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 non-negative 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.
- create a two-way conversation with another user who is currently logged in. (Once created, users can leave and more users can be added.)

Initially, a conversation contains only the user who created it, or the two users involved in a two-way conversation. Upon creation, the user may specify a unique name for the conversation or have a unique name automatically generated by the server. When creating a two-way conversation, the server always automatically generates a name for the conversation. 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.

Implementation

Conversations in guichat are represented by instances 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 to every User in this.users a message with the given sender, message text, and message ID.
- add(User u)
  If u is non-null and this.users does not already contain u, adds u to this.users, 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 u is non-null and 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.
- getName()
  Accessor method for this.name.
- toArray()
  Returns an array representation of this.users.
- 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, toArray, 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 and toString
  We test constructors and toString by passing in name, (name, user), or (name, user, user), and see if the toString() method gives the right state.
- add
  We set up one or more instances of TestClient (see below), add users to it, call add(user) method on a conversation and check that
    - 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 get a notification for the new user
        - We will also check if the behavior is expected when the user is already in the conversation.
- remove
  We set up one or more instances of TestClient (see below), add users to it, call add(user) method on a conversation and check that
    - 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
  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.
- 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
  The tests above for the constructor also test toString.

Server

Description

The server 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 name.
- conversations
  A map from String to Conversation that keeps track of all conversations, active and empty. 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. When username is null or is not contained in this.users, the method returns false:

- 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 the 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 the User with the given username and adds it to this.conversations. Sends an entered conversation message to the client. 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 sends to the given User a entered conversation

message. If the User was not already in the conversation, sends an added to conversation message to every other client in this conversation. 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)
  If u is null, returns false. If u.username is not null or the empty string and is already in this.users, sends a disconnected message to u and returns false. If u.username is null or the empty string, generates a unique username and sets u.username. If u.username is null or the empty string or is not already in this.users, adds u to this.users, sends an initial users list message to u and a connected message to all other Users in this.users.
- retrieveParticipants(String username, String convName)
  If there is a conversation associated with convName, sends a participants message to the client specified by username. Returns whether or not a participants message was successfully generated.
- twoWayConv(String username1, String username2)
  If username1 and username2 are non-null, refer to Users in this.users, and are not equal, creates a new Conversation with a unique, auto-generated name containing only the Users with names username1 and username2 and adds it to this.conversations. Sends an entered conversation message to each user.

The private method userByUsername is used to look up a User in this.users by username. The method close closes the IMServer by closing serverSocket and interrupting all Users in this.users.

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.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.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.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.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.
- retrieveParticipants is thread-safe because it synchronizes on this.conversations while getting the specified conversation.
- userByUsername is thread-safe because it synchronizes on this.users while checking if the username is in this.users and getting the corresponding User.
- close is thread-safe because it synchronizes on this.users while getting a collection of the values of this.users.

Testing

To test the server, we use one or more instances of TestClient to connect to the server and make sure that the correct messages are received in response to a given sequence of client-to-server messages. This tests the corresponding methods in IMServer and the private helper methods for run in User, as well as the server performance as a whole. Since these methods each rely on the behavior of many components of the system, they cannot be tested individually. However, by testing certain sequences of messages to the server, we can unit test independent aspects of the server's behavior.

The methods used to test the server are located in ServerTest. For a full description of the tests, see the Testing Report document.

- 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 TestClient to see if we can communicate with the server.
- sendMessage method - We'll connect a few TestClients as users. Then call sendMessage() to see if the TestClient 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 TestClient 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 testClient, 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 TestClient(s) 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 TestClient, 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. When the User has been added to its server, name is non-null, non-empty, contains at most 256 characters, and contains no newline characters.
- conversations
  A set of Conversation objects, representing 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. Once this has been added to this.server, this.name should not be changed.)

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, exitConv, retrieveParticipants, and twoWayConv 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.
- setUsername(String s)
  Sets this.name to the given String. Should not be called when this has been added to server.users or on any thread other than this.

User uses following methods to format and send the network messages to the client using send. (For full descriptions of these methods, see the Javadocs in guichat/doc.)

- 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)
- sendParticipantsMessage(Object[] users)
- sendErrorMessage(String m)

Concurrency

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

Testing

The run method and its private helper methods (handleConnection, handleRequest, and the methods called in each case of network message from handleRequest) are tested via network messages in ServerTest. The methods that are be tested independently have tests located in UserTest. For a full description of the tests run for each method, see the Testing Report.

Many methods in User are tested by using one or more instances of TestClient, because most of its behavior is through the network. We'll connect a TestClient as a test client, set the user's socket through the constructor properly. Then we'll test each individual part through the network protocol.

