
  If this.users does not already contain u, adds u to this.users and sends an added to conversation message to every other client in this conversation, and sends to u a entered conversation message. Returns whether or not this.users changed as a result of the call to add.
- remove(User u)
  If this.users contains u, removes u from this.users and sends a removed from conversation message to every other client in this conversation. Returns whether or not this.users changed as a result of the call to remove.
- contains(User u)
  Returns whether or not this.users contains u.
- isEmpty()
  Returns whether or not this.users is empty. Note: all empty conversations are immediately removed from IMServer.conversations.
- getName()
  Accessor method for this.name.
- toString() (Override)
  Returns the conversation name and the names of the Users in this.users according to ENTERED_CONV_CONTENT in the server-to-client network described in the Network Protocol section below.

Concurrency

Conversation is thread-safe because all methods that access or modify this.users are synchronized (including sendMessage, add, remove, contains, isEmpty, and toString). This prevents concurrent modification of this.users. Since this.name is immutable, Conversation is thread-safe.

Testing
Conversation object contains the name of the conversation and the users who are in it. The toString method gives a representation of the conversation. So we can use the toString method to test whether the state of the conversation is consistent.

- Constructors - We'll test both constructors by passing in name or (name, username) and see if the toString() method gives the right state
- add method - We set up one or more TestUtility(s) (See below), add users to it, and call add(user) method on a conversation and see
    - The user is added to the conversation
    - The conversation is added to the user by using contains method and toString()
    - The user gets a list of other users in the conversation (Through testClient)
    - The other users in the conversation gets a notification for the new user
        - We will also check if the behavior is expected when the user is already in the conversation.
- remove mothod - We set up one or more TestUtility(s) (See below), add users to it, and call add(user) method on a conversation and see
    - The user is removed from the conversation
    - The conversation is removed to the user by using contains method and toString()
    - The other users in the conversation gets a notification for the leaving user
        - We will also check if the behavior is expected when the user does not exist in the conversation.
- contains method - We set up a Conversation with some users, and check if contains returns true or false depending on whether the User is already in. We also check if adding and removing users correctly changes the return value for some user.
- isEmpty Method - We set up a conversation, check if it ismpty, add users and remove them to see if isEmpty behaves in the right way
- hashCode - We set up conversations with same name to see if hashCodes are equal, and different names to see if they are unequal. Also adding or removing users from conversations should not change the hashCode.
- toString - this already has been tested

Server

Description
The server has a main thread that listens to a specific socket, corresponding to a specific port and IP address, for user connections. The server also keeps a thread for each client, which handles the network requests from that client.

Implementation

The server is implemented in the IMServer class, which implements Runnable and is located in the server package. Instances of IMServer keep track of the following (final) instance variables:

- serverSocket
  The ServerSocket on which the server listens for user connections.
- users
  A map of from String to User that keeps track of clients currently connected to the server. The key for a given User is the user's username.
- conversations
  A map from String to Conversation that keeps track of active conversations. The key for a given Conversation is the conversation's name.

IMServer also contains the following methods, which are called from the run method of User when network messages are received from a client, all of which return a boolean indicating success, with the exception of disconnectUser:

- sendMessage(String username, String convName, int messageId, String m)
  Sends the message m to all clients in the conversation specified by convName with sender name username and ID messageId. The variable messageId should be unique among all messages sent by this client to the conversation associated with convName. (In practice, this number will be globally unique among *all* messages this client has sent, since this is easier to implement.) Returns whether or not the message was properly sent. Note: since messages will be logged on the client side, there is no need for the server to store messages at all.
- newConversation(String username, String convName)
  If convName is non-null and non-empty and there is no Conversation associated with convName in this.conversations, creates a new Conversation with name convName containing only a User with the given username and adds it to this.conversations. If convName is null or empty, creates a new Conversation associated with a unique String containing only a User with the given username and adds it to this.conversations. Sends a new conversation receipt message to the client with the given username with information on whether or not the conversation was successfully created. Returns whether or not the conversation was successfully created.
- addToConversation(String username, String convName)
  If there is a Conversation associated with convName, adds the User corresponding to username to that Conversation and sends an added to conversation message to every other client in this conversation, and sends to the given User a entered conversation message. If there is no Conversation associated with convName,

sends a removed from conversation message to the client associated with the given username. Returns whether or not the User was successfully added.

