Concurrency Strategy

Server

- Maintains a ConcurrentHashMap of client usernames to client userInfo objects which contain the client's username, color, and socket
    - ConcurrentHashMap allows for synchronization in the case of searching for an element which is not there (once the item is not found, the map is synchronized, the item is looked for again (in case it was added before the synchronization) then the item is added)
    - For most other methods, the map is not needlessly locked - only the individual buckets are locked by threads accessing them - this eliminates needless overhead encountered when synchronizing a regular HashMap
    - Iteration through a ConcurrentHashMap ensures any updates or removals of items will be correctly reflected in their behavior
    - Performs better than a HashMap with synchronization under load which ensures our system is scalable while maintaining thread safety
- Every function that reads from or writes to a socket first synchronizes the socket - this eliminates race conditions such as the possibility of sending a message to participants in the conversation as a participant is signing off
- We will set an order to locking the server socket then the client socket so no deadlock will occur

Client

- Handling the client's connection happens in two steps to eliminate duplicate username issues:
    - the client connects via a socket and requests to login with a desired username and color
        - if the server sends a message that the username is taken, the socket is closed and can try to login again from the login screen
        - if a message with the online users is reported, the client adds these usernames to its record of online users (a ConcurrentHashMap of string usernames to userInfo objects - the ConcurrentHashMap eliminates potential concurrency issues from the user's connection handler and the GUI which run on separate threads)
    - once the client successfully chooses a username, the main connection handler is run from the User class and the ConversationView is opened
- The synchronization of Conversation convo of all methods that access (to prevent display inconsistencies) and mutate the convo (by adding messages) ensures the integrity of a user's Conversations
- The client system is free deadlock because no cycles can exist since we are only synchronizing Conversations which do not access other Conversations

Testing Strategy

JUNIT Method Tests:

ChatClient.java

- Method: `ChatClient()`
    - Strategy: cannot be tested effectively with unit test since makes GUI and other stuff, so will test in System test
- Method: `setUser(String username, String color, Socket socket) {`
    - Strategy: will test with mock socket instance and check that the user is set
- Method: `attemptLogin(String IP, String port, String username, String color)`
    - Integration testing with Server - run ChatServer and create ChatClient and try to call this method with both unique and duplicate username (partitioning input space)

ClientThread.java

- Method: `ClientThread(Socket socket)`
    - Strategy: unit test with mock socket object
- Method: `run()`
    - Strategy: difficult to test in isolation so will use Integration testing by making sure it works in conjunction with User

Conversation.java

- Method: `Conversation(ConcurrentHashMap<String,UserInfo> participants)`
    - Stategy: create new Conversation with mock participants map, assert the instance variables are set correctly
- Method: `addMessage(Message msg)`
    - Strategy: add a message to a Conversation and access last message to verify it matches
- Method: `getMessages()`
    - Strategy: get messages from a Conversation
- Method: `getConvoID()`
    - Strategy: make sure a convoID is returned as expected
- Method: `alphabetizeHashMap(ConcurrentHashMap<String, UserInfo> participants)`
    - Strategy: partition input space: two participants with same first letter(s) (at least one letter must eventually be different since we use it for chat users who must have unique names), test adding person with later letter to particpants first then adding the alphabetically first person second to participants, test adding participants in alphabetical order then running the method, similarly test with mixture of capital letters bearing in mind all capital letters come before any lowercase letters

ConversationView.java

This is the main component of our GUI and is what allows users to send and receive Instant Messages. We manually tested it by:

- Making sure if a person logging in is not the first person, the other online users are displayed correctly
- Making sure that users who login when there are already people logged in show up on the previously logged in peoples' user lists
- Making sure that when somebody logs out they disappear off the other user's user list and open conversations with the logging out person end on the receivers' sides
- Making sure that a single conversation between two logged in users can happen (history filled appropriately and displayed correctly)
- Making sure that conversations involving two users can happen and that on all users' sides, the correct messages are displayed in the correct panes
- Making sure that duplicate conversations cannot happen - this is why the number of users in a conversation is fixed: if a conversation starts out with three people and one person exits or logs out, the conversation should be terminated rather than becoming a conversation with the two remaining users otherwise there will be a duplicate conversation which we decided against allowing for sake of clarity
- Making sure that conversations with more than two participants happen correctly (i.e. messages displayed correctly etc.)