- In each of the following methods, we 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.
    - addConversation
    - removeConversation
    - sendInitUsersListMessage
    - sendEnteredConvMessage
    - sendAddedToConvMessage
    - sendRemovedFromConvMessage
    - sendConnectedMessage
    - sendDisconnectedMessage
    - sendIMMessage
    - sendNewConvReceiptMessage

- 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'll verify that the hashcode is based on the username only.
- getUserName and setUsername - These methods are also easy to test, we'll set up some user name using the test client, and see if the correct username is returned.

Network Protocol

There are nine 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.
8. RETRIEVE_PARTICIPANTS
   A retrieve participants message is sent to request the names of the users in a given conversation.
9. TWO_WAY_CONV
   A two way conversation message is sent as a request to create a new conversation with a specified user.

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 | RETRIEVE_PARTICIPANTS | TWO_WAY_CONV
```

```
MESSAGE_CONTENT = CONNECT_CONTENT | IM_CONTENT | NEW_CONV_CONTENT
        | ADD_TO_CONV_CONTENT | ENTER_CONV_CONTENT | EXIT_CONV_CONTENT
        | DISCONNECT_CONTENT | RETRIEVE_PARTICIPANTS_CONTENT |
        TWO_WAY_CONV_CONTENT

CONNECT_CONTENT = NEW_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
RETRIEVE_PARTICIPANTS_CONTENT = CONV_NAME
TWO_WAY_CONV_CONTENT = USERNAME