- removeFromConversation(String username, String convName)
  If there is a Conversation associated with convName, removes the User with the given username from the Conversation, sends a removed from conversation message to every other User in the Conversation, and removes the Conversation from this.conversations if the Conversation contains no Users after removing the User with the given username. Returns whether or not the User was successfully removed.
- disconnectUser(String username)
  Removes the User with the given username from each of his conversations, sending a removed from conversation message to every other user in the conversation, using remove, and removing empty conversations from this.conversations.
  Removes the given User from this.users. Sends a disconnected message to all clients.
- connectUser(User u)
  Adds u to this.users if u is non-null. Sends an initial users list message to u and a connected message to all other Users in this.users if u is not null and not already in users. Sends a disconnected message to u if u is not null and not already in this.users. Returns whether or not u was successfully connected.

We use the publish-subscribe pattern. When a user enters, leaves, or sends a message to a conversation, we fire a notification to all the other users in the conversation, which allows their clients to update the displayed list of participants or message history.

The server also contains a method run which makes the server listen for user connections on the calling thread. When a user connection is received over serverSocket, run creates and starts an instance of User, which is a subclass of Thread. The User then waits for a connect message. When the connect message is received, the User tries to add itself to server.users (where server is the IMServer passed into the User constructor) by using connectUser (which is thread-safe), succeeding if the provided username is available.

Concurrency

By getting a lock on the necessary instance variable for all sets of instructions that should be atomic, each method in IMServer avoids concurrency issues:

- sendMessage is thread-safe because it synchronizes on
  - this.users while checking if the username is in this.users and getting the corresponding User,
  - this.conversations while checking if the conversation name is in this.conversations and getting the corresponding Conversation,

    - the Conversation while checking if it contains the given User and sending the given message to all Users in it.
- newConversation is thread-safe because it synchronizes on
    - this.users while checking if the username is in this.users and getting the corresponding User,
    - this.conversations while checking if the specified or generated conversation name is in this.conversations and adding the new Conversation to this.conversations, in order to prevent multiple users from creating new conversations of the same name simultaneously.
- addToConversation is thread-safe because it synchronizes on
    - this.users while checking if the username is in this.users and getting the corresponding User,
    - this.conversations while getting the specified Conversation. (It later checks if the Conversation returned is null.)
    - the Conversation while checking if it contains the given User and, if it doesn't, adding the given User.
- removeFromConversation is thread-safe because it synchronizes on
    - this.users while checking if the username is in this.users and getting the corresponding User,
    - this.conversations while getting the specified Conversation. (It later checks if the Conversation returned is null.)
    - the Conversation while
        - checking if it contains the given User,
        - if it does, while removing the given User,
        - while checking if the Conversation is empty and removing it from this.cconversations if it is.
            - When this happens, we synchronize on this.conversations during removal.
- disconnectUser is thread-safe because it synchronizes on this.users while checking if the username is in this.users, removing the corresponding User if it is, and getting an array of Users to which to send a disconnected message to indicate that the given User has disconnected.
- connectUser is thread-safe because it synchronizes on this.users while checking if the username is in this.users, adding the corresponding User if it isn't, and getting an array of Users to which to send a connected message to indicate that the given User has connected.

Testing

The server has a set of users, conversations. So we can check if these fields are equal to the expected values to see if the state of the server is as expected.

- Constructor will be tested by making an instance, and checking if the user set and

conversation set s empty.

