Design Milestone 2 -- UPDATED

Sresht Rengesh (sresht)
Jiarui Huang (jiarui)
Henry Nassif (hnassif)

Testing Strategy – Chat via Telnet

We will design tests holistically to test logging in, logging out, creating chatrooms, joining chatrooms, and sending messages. We will also test invalid commands (nonsensical commands as well as incorrectly-formatted commands), and invalid combinations of commands (for example, logging out before logging in). Below is a list of 19 holistic tests to ensure that both the client and the server are functioning as we expect it to. We add a checkmark when we manually test these using telnet and multiple terminals. We wrote JUnit Test Suites to run tests on Time (which is our method to create a timestamp) and Conversation (which initializes and modifies chatrooms). The rest of our testing was done manually.

Finally, we test as many of these things as are allowed by our GUI, and we will put a red check when they pass on the GUI.

We performed a code reviewing session with another team upon the completion of our chat client in which we observe actions that the other team takes in running through our client. We improved components that proved difficult to use (such as our initial chatroom that didn't have a send message button), and take away unnecessary components of our design. Finally, we tested other people in order to get a better idea of alternative implementation strategies and to observe differences in their implementations.

Our tests are divided into three different parts:

a) valid commands,
b) invalid commands that should return an error message but re-prompt the user, and
c) invalid commands that kill the socket.

Type A Tests

1. Login with valid username ✓ ✓
2. Login with uppercase username ✓ ✓
3. Login, then logout, then login again ✓ ✓
4. Create chatroom successfully ✓ ✓

5. Create chatroom and then leave chatroom ✓✓
6. Create chatroom and then logout ✓ ✓(destroys the chatroom on the GUI)
7. Create >1 chatrooms by the same person ✓ ✓
8. Have three users make mutual chatrooms ✓ ✓
9. Create chatroom 1, leave chatroom 1, create chatroom 2 ✓
10. Type messages to each other (sub-tests included below)
    a. Have three chatrooms, in which three users will have mutually exclusive conversations (i.e. chatroom 1: AB, chatroom 2: BC, chatroom 3: AC, and send messages back and forth) ✓✓
    b. Have three chatrooms, in which three users will all participate in all three chatrooms ✓ ✓

Type B Tests

11. Nonsense keywords (ex: die, disconnect, kill, etc) ✓
12. Login with valid username then try logging in again (ignore the second attempt to login, regardless of whether it's valid or not) ✓ ✓
13. Create chatroom with an invalid name specification (i.e., Henry tries to create chatroom under Sresht's name) ✓
14. Send message with an invalid name specification ✓
15. Leave chatroom in which the user is not already participating ✓
16. Send message to a chatroom that the user is not in ✓
17. Specify non-numerical chatroom ID (i.e., create foo Henry) ✓ ✓
18. Try to join the chatroom when the user is already in the chatroom ✓ ✓
19. Logout before logging in ✓

Type C Tests

*We decided to close the socket on these invalid commands because these would result in our login screen returning a warning and then clearing the text input boxes. In essence, the GUI would "refresh" upon running into these errors without actually creating a socket for the user. Therefore, we should not have these errors maintain a socket.

20. Login with invalid username ✓ ✓
21. Login with taken username ✓ ✓

Testing Strategy – Graphic User Interface

Before testing the GUI, we want to ensure that the entire server-side design is working as expected. Now, we can turn our attention to GUI testing.

Firstly, we will ensure that **all JButton objects can be clicked**, all JTextField objects have appropriate eventListeners when the user pressed the return. ✓

Secondly, we will ensure that the eventListeners **correspond to the right action**. ✓

Regarding design choices, we will allow for the window to **resize appropriately unless resize is disabled**, and make sure that there are no overlapping or conflicting problems on the interface when the window is resized. ✓

To ensure that, we will also make sure that **everything is appropriately sized and spaced**. ✓

Since we have a login screen, we will test to make sure that **appropriate logins work while inappropriate logins don't**, and instead clear the input JTextField and also return a warning message. ✓

When a user opens multiple chats, we will ensure that the newest chat window is created and does not overwrite the existing chat window, but instead creates a new window. ✓

