Client Design

GUI Appearance

The client will look as pictured in client-drawing.pdf.

The workings of the Connect dialogue are self-explanatory. Once the user clicks the "Connect" button, the client will wait for the server's response before deciding what kind of window to open next.

During the waiting time, all editable fields in the Connect dialogue become disabled and the "Connect" button turns into a "Cancel" button so that the user may interrupt a connection that is taking too long.

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 Client.handleServerMessage(MessageFromServer message) which does this; it will probably have helper methods.

Generally, our startegy 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.

Conversation history display

The conversation history display will controlled by the object ConversationHistory conversationHistory, which has the following data fields:

- String receivedMessages – represents all messages which have been received as of yet
- List<OutgoingChatMessage> pendingMessages – represents all chat messages which are still pending (i.e., have not been sent back to us by the server).

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

The class OutgoingChatMessage will have two fields:

- String messageText
- int uniqueId – this is how we 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 so, remove it from pendingMessages. Then, we append the message (with appropriate byline) to receivedMessages.