USERNAME = [^\t\n]{1,256}
NEW_USERNAME = [^\t\n]{0,256}
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
RETRIEVE_PARTICIPANTS = 7
TWO_WAY_CONV = 8
```

There are ten 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, with the connecting user's name first.
2. IM

An IM message is used to send a message to all users in a conversation.

3. 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.
4. 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..
5. 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.
6. CONNECTED
   A connected message notifies the client that a new user has connected to the server.
7. DISCONNECTED
   A disconnected message notifies the client that a given user has disconnected from the server.
8. PARTICIPANTS
   A participants message informs the client of the current participants in a given conversation, to resolve potential issues arising from network latency.
9. ERROR
   An error message informs the client of a bad request, possibly as a result of network latency.

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 | ADDED_TO_CONV | ENTERED_CONV |
    REMOVED_FROM_CONV | CONNECTED | DISCONNECTED | PARTICIPANTS |
    ERROR
MESSAGE_CONTENT = INIT_USERS_LIST_CONTENT | IM_CONTENT |
    ADDED_TO_CONV_CONTENT | ENTERED_CONV_CONTENT |
    REMOVED_FROM_CONV_CONTENT | CONNECTED_CONTENT |
    DISCONNECTED_CONTENT | PARTICIPANTS_CONTENT | ERROR_CONTENT

INIT_USERS_LIST_CONTENT = (USERNAME TAB)* USERNAME
IM_CONTENT = USERNAME TAB CONV_NAME TAB IM_ID TAB MESSAGE_TEXT

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
PARTICIPANTS_CONTENT = CONV_NAME (TAB USERNAME)*
ERROR_CONTENT = MESSAGE

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

INIT_USERS_LIST = 0
IM = 1
ADDED_TO_CONV = 2
ENTERED_CONV = 3
REMOVED_FROM_CONV = 4
CONNECTED = 5
DISCONNECTED = 6
PARTICIPANTS = 7
ERROR = 8

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

Client

Description
The client provides a GUI for users to connect to and to interact with a running IMServer. Upon launch, the client prompts for the address and port of the server. Then, the client prompts for a username with which to connect. The user may choose to specify a unique username or have the server generate one. Upon connection, the client displays a window with an area displaying the other currently connected users, an area displaying chat history, and an area containing buttons to perform actions. The available actions are

- New Room
  Opens a dialog box prompting for a conversation name. When the name is specified, the client requests a conversation of that name and opens it in a new tab if

there is no other conversation of that name on the server. When left blank, the client creates a new conversation with a unique name generated by the server and opens it in a new tab.

- New One-on-One Chat
  Opens a dialog box prompting for a username. If the specified user is also connected to the server, the client creates a new conversation with a unique name generated by the server containing the specified user. (Other users may later be added.) One may instead double-click on a username in the panel labeled Other Users to create a one-on-one chat with that user.
- Join Conversation
  Opens a dialog box prompting for a conversation name. If the conversation exists on the server, the client adds the user to the conversation and opens it in a new tab.
- Disconnect
  Terminates connection with the server, prompting again for an address and port number at which to connect to a chat server. One may instead close the window to log out.

When the user joins or creates a conversation, the client opens a new tab in which it displays all messages it receives and allows the user to send his own messages. Pending messages remain grey until the server acknowledges that they have been sent. The user may invite other users to the conversation by typing a name into the Invite User textbox. To see past chat history, the user must return to the Top Level tab and double-click the conversation name under Past Conversations. The user may leave a conversation by closing its tab.

Implementation

The ClientGUI class is a subclass of JFrame that displays the chat window upon connection. In addition to JComponents for the GUI and a socket, reader, and writer for communication with the server, ClientGUI keeps track of the following instance variables:

- otherUsers
  A set of usernames of the other clients connected to the same server. This is initialized upon receipt of an initial users list message and updated upon receipt of connected and disconnected messages.
- conversations
  A map of the conversations the client is engaged in. The key of a conversation is its name.
- serverName
  The name of the server (the address at which it is found).
- myUsername

The username used to connect to the server.

- conversationHistory
  An instance of ConversationHistory. Stores messages received in all conversations while logged in.
- incomingMessageManager
  An instance of IncomingMessageManager, which runs a thread to dispatch commands in response to messages from the server.
- outgoingMessageManager
  An instance of OutgoingMessageManager, which runs a thread to receive MessageToServers, format them, and send them to the server.

Most of the methods of ClientGUI handle a certain message type from the server or prompt the user for input:

- removeConversation(String convName)
  Removes convName from the collection of names of conversations that we think we are in. (Does not perform any GUI modifications.)
- tryToEnterConv(String convName, String _otherUsers)
  Opens a new tab corresponding to the given conversation unless we already have a tab open for it, in which case an informative message is posted to the status bar.
- tryToRemoveUserFromConv(String username, String convName)
  Removes the given user from our list of participants in the given conversation, if we actually thought the user was in the conversation in the first place.
- tryToAddUserToConv(String username, String convName)
  Adds the given user to our list of participants in the given conversation, if we didn't already think the user was in the conversation.
- handleConnectedMessage(String username)
  Adds the given user to our list of other users connected to the server, if we didn't already think the user was connected.
- handleDisconnectedMessage(String username)