The following tests are sequence-oriented GUI tests (which we will test manually when we build the GUI):

- Login with invalid username (should return a warning as a popup or text message) ✓
- Login and then force close out of the GUI, and then login again ✓
- Login, create a chatroom, force close out of the chatroom, then create the same chatroom (we couldn't get this to work. See future improvements)
- Login, create chatrooms, logout (which should close all existing chatrooms), then login with the same username and then create the same chatrooms ✓

Concurrency Strategy

To deal with concurrency, we have to ensure that only one thread is accessing each variable at a time; in other words, no more than one thread is accessing the same object simultaneously. Therefore, our strategy is to confine variables to be used only within the class and lock every mutable object when a thread tries to access it.

Server-side concurrency

There are two types of processors running on the server,

1. Server thread (Server.java: main thread)
2. Client thread (Client.java: thread handler)

Each main thread initializes a client handler once a user is connected to the socket to handle request from the user. Thus, while there is only one server for each session, there are a number of client threads running in parallel with the server. Each client thread has a reference to the server so that they can communicate with the server and tell the server to perform some operations according to the messages it receives from the user it is handling.

In order to achieve our goal, we synchronize every mutable object in the Server once a thread accesses it, namely activeUsers and allConversations (We don't need to worry about serverSocket and a debug since they are immutable) assuming that our Conversation class is thread-safe (described in the next paragraph). We also don't need to worry about the client thread since once it is initialized, the no other thread will touch it again and all the variables are confined within itself. We will not let a client thread retrieve references to these object directly; it can only access the data using server's method. Therefore, if we synchronize every mutable object that gets accessed at the time, we will be guaranteed that our server is thread-safe.

We also need to make sure that our conversation class is thread-safe. We also achieve this by synchronizing every mutable object within the class and allowing no other object to retrieve the reference to the object.

Since we are using multiple locks, we always apply the lock in the same order to prevent deadlock. Foe example, in Server.java, we always lock activeUsers first and then allConversations if two of them get accessed at the same time.

Client-side concurrency

We consider using two types of threads to implement client applications:

1. Client Model and GUI thread – the main client thread that stores all the data and mange GUI
2. Client Listener – a thread that handles requests from the server

The main strategy is the main client thread (Client Model and GUI thread) initializes a Client Listener thread to handle requests from the server. Receiving a message from the server, a Client Listener thread processes it and tells the model how it should process the request (this is similar to the client handler thread in the server.) The main client thread updates the client model and GUI accordingly. On the other hand, GUI thread handles actions performed by user using ActionListeners.

We stick with the same concurrency strategy: only one thread can access a variable at a time. Therefore, we lock every shared data using synchronized keyword and make sure that there is no deadlock.

Because the server is concurrent, it is unnecessary to use SwingWorker (an atomic version of Java Swing) for our GUI.

Testing Strategy - Concurrency

We tested multiple scenarios that can lead to concurrency problems with the built-in Eclipse debugger. However, we did not have enough time to test concurrency using JUnit tests. Therefore, we have documented several different tests that we would have done through JUnits if we had more time.

- 2 people try to login with the same username at the same time. We will make sure only one of them can login with that username and the other is prompted to choose a different username.

- 2 people create a chatroom at the same time. This should allow both users to join that chatroom without any problems.

- 2 people send messages to one another at the same time. We will make sure both receive the message destined to them.

- 2 people send a message to a third person at the same time. We will make sure that the third user receives both messages.

-2 people logout at the same time. We will make sure that all their conversations are cancelled.

-One user logs out and one user logs in with that same username at the same time. This will be processed in one of two orders: either everything will work as if the first user logged out first, or the second user will be prompted to try a different username and then the first user will be logged out. Then, if the second user tries the same username some short period of time later (as long as it takes to process a logout request), he will be allowed to successfully login.

Future Improvements

While the project and our implementation was overall a success, we wish that we could have had enough time to write a more sophisticated JUnit suite. We provide JUnits to make sure that the different components of the program work and we test to make sure that the entire program works using Telnet and the GUI. However, we did not have time to implement a full suite of tests to our content, and instead just outlined what we would have done.