Concurrency Strategy

Client:

To prove that our code is free of race conditions, we simply need to consider the variables touched by multiple threads and show that they are accessed in a thread-safe manner. For reference, there are three threads in the client (besides the main thread, which does no important work) – the event dispatching thread, the socket input processing thread (CSocketInputWorker), and the socket output processing thread (SocketOutputWorker).

The BlockingQueue in the SocketOutputWorker class is accessed by both the SocketOutputWorker thread and the CSocketInputWorker thread, but the Java API specification for BlockingQueue guarantees that all accesses to it will be thread-safe. The ConversationListener map in the ClientInputProcessor class is accessed by both the CSocketInputWorker thread and the event dispatch thread, so we make it a thread-safe map using the Collections.synchronizedMap() method. The threads that access the ConversationListener map also call methods on the ConversationListeners in the map and the single UserStatusListener in ClientInputProcessor, but these methods modify object state – in this case, all of the GUI elements – only on the event dispatching thread using SwingWorkers or SwingUtilities.invokeLater(). We do not have separate model data structures whose thread-safety we have to worry about – all of the client-side data is stored in the GUI elements. This covers all important client-side state.

There can be no deadlocks in the client because there is no synchronization or locking whatsoever in the code. (The only synchronization is performed internally by Java API data structures, which we can assume to be free of deadlocks.)

Server:

We use the same approach as above to demonstrate the absence of race conditions in the client. For reference, there are 2N + 1 threads on the server, where N is the number of connected clients – a socket input processing thread for each client (SSocketInputWorker), a socket output processing thread for each client (SocketOutputWorker), and the server's main thread, which waits for incoming connections.

The major state on the server is found in the ServerInputProcessor instance's Conversation map, the SocketOutputWorker map, and the map (newly added to the design) from server-assigned temporary handles to SSocketInputWorkers whose clients have not claimed a handle. These maps are accessed concurrently by up to N SSocketInputWorkers, so we make them thread-safe using the Collections.synchronizedMap() method as described above.

The SSocketInputWorkers also access the Conversations in the Conversation map. A Conversation's ID is immutable, so we do not need to worry about that field. The only other field is the List of handles, which we can make thread-safe by calling the method Collections.synchronizedList(). Additionally, the SocketOutputWorkers in the SocketOutputWorker map are accessed by multiple threads, but their only state is the thread-safe BlockingQueue (see above section). Finally, by design, each of the SSocketInputWorkers in the SSocketInputWorker map described above can only be accessed by itself (when an SSocketInputWorker processes a REGISTER_HANDLE request and verifies that the submitted handle is available, it updates its own handle field), so race conditions cannot occur.

There can be no deadlocks for the same reason described in the above section.

Testing Strategy

General Testing Strategy:

Our first goal is to perform extensive manual tests on all user-facing parts of the client (see below for examples). Our design also allows for extensive automated testing. Our design enables easy automated testing of all message datatypes and message serialization and deserialization, which constitutes the bulk of both client and server logic. We can perform end-to-end tests of the server by either launching the server in regular form and using a mock client to send text messages to the server and analyze the responses, or by launching the server with mocked Sockets that have some preloaded data in their input streams and whose output streams we can read. We can perform some automated end-to-end testing of the client by mocking up ConversationListeners and UserStatusListeners (rather than using ConversationFrame and MainFrame, which have many GUI components that are difficult to automatically examine).

Possible Manual Tests:

- **Client.** *"The client is a program that opens a network connection with the IM server at a specified IP address and port number. The client should have a way of specifying the server IP, port, and a username. Once the connection is open, the client program presents a graphical user interface for performing the interactions listed below."*
  - Opening a program should give choices of Server IP, port, and username
  - Once those are filled out and submitted, Chat list GUI should appear
- **Server.** *"The server is a program that accepts connections from clients. A server should be able to maintain a large number of open client connections (limited only by the number of free ports), and clients should be able to connect and disconnect as they please. The server also has to verify that client usernames are unique and handle collisions gracefully. The server is responsible for managing the state of both clients and conversations."*
  - Check whether a user's online/offline status changes when closing the client and opening it back up again (and claiming the same handle as before).
  - Check what happens when client tries to claim an unavailable handle
- **Conversations.** *"A conversation is an interactive text-exchange session between some number of clients, and is the ultimate purpose of the IM system. The exact nature of a conversation is not specified (although the hints section details a couple of possibilities), except to say that it allows clients to send text messages to each other. Messaging in a conversation should be instantaneous, in the sense that incoming messages should be displayed immediately, not held until the recipient requests them. You should visually separate messages of different conversations (e.g., into distinct windows, tabs, panes, etc)."*
  - Multiple conversations on one screen visibly separated

    - Ability to add people to conversation
    - Check that message sent appears on both sender’s/receiver’s screen
    - Check that message sends to all people in multi chat
    - Check that when multiple conversations present on the server, the server forwards messages to the correct conversations
    - Check when that multiple clients opened on the same computer, the server sends messages to the right client
- **Client/server interaction.** *“A client and server interact by exchanging messages in a protocol of your devising — the protocol is not specified. Using this protocol, the user interface presented by the client should: Provide a facility for seeing which users are currently logged in; Provide a facility for creating, joining and leaving conversations; Allow the user to participate in multiple conversations simultaneously; Provide a history of all the messages within a conversation for as long as the client is in that conversation.”*
    - Check that everyone who is supposed to be online is online
    - Check that when a person leaves a conversation, the other people in the conversation are notified.
- **No authentication.** *“In a production system, logging in as a client would require some form of password authentication. For simplicity, this IM system will not use authentication, meaning that anyone can log in as a client and claim any username they choose.”*
    - Check that server allows clients to choose arbitrary usernames as long as they have not been claimed by another client