Thread Safe Chat-Server

1.1 Design Summary

Our chat server works by forking two threads. One thread is responsible for dispatching messages to users and groups, ensuring they are delivered in correct order. The other thread is responsible for the management of users and groups, including tasks such as addition, creation and removal.

1.2 Interaction With the Chat Server

Below are the variables/data structures/classes/methods we will be adding.
`UserManager` is the class responsible for management of users and groups.

```
Variables added:
HashMap (synchronized) - userHash: A mapping of usernames to threads.
    Used to keep track of all users current logged onto the server.
HashMap (synchronized) - groupHash: A mapping of group names to Group objects.
    Used to keep track of all current groups on the server.

Methods added:
addUser, addUserToGroup, removeUserFromGroup, listGroups, createGroup,
deleteGroup, listUsersOfGroup, removeUser, getUserByName, getNumberOfUsers, getListOfUse
```

`MessageDispatcher` is the class responsible for dispatching messages.
Variables added:

```
Queue - messageQueue: A queue of Message objects in the order of delivery.
```

Methods added:

```
enqueue(message): enqueues a message in the messageQueue
```

1.2.1 Design Decisions

1.2.2 Tests

1.3 Message Delivery

To address the feature of message delivery, we have created a Message class.
`Message.java`

```
String - receiver
String - sender
TimeStamp - timestamp
Boolean - error
```

The `error` boolean is set to true when a message is not successfully delivered.

1.3.1 Design Decisions

1. If we did not create a class to group those fields, then during message delivery programmer would have to keep track of all four fields. This is very inconvenient.
2. We could have created an array to hold those fields; however, using array requires programmer to remember which index represent what field. If in the future, the list of fields grows, an array becomes a less usable data structure to use.
3. The use of Message class allows programmers to encapsulate the data structure used to represent those fields.

Upon constructing a message object, the message constructor shall take in three parameters: String sender, String receiver, and String text. These parameters will initialize their respective instance variables. The constructor will also initialize the timestamp instance variable to the time the message was created.
The error boolean variable, when set to true will signal that this message was not successfully received. This variable is set to false when message object is initially instantiated.

1.3.2 Features

- A user can send a message to another user
- A user receives messages successfully from individual users
- Sending a a message to a user that does not exist should fail explicitly and should mark the message in the log to be error.

************* WE NEED TO EXTEND BASEUSER CLASS =/ *********************
When we create a user object, we need to pass two parameters to the constructor: String name, ChatServer chatServer, and use these two parameters to initialize the instance variables: name and chatServer in the BaseUser class. Name is obvious because every user is associated by its user name. We also pass in chatServer because when users want to send messages to another user, they need to use the chatServer as a handle to send and let the chatServer to send the message to another user.
The `BaseUser` class has the method:
public void send(String dest, String msg);
To allow a user to be able to send messages to another user, we will instantiate a new message object within the method. The new message object will be constructed with the input parameters from the send method and the name instance variable of the BaseUser class, where this.name is the source:
msgPacket = new Message(this.name, dest, msg);
Once we have created the message object to send, we will put this message object into the log, so that the user will have a log of the conversation. The User will use the chatServer handle to call chatSever.send(msgPacket) to send the message to the chat server and let the server to resolve message delivery.
Once the message object is sent to the server, the message object is passed onto the message queue, so that the message can be delivered to another user. The MessageDispatcher class is responsible for managing the message queue. To pass the the message object to the message queue, we will need to call the enqueue method of the MessageDispatcher. This will effectively enqueue the message onto the message queue.
When the chat server starts, the chat server forks a thread to run MessageDispatcher. This will keep the MessageDispatcher in a loop, so that it will continue to deliver messages. The MessageDispatcher has a deliver method that is responsible for dequeue the messages. In the deliver method, MessageDispatcher will dequeue a message object from the message queue, and examine this message object to get the dest (receiver) and the