  Removes the given user from our list other users connected to the server, if we actually thought the user was connected in the first place.
- handleParticipantsMessage(String convName, String users)
  Handles a PARTICIPANTS message from the server, which tells us which users are participating in a given conversation. We already have some idea of which users we think are participating; this message serves as a correction to compensate for issues of network latency.
- handleErrorMessage(String rejectedInput)
  Handles an ERROR message from the server, which says that it didn't like our previous message
- registerIM(String username, String convName, String _messageId, String text)

Takes in data about an IM message as received from the server and processes the message, updating the GUI and logging in history as appropriate.
- promptForNewRoom()
  Prompts the user to enter a new conversation.
- promptForTwoWayConv()
  Prompts the user to start a new two-way conversation.
- promptToJoinConv()
  Prompts the user to choose a conversation to join.
- disconnect()
  Ceases communication with server.
- addCloseButton(ConversationPanel conv)
  Makes the tab in this.tabbedPane with component conv have a close button.
- setStatusText(String s)
  Sets the text of the status bar.

ConversationPanel

ConversationPanel is a subclass of JPanel and displays the users and messages in a conversation. It provides text fields and buttons for inviting users and sending messages. In addition to displaying a GUI hierarchy (as drawn out in the UI hand-sketch), it stores the following data fields relating to the conversation:

- myUsername
  The username used to connect to the server.
- clientGUI
  The instance of ClientGUI to which the ConversationPanel belongs.
- convName
  The name of the conversation.
- messagesDoc
  A stylized document containing the history of the conversation since the ConversationPanel has been open. (For older messages, the user must navigate to PastConversations in the Top Level tab.) Updated upon receipt of an IM message and upon entering a message to be sent. See the description of MessagesDoc below.
- otherUsersSet
  A set containing the names of the other users in this conversation.

ConversationPanel has the following methods:

- createInviteMessage()
  Invites the user whose name is in the "invite" field to this conversation. Complains if

the contents of the field are invalid. Upon successfully sending the message, clears the contents of the invite field.

- createIMMessage()
  Sends out an IM message according to the user input. Complains if the contents of the message field are invalid. If the contents are valid, registers the message as pending and increments messageId.
- close()
  Cancels all pending messages in this conversation, removes this conversation from our collection of ongoing conversations, and sends an exit conversation message to server to let it know that we are leaving the conversation.

The first two methods are called by listeners (InviteListener and SendIMListener) attached to the appropriate GUI elements. The listener DoubleClickUsernameListener listens for double-clicking of username in the Other Users panel and generates one-on-one chats.

ConversationPanel's are added to the view in a JTabbedPane (the field tabbedPane in the ClientGUI class), which uses a ConversationTabComponent for each ConversationPanel to provide a facility for switching between and closing conversation tabs. The ConversationTabComponent class contains a JLabel to show the text and a JButton (subclassed as TabButton) to close the tab it belongs to. Attached is a MouseListener to allow the user to switch between tabs.

TopLevelPanel

ClientGUI maintains one tab in tabbedPane for a TopLevelPanel, which displays other currently connected users, chat history, and buttons to perform actions (the available actions are described in the description of the client above). TopLevelPanel keeps a reference to the ClientGUI it is in and references to swing components—one button for each action, a JScrollPane for displaying the other connected users, and a DefaultListModel, JList, and JScrollPane for keeping track of and displaying past conversations. DoubleClickHistoryItemListener is the MouseAdapter used to listen for double-clicks on past conversations. It pulls up a window displaying the associated list of messages, which are stored in clientGUI.conversationHistory (see the section on ConversationHistory).

MessagesDoc

MessagesDoc represents a stylized display of the messages in a conversation. This includes pending messages, which are be displayed beneath the non-pending ones. MessagesDoc keeps track of the following instance variables, in addition to variables used for styling:

- endOfReceived
  An integer representing the position in the string underlying this display at which received (non-pending) messages end.
- endOfPending
  An integer representing the position in the string underlying this display at which pending messages end.
- pending
  Maps messageId to message. messageId is the unique integer associated with a message, and the message is an instance of IMMessage used for representing IM messages.
- myUsername
  The user's username.
- convName
  The name of the conversation to which this MessagesDoc belongs.

MessagesDoc has the following methods:

- addMessage(IMMessage m)
  Adds a message to the display.
- receiveMessage(IMMessage m)
  Handle receipt of a message from the server. Registers a message as received from the server. Adds it to the display, or unpends an existing message if it exists.
- unpend(int messageId)
  Finds a sent message by its messageId and unpends it.

MessageToServer

IMMessage is used to store data in a chat message (sent as an IM message over the network). It has the instance variables username, message, convName, pending, messageId, canceled, as well as accessor methods for all of them and a method cancel that sets this.canceled, a volatile boolean, to true.

IMMessage is one of the two classes that implement the MessageToServer interface, which has the following methods:

- getMessageText
  Returns the text of the message, according to the network protocol grammar.
- isCanceled
  Checks whether the message has been canceled.
- cancel
  Cancels the message.