GUIThread.java

This will be tested in the system test by ensuring that despite all possible actions, none of the GUI components hangs

LoginView.java

Since this is a GUI component it will be tested manually by:

- Trying to login to a running server with appropriate login info and make sure ConversationView opens and login screen closes
- Trying to login with invalid IP address and making sure dialog with Invalid IP message is displayed and can proceed to login with correct info
- Trying to login with invalid port number and making sure Invalid Port dialog shows up and can proceed to login with correct info
- Trying to login with a duplcate username and making sure Username already in use dialog shows up and can login with proper username
- Testing getting ALL the dialog boxes then still being able to login afterwards with correct login info

Message.java

- Method: `Message(UserInfo sender, Conversation convo, String text)`
  - Strategy: make sure things are set correctly
- Method: `getSender()`
  - Strategy: make sure sender is returned
- Method: `getConvo()`

    - Strategy: make sure the Conversation being returned is the same instance as the one the Message was made with
- Method: `getText()`
    - Strategy: make sure correct text returned

TabPanel.java

Since this is a custom JComponent, it is difficult to test in isolation and will be tested in Integration testing with ConversationView. We will make sure the appropriate conversations show up in the panels and are updated properly.

User.java

- Method: `User(String username1, String color1, Socket socket1)`
    - Strategy: est creation of a User with mock socket and ensure things are set correctly
- Method: `main()`
    - Strategy: since main creates a new thread for the client connection, it is difficult to test in isolation; therefore, this will be tested via the System test and with Integration tests with the Server
- Method: `handleConnection(Socket socket)`
    - Strategy: since this method is pointless if the socket has no server to communicate with, unit tests are impractical so it will be tested in Integration tests with the Server
- Method: `handleRequest(String input)`
    - Strategy: partition the input space and test all possible inputs that do not affect the GUI and use Integration testing with ConversationView
- Method: `sendMessageToServer(String text)`
    - Unit test by having something listening to a mock socket output
- Method: `startConvo(Object[] usernames)`
    - Strategy: unit test that tests the correct message is returned when making a convo from username objects
- Method: `closeConvo(Conversation convo)`
    - Strategy: unit test that the right message is sent from the socket given a conversation
- Method: `updateConvo(String input)`
    - Strategy: this involves updating the ConversationView so must be Integration tested with ConversationView
- Method: `login()`
    - Strategy: make sure appropriate message sent to socket outputstream
- Method: `quit()`
    - Strategy: since this accesses the UserGUI, shoiuld be tested with Integration tests with the GUI
- Method: `setActiveConvo(Conversation convo)`

    - Strategy: make sure the active convo is set to the correct Conversation
- Method: `setOnlineUsers(ConcurrentHashMap<String,UserInfo> userMap)`
    - Strategy: make sure correct users are set
- Method: `addOnlineUser(UserInfo user)`
    - Strategy: ensure users added correctly
- Method: `removeOnlineUser(String user)`
    - Strategy: test that user can be removed from online user list properly
- Method: `getUsername()`
    - Strategy: assert correct username returned
- Method: `getColor()`
    - Strategy: assert correct color returned
- Method: `getSocket()`
    - Strategy: assert correct socket instance returned
- Method: `getOnlineUsers()`
    - Strategy: assert that online users returned
- Method: `getMyConvos()`
    - Strategy: assert correct conversations returned
- Method: `addNewMyConvo(Conversation convo)`
    - Strategy: since this accesses ConversationView needs to be Integration tested with ConversationView
- Method: `removeMyConvo(Conversation convo)`
    - Strategy: make sure correct conversation removed from convos
- Method: `checkDuplicateConvo(Conversation convo)`
    - Strategy: unit test to test expected DuplicateConvoException and to test unique conversation passes (partitioning input space)

ChatServer.java

- Method: `addUser(String, String, Socket)`
    - Strategy: Make a mock socket, input a username string and color, and see if the correct ServerMessage arrayList is returned
- Method: `closeConvo(String)`
    - Strategy: input a valid CLOSE_CONVO message and see if it returns CLOSE_CONVO messages for all other users involved in the convo
- Method: `everyoneButMe(String)`
    - Strategy: input a username String and see if all other “logged on” users have a ServerMessage returned - should be null. Also will need to be tested in an integration / system test.
- Method: `everyoneInConvoButMe(String convoId, String username)`
    - Strategy: make mock sockets and userInfos in an infoMap and make sure that the sockets not associated with username are returned