- run method - We'll start the run method, and set us TestUtility to see if we can communicate with the server.
- sendMessage method - We'll connect a few testUtilities as users. Then call sendMessage() to see if the TestUtility registers the right responses in its socket. To test for concurrency issues, we'll set up several threads and call sendMessage from all of them running simultaneously, and verify if the results are correct.
    - We'll also check some boundary cases, like when the username/conversation name is invalid or does not exist, or the message contains invalid characters.
- newConversation method - We'll send messages from the TestUtility to set up new conversation, then check
    - If the set of conversations in the IMServer gets updated.
    - The conversation has the user.
    - The user gets the right messages from the server
        - We'll also take care of the case when the conversation already exists, or if the name of the conversation/user is invalid.
        - Concurrency will be verified by using multiple threads calling the same method.
- addToConversation method - This will be tested in a similar way as before. We set up a testUtility, and set up users and conversations, then add an user to the conversation (with invalid username/conversation as well) and see if the internal representation changes in the right way and the correct messages are received.
- removeFromConversation method - We Will follow up the same testing strategy in the previous case with some removals. Then we will add the person back and test if everythig holds the way it should. Concurrency will also be tested in all cases, by constructing multiple thread which simultaneously call these methods with same username/conversation. One very importnt point is that if all users remove themselves from a conversation, the conversation should be deleted. We'll specifically test for this.
- connect/disconnectUser method - As in previous cases, we'll set up TestUtilitiy(ies) and connect and disconnect users (also check for invalid cases or the cases when user does not exist when disconnecting). In either case, every other user gets a notification. In the case of connect, the connecting user gets a list of other connected people. These messages will be verified through TestUtility, and the internal representations, userSet, will be verified too. Concurrency will be checked through multiple threads calling connect/disconnect on different users.

User

Description
Instances of User are used by the server to keep track of connections to clients. Note that we frequently use User to refer to the client represented by an instance of User.

Implementation
User is a subclass of Thread, is located in the server package, and has the following instance variables:

- socket
  The socket over which the server communicates with this client.
- name
  A String containing the client's username. username is non-null, non-empty, contains at most 256 characters, and contains no new-line characters.
- conversations
  A set of Strings representing the names of all the conversations that this client is in.
- out
  A PrintWriter the writes to this.socket.
- in
  A BufferedReader that reads from this.socket.
- server
  The instance of IMServer that created this User, and whose users map this will add itself to when the client specifies a valid username using a connect message.

All instance variables except this.name are final. (this.name is not final because it is specified after initialization, upon receipt of a connect message.)

The run method of User listens to the socket, parses network messages when they are received, and either calls the corresponding IMServer method or throws an InterruptedException to disconnect upon receiving a disconnect message. run uses the private helper methods handleConnection, handleRequest, im, newConv, addToConv, enterConv, and exitConv to do this. The private method removeFromAllConversations removes the User from each of his conversations, sending removed from conversation messages to other users in each conversation, and is called when an Exception is raised. User also has the methods

- send(String s)
  Sends s to the client by writing to this.socket.
- addConversation(Conversation conv)
  Adds conv to this.conversations and sends an entered conversation message to the client corresponding to this. (addConversation is called by conv with itself as a parameter. conv deals with sending added to conversation messages to its Users.)

Returns whether or not conv was added, i.e., whether or not this.conversations has changed as a result of the call to addConversation.

- removeConversation(Conversation conv)
  Removes conv from this.conversations. (removeConversation is called by conv with itself as a parameter. conv deals with sending removed from conversation messages to its Users.) Returns whether or not conv was added, i.e., whether or not this.conversations has changed as a result of the call to addConversation.
- getUsername()
  Accessor method for this.name.

User uses following methods to format and send the network messages to the client using send.

- sendInitUsersListMessage(Object[] users)
- sendEnteredConvMessage(Conversation conv)
- sendAddedToConvMessage(User u, String convName)
- sendRemovedFromConvMessage(User u, String convName)
- sendConnectedMessage(User u)
- sendDisconnectedMessage(User u)
- sendIMMessage(User u, String m, int messageId, String convName)
- sendNewConvReceiptMessage(boolean success, String convName)

Concurrency

User avoids concurrency issues because each method that modifies or accesses this.conversations is synchronized. Server is thread-safe, and all other instance variables are never modified. Moreover, this.conversations is private and not exposed.