msg. To get the user who will receive the message, MessageDispatcher will have a handle to the UserManager. MessageDispatcher will call the getUserByName method from the UserManager and pass in the dest string. GetUserByName will effectively return the receiving user object or it will return null if the user object does not exist. MessageDispatcher will then invoke the msgReceived method from the receiving user and pass in the msg. Up to this point, users can send and receive messages from each other.
If getUserByName returns null, meaning the user does not exist, then MessageDispatcher will set the error instance variable of the message object to be true. This says that the message was not received by a user. MessageDispatcher will not try to call msgReceived method on a null value.
When a user display his/her log of conversation, if the message object is marked error, then it shall simply print out message was not received.

- A user can send a message to the group she belongs to
- A user receives message successfully from the group
- Sending a a message to a group that does not exist should fail explicitly and should mark the message in the log to be error.

To allow a user to send a message to the group she belongs to, a user will invoke the method in the BaseUser class:

```
sendGroupMessages(String groupname, String msg)
```

where groupname is the name of the group, the user is sending message. Msg is the message being sent. Inside sendGroupMessages method, the user will use the chatServer handle to invoke the method:
getUserGroup(BaseUser user, String groupname)
The user will pass itself and the groupname from the parameter of sendGroupMessages method to getUserGroup. GetUserGroup will return the group with specified groupname that this user has joined. GetUserGroup will return null if the group does not exist.
The Group class provides a method that will return a list of users by name in this group :
listUsers()
The next step is very similar to send messages to single user. We will call the send method of the BaseUser class. With the source and message being fixed, we are going to iterate through the list of users by name in the group and set those names as the destinations of the send method. This will effectively send the message to all users in the group. Because sending messages to group has the same mechanism as sending message to single user, we can expect each individual user in the group can receive the message the same way as if the message was sent as one on one basis.
If getUserGroup returns null, the message object will have the error marked true and stored in the log to signal that this message was not received by the group. We will not try to call listUsers() if getUserGroup returns null.
** what if listUsers() return empty list. Do we error the message? or what do we do?

1.3.3 Tests

1.4 In-Order Delivery

We want to ensure the following message delivery semantics:

1. Message must be consumed in the order they are received. Message order must be preserved across all users in a group and between pairwise user conversations.

We will be using queue to store the message objects that are sent from the users. Queue allows us to ensure that message orders are preserved because queue will dequeue in FIFO order. More importantly, when we enqueue the message object, we will need to make sure to serialize (lock) the action. Therefore, at any given one time, only one message can be enqueued to the message queue.

2. Only the current members of the group can consume messages posted to the group.

   `Group` is the class we will create to represent a user chat group.

   ```
   Variables added:
   ArrayList (synchronized) - userList: A list of the user threads in this group.

   Methods added:
   addUser, removeUser, listUser, userCount
   ```

   Because we actually use a Group object to store all users within a group, when users send messages, we can make sure that messages are only delivered to users within the group. The tricky part will be to ensure that when a user leaves the group and receives a message at the same time, the user should not receive the message. To ensure this, we have a writer and reader situation. Writer will be updating the user ArrayList (such as remove or add) and reader will be getting a list of users by name from the ArrayList. We need to lock the critical section where writer or reader try to access the ArrayList, and writer should have priority to perform the action. This way we can ensure the reader will be getting the most up-to-date list of users.

3. The order of events and message posts should be consistent with the expected message-delivery semantics (i.e. a user cannot see a message that was posted before she joined the group).

   Any user object will keep an ArrayList of message object. This allows us to log the conversation. Any time a message is sent or received, we will put the message object onto the ArrayList.

   To lock the message queue will only ensure that message objects sent to the server will be sent to another user in the order they were sent. However, it does not guarantee that message received on the user's side will be in order.

   To ensure that the messages are received or sent in the order, we also need to lock the critical section where we try to update the message ArrayList.

1.4.1 Tests