Updated Concurrency Argument

This Server is threadsafe due to 3 categories of threads which take care of separate tasks:

Main Thread - The main thread repeatedly reads requests from the main queue, handles them, and sends messages back to the clients involved through their sockets. This thread is also sequential in that only one request is handled by the server from the queue at a time, so there are no multiple accesses from the main thread. The requests that the main thread handles are those such as creating new conversation threads or sending the userList out to newly logged on users.

Client Threads - The client thread accepts connections from clients through sockets, waits for user to send a valid username and then logs them on, sending the new user startup data through their socket. Then the thread maintains a constant read from the client's socket, placing requests in the correct queue. Finally, when the user closes, the server handles the disconnect and updates the userList.

Conversation Threads – The conversation thread handles all the messages that are directed toward a conversation. When a message is received by a client thread it puts that request in the correct conversation queue (assuming it is not intended for the main queue).

Thread Interactions - Here are a list of the data fields used by either of the three threads and a note of their thread safety:

ConcurrentHashMap<String, Socket> socketNames - This HashMap is accessed by all threads, so we chose a ConcurrentHashMap. This class takes care of the thread safety in get, replace, and remove actions. It also allows for iterators to perform correctly over them by having it loop over the most recent completed copy of the HashMap.

ConcurrentHashMap<Integer, ServerConversationThread> convoThreads – Same as above

ConcurrentHashMap<Integer, ServerConversation> conversationMap – Same as above

ConcurrentLinkedQueue - This class is made to be accessed by multiple threads at once. Handles all concurrency issues within the class.

Sockets – All sockets can read and write simultaneously. Thus, since the sockets are always being read by only one thread (the client thread) we must ensure that there is never more than one write happening to a socket at a time. We did this by placing all socket write access into synchronized blocks synchronized over the socket. This way, they must attain the lock before executing the code and thus only one write operation at a time.