Thread-Safe Chat Server

Tony Dear, Jason Jee, Harry Jiang, Dustin Kwong, Paul Ruan
cs162-ba, cs162-ab, cs162-ai, cs162-at, cs162-bb

Overview

In almost all systems today, concurrency and synchronization are major problems that designers need to consider. We aim to control these issues in a thread-safe chat server, in a consistent yet efficient manner for all users of the system. In particular, functionalities that are vulnerable to potential conflict include users logging in and out of the system, joining and leaving chat groups, and sending and receiving messages. For many of these, locks and queues will be the main solution. We implement the former to ensure that operations are atomic and completed with no interruptions, while the latter will be used to assist with requests that arrive while the host is busy. With these tools in place, we hope to be able to control concurrency and apply synchronization in a safe way.

Specifications

At the core, we have five different classes of objects. The ChatServer, which implements a ChatServerInterface, will serve as the main mechanism behind the system. It is in charge of logging Users in and out, as well as adding them and taking them out of ChatGroups. It will keep track of how many Users and ChatGroups currently exist on the server. This information is also publicly viewable via getter methods, should an external process wish to access this information. Users are expected to log in with a unique username. They can also join ChatGroups (which also require a unique name), as well as leave one. Finally, Users may send and receive Messages, either to other Users or a ChatGroup. For debugging purposes, it is also useful to have a ChatLog class, of which each user may have many.

Considerations

Because the ChatServer can be easily subject to heavy traffic by different Users attempting different activities, it is important that we consider how it will perform under corner cases and a variety of situations. At its core, the handling of concurrency will play a key part in the correctness of the system. Key points include the following:

- How do we ensure that Users logging in and out at the same time will not confuse the ChatServer? This is especially critical when the number of Users in it is around the maximum limit. For example, if one User logs out and leaves one open spot for a new one to come in, it can become a race condition, or maybe both will slip past the ChatServer and get in unnoticed.

- A User can join and leave multiple ChatGroups. As with users logging in and out of the ChatServer, it is possible that a user has not joined or left the ChatGroup properly because it was busy dealing with someone else.
    - If a User tries to join a ChatGroup that does not exist, he automatically creates it.
    - A ChatGroup should not be empty. Whenever one is born, the User who created it should be automatically added into the ChatGroup. Similarly, when the last User in a Group leaves, it should automatically destroy itself.

- Uniqueness of names must be enforced, applying to the combination of both Users and ChatGroups. In addition to maintaining data structures to keep track of what has been used so far, we need to ensure concurrent reading of such lists for efficiency but synchronize modification for correctness. Otherwise, two different readers may get the wrong information or edits can be lost.

    - Getters and setters of such lists would thus have different priorities with respect to each other. Setters will have to be mutually exclusive, while readers are not.
    - Getters are also only available to valid Users, meaning they are actually on the server.

- In addition to name uniqueness, Users can also be rejected from the ChatServer if it has hit its maximum limit of 100 Users. Concurrency is handled here within the logging functionality mentioned above. A similar principle applies with the maximum of 10 Users in any ChatGroup.
    - Any User that attempts to log in or join past the limit is explicitly rejected.

- How do we ensure that Message delivery is done properly? It is possible for two Users to send one another Messages at the same time. To ensure that both Users process the same one first, sending (and receiving a successful confirmation) and receiving messages should be exclusive.
    - All Users have a queue for sending messages. Pending messages are placed on the send queue, and they will be processed at the User thread's convenience.
    - Upon wakeup, a User examine its queue to see if any messages are waiting.
        - If there are any to send, the User will forward them appropriately, which also includes getting the recipient to receive them.

- Similarly with ChatGroups, they should only send out one message at a time to all its Users.
- Again, only Users who are in a ChatGroup can send and receive Messages within it.
    - Because the ChatGroup has no memory of Messages in the past, Users who join a ChatGroup later will not see any Messages before then.

- It is always possible for Message delivery to fail. This may occur when either the sender or the recipient is invalid. In either case, the sender should receive either a success or fail confirmation appropriate for the situation.
    - A ChatGroup should confirm that it has received a sender's Message, although that does not mean every other User in the group will receive the Message. It is possible for a User to leave while it is in transit, in which case message delivery silently fails.
- Another issue is the finite size of the message buffer. We must impose some limit on the queue size to prevent it from being too full and potentially causing some sort of memory error. If this limit is reached, then messages must be dropped.