The other implementing class is DefaultMessageToServer, which is used for non-IM messages, and simply has a field messageText containing the full text of the message and a volatile boolean canceled.

ConversationHistory

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

- history
  A map from strings to lists of IMMessages. Each conversation that the user has been in is a key, and its value is the list of messages received in that conversations (including non-pending sent messages).
- listModel
  The DefaultListModel of the GUI element in the Top Level panel that displays Past Conversations.
- myUsername
  The user's username.
  Displays all chat messages which are still pending (i.e., have not been sent back to us by the server).

ConversationHistory has the methods

- logNew(IMMessage message)
  Logs a new message in history. Only received messages should be logged, not pending messages. If the conversation to which this message belongs is not already in this.history, adds the conversation's name as a row in this.listModel, to display it in the past conversations table in the top level panel of the GUI.
- historyDocument(String convName)
  Create a MessagesDoc to display the history of the conversation with the given name. Requires that the conversation name be in this.history.

Communication with the Server

The class IncomingMessageManager handles handles incoming communication from the server. It runs a Publisher thread which enqueues messages and a Subscriber thread which dispatches IncomingMessageWorkers, which are SwingWorkers, to handle them. IncomingMessageManager keeps a reference to a BlockingQueue called incomingMessages, which is shared between producer and consumer.

IncomingMessageWorker is a SwingWorker that executes only on the event-dispatch

thread. (doInBackground does nothing.) In it its done method, it parses the given incoming message, updating ClientGUI's data fields and modifying its GUI elements as necessary. Modifying these only from the event-dispatch thread ensures the thread-safety of ClientGUI. The done method sets the status bar of the GUI if it encounters a bad message.

OugoingMessageManager handles outgoing communication to the server. It runs a Subscriber thread which sends the message only if it has not been canceled, by keeping a reference to a BlockingQueue of messages and taking from it. The add method allows other threads to act as producers by adding messages to the BlockingQueue. Only sending uncanceled messages ensures that closing a conversation tab prevents pending outgoing messages for that conversation from being sent.

Generally, our strategy is to discard server messages that don't make sense with our current view of the world. If the server tells us that a message has been received for a conversation that we don't think we're in, we complain passively in the status bar. If the server tells us that a message has been received from a user who we don't think is in the conversation, we send a retrieve participants message to refresh our list of the conversation's participants. This provides a fix for some but not all of the issues 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 users, he can close and re-enter the conversations. If he wants to refresh the list logged-in users, he can log out and then log back in.

ConnectWindow

ConnectWindow is the initial window used to prompt the user for an address and port at which to connect to the chat server. ConnectWindow is a JFrame with two JTextFields, the necessary JLabels, and a JButton that calls the method tryToConnect when clicked. tryToConnect collects information from the text fields and uses it to attempt to connect to the server. It complains if the contents of the fields are invalid by displaying a popup message. If the client fails to connect to a server at the specified address, it prints the error message to stderr.

UsernameSelectWindow

UsernameSelectWindow is window displayed upon successfully identifying the server. It is a JFrame that prompts the user for a username to use on the server. It complains if the user input is invalid by displaying a popup message.

The user can specify a username in the JTextField username or select the JRadioButton generateUsername. Upon hitting enter from the JTextField or clicking the JButton okButton, tryToRegisterUsername is called to send a connect message with the specified username (or no username). It complains if the user input is invalid, waits for server response, and lets us know if the name is taken. Upon successfully registering the username, it spawns a ClientGUI for our session.

The method dialogAndReenable displays an alert dialog with the specified text and then re-enables the GUI elements.

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.

Methods that access and modify otherUsersSet conversations, and conversationHistory in ClientGUI all run in the event-dispatch thread, thus preventing concurrent modification. All methods that access and modify a ConversationPanel.otherUsersSet also runs in 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 run manual tests for all edge cases and observe that the behavior is consistent. 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 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 TestClient 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

TestClient

Description

We use the TestClient class, located in the test package, as a mock version of the client to test different parts of the server. TestClient provides an automated way to send messages to and read responses from the server.

Implementation

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

- send(String s)
  Sends the given String to the server.
- readLine()
  Read the next line in the socket, sent from the server or client. Blocks until a newline is printed in the socket.
- close()
  Closes this TestClient by closing each of this.in, this.out, and this.socket. Should be called whenever communication is completed.
- ready()
  Tells whether the TestClient’s input stream is ready to be read.

guichat: Testing Report

Summary

This document details all the JUnit and manual tests that we run to ensure that each component of our server and client function as expected. For methods that can be individually tested, we test every case allowed by the method specifications. For method specifications, see the Javadocs located in guichat/doc, or the descriptions in the design document.