- Method: `getInfoMap()`
    - Strategy: make mock infoMap, set, and make sure it is returned
- Method: `closeConvo(String)`
    - Strategy: make sure appropriate message sent to socket outputstream of appropriate sockets
- Method: `handleClientRequest(String input, Socket socket)`
    - Strategy: Partition the input space and check that appropriate response returned for all testable input
- Method: `handleConnection(Socket socket)`
    - Strategy: Cannot be tested with unit tests effectively but tested in System test
- Method: `justMe(String)`
    - Strategy: ensure socket returned corresponds to username given as input
- Method: `logout(String)`
    - Strategy: ensure that username is properly removed from the infoMap, and that the ServerMessage returned contains the recipients, flag, and username
- Method: `main(String[] args)`
    - Strategy: difficult to test with unit test, but tested in System test
- Method: `runChatServer(int)`
    - Strategy: Cannot be tested with unit tests effectively but tested in System test
- Method: `startConvo(String)`
    - Strategy: Ensure the correct ServerMessage is generated including correct recipients and message
- Method: `updateConvo(String message)`
    - Strategy: unit test to make sure correct messages returned to be sent to right sockets
- Method: `serve()`
    - Strategy: Cannot be tested with unit tests effectively but tested in System test

ServerMessage.java

- Method: `getRecipients()`
    - Strategy: Make a ServerMessage object and make sure it returns the same Sockets you put in
- Method: `ServerMessage(ArrayList<Socket>, String)`
    - Strategy: Make various ServerMessages with mock Sockets and text Strings. Mostly to check for no errors.
- Method: `getText()`
    - Strategy: Make a ServerMessage object and make sure it returns the correct text
- Method: `toString()`
    - Strategy: Make a ServerMessage object and call toString on it, make sure it prints reasonably and in a user-friendly way

System (Integration) Tests:

JUNIT

- Tests running the server and one or more clients
- We will test all possible interactions between the server and client, which are detailed in the "Client/Server Protocol" section below, and ensure that the server and client respond with the appropriate messages dictated by our protocol.
- In the case where these messages correspond to updates in the GUI, we will verify the appropriate changes visually, as described below.

Visual

- In addition to the JUNIT tests, we will visually test the GUI to ensure the user experience matches our specifications.
- Below in the User Experience section we have outlined the behaviour of our GUI in response to any inputs from the User. We will perform User testing on our GUI by stepping through all viable combinations of user inputs to ensure the behavior matches what we expect.

User Experience

Login view presented on startup

- Text box: Server IP
- Text box: Port number
- Text box: Username entry
- Button: Login
    - Action: Checks for valid IP, port, username
        - If valid: Go to conversation view
        - Else: Error popup, returns to login screen.
- Color selection

Conversation view

- Tabbed view
    - Tabs for each open conversation (on the left)
    - Frame containing active conversation's history
- Text field with send button for new messages
- New conversation starter
    - Select recipient from dropdown list of online users
- Buttons to close open conversations
- Logout button
    - Logs user out and returns to login view

Classes

User: Object representing user of a client

- `public User(String username, Color color, Socket socket);`
    - @param username, a String representing the User's name

    - @param color, a Color associated with the User in the UI
    - @param socket, the Socket associated with this User
- `private ConcurrentHashMap<String, UserInfo> onlineUsers;`
    - Maps username strings to UserInfo object containing info about all other online users.
- `private Conversation activeConvo;`
    - The current conversation the user has open.
- `private ConcurrentHashMap<String, Conversation> myConvos;`
    - Maps convoID strings to Conversation object for all active conversations this User is part of.
- `private ConcurrentHashMap<String, Color> colorMap;`
    - Maps String color names to actual Color objects
- `public void main() throws IOException;`
    - calls handleConnection
- `public void sendMessageToServer(String text);`
    - Sends text to the server, using this.socket.
- `public static void handleConnection() throws IOException;`
    - Handle connection to the server, using its this.socket
    - Calls handleServerRequest()
    - @throws IOException if connection has an error or terminates unexpectedly
- `public static String handleServerRequest(String input);`
    - handler for server messages. Update conversations, etc based on the message received.
    - @param input, the input from the server, from the grammar
    - @return the message (from the grammar) to the client
- `public String getUsername();`
    - @return String representing the User's username
- `public Color getColor();`
    - @return Color representing the User's color
- `public Socket getSocket();`
    - @return Socket representing the User's socket
- `public ConcurrentHashMap<String, UserInfo> getOnlineUsers();`
    - @return ConcurrentHashMap mapping usernames of all online users to their UserInfo objects
- `public ConcurrentHashMap<String, Conversation> getMyConvos();`
    - @return ConcurrentHashMap mapping convoID's to Conversations
- `public void setActiveConvo(Conversation convo);`