Implementation

As the specifications above imply, we will design one class for each of the above types of objects. However, the only threads that are actually running are Users. The ChatServer and ChatGroup objects are mainly used by Users to interact with one another. Messages and ChatLogs also exist, but only for logging purposes. The main functionalities include the following:

Logging in (Fig. 1): The ChatServer performs a series of tests before allowing a User in. First it checks itself to make sure that it is still up and running; if not, it could have been shutdown() some time before. If that passes, it then checks the userlist to see if the new user has a unique name. Finally, if that passes, the ChatSever checks the number of Users currently on the system. If it is below the limit, then the new User can successfully log in and is notified of success. Otherwise, it gets back the USER_REJECTED error.

An alternative that we had considered but did not implement was to have a buffer for Users waiting to get into a full ChatServer. In that case, they wouldn't be explicitly rejected, but instead be placed on a queue operating on a FCFS basis. In this scenario, our ChatServer thread would actually be doing something in constantly checking the queue and then logging in Users when the right time comes.

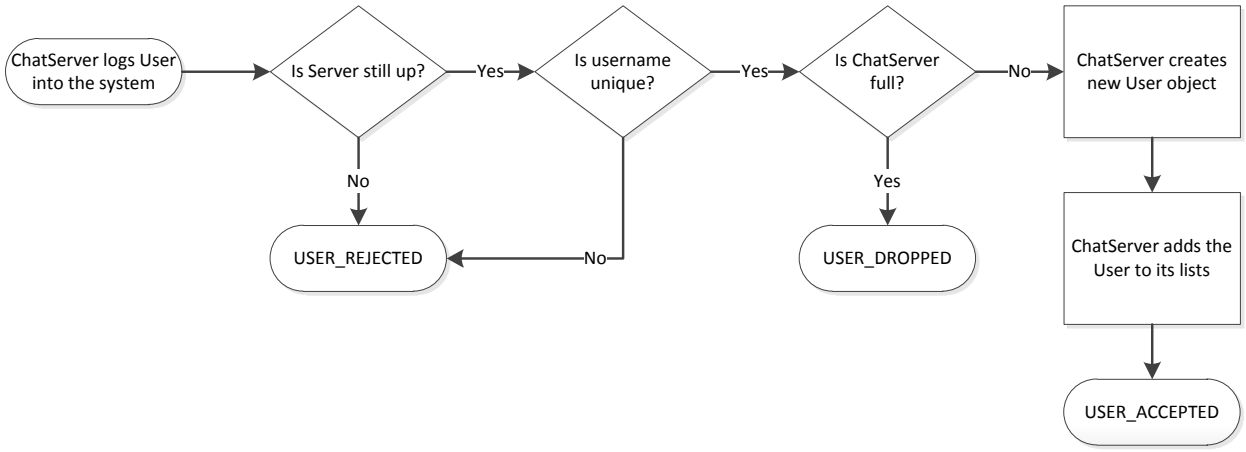

Figure 1: Logging in

Logging off (Fig. 2): Logging out of the systems is relatively simpler than its opposite process. After checking that the User is valid, the ChatServer first forcibly removes the User from all his ChatGroups, and then it removes the User from its lists. Finally, the ChatServer calls User's logoff() method to set its new state.

Figure 2: Logging off

Shutting down (Fig. 3): This is a straightforward process. After locking its lists, the ChatServer clears all records Users and ChatGroups and sets the isDown field to true. At this point, it no longer contains any references to what was previously inside, and no one new can come in.

