Ranna Zhou
Derek Vaughn
Jennifer Liu
rannaz-dvaughn-jkliu

6.005 Project 2: Instant Messaging
Milestone 2

CONCURRENCY STRATEGY

The Client

This class maintains two separate threads, one for sending requests to the server and one for receiving them. To update the Conversation model with the responses from the server, we instantiate a SwingWorker for each. That way the Client can keep processing server responses as they are received without being backlogged. The SwingWorker holds a lock on the Conversation model while updating it to prevent more than one thread being able update a conversation a time and thus to prevent race conditions. The Conversation models are only modified when server requests are handled--not when requests are sent to the server--thus each Conversation is modified sequentially, ensuring thread-safety.

The Server

****NOTE**** Tuesday May 8th we decided to change the server somewhat to introduce concurrent threads in the server (one for each conversation) to improve performance. Since we had this design change the day of this deliverable, the Concurrency Strategy for the Server below is for when there was only 1 thread in the server handling requests. When the design of the new server is complete we will update Milestone 2 folder with a revised Server Concurrency Strategy PDF.

This Server is thread safe due to 2 categories of threads which take care of separate tasks:

-**Main Thread** - The main thread repeatedly reads requests from the 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.

-**Client Thread** - 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 queue. Finally, when the user closes, the server handles the disconnect and updates the userList.

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

--ConcurrentHashMap<String, Socket> socketNames - This HashMap is accessed by both 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. The link for ConcurrentHashMap is below.
http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentHashMap.html

--HashMap<int, Conversation> - This is only accessed by the Main Thread in one of the handleX() functions. Since no multiple accesses, safe here.

--HashSet<Conversation> - This is only accessed by the Main Thread in one of the handleX() functions. Since no multiple accesses, safe here.

--ConcurrentLinkedQueue - This class is made to be accessed by multiple threads at once. Handles all concurrency issues within the class. The link for ConcurrentLinkedQueue is below.
http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html

TESTING STRATEGY

Basic cases to test:

- Multiple users logging on
- Starting a conversation, exiting a conversation, and logging off (making sure all users online are updated)
- All users exiting a conversation (make sure it is removed from the server)

Edge cases to test (should throw user-friendly error messages):

- User tries to sign on with a username that is already taken
- User tries to sign on with a username with invalid characters
- User enters invalid server IP
- User tries to start a conversation with someone with whom they are already engaged in a 1-on-1 conversation
- User tries to start a conversation with him/herself
- User tries to add a person to the conversation that is already in the conversation

For the Server:

There is a ServerClientThread to receive client requests and a ServerMainThread to process these requests. We will use the mock/stub testing strategy and create trivial clients to send requests to the server to test queueing and check that the server handles and responds to these requests correctly. For the backend of the server, we will write white box tests to test that the server keeps track of the users in a conversation and the users online throughout.
Finally, our end-to-end testing consists of testing interactions with the server from separate computers on different operating systems. For instance, having the server hosted on computer A and having a conversation between computer B & C.

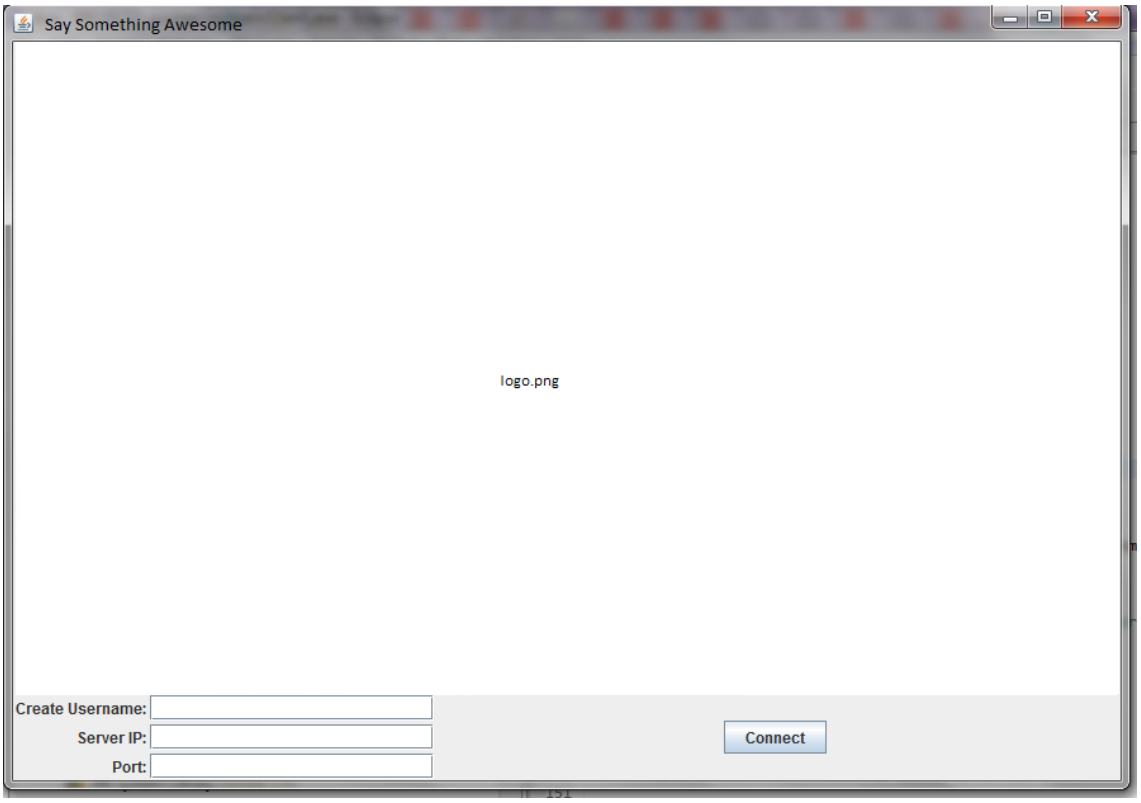
For the Client:

