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. 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 select the handle(s) of these users in the list of all online users that appears. 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. Furthermore, a user cannot deny a conversation – if a message is sent to them in a new conversation, a new chat window automatically appears. When the initiating user sends the first message to the server for this conversation, the server assigns the conversation a unique ID, and stores the list of communicants. It then sends this ID back to the initiating user and then forwards the message to each recipient along with the ID and the handles of all other communicants. In all future messages sent from clients for this conversation, only the ID and the message text are sent to the server.

Classes:

1. Server-to-client messages
    a. Classes that represent server-to-client messages all implement the interface SMessage with the following method
        i. accept(SMessageVisitor)
    b. The class SMessageImpls contains as static inner classes all of the classes that implement SMessage, which are (these represent each of the possibilities for SC_MESSAGE in the server-to-client grammar)
        i. UserList, a container with one variable: List<String> of handles
        ii. InitialChat, a container with the following variables
            1. Long id
            2. String senderHandle
            3. UserList otherCommunicants
            4. String chatMessage
        iii. NormalChat, a container with the following variables (NormalChat also implements CMessage, see 2)
            1. Long id
            2. String senderHandle
            3. Action action; Action is a container with the following variables
                a. ActionType at; Action is an enum nested in Action with three values: TEXT_MSG, EXIT_CHAT, and ADD_USER
                b. List<String> handles (may be null if not ADD_USER action)
                c. String chatMessage (may be null if not TEXT_MSG action)
        iv. AvailabilityInfo

1. String handle
2. Status s – a nested enum with values OFFLINE and ONLINE (can easily be expanded to include other statuses)

v. BadHandle
1. Has no data, simply implements the accept() method of SMessage

c. SMessageImpls also contains a factory method to convert a String server-to-client message into the appropriate SMessage

2. Client-to-server messages
   a. The datatypes representing client-to-server messages are structured very similarly to those for server-to-client messages
      i. CMessage interface with one method: accept(CMessageVisitor)
      ii. CMessageImpls class that contains as static inner classes all of the classes that implement CMessage, which together cover all of the possibilities for CS_MESSAGE in the client-to-server grammar below
      iii. CMessageImpls also includes a factory method to convert a String client-to-server message into the appropriate CMessage
      iv. The remaining details are omitted for brevity
3. Other datatypes used by both server and client
   a. SocketOutputWorker, extends Thread
      i. Constructor takes a socket whose output stream we should write to
      ii. Has a BlockingQueue in which messages to be written to the output stream are stored
      iii. run() method sits in a loop calling take() on the BlockingQueue and writes any data it pulls from the queue
      iv. Has add() method that takes a String to be written to the socket output stream, adds this String to the BlockingQueue
4. Server-specific datatypes
   a. Conversation – more or less a container class with getters and setters for the following variables
      i. List<String> communicants – a list of the handles of the people in this conversation
      ii. Long id – the server-assigned ID for this conversation
   b. One SSocketInputWorker per client – reads input from the socket for the client
      i. Extends abstract class SocketInputWorker, which itself extends Thread
         1. Constructor takes a socket
         2. run() method reads input stream for socket and calls the abstract method process(String) with the read data
      ii. SSocketInputWorker constructor takes both a socket and an instance of CMessageVisitor (in this case a ServerInputProcesser)
         1. process() is implemented to use the CMessageImpls factory method to produce a CMessage from the input String. It

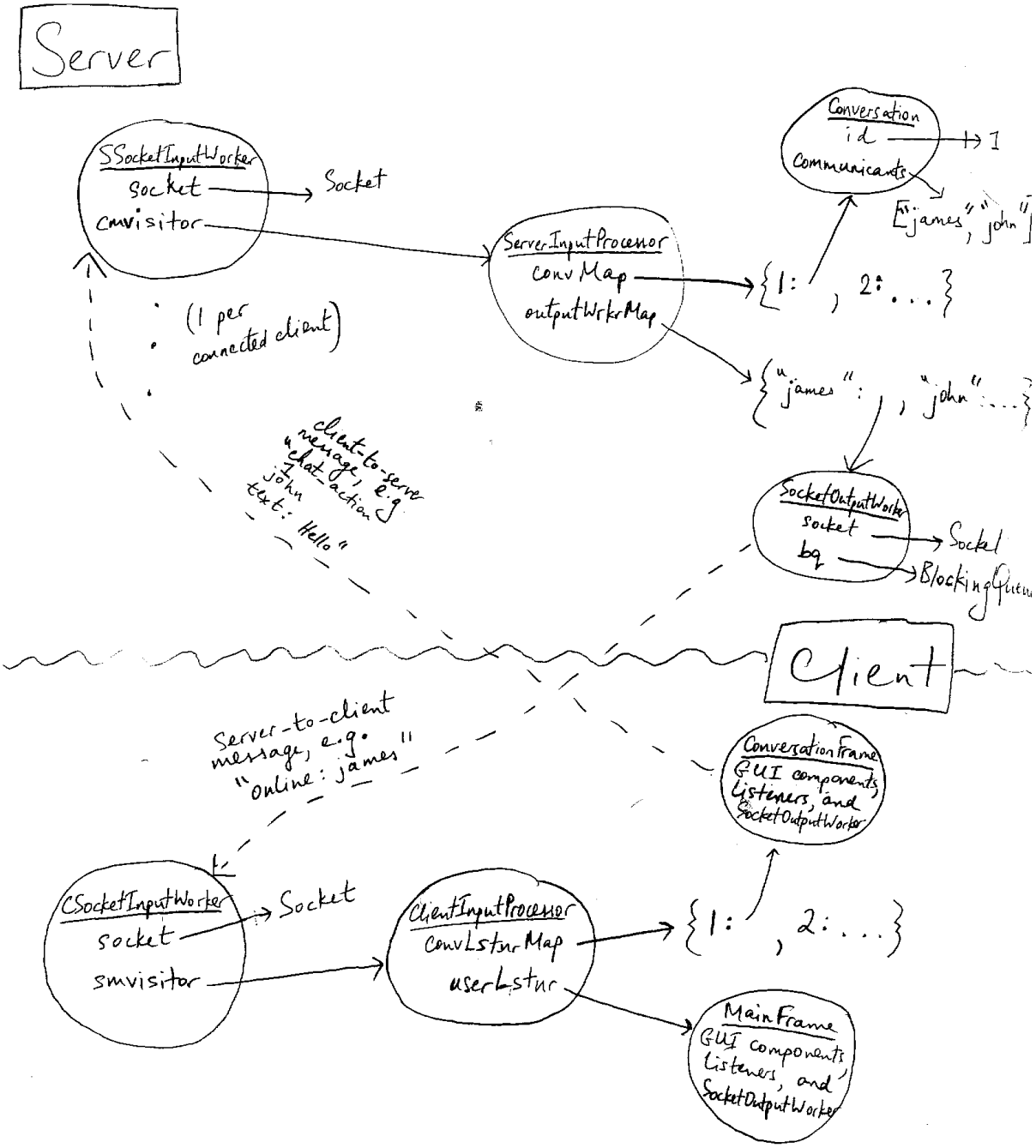
then calls accept on this CMessage with its CMessageVisitor as the argument