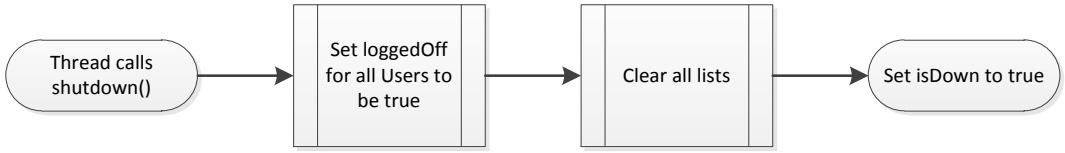

Figure 3: Shutting down

Joining groups (Fig. 4): Whenever a User wishes to join a ChatGroup, after checking if he is valid, the ChatServer must also check if the group exists yet. If not, then this function is equivalent to creating a new ChatGroup, as long as the name is not already taken by a User on the server. In this case, the server allocates room for one and adds it to its list; then it simply adds the creator to the new ChatGroup.

On the other hand, if it already existed prior, then the server has to ask the group it to allow the User to join. The ChatGroup performs a series of checks similar to the ChatServer when logging Users in: It checks if the User is already in the group, checks the number of Users to make sure of its User limit, and finally adds the User to its list if all the tests have passed. At this point, the User is informed that he has successfully joined.

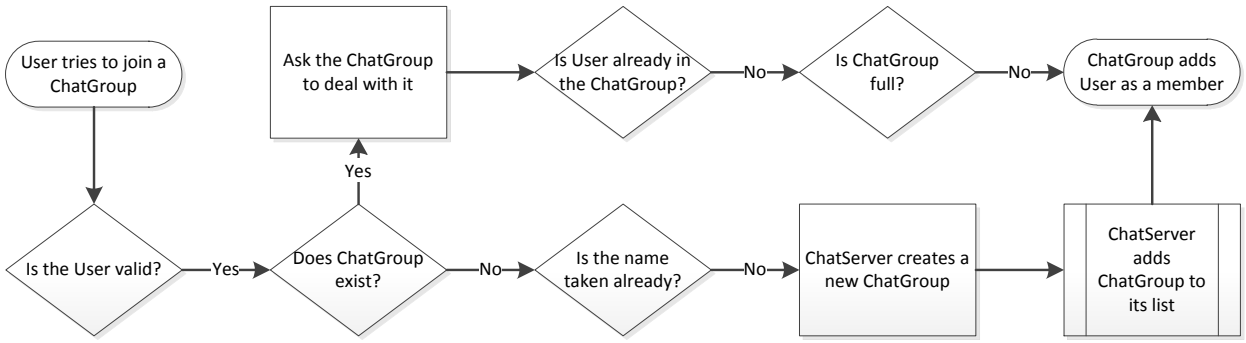

Figure 4: Joining a group

Leaving groups (Fig. 5): Whenever a valid User leaves a valid ChatGroup, the ChatServer checks if this causes the ChatGroup to become empty. If it does, then the ChatServer simply deletes it by removing it from its list. Otherwise, it tells the ChatGroup to remove the User from its member list.

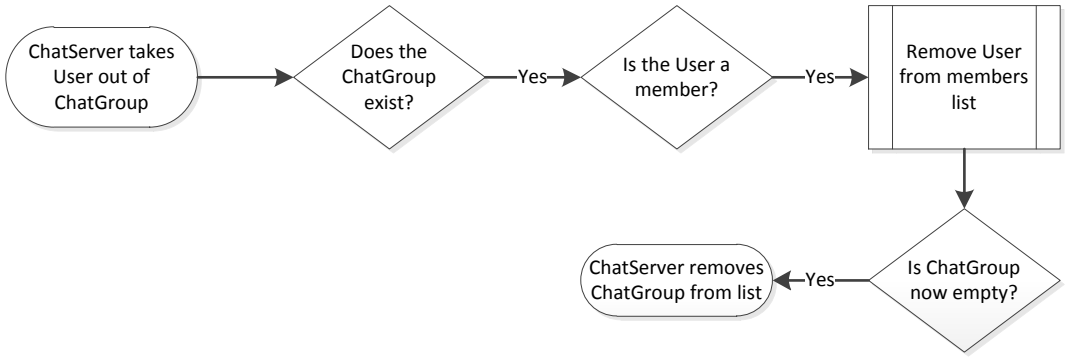

Figure 5: Leaving a group

Message processing (Fig. 6): Message processing is the most complex functionality that the ChatServer provides. The first step taken when a User wishes to send a message (in the form of a String) is placing it on the User's send queue. For this to occur successfully, we must check that the User is valid (actually logged in) and that his queue is not already full. Now when the User thread wakes up and runs, it will check this queue to see the potential messages waiting to be sent.