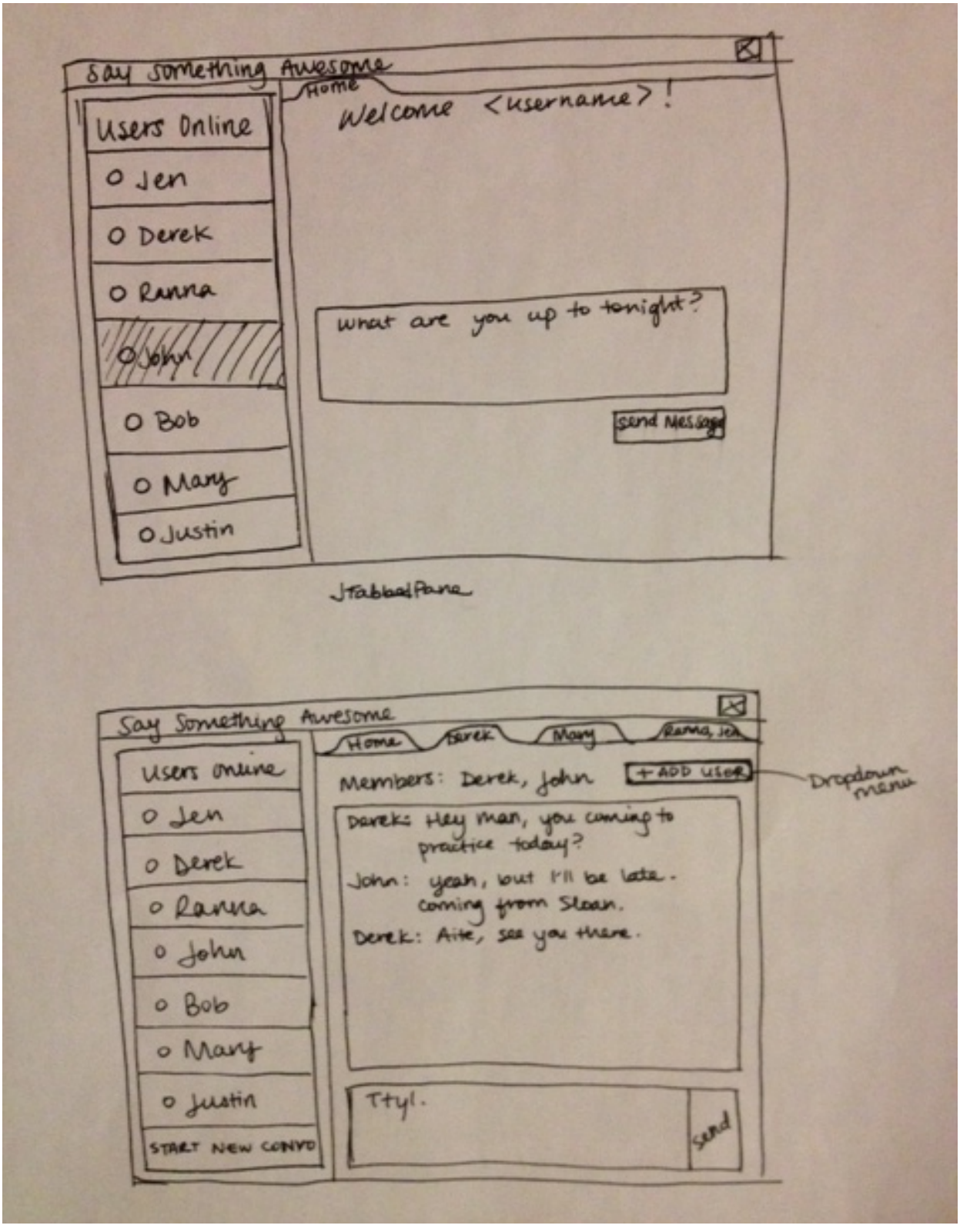
In order to test the Client, we have a series of JUnit tests to ensure that the ServerRequests (UserList, LogOnUser, LogOffUser, AddUser, RemoveUser, and Message) are all being parsed and handled properly, and that the Conversation models are being updated accordingly. We will also write JUnit tests to check if the message from the client to the server are being formatted and sent properly. Additionally, we will use JUnit tests to ensure that the checkUsernameAvailability method works.

For the GUI:

We will test the GUI using FEST to simulate performing various tasks, and to make sure that no matter what, the UI is responsive at all times. This is where we will also test the full implementation of the chat client. We will also manually test the GUI on multiple computers to make sure it is functional and user-friendly.

UI SKETCHES




Say Something Awesome
Home
Users Online
Jen
Derek
Ranna
John
Bob
Mary
Justin
Welcome <username>!
what are you up to tonight?
send Message
JTabbedPane
Say Something Awesome
Home
Derek
Mary
Ranna, Jen
Users Online
Jen
Derek
Ranna
John
Bob
Mary
Justin
START NEW CONVO
Members: Derek, John
+ ADD USER
Dropdown menu
Derek: Hey man, you coming to practice today?
John: yeah, but I'll be late. coming from Sloan.
Derek: Aite, see you there.
Ttyl.
send