    - updates the stored value for current conversation
    - @param convo, the new active conversation
- `public void startConvo(Conversation convo);`
    - call sendMessageToServer("-s" + convo.convo_id)
- `public void closeConvo(Conversation convo);`
    - call sendMessageToServer("-x" + convo.convo_id)
- `public void addMsgToConvo(Conversation convo, String text);`
    - call sendMessageToServer on the text from the message, the convo_id, and the username, according to the grammar.
- `public void quit();`
    - call sendMessageToServer("-q" + this.username) and closes its socket's connection.
- `public void setOnlineUsers(ConcurrentHashMap<String,UserInfo> userMap);`
    - Replaces the onlineUsers ConcurrentHashMap with userMap.
- `public void addOnlineUser(UserInfo user);`
    - Adds user to the onlineUsers ConcurrentHashMap.
- `public void removeOnlineUser(UserInfo user);`
    - Removes user from the onlineUsers ConcurrentHashMap.

Message: Object representing an instant message

- `public Message(UserInfo sender, Conversation convo, String text);`
    - @param sender, the UserInfo of who sent the message
    - @param convo, the conversation the message is part of
    - @param text, the body of the instant message
- `private final UserInfo sender;`
    - the sender of the message
- `private final Conversation convo;`
    - the conversation the message is part of
- `private final String text;`
    - the text of the message
- `public UserInfo getSender();`
    - returns sender of message
- `public Conversation getConvo();`
    - returns conversation
- `public String getText();`

    - returns text

Conversation: Object representing a Conversation

- `public Conversation(ConcurrentHashMap<String, UserInfo> participants);`
- @param participants, ConcurrentHashMap mapping usernames to UserInfo for all Users who are participating in the conversation
- `String convoID;`
    - Identifier for conversation
    - Format: usernames of conversation participants in alphabetical order, separated by spaces.
- `ConcurrentHashMap<String, UserInfo> participants;`
    - Username mapping to UserInfo for everyone participating in the Conversation
- `List<Message> history;`
    - ArrayList of all messages in the conversation
- `public List<Message> addMessage(Message message);`
    - @param message, new message that will be added to the history.
- `public void getMessages();`
    - @returns the history
- `public String getConvoID();`
    - @returns the convoID
- `public ConcurrentHashMap<String, UserInfo> getParticipantsMap();`
    - @returns the HashMap of participants to their UserInfo.

ChatServer:

- `public ChatServer(int port) throws IOException;`
- `private ConcurrentHashMap<String, UserInfo> infoMap;`
    - maps String usernames to UserInfo objects which have a record of the user's relevant information (name, color, socket used for communication)
    - uses threadsafe implementation of HashMap
- `private ConcurrentHashMap<String, Color> colorMap;`
    - maps strings
- `public void serve() throws IOException;`
    - Run the server, listening for client connections and handling them. Never returns unless an exception is thrown.
    - Uses a ServerThread class that extends runnable and calls

handleConnection in its run() method to make a new thread each time a new client connects.
    - @throws IOException if the main server socket is broken.
- `public static void handleConnection(Socket socket) throws IOException;`
    - Handle a single client connection. Returns when client disconnects. Calls handleClientRequest()
    - Sends messages to clients based on the ServerMessages returned by handleClientRequest()
    - @param socket socket where the client is connected
    - @throws IOException if connection has an error or terminates unexpectedly
- `public static ServerMessage handleClientRequest(String input, Socket socket) throws IOException;`
    - handler for client input. Make requested mutations and returns appropriate Server Message which needs to be returned to clients.
    - @param input, the input from the client, from the grammar
    - @return a ServerMessage containging message and clients which need to receive it
- `public static void main(String[] args)`
    - Start a server running
    - @param args contains the desired port for the server
- `public static void runServer(int port) throws IOException`
    - start a server running on specified port
    - @param port, the port to use
- `private ServerMessage addUser(String username, String color, Socket socket)`
    - adds a user to the infoMap unless it is a duplicate username, in which case it sends a INVALID_USER message
    - @param username, the potential username
    - @param color, the color
    - @param socket, the socket this user is connected by
    - @return INVALID_USER message to just me if username already in use, ONLINE_USERS message to everyone but me if username is valid
- `private ServerMessage logout(String username)`
    - deletes a user from the infoMap.
    - @param user, the username to be deleted
    - @return OFFLINE message to everyone but me