For each of these messages, the User thread calls ChatServer's processMessage() method, which wraps the String along all other necessary information into a Message object. Before it does anything else, the ChatServer first checks if the sender is actually a valid User who is logged in on the server. Then we can have one of three cases for the recipient: it is either a User, a ChatGroup, or neither.

If the recipient is a valid User, then the ChatServer simply calls the recipient's acceptMsg() function. On the other hand, if the recipient is a ChatGroup, the server calls the group's forwardMessage() function. There a further check is performed to make sure that the User is a valid member of this group. If that passes, the ChatGroup atomically tells all its members to acceptMsg(). If the Message is successfully forwarded in either of these cases, then the sender is notified of success with a MESSAGE_SENT.

Otherwise, a number of issues could have caused it to fail, and the ChatServer tells the sender appropriately. It could have been that the sender was invalid, in which case INVALID_SOURCE is

returned. If the User was not a member of the ChatGroup to which he sent the Message, we return NOT_IN_GROUP. Finally, if it turns out that the recipient is neither a User or a ChatGroup at the time of sending, it is a INVALID_DEST.

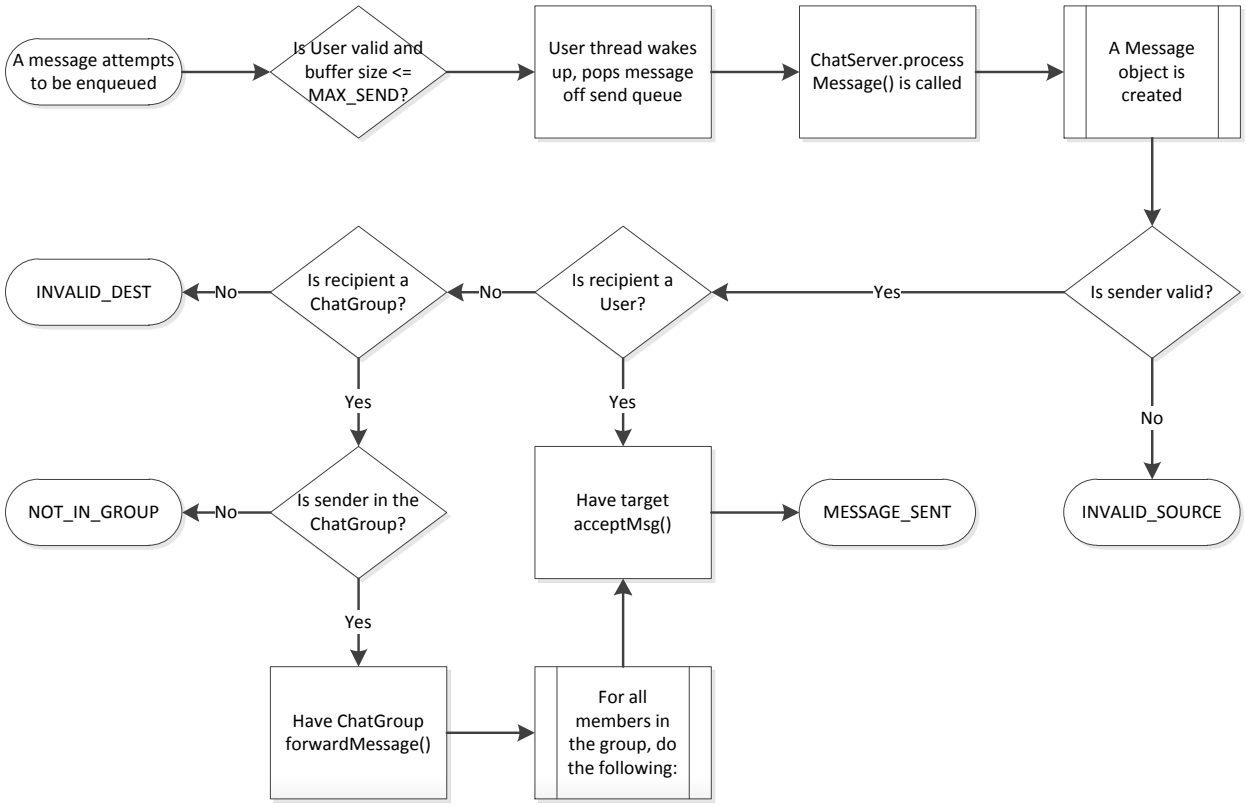

Figure 6: Message processing

Accepting a message (Fig. 7): This functionality actually occurs within the same thread that the sender runs in. Accepting a Message mainly consists of logging it in a ChatLog. First we check if the Message is from a ChatGroup to see what the reference to the ChatLog should be. Now we have two cases: either the ChatLog already exists (which means the conversation has already started with the other party), or it does not. If not, then we simply create a new one and add it to the User's ChatLog list. Then at the end, we actually add the Message to the ChatLog.

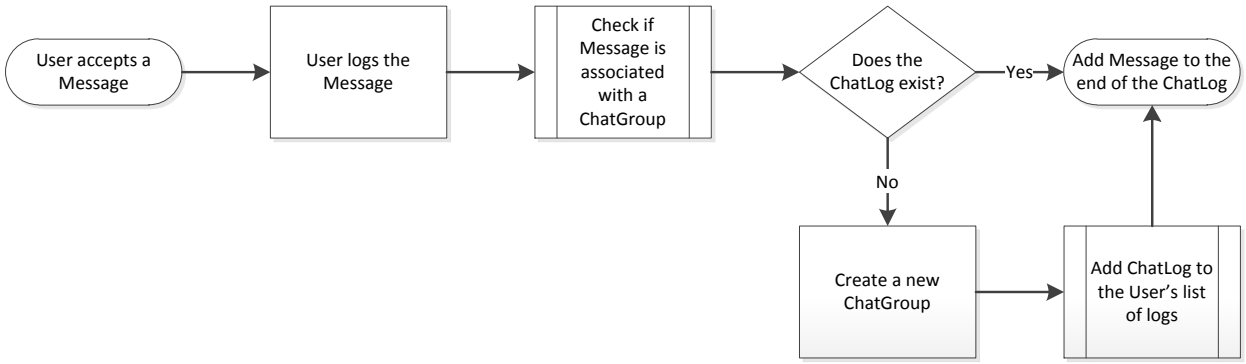

Figure 7: Message receiving