The Conversation class is tested in ConversationTest, located in the server package. The User class has some methods that are tested individually in UserTest, also located in the server package, and others that are tested in ServerTest, also located in the server package, that cannot be tested individually because of dependence on methods in IMServer. The tests in ServerTest test proper functioning of run and its private helper methods in User, as well as the corresponding methods in IMServer. ServerTest also provides methods to test for proper functioning of the server as a whole, by sending network messages and reading server responses using instances of TestClient.

The client is tested manually by dividing the space of sequences of inputs from the user into modular components. We ensure that each of these independently tested sequences of inputs receives the correct response from the GUI, with one or many clients connected to the same server.

ConversationTest, UserTest, and ServerTest are no_didit tests because they all use networking. (To create an instance of User, we must have a valid socket.) Since didit restrictions prevent us from automatically running these tests, we must run them without didit.

guichat passes all tests described here.

ConversationTest

In ConversationTest, we test toString, add, remove, contains, getName, toArray, and hashCode. sendMessage is tested in ServerTest, since it requires the IMServer to work properly. The tests for each method are explained below:

- toString (and constructors)
    - noUserConstructorTest
      Check that constructor with no initial users and toString work.

    - oneUserConstructorTest
      Check that constructor with one initial user and toString work.
    - twoUserConstructorTest
      Check that constructor with one initial user and toString work.
- add
    - addFirstTimeTest
      Expect that add returns true when a User is added for the first time.
    - addSecondTimeTest
      Expect that add returns false when a User is added for the second time.
    - addNullTest
      Expect that add returns false when a null User is added.
- remove
    - removeNewUserTest
      Expect that remove returns false when a User not in a conversation is removed.
    - removeExistingUserTest
      Expect that remove returns true when a User in a conversation is removed.
    - removeNullTest
      Expect that remove returns false when a null User is removed.
- contains
    - containsNewUserTest
      Expect that contains returns false when a User is not in the conversation.
    - containsExistingUserTest
      Expect that contains returns true when a User is in the conversation.
    - containsNullTest
      Expect that contains returns false when given null.
- getName
    - getNameTest
      Expect that getName correctly returns the conversation's name. Since this.name is non-null, there is only one case.
- toArray
    - emptyToArrayTest
      Expect that toArray returns an empty array when conversation is empty.
    - nonEmptyToArrayTest
      Expect that toArray returns an an array of the correct users when non-empty.
- hashCode
    - hashcodeTest
      Expect that hashCode returns the Conversation's name's hashCode.

ServerTest

To test the server, we use one or more instances of TestClient to connect to the server and make sure that the correct messages are received in response to a given sequence of client-to-server messages. This tests the corresponding methods in IMServer and the private helper methods for run in User, as well as the server performance as a whole. Since these methods each rely on the behavior of many components of the system, they cannot be tested individually. However, by testing certain sequences of messages to the server, we can unit test independent aspects of the server's behavior.

The tests in ServerTest are organized below by the type of network message being tested. (In some, a few network messages are sent, but one marks the primary path being tested.)

- Logging in: connect messages
    - oneUserLoginTest
      Ensure that correct message is returned from the server when one user logs in.
    - nameTakenLoginTest
      Ensure that disconnected message is returned from the server when a user logs in with a name that is already taken.
    - multipleUserLoginTest
      Ensure that correct message is returned from the server when a user logs in after other users are logged in.
    - largeNumberOfUsersLoginTest
      Ensure that correct message is returned from the server when a lot of users are logged in.
- Sending chats: IM messages
    - inConvMultipleUserIMTest
      Expect IM message to be sent to all clients in a conversation when the sender of an IM message is in the conversation and other clients are also in the conversation.
    - inConvOneUserIMTest
      Expect IM message to be sent to the sender when the sender of an IM message is in the conversation but no other clients are in the conversation.
    - notInConvIMTest
      Expect error message to be sent to sending client when the sender of an IM message is not in the conversation but is connected to the server.
    - notConnectedMultipleUserIMTest
      Expect error message to be sent to sending client when the sender of an IM message is not in the conversation or connected to the server.

    - convDoesNotExistIMTest
      Expect error message to be sent to sending client when an IM message is sent to a conversation that doesn't exist.
- Making conversations: new conversation messages
    - unspecifiedNameNewConvTest
      Expect a conversation name to be autogenerated when a user sends a new conversation message with an empty name.
    - unusedNameAndConnectedNewConvTest
      Expect enter conversation message to be sent to the sender of a new conversation message when the conversation name is unused and the sender is connected to the server.
    - usedNameNewConvTest
      Expect no message to be sent to the sender of a new conversation message when the conversation name is used.
    - unusedNameAndDisconnectedNewConvTest
      Expect an error message to be sent to the sender of a new conversation message when the conversation name is unused and the sender is not connected to the server.
- Inviting users to conversations: add to conversation messages
    - inConvAndConnectedAndUserExistsAndIsNotInConvAddToConvTest
