Conversation Design

We try to emulate the design of the web-based Google Talk client. The client displays a list of online users. If a user wants to start a conversation with some set of users of size greater than or equal to one, he clicks on one of the desired users in the online users list and a chat window appears. His client assigns a client-unique ID (a client-unique ID should be different from those for other conversations currently active on on this client) to this conversation. To add any additional users to the conversation, the initiating user (or any other user currently in the conversation) simply needs to hit the "plus" button in the chat window and enter the handle(s) of these users. Conversations do not have names, so individuals not currently in a conversation cannot add themselves to it – they must be added by a current communicant. When a user sends a message to the server for this conversation, the server distributes it to all other communicants, and if those communicants do not yet have windows for this conversation, their clients open them and assign client-unique IDs to this conversation. The server stores no state for conversations – it simply relays messages from senders to recipients. (To enable this, every message includes all desired recipients, as detailed further in the protocol section.) A client knows what conversation an incoming message from the server corresponds to because it is tagged with the unique ID the client assigned to it. (Unless, of course, this is the first incoming message from the server for a particular conversation, in which case the client opens a new window and assigns a new ID to this conversation, which it then relays to the server in all future messages in this conversation.)

Client/Server Protocol

Client-to-server grammar:
CS_MESSAGE := CHAT_MESSAGE | REGISTER_HANDLE_MESSAGE | GET_USERS_MESSAGE
CHAT_MESSAGE := SENDER_FIELD newline COMMUNICANTS_LIST newline DATA
SENDER_FIELD := "sender: " HANDLE
COMMUNICANTS_LIST := "clist: " (COMMUNICANT ",")+ COMMUNICANT /* includes the sender, as well as any users who were just added if this message is for an 'add user' operation */
COMMUNICANT := HANDLE (NUM | "new") /* the number is the ID the communicant has assigned to this chat, or "new" if the communicant has not yet communicated an ID for this chat */
HANDLE := [a-z0-9]+
NUM := [0-9]+
DATA := CHAT | EXIT_CHAT_NOTICE | ADD_USER_NOTICE
CHAT := "chat: " TEXT
TEXT := [^\n]+
EXIT_CHAT_NOTICE := "exit"
ADD_USER_NOTICE := "add: " HANDLE
REGISTER_HANDLE_MESSAGE := "handle: " HANDLE
GET_USERS_MESSAGE := "getusers" /* want list of all online users */

Server-to-client grammar (uses productions defined above):
SC_MESSAGE := CLIENT_ORIG_MESSAGE /* stands for client-originated message, one that is simply being relayed to an intended recipient */ |
USER_ONLINE_MESSAGE | BAD_HANDLE_MESSAGE | USER_LIST_MESSAGE
/* If we are sending this to a client that has not yet communicated its ID for this conversation, then we include the full communicants list. Otherwise, we just include the client's ID (RECIPIENT_ID), as we assume the client has stored the list of communicants and their IDs for this conversation. */
CLIENT_ORIG_MESSGE := SENDER_FIELD newline (COMMUNICANTS_LIST | RECIPIENT_ID) newline DATA
RECIPIENT_ID := "rid: " NUM
USER_ONLINE_MESSAGE := "online: " HANDLE
BAD_HANDLE_MESSAGE := "nohandle" /* the requested handle was not available */
USER_LIST_MESSAGE := HANDLE ("," HANDLE)* /* list of all online users */

Note about client/server communication:
Consider the situation where a client has received messages for a particular conversation but has not sent any of its own. This means that the client will not be able to communicate its ID for this conversation to the server, since this only happens when the client performs some action (sends a chat message, adds a new user to the chat, etc.). So, before it assigns an ID to a conversation, the client stores a HashMap containing the handles (keys) and IDs (values) of the other communicants in this conversation. (If another communicant also has not assigned an ID to this conversation, the value in the HashMap can be null.) When it receives a new message for the conversation, we can guarantee that its communicants list matches what the client has in the HashMap exactly (**Claim 1**, some explanation found in architecture section; the one exception is that the sender of the message may have set its ID, but this is easily detected and does not affect anything), enabling the client to match the message to the conversation without using an ID. As long as the client updates its HashMap when users are removed from and added to this conversation (and when clients assign IDs to the conversation), this invariant will continue to hold. When the user finally performs an action in this conversation, the client can assign an ID to the conversation and the HashMap can now be associated with this ID so that we do not need to check the contents of the HashMap to determine whether it corresponds to a conversation for which we have just received a message, since our ID will be present in the message if it does indeed correspond. (We must continue to make the same updates to the HashMap as before, however – we need to know the correct recipients for each message and their IDs.)

General Client/Server Architecture (not required for milestone, but we wanted to think about it anyway)

1. When the client is launched, it opens a TCP connection with the server that remains alive until the client is terminated. The client must send a handle to the

server before it can communicate further with the server (e.g. to retrieve information about who's online).

2. Threads on the server
    a. A main server thread that waits for new incoming connections, creates dedicated sockets for those connections, and then spins off two threads (client worker threads) to handle each created socket (one to read input from the socket, one to write output).
    b. The input client worker thread for a particular socket reads input from the socket and turns it into jobs that are put on the BlockingQueues of all relevant output client worker threads.
        i. For example, input that specifies a message to be sent to some communicants will be transformed into the server-to-client format detailed above and inserted into the BlockingQueues of the output threads for the sockets of all communicants.
        ii. Another interesting case is an input that registers a handle. If this handle is available, the input client worker thread places a USER_ONLINE_MESSAGE into the BlockingQueue of every output client worker thread.
        iii. When an input client worker thread wants to write to a set of BlockingQueues for output client worker threads, it first obtains the locks for all of these BlockingQueues – this ensures that **Claim 1** above is always true, because there is no way for some client A to receive a message from the server that reflects more than the one update from its current state, since this would mean that updates did not appear in the same order in all BlockingQueues (which is impossible given the complete locking of the BlockingQueues).
    c. The output client worker thread for a particular socket reads from its BlockingQueue and relays these messages to the client.
3. Threads on the client
    a. Other than the event handling thread, each client has two main threads – one that handles input from the server and another that sends output to the server.