Alternative designs: We considered a number of alternatives in designing the message processing functionality. For example, we originally did not have any queues for sending messages. In this case, the external thread that tells the sender to send the message would just do the whole job itself. In other words, the User thread would not have any formal role in this. We decided that this was not exactly consistent with the project spec, since we were not using the User thread for anything.

Another approach was to have both send and receive queues. In this scenario, processMessage() would enqueue the Message object on the recipient's queue. Then the latter would only "receive" the Message when its thread wakes up and pops it off, in the same manner as the sender handles its queue. We decided to discard this approach as well, due to weird timing inconsistencies that can occur. For example, a User can have group Messages still on his queue after he leaves.

In the end, we decided to define messages that have been enqueued in a User's send queue but not yet accepted by the recipient to be "in transit." In this model, "graceful handling" of messages is undertaken upon the User leaving a group, logging off, and shutting down the server. In doing so, we drop all messages on the sender's queue; then the intended recipient will never receive them, and both sender and recipient have their ChatLogs remain consistent.

Concurrency handling: Within any of these functionalities, concurrency must be handled properly. We use one of Java's built-in locks to deal with almost all of these cases. For example, in User we have a ReentrantReadWriteLock called sendLock. As its name suggests, it places a lock on the send queue when a String message is being enqueued or dequeued. That way, we prevent any other process from messing with the queue while this is being done.

Another location where locks are heavily used is in the ChatServer. All methods that modify the lists in some way must utilize a writeLock. This includes logging in and out, joining and leaving ChatGroups, and shutting down the server. Otherwise, it is possible for subtle bugs to occur. For getter methods, it is enough to place a readLock on the lists while the method is being run. The usage of locks in this way ensures that the reader-writer problem discussed in class is dealt with appropriately. Specifically, multiple readers can access the structure at once, while each writer must have exclusive access.