      Expect enter conversation message to be sent to a user and an added to conversation message to every other user when a user is added to a conversation by a user who is already in the conversation.
    - inConvAndConnectedAndUserExistsAndIsInConvAddToConvTest
      Expect an entered conversation message to be sent when a user is added to a conversation by a user who is already in the conversation if the user he adds is also already in the conversation. Expect no added to conversation messages to be sent to other users.
    - inConvAndConnectedAndUserDoesNotExistAddToConvTest
      Expect an error message to be sent when a non-existent user is added to a conversation by a user who is already in the conversation.
    - notInConvAndConnectedAddToConvTest
      Expect an entered conversation message to be sent when a user is added to a conversation by a user who is not already in the conversation but is connected to the server. Expect an added to conversation message to be sent to everyone else in the conversation.
    - notConnectedAddToConvTest
      Expect an error message to be sent when a user is added to a conversation by a user who is not connected to the server.
- Entering conversations: enter conversation messages
    - notInConvEmptyEnterConvTest

Expect an entered conversation message to be sent when a user who is not in a given conversation sends an enter conversation message requesting to join it, when the conversation is previously empty.
    - notInConvNonEmptyEnterConvTest
      Expect an entered conversation message to be sent when a user who is not in a given conversation sends an enter conversation message requesting to join it, and that an added to conversation message is sent to everyone else in the conversation, when the conversation is previously nonempty.
    - inConvEnterConvTest
      Expect an entered conversation message to be sent when a user who is already in a given conversation sends an enter conversation message requesting to join it.
    - notConnectedEnterConvTest
      Expect an error message to be sent when a user who is not connected to the server sends an enter conversation message.
    - convDoesNotExistEnterConvTest
      Expect an error message to be sent when a user tries to enter a conversation that does not exist.
- Exiting conversations: exit conversation messages
    - notInConvExitConvTest
      Expect an error message to be sent when a user who is not in a given conversation sends an exit conversation message requesting to leave it.
    - inConvExitConvTest
      Expect a removed from conversation message to be sent to all other users in the conversation when a user who is in a given conversation sends an exit conversation message requesting to leave it.
    - notConnectedExitConvTest
      Expect an error message to be sent when a user who is not connected to the server sends an exit conversation message for an existing conversation.
    - convDoesNotExistExitConvTest
      Expect an error message to be sent when a user tries to exit a conversation that does not exist.
- Disconnecting: disconnect messages
    - connectedDisconnectTest
      Expect a disconnected message to be sent to all other connected users when a user who is connected to the server sends a disconnect message.
    - notConnectedDisconnectTest
      Expect an error message to be sent when a user who is not connected to the server sends a disconnect message.
- Retrieving participants of conversations: retrieve participants messages
    - convExistsAndEmptyRetrieveParticipantsTest

Expect a participants message to be sent when a user who is connected to the server sends a retrieve participants message to the server for a conversation that exists but is empty.
    - convExistsAndNonEmptyRetrieveParticipantsTest
      Expect a participants message to be sent when a user who is connected to the server sends a retrieve participants message to the server for a conversation that exists and is non-empty.
    - convDoesNotExistRetrieveParticipantsTest
      Expect an error message to be sent when a user who is connected to the server sends a retrieve participants message to the server for a conversation that does not exist.
    - notConnectedRetrieveParticipantsTest
      Expect an error message to be sent when a user who is not connected to the server sends a retrieve participants message to the server.
- Making one-on-one chats: two-way conversation messages
    - bothConnectedTwoWayConvTest
      Expect entered conversation messages to be returned when a user sends a two way conversation message requesting a conversation with another connected user.
    - user1ConnectedTwoWayConvTest
      Expect error message to be returned when two way conv message is sent, user sending the message is connected, but the username sent in the message is not connected.
    - user2ConnectedTwoWayConvTest
      Expect error message to be returned when two way conv message is sent, the user sending the message is not connected, but the username sent in the message is connected.
    - neitherConnectedTwoWayConvTest
      Expect error message to be returned when two way conv message is sent, the user sending the message is not connected, and the username sent in the message is not connected.
    - sameNameTwoWayConvTest
      Expect error message to be returned when two way conv message is sent, both users are connected, and both users have the same name.

UserTest

We tested the following User methods in UserTest, without dependence on an IMServer. The tests for each method are explained below:

- addConversation

    - addNewConversationTest
      Expect that addConversation sends an entered conversation message and returns true when the conversation is not already in User.conversations.
    - addExistingConversationTest
      Expect that addConversation returns false when the conversation is already in User.conversations.
- removeConversation
    - removeNewConversationTest
      Expect that removeConversation returns false when the conversation is not already in User.conversations.
    - removeExistingConversationTest
      Expect that removeConversation returns true when the conversation has been added to User.conversations.
- getUsername and setUsername
    - unsetUsernameTest
      Expect that getUsername returns null when the username is unset.
    - setUsernameTest