Testing

The User class will be tested primarily through the TestUtility, because most of its behavior is through the network. We'll connect a testUtility as a test client, set the user's socket through the constructor properly. Then we'll test each individual part through the network protocol

- send method - We'll send some strings to the test client and check if the test client's read method returns the right string.
- run method - The main thing to test for this method is whether the thread is correctly listening to new input from the test client. One important thing to check is the way user gets its name assigned. According to our design, the user object will be created and the connection will be established before the name is determined. The user should be unable to create or join conversations before the name is assigned. Also we'll interrupt the thread and see if the exception is correctly handled. One important thing to check is the way.

- handleRequest method - We'll send each type of messages from the test client with valid and invalid arguments (according to the network protocol) and see if the behavior is consistent internally (field values) and externally (network messages).
- im
- newConv
- addToConv
- enterConv
- exitConv
- removeFromAllConversations
    - Each of the above will be tested seperately by calling the respective functions with appropriate arguments (array of strings according to network protocol)
- addConversation
- removeConversation
- sendInitUsersListMessage
- sendEnteredConvMessage
- sendAddedToConvMessage
- sendRemovedFromConvMessage
- sendConnectedMessage
- sendDisconnectedMessage
- sendIMMessage
- sendNewConvReceiptMessage
    - In each of the above methods, we'll call them with valid and invalid arguments. The job of each of them is to convert them into a suitable network message and send to the client. Using the testClient we'll be able to verify that the messages are correct.
- equals method - We'll test equals method with objects that are instances of USer or not, and verify that the equality is based on the username. Adding conversations or other things, should not change the equality condition.
- hashCode method - We'l verify that the hashcode is based on the username only too.
- isInConversation method - This is easy to test, we'll add some conversation and remove it. And check if the behavior of this method is consistent.
- getUserName - This method is also easy to check, we'll set up some user name using the test client, and see if the correct username is returned.

Network Protocol

There are seven types of network messages sent from the client to the server:

1. CONNECT
   A connect message is sent as a login request to the server.
2. IM
   An IM message is sent as a request to send a given message to all users in a given conversation.
3. NEW_CONV
   A new conversation message is sent as a request to create a new conversation. The user may specify a name to request a conversation with that name.
4. ADD_TO_CONV
   An add to conversation message is sent as a request to add a given user to a given conversation.
5. ENTER_CONV
   An enter conversation message is sent as a request to be added to a given conversation.
6. EXIT_CONV
   An enter conversation message is sent as a request to exit a given conversation.
7. DISCONNECT
   A disconnect message is sent as a logoff request from the server.

The client-to-server network protocol grammar is as follows:

```
MESSAGE = MESSAGE_TYPE TAB MESSAGE_CONTENT
MESSAGE_TYPE = CONNECT | IM | NEW_CONV | ADD_TO_CONV | ENTER_CONV |
        EXIT_CONV | DISCONNECT
MESSAGE_CONTENT = CONNECT_CONTENT | IM_CONTENT | NEW_CONV_CONTENT
        | ADD_TO_CONV_CONTENT | ENTER_CONV_CONTENT | EXIT_CONV_CONTENT
        | DISCONNECT_CONTENT

CONNECT_CONTENT = USERNAME
IM_CONTENT = CONV_NAME TAB IM_ID TAB MESSAGE_TEXT
NEW_CONV_CONTENT = NEW_CONV_NAME
ADD_TO_CONV_CONTENT = USERNAME TAB CONV_NAME
ENTER_CONV_CONTENT = CONV_NAME
EXIT_CONV_CONTENT = CONV_NAME
DISCONNECT_CONTENT = NOTHING

USERNAME = [^\t\n]{1,256}
NEW_CONV_NAME = [^\t\n]{0,256}
CONV_NAME = [^\t\n]{1,256}
IM_ID = [0-9]{1,9}
MESSAGE_TEXT = [^\t\n]{1,512}
```