One exception to using locks is the usage of Synchronized for the forwardMessage() function in the ChatGroup class. Here the Synchronized essentially "locks" the method, preventing someone else from calling the same method in another process. This is necessary here since the ChatGroup should atomically forward the message to all its members. If it is interrupted before it finishes, it is possible that some Users fail to receive it, or some may receive it in an order different from other Users.

Enums used in our design

LoginError:	USER_ACCEPTED	//User logs in successfully.
	USER_ DROPPED	//ChatServer is too full for User to log in.
	USER_REJECTED	//Username is invalid, fails to log in.
MsgSendError:	MESSAGE_SENT	//Message successfully sent.
	INVALID_DEST	//Recipient is neither a valid User nor valid ChatGroup.
	INVALID_SOURCE	//Sender is not a valid User.
	NOT_IN_GROUP	//Sender is not in the group that he tries to send to.

ChatServer class

The ChatServer object is in charge of all functionality. It handles the logging in and out of users, the management of ChatGroups, and message processing. In addition, it maintains lists of users and groups on the server, since those are the only handles to them that exist on the system. Finally, it handles any illegal actions by users in a graceful manner.

The tasks of keeping lists and forwarding messages is relatively trivial, but the challenge is handling concurrent requests correctly and efficiently. Any actions that Users take must go through this one ChatServer. Our first idea was to just synchronize all methods so no that two threads (or Users) can access the server at once. However, we later decided that it would make more sense to allow reads to happen concurrently. We implement it in a way similar to the "readers and writers" problem discussed in lecture but much more simply with already implemented locks in Java, specifically the ReentrantReadWriteLock.

Description of Fields

final static int MAX_USERS
HashMap<String, User> users //Maps existing usernames to Users.
HashMap<String, ChatGroup> groups //Maps existing chat group names to ChatGroups.
HashSet<String> allNames //Set of all names that are in use.
boolean isDown //True after server's shutdown() method is called.
ReentrantReadWriteLock lock //Protects reads and writes to users, groups, and allNames.

The above lock is our main mechanism for controlling concurrent reads and writes to our list of users and groups. Readers can read at the same time when there are no writers, and they can only start when there are no writers waiting or writing. Writers are mutually exclusive with any other accessors, so they are allowed to execute only when no one is using the list.

Description of Methods

BaseUser getUser(String username) //Returns User with username if it exists in user, or null.
List<BaseUser> getUsers() //Returns a list of Users that are logged in.
ChatGroup getGroup(String name) //Returns ChatGroup with given name, or null.
List<ChatGroup> getGroups() //Returns a list of names of existing groups.
int getNumUsers() //Returns number of all users.
int getNumGroups() //Returns number of all groups.
LoginError login(String username)
boolean logoff(String username)
boolean joinGroup(BaseUser user, String groupName)
boolean leaveGroup(BaseUser user, String groupName)
boolean shutdown()
MsgSendError processMessage(String source, String dest, String mst, int sqn, String timestamp)

The above getter methods simply access information from relating to Users and ChatGroups. They all acquire the read lock to run, allowing other readers at the same time to concurrently access them.

LoginError login(String username)
If the username is not in allNames and the number of users is less than MAX_USERS, logs the user in: username is added to allNames, a BaseUser is created with the username, and a mapping between username and user is added to users. A loginError is returned to indicate success, the ChatServer being too full, or that the username was taken already.

Because multiple Users may try to login at the same time, concurrency errors in allowing the same names to be used or allowing more than MAX_USERS can occur. We prevent these errors by acquiring the write lock. Then while a User logs in, users, groups, and allNames can't be read nor written to. The result is that only one user may log in at any one time, making it serial.

```
Acquire write lock
   # of users == MAX_USERS?
      Release write lock
      Return USER_DROPPED
   Name exists already?
      Release write lock
      Return USER_REJECTED
   Add user, have the user connect
Release write lock
Return USER_ACCEPTED
```

boolean logoff(String username)
If the username is not mapped to a User in users, returns false. Otherwise, logs the User off and returns true: User is removed from users, and username is removed from allNames. User also leaves all groups.