      Expect that setUsername properly sets the username and is retrieved by getUsername. (The preconditions for setUsername ensure that their is only one case for this, and no edge cases to test.)
- sendInitUsersListMessage
    - sendInitUsersListMessageEmptyUsersTest
      Expect that sendInitUsersListMessage properly sends an initial users list message, following the grammar, when given an empty array.
    - sendInitUsersListMessageNonEmptyUsersTest
      Expect that sendInitUsersListMessage properly sends an initial users list message, following the grammar, with the calling User's name first, when given a nonempty array.
- sendEnteredConvMessage
    - sendEnteredConvMessageTest
      Expect that sendEnteredConvMessage properly sends an entered conversation message, following the grammar.
- sendAddedToConvMessage
    - sendAddedToConvMessageTest
      Expect that sendAddedToConvMessage properly sends an added to conversation message, following the grammar.
- sendRemovedFromConvMessage
    - sendRemovedFromConvMessageTest
      Expect that sendRemovedFromConvMessage properly sends an removed from conversation message, following the grammar.
- sendConnectedMessage

    - sendConnectedMessageTest
      Expect that sendConnectedMessage properly sends a connected message, following the grammar.
- sendDisconnectedMessage
    - sendDisconnectedMessageTest
      Expect that sendDisonnectedMessage properly sends a disconnected message, following the grammar.
- sendIMMessage
    - sendIMMessageTest
      Expect that sendIMMessage properly sends an IM message, following the grammar.
- sendParticipantsMessage
    - sendParticipantsMessageEmptyUsersTest
      Expect that sendParticipantsMessage properly sends a participants message, following the grammar, when given an empty array.
    - sendParticipantsMessageNonEmptyUsersTest
      Expect that sendParticipantsMessage properly sends a participants message when given a nonempty array, according to the network protocol grammar.
- sendErrorMessage
    - sendErrorMessageTest
      Expect that sendErrorMessage properly sends an error message, according to the network protocol grammar.
- equals
    - nullEqualsTest
      Expect that equals returns false when argument is null.
    - notUserEqualsTest
      Expect that equals returns false when argument is not of type User.
    - differentNameEqualsTest
      Expect that equals returns false when argument has a different name.
    - sameNameEqualsTest
      Expect that equals returns true when argument has the same name.
- hashCode
    - noNameHashcodeTest
      Expect that hashCode returns 0 when the user's name is not set.
    - namedHashcodeTest
      Expect that hashCode returns the User's name's hashCode when the user's name is set.

Each component of the server passed all of its tests, so we conclude that it works as expected.

Manual Testing of Client

To test the client, we performed the following series of actions and observed the expected behavior. Note that text fields are "valid" if they follow the grammar (for username, message, or new conversation name, depending on context).

- Connect to a valid server. Expect: UsernameSelectWindow generated.
- Connect to an invalid server. Expect: Popup displayed.
- Connect to a server when address text field is length blank. Expect: Popup displayed.
- Connect to a server when port text field is length blank. Expect: Popup displayed.
- Submit generated username. Expect: ClientGUI generated.
- Submit specified username. Expect: ClientGUI generated.
- Submit specified username that is taken. Expect: Popup displayed.
- Hit New Room button and submit conversation name. Expect: ConversationPanel generated in new tab.
- Hit New Room button and submit conversation name that is taken. Expect: Dialog box closes and no new tab is generated.
- Hit New Room button and submit empty conversation name. Expect: ConversationPanel generated in new tab with autogenerated name.
- Hit New Room button from two clients and submit same conversation name from both. Expect: ConversationPanel generated in new tab in one, not in other.
- Hit New One-on-One Chat button and submit invalid username. Expect: Popup displayed.
- Hit New One-on-One Chat button and submit valid username who is not logged in. Expect: No new ConversationPanel generated; status bar updates.
- Hit New One-on-One Chat button and submit valid username who is logged in. Expect: New ConversationPanel generated in new tab, with autogenerated name, with both users in it.
- Double-click on a user in Other Users. Expect: New ConversationPanel generated in new tab, with autogenerated name, with both users in it.
- Hit Join Conversation and submit invalid conversation name. Expect: Popup displayed.
- Hit Join Conversation and submit valid conversation name that does not exist. Expect: No new conversation tab generated; status bar updates.
- Hit Join Conversation and submit valid conversation name that is in use. Expect: New conversation tab generated with conversation and current users.
- Hit Disconnect. Expect: ConnectWindow generated, and other users have the user removed from each conversation and from the Top Level panel list of Other Users.
- Send an invalid message. Expect: Popup displayed.
- Send a valid message in an empty conversation. Expect: Message displayed as

grey and then (quickly) replaced as black, with correct username.

- Send a valid message in a nonempty conversation. Expect: Message displayed as grey and then (quickly) replaced as black, with correct username, in sender's tab, and displayed as black, with correct username, in other users' tabs for that conversation.
- Leave a conversation and double-click its name in Past Conversations. Expect: Popup with history displayed.
- Stay in a conversation with messages and double-click its name in Past Conversations. Expect: Popup with history displayed.
- Stay in a conversation with messages and double-click its name in Past Conversations. Then send more messages and double-click again. Expect: Popup with history displayed, then another popup displayed with updated history.
- Leave a large conversation. Expect: users list updated for other users.
- Leave a conversation with only one user. Create New Room of same name. Expect: No new conversation tab generated. (Recall that room names are not reusable).

The client passed these tests, and is in general very usable. We conclude that it works.