iii. ServerInputProcessor, implements CMessageVisitor
    1. It has the following variables
        a. Map from Long (conversation ID) to Conversation
        b. Map from String (user handle) to SocketOutputWorker (for the socket for that particular user)
    2. For each possible CMessage, it decides what (if any) SMessage to send to the clients that should be notified. It constructs this SMessage and calls toString() on it to generate the message to pass to the SocketOutputWorkers for the recipient clients.

5. Client-specific datatypes
    a. Each client has one CSocketInputWorker, which extends the aforementioned abstract class SocketInputWorker
        i. Works much like to SSocketInputWorker, with CMessage* replaced by SMessage* see 4bii
    b. ClientInputProcessor implements SMessageVisitor, has the following variables (can be thought of as a pseudo-model for the client, although its only duty is to notify the view of data changes – the actual data storage is delegated to the view, since we deemed it pointless to store one copy of the data in a model data structure and another identical copy in the view itself; we can easily start storing data in the model if at any point we change the view to display only a subset of the total relevant data)
        i. Map from Long (conversation ID) to ConversationListener (in this case the implementation is ConversationFrame, see d)
        ii. UserOnlineListener ul (in this case the implementation is MainFrame, see c)
    c. MainFrame, extends JFrame, implements UserOnlineListener (can be thought of as the view and the controller for the user list part of the client GUI)
        i. It implements the methods in UserOnlineListener, which are
            1. addOnlineUser(String handle)
            2. removeOfflineUser(String handle)
        ii. When the user closes the MainFrame, it creates the datatype corresponding to an EXIT_CHAT_NOTICE (see grammar), serializes it, and passes it to this client's SocketOutputWorker
    d. ConversationFrame, extends JFrame, implements ConversationListener, one per active conversation on the client (can be thought of as the view and the controller for an individual conversation in the GUI)
        i. It implements the methods in ConversationListener, which are
            1. addMessage(String sender, String message)
            2. removeUser(String handle)
            3. addUser(String handle)

ii. When the user enters a message for this conversation, adds a user to it, or closes the ConversationFrame, the ConversationFrame creates the datatype corresponding to the action, serializes it, and passes it to the client's SocketOutputWorker (the user's actions only show up in the view once they're passed back from the server)

Snapshot Diagram:



Client/Server Protocol

Client-to-server grammar:
CS_MESSAGE := INITIAL_CS_CHAT_MESSAGE | NORMAL_CHAT_ACTION | REGISTER_HANDLE_MESSAGE | GET_USERS_MESSAGE
INITIAL_CS_CHAT_MESSAGE := "register" newline HANDLE /* initiator */ newline USER_LIST /* the other users in the chat */ newline TEXT_MSG /* the first message in the conversation */
NORMAL_CHAT_ACTION := "chat_action" newline CHAT_ID newline HANDLE /* sender */ newline ACTION
USER_LIST := HANDLE ("," HANDLE)*
HANDLE := [a-z0-9]+
CHAT_ID := [0-9]+
ACTION := TEXT_MSG | EXIT_CHAT_NOTICE | ADD_USER_NOTICE
TEXT_MSG := "text: " TEXT
TEXT := [^\n]+
EXIT_CHAT_NOTICE := "exit"
ADD_USER_NOTICE := "add: " USER_LIST
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 := INITIAL_SC_CHAT_MESSAGE | NORMAL_CHAT_ACTION | USER_ONLINE_MESSAGE | USER_OFFLINE_MESSAGE | BAD_HANDLE_MESSAGE | USER_LIST_MESSAGE
INITIAL_SC_CHAT_MESSAGE := "initial" newline CHAT_ID newline HANDLE /* sender */ newline USER_LIST newline TEXT_MSG
USER_ONLINE_MESSAGE := "online: " HANDLE
USER_OFFLINE_MESSAGE := "offline: " HANDLE
BAD_HANDLE_MESSAGE := "nohandle" /* the requested handle was not available */
USER_LIST_MESSAGE := "users:" USER_LIST

State of client and server:
This is covered extensively in the conversation design part of this document. In summary, the server stores an ID for each conversation and the communicants in each conversation. It does not store the messages in the conversation. The client stores the ID for each conversation, and the client's view (but no outside data structures) contains the list of online users, the communicants in each conversation, and the messages in each conversation.