As with login(), the write lock is acquired, as we must make changes to allNames and users. Then we must tell the groups in which the User is a member to delete the User. Concurrency issues with deleting the user from the groups is handled in the ChatGroup class.

```
Acquire write lock
   User exists?
      Remove user
      Have user leave all its groups
      Release write lock
      Return true
Releases write lock
Return false
```

boolean joinGroup(BaseUser user, String groupName)
Fetches ChatGroup from groups and have the group add the user. If the User is already in the ChatGroup, or if there's no more room in the group, returns false. Otherwise, if the group doesn't exist, add the group first by adding a mapping between groupName and the ChatGroup in groups, and then add the User to the ChatGroup as the first member. If all this completes successfully, return true.

Because it is possible that the ChatGroup doesn't exist yet, we must acquire the write lock in case we have to add to groups. The issue of joining groups and having too many users in a group because of concurrent joining is handled in the ChatGroup class.

```
Acquire write lock
   Group exists?
      Join group
      Release write lock
      Return joined successful?
   Groupname not taken by another user?
      Add group
      Join group
```

```
    Release write lock
    Returns joined successful?
```

boolean leaveGroup(BaseUser user, String groupName)
Fetches ChatGroup from groups and removes User from group. Returns false if the group doesn't exist, or if the user is not a member of the group. Otherwise returns true.

Similar to joinGroup, there's a possibility that we have to modify groups; if the User is the last person to leave the ChatGroup, it will be deleted. Therefore, we first acquire the write lock.

```
Acquire write lock
  Group exists?
    Try to have ChatGroup handle User leaving
      Leave successful?
        # of Users left == 0?
          Delete group
        Release write lock
        Return true
      Release write lock
      Return false
```

void shutdown()
After acquiring the write lock, we force all Users to log off by iterating through the users list. Then we clear all lists (essentially deleting the ChatGroups as well) and set isDown to true.

MsgSendError processMessage
(String source, String destination, String message, int sqn, String timestamp)
Returns success if the recipient successfully receives the Message object that this method creates. Depending on if the destination is a User or ChatGroup, processMessage will tell it to either accept the Message or forward it to all group members (which tells them all to accept). Otherwise, we return an appropriate failure message.

```
Make a new Message with given parameters
Acquire read lock
  Is source a valid user?
    Is recipient a user?
      Have recipient acceptMsg()
    Is recipient a group?
      Try group.forwardMessage()
      Failed?
        Release read lock
        Return NOT_IN_GROUP
    else
      Release read lock
      Return INVALID_DEST
  else
    Release read lock
    Return INVALID_SOURCE
  Release read lock
  Returns success message - MESSAGE_SENT
```

User class

The User class represents an individual who can interact with other Users via the ChatServer. All Users can join and leave groups, as well as and send and receive messages, either with other Users or with ChatGroups. Each user also maintains his own ChatLog of each conversation he participates in, starting from the time it joins the conversation to the time it leaves.

Description of Fields

```
final static int MAX_SEND              //Maximum size of the send queue.
ChatServer server                      //Handle to the ChatServer.
String username
LinkedList<String> groupsJoined        //List of ChatGroups of which User is a member.
HashMap<String, ChatLog> chatlogs      //All ChatLogs belonging to the User.
LinkedList<MessageJob> toSend          //Send queue.
ReentrantReadWriteLock sendlock
int sqn                                //Sequence number used for sending messages.
boolean loggedOff                      //True if the User is no longer logged in.
```

As with the ChatServer, we maintain a ReentrantReadWriteLock to ensure that our send queue is dealt with in a safe manner in concurrent situations. Whenever we enqueue or dequeue, we acquire a write lock so as to prevent other threads from taking our spot before we finish. That way, only one writer can use the queue at any one time.

The sequence number is dynamic; each time the User sends a message, it is increased so that the next message will have a different one. That way, we ensure that every message sent out by this User is unique in at least one way.