TAB = \t
NOTHING = .{0}

CONNECT = 0
IM = 1
NEW_CONV = 2
ADD_TO_CONV = 3
ENTER_CONV = 4
EXIT_CONV = 5
DISCONNECT = 6

There are eight types of network messages sent from the server to the client:

1. INIT_USERS_LIST
   An initial users list message is sent upon connection with the server. The message specifies all other users that are currently logged in.
2. IM
   An IM message is used to send a message to all users in a conversation.
3. NEW_CONV_RECEIPT
   A new conversation receipt message confirms that a given conversation was properly created and that the client was added to the new conversation.
4. ADDED_TO_CONV
   An added to conversation message notifies the client that a user has been added to a conversation that the client is in.
5. ENTERED_CONV
   An entered conversation message notifies a user that he has entered a conversation. It specifies the users in the conversation, including the user entering the conversation..
6. REMOVED_FROM_CONV
   A removed from conversation message notifies the client either that a user has been removed from a given conversation or that the server failed to add the client to the given conversation.
7. CONNECTED
   A connected message notifies the client that a new user has connected to the server.
8. DISCONNECTED
   A disconnected message notifies the client that a given user has disconnected from the server.

The server-to-client network protocol grammar is as follows. Note that some pieces of the grammar may be different from their counterparts with the same name in the grammar for

the client-to-server network protocol grammar; for example, IM_CONTENT in this grammar contains the sending user's username:

```
MESSAGE = MESSAGE_TYPE TAB MESSAGE_CONTENT
MESSAGE_TYPE = INIT_USERS_LIST | IM | NEW_CONV_RECEIPT | ADDED_TO_CONV
        | ENTERED_CONV | REMOVED_FROM_CONV | CONNECTED | DISCONNECTED
MESSAGE_CONTENT = INIT_USERS_LIST_CONTENT | IM_CONTENT |
        NEW_CONV_RECEIPT_CONTENT | ADDED_TO_CONV_CONTENT |
        ENTERED_CONV_CONTENT | REMOVED_FROM_CONV_CONTENT |
        CONNECTED_CONTENT | DISCONNECTED_CONTENT

INIT_USERS_LIST_CONTENT = (USERNAME TAB)* USERNAME
IM_CONTENT = USERNAME TAB CONV_NAME TAB IM_ID TAB MESSAGE_TEXT
NEW_CONV_RECEIPT_CONTENT = (SUCCESS | FAILURE) TAB CONV_NAME
ADDED_TO_CONV_CONTENT = USERNAME TAB CONV_NAME
ENTERED_CONV_CONTENT = CONV_NAME TAB (USERNAME TAB)* USERNAME
REMOVED_FROM_CONV_CONTENT = USERNAME TAB CONV_NAME
CONNECTED_CONTENT = USERNAME
DISCONNECTED_CONTENT = USERNAME

USERNAME = [^\t\n]{1,256}
CONV_NAME = [^\t\n]{256}
IM_ID = [0-9]{1,9}
MESSAGE_TEXT = [^\t\n]{1,512}
TAB = \t
FAILURE = 0
SUCCESS = 1

INIT_USERS_LIST = 0
IM = 1
NEW_CONV_RECEIPT = 2
ADDED_TO_CONV = 3
ENTERED_CONV = 4
REMOVED_FROM_CONV = 5
CONNECTED = 6
DISCONNECTED = 7
```

Network protocol constants and regular expressions are contained in the NetworkConstants class, in the main package.

Client

Description

The client will provide a GUI for users to connect to and interact with a running IMServer. Upon launch, the client will prompt for the address and port of the server and attempt to connect. Upon connection, the client will spawn a tabbed window which displays the other currently connected clients (by username) and allows the user to start or join conversations. Once the user joins a conversation, the client will open a new tab in which it displays all messages it receives and allows the user to send his own messages. The user may leave a conversation by closing its tab.

