Design Specifications  
Eric Makino   
02/27/2012

**Description**

This is a simple console Instant Messenger Program. It uses a very primitive console interface, therefore doesn't provide a very "friendly" interface. Its primary purpose is to demonstrate a multiple client and single server setup through the use of threads and socket programming.

**IMclient.cpp**

IMclient is the client program that connects to IMserver. The IMclient allows the client to talk to users connected to IMserver. The IMclient will enter a username for authentication to see if username already exist in server. The IMclient allows the client to select from three choices: CHECK LIST, TALK, MENU and QUIT. The CHECK LIST allows the client to see all the users that are connected to the server. The MENU function allows the client the option of displaying the menu of choices again. The TALK function allows the client to choose a user that’s connected the server and send them a message. Last, QUIT allows the server to disconnect the client and remove client from server. These four operations are part of the protocol, which is send along with the message.

**IMserver.cpp**

IMserver is the server program for the IM application. The server accepts users connections and stores usernames that the users specify in a struct (UserInfo) with username (name) and socket number (sock). With the username and socket number, the server will be able to keep track of all its users and their socket number for sending messages between different users. The server will parse the protocol in messages into fields and provide different functions according to the protocol’s specifications (more detail explained later). The server will remove the name if a user chooses to QUIT or disconnects with the server. The server will also send the CHECK LIST of all users that is connected to all users connected to the server whenever another user joins or leaves the application.

**IMclient.cpp & IMserver.cpp**

Both the IMclient and the IMserver work together to create the IM Application. The client is first connects to the server via the indicated port and asks to input a username. If the username does not exist in the server already, the server allows the user to join the server and adds client information into UserInfo. The client will than be ask to choose from three operations. Client’s message is sent to the server via the IM protocol. The server than takes the protocol and parses it to understand what functions the users want. The server than executes the function according to user’s specifications. When a user leaves the server, the server removes the user from the struct.

**Protocol**

**Packet Structure:**



COMMAND – The function to execute for the corresponding message

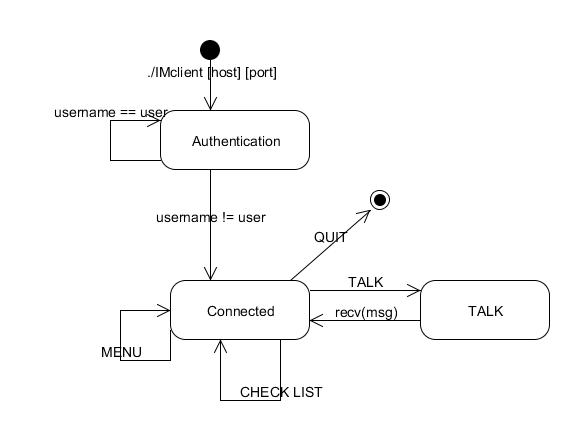
USER\_FROM – The username that sent the message.

USER\_TO – The username that receives the message.

MSG\_LENGTH – The length of the whole message including the protocol.

MSG – The actual message itself. The message is limited at 100 characters.

**Finite State Machine:**

****

**Design Decisions**

**Prior Design Ideas & Issues**

**Protocol.** At first, the protocol only had MSG, MSG\_LENGTH, USER\_FROM, and USER\_TO. We later discovered that the program would have to do more than just sending messages between users. Then we added COMMAND to the protocol to distinguish user’s choice: LIST, TALK, and QUIT.

**Send & Recv.** Our prior design also had a hard time synchronizing the sending and receiving between the client and the server causing the server and client to be blocked sometime.

Than we ran into a problem similar to the proxy where the server would always have to be ready to receive information from client. However, in this case, the client would also always have to be ready to receive form the server because at anytime point, another user may send a message to you.

**Thread.** The original idea on threading was to have the thread to only take care of the sending and receiving in the server. However, we run into the problem where the thread will have to exist after connection have been made in order for client to stay connected with the server.

**The Final Design**

**Protocol.** We added in the COMMAND field to the protocol to know what function the user will like to execute.

**Send & Recv.** We looked at our proxy and implemented a select statement in both the client and server. This is needed to determine whether an outside source is trying to send messages to the program so the select can quickly switch to receiving in order to receive the messages.

When a user is in the talking state, the user appears offline to other users. This modification allows the user to receive messages only when not in talking state; messages, are therefore, not lost when the user are trying to send and not receive.

**Thread.** We change the thread so that each thread takes care of each users and exits only when the user disconnects from the server. The threads main job is to connect the user to the server and store the corresponding user’s socket number into the UserInfo struct.

The select statement controls the thread synchronization because the server will either be in a receiving state or a sending state, separating the concurrent reading and writing process.

**Functionality.** We also added switch statements at both the server and the client to process different COMMANDS. This helps control the flow of the program so that not all functionalities are meshed up together making it hard to understand and debug.