- `private ServerMessage startConvo(String message)`
    - @param message, the START_CONVO message
    - @return START_CONVO message to everyone in convo but me
- `private ServerMessage updateConvo(String message)`
    - @param message, the ADD_MSG message
    - @return UPDATE message to everyone in convo but me
    - called when the server receives a ADD_MSG message.
- `private ServerMessage closeConvo(String message)`
    - @param message the CLOSE_CONVO message
    - @return CLOSE_CONVO message to everone in convo but me
- `private static ArrayList<Socket> justMe (String username);`
    - makes an ArrayList of sockets to send to if just returning to same client
    - @param username the username of the client to send to
    - @return ArrayList containing socket just of that username
- `private static ArrayList<Socket> everyoneButMe(String username);`
    - makes an ArrayList of sockets for everyone else but the username
    - @param username the username not to receive the message
    - @return ArrayList with sockets for everyone else
- `private static ArrayList<Socket> everyoneInConvoButMe (String convoID, String username)`
    - makes an ArrayList of sockets for everyone else in conversation but username
    - @param convoID the conversation to be updated
    - @param username the username that shouldn't be updated
    - @return ArrayList withs ockets of everyone relevant

ServerMessage: Object to handle sending a message to multiple clients.

`public class ServerMessage(ArrayList<Socket> recipients, String text);`

- `private final ArrayList<Socket> recipients;`
    - a list of Sockets to send the message to
- `private final String text;`
    - the message to be sent

UserInfo: Object to hold information about other users. First constructor is what the ChatServer class uses for its infoMap, and the second is for Users to keep track of other Users' names and colors.

`public class UserInfo(String username, String color, Socket socket);`

`public class UserInfo(String username, String color);`

- `private final String username;`
    - the username of the user
- `private final String color;`
    - the color associated with the user
- `private final Socket socket;`
    - the socket used by the user
- `public String getUsername();`
    - @return the username
- `public String getColor()`
    - @return the color
- `public Socket getSocket()`
    - @return the socket

UserGUI:

`public class UserGUI extends JFrame;`

- `public UserGUI(User user);`
    - `private final User user;`
        - `the User associated with the GUI`
- `public MainWindow()`
    - Creates a LoginView, displays to User
    - When confirmation is received from server, creates a ConversationView and displays to User
- Has ActionListeners for each field that call methods in User to relay the information to the server.
- Ex:
    - Listener for the “Send” button: calls AddMsgToConvo, using user.activeConvo as the Conversation and the text in the text field as the text

LoginView:

`public class LoginView extends JPanel;`

- `private final JTextField ipAddress;`
- `private final JTextField portNumber;`
- `private final JTextField username;`
- `private final JComboBox colorDropDown;`
- `private final JDialog errorDialog;` //in case of invalid username

ConversationView:

`public class conversationView extends JPanel;`

- `private final JTabbedPane tabby;`
- `private final JScrollPane scrolly;`
- `private final JTable messages;`
- `private final JButton logout;`
- `private final JLabel newConvoLabel;` //"select a user to chat with"
- `private final JComboBox userlist;`
- `private final JButton newConvoButton;`
- Appropriate listeners for each of these JComponents

Client/Server Protocol

Client to server:

MESSAGE :== [ADD_MSG START_CONVO CLOSE_CONVO LOGIN QUIT] "\n"
LOGIN :== -l USER_INFO
QUIT :== -q USERNAME
ADD_MSG :== -c CONVO_ID -u USERNAME -t TEXT //-u is the sender
START_CONVO :== -s CONVO_ID -u USERNAME
CLOSE_CONVO :== -x CONVO_ID -u USERNAME
CONVO_ID :== USERNAME+ //alphabetized
USER_INFO :== USERNAME COLOR
USERNAME :== [a-zA-Z]{1,10} //10 character limit, no numbers
COLOR :== ["red" "orange" "yellow" "green" "blue" "pink"]
TEXT :== [^NEWLINE]+
NEWLINE :== "\r?\n"

Server to client:

MESSAGE :== [START_CONVO UPDATE CLOSE_CONVO ONLINE OFFLINE INVALID_USER ONLINE_USERS] "\n"
ONLINE_USERS :== -f (USER_INFO)* //list of all online users' names and colors
INVALID_USER :== -i USERNAME //error sent if username is already in use
ONLINE :== -o USER_INFO //lets clients know a new user is online
OFFLINE :== -q USERNAME //lets clients know a user has logged off
START_CONVO :== -s CONVO_ID -u USERNAME//sent to recipient when new convo started
CLOSE_CONVO :== -x CONVO_ID -u USERNAME //sent to other participant when one

person exits
UPDATE :== -c CONVO_ID -u USERNAME -t TEXT //-u is the sender
CONVO_ID :== USERNAME+   //alphabetized
USER_INFO :== USERNAME COLOR
USERNAME :== [a-zA-Z]{1,10} //10 character limit, no numbers
COLOR :== [“red” “orange” “yellow” “green” “blue” “pink”]
TEXT :== [^NEWLINE]+
NEWLINE :== “\r?\n”