Implementation

Data Hierarchy

The Client will have the following data fields:

- Set<String> otherUsers
  A set of usernames of the other clients connected to the same server. This will be initialized upon receipt of INIT_USERS_LIST and updated upon receipt of CONNECTED and DISCONNECTED messages.
- Set<ConversationTab> conversations
  A set of the conversations the client is engaged in.
-

ConversationTab is a subclass of JComponent. In addition to displaying a GUI hierarchy (as drawn out in the UI hand-sketch), it will store the following data fields relating to the conversation:

- Set<String> users
  The set of other users involved in the conversation. Will be updated upon receipt of an ADDED_TO_CONV message or a REMOVED_FROM_CONV message.
- ConversationHistory conversationHistory
  The history of the conversation as far as the client has been present to hear. Will be updated upon receipt of an IM message. See the subsection on conversation history display below.

Conversation History Display

The conversation history display is controlled by an instance of ConversationHistory, which has references to the following GUI elements as instance variables:

- receivedMessages
  Displays all messages which have been received.
- pendingMessages
  Displays all chat messages which are still pending (i.e., have not been sent back to us by the server).

receivedMessages is displayed at the top of the conversation history display. Next, the pending messages will be listed in order, with a "[P]" before each one.

OutgoingChatMessage is an immutable class with two fields:

- String messageText
- int messageId
  An integer unique among messages sent by the client. messageId allows us to communicate clearly with the server about exactly which message we're talking about.

When we receive a new chat message from the server, we check whether it is one of our pending messages, and, if it is, remove it from pendingMessages. Then, we append the message (with appropriate byline) to receivedMessages.

Communication with Server

The client will keep a BlockingQueue<MessageToServer> messagesToServer of messages it wishes to send to the server. There will be a single consumer thread taking messages from this queue and sending them off to the server.

MessageToServer will have two fields:

- String messageText
- volatile boolean cancelled = false

The consumer will only send a message off to the server if its cancelled field is false. In this way, the client can ensure that closing a conversation tab cancels all pending outgoing messages for that conversation.

Similarly, the client will keep a BlockingQueue<MessageFromServer> messagesFromServer of messages it has received from the server. A consumer thread will create a SwingWorker for each server message.

MessageFromServer is just a wrapper around String. For purposes of extensibility later, and for parallism with MessageToServer, we'll tolerate the overhead.

The Swing event dispatch thread will take care of both GUI events (as triggered by ActionListeners) and server messages (since the handling of each message from the server will be done in the corresponding SwingWorker's done() method, which runs in the event dispatch thread). Thus, all GUI updates occur within the event dispatch thread, so concurrent access to GUI elements is not a problem.

Handling of Server Messages

As we said above, there will be a consumer thread taking server messages from the queue (messagesFromServer) and creating an appropriate swingWorker to handle them. Some intricate logic will have to happen here, since the order in which we receive messages from the server may differ from the order in which the server sent them. There will be a method handleServerMessage(MessageFromServer message) which does this. It will have a helper method for each type of server-to-client method mentioned in the network protocol:

- handleInitUsersList(String m) -- Initializes the list of other connected users according to the message m. (If the client doesn't expect to be receiving an INIT_USERS_LIST message right now, ignore it and complain passively in the status bar.)
- handleIM(String m) -- (parses the IM message and) appends the new message to the appropriate conversation. If the new message is one of the currently pending messages, unpend it. (If we didn't think we belonged to the conversation this IM came from, ignore it and complain passively in the status bar.)
- handleNewConvReceipt(String m) -- If the new conversation creation was reported successful, open a new tab corresponding to the new conversation (which the user has been added to). Initialize the list of other users in the conversation to contain just me. If the new conversation creation was reported a failure, spawn an alert window letting the user know. (If we thought the given conversation already existed and we were part of it, ignore the message and complain.)
- handleAddedToConv(String m) -- Update the list of participants in the relevant conversation to include the new user. If the new user was already there or we didn't think we belonged to that conversation, ignore the message.
- handleEnteredConv(String m) -- Opens a new conversation as specified by the ENTERED_CONV message. Initializes the list of other users in the conversation to that given in the message m. If we thought we were already part of the given conversation, ignore the message.
- handleRemovedFromConv(String m) -- Update the list of participants in the relevant conversation by removing the specified user. If we didn't think that user was part of the conversation in the first place, ignore the message.
- handleConnected(String m) -- Update the list of other users connected to the server. If we thought that user was already connected, ignore the message.
- handleDisconnected(String m) -- Update the list of other users connected to the

server. If we thought that user wasn't connected in the first place, ignore the message.

Generally, our strategy is to discard server messages that don't make sense with our current view of the world. So if the server tells us that a message has been received for a conversation that we don't think we're in, we will do nothing (except maybe complain passively in the status bar). Our philosophy is that, if network communication causes us to lose information, that's the way the cookie crumbles. We would run into fundamental problems if we tried to "fix" the issue of network latency.
However, we should note that network latency at its worst can cause unfaithful representations of reality on the part of the client. For example, if Ben creates a conversation, then later Vinay connects and quickly disconnects, it is possible that Ben will receive the REMOVED_FROM_CONV message before he receives the ADDED_TO_CONV message. Thus, on Ben's client, it will appear as if Vinay is in the conversation even though in reality Vinay is not in the conversation. Lots of commercial IM clients such as Gmail and Facebook also suffer from this problem. If a user wants to refresh the list of logged-in users, he can log out and log back in. (Later, if we have time, we may implement a "refresh" feature that does not require disconnect and reconnect.)

Non-Obvious Parts of GUI Interaction

- When the user closes a tab corresponding to a conversation, we do the following: Leave the conversation, cancel all pending outgoing messages which have not yet been taken out of the queue MessagesToServer, and close the corresponding tab in GUI.

Concurrency
All modifications of GUI elements take place on the event dispatch thread, either by being called in the actionPerformed method of an EventListener in response to user input, or by being called in the done method of a SwingWorker that is dispatched when the client receives a network message from the server. This prevents concurrent modification of GUI elements.

Since OutgoingChatMessage is immutable, it is thread-safe. Methods that access and modify ConversationTab.users synchronize on ConversationTab.users to make the modfication atomic and prevent concurrent modification. ConversationHistory objects contain GUI elements that are modified only on the event dispatch thread.

Testing
We will of course run manual final tests using our server with our client, which will give a general check of typical behavior. In addition, we will use the TestUtility to run rigorous, automated tests of edge cases and unusual stuff. Namely:

- Make sure INIT_USERS_LIST message is handled properly
- Test behavior upon receiving IM message for a conversation that we are in
- Test behavior upon receiving IM message for a conversation we don't think we're in
- Test behavior upon receiving NEW_CONV_RECEIPT failure
- Test behavior upon receiving NEW_CONV_RECEIPT success, even if we didn't request the room
- Test ADDED_TO_CONV on someone we already thought was in the conversation
- Test REMOVED_FROM_CONV on someone who we didn't think was in the conversation
- Test ENTERED_CONV and initialization of other users list
- Test ENTERED_CONV on a room that we're already in (we should ignore and complain)
- Test CONNECTED on someone we thought was already connect
- Test DISCONNECTED on someone we thought was not connected in the first place

For debugging purposes, we will temporarily have a sentinel message which delays before sending out to server (like the asterisks in Pset4). Thus:

- Test closing a conversation when there are several pending messages that have not yet been sent (they should be cancelled, and the TestUtility can check to make sure they are not sent).

Also test the connect screen:

- Failure to connect to server
- User cancelling connection to server using "Cancel" button if the connection is taking too long
- Popups for especially problematic error messages, e.g., "Server disconnected unexpectedly"
- Failure to reserve a nickname because it is taken

TestUtility

Description

We'll use the TestUtility class as a mock version of the client or server to test different parts of the server or of the client. We can connect it using any socket to an User object and send messages to and read responses from the server.

Implementation

TestUtility's constructor takes a port number. The constructor initiates a connection with the server on that port number. TestUtility has the following methods:

- send(String s)
  Sends a String to the server.
- read()

Reads what is currently in the socket, sent from the server.