Description of Methods

```
String getUsername()                   //Returns username.
List<ChatGroup> getUserGroups()        //Returns all groups of which User is a member.
Set<String> getAllUsers()              //Returns all Users in the ChatServer.
Set<String> getAllGroups()             //Returns all ChatGroups in the ChatServer.
int getNumUsers()                      //Returns total number of Users in the ChatServer.
int getNumGroups()                     //Returns total number of ChatGroups in the ChatServer.
ChatLog getLog(String name)            //Returns the ChatLog corresponding to name.
Map<String, ChatLog> getLogs()         //Returns all ChatLogs.
void send(String dest, String message)
void acceptMsg(Message msg)
void msgReceived(String msg)           //Prints out msg.
void logRecvMsg(Message msg)
void logoff()                          //Sets loggedOff = true.
void run()
```

Most of these methods are relatively trivial and were used in our testing. The bulk of the activity for a User lies in message processing; this occurs in the methods described below.

void send(String dest, String message)
First we acquire a write lock before doing anything. Then we check that the User is actually logged in and the send queue has room. If both tests pass, then we create a new msgJob, which includes the message information in addition to the sqn and timestamp, and place it on the queue.

Acquire write lock
  Is user logged off?
    Release write lock and return
  Is send queue full?
    Notify that User timed out, release write lock, and return
  Make new MsgJob and enqueue
  Release write lock

void run()
While the User is logged in, the thread constantly checks the toSend queue for msgJobs. If it finds that there are any, it pulls it out and calls ChatServer's processMessage() function on it. Otherwise, we're logged out, and we pull out any msgJobs on the queue and drop them. As always, we have to worry about concurrency and use locks when dealing with the queue.

While not logged off:
  Acquire write lock
    If toSend queue is not empty:
      Pull a msgJob and process it
  Release write lock
Otherwise acquire write lock
  While toSend queue is not empty:
    Pull a msgJob and drop it
  Release write lock

void acceptMsg(Message msg)
Calls logRecvMsg(), followed by msgReceived() to print it out.

void logRecvMsg(Message msg)
Here we actually add the Message to a ChatLog according to the implementation described above. First we check if msg is sent from a group to determine the right reference. Then we see if it exists already; if so we can just pull it out from the list, and if not then we just create a new one according to the reference. After adding the new ChatLog into the list, we can just tack the new Message onto the end of the log.

Message class

The Message class acts as the middleman between the chatting Users. In addition to containing the message content itself, a Message object keeps track of a timestamp and sequence number, so that Users receive Messages in the correct order. The sequence number tells us how many messages that the sender has sent out so far. Because it also contains source and destination information, it also plays a role in helping determine if the send request is valid or not.

Description of Fields
String timestamp
String source
String dest
String content
int sqn //Acts as a counter for messages from a particular User
boolean isFromGroup //True if message is associate with a ChatGroup.

Description of Methods

void setSQN(int num) //Set SQN to num.
void setIsFromGroup() //Set isFromGroup to true.
String getTimeStamp()
String getSource(
String getDest()
String getContent()
int getSQN()
boolean isFromGroup()
String toString() //Return Message in the following format: SRC DEST TIMESTAMP SQN.

MessageJob class

The MessageJob can be thought of as an incomplete version of a Message. It contains all the background information: destination, message content, timestamp, and SQN. It is used primarily for enqueueing onto the User's send queue prior to being popped.

Fields:

String dest
String msg
String timestamp
int sqn

ChatLog class

The ChatLog class is used by Users to maintain a list of Messages received in the correct order. It is simply a linked list of Message objects in the order they were received by the user. There are two different types of ChatLogs, as mentioned above. For a 1-on-1 conversation, ChatLogs are referenced by the source User. For ChatGroups, we reference the log using the ChatGroup name. Hence, we have two constructors, one with and one without a ChatGroup name. In the case that we do not have one, we leave it as null and never reference it.

Description of Fields

LinkedList<Message> log //The chatlog itself as a linked list of Messages.
User user //The User who owns the log.
String source //For a 1-on-1 conversation, the party with whom user is chatting.
String chatgroup //For a ChatGroup conversation, the ChatGroup name.

Description of Methods

List<Message> getLog()
BaseUser getUser()
String getSource()
String getGroup()
void add(Message message) //Adds the message to the end of the ChatLog
String toString() //Returns a string form of the ChatLog, mainly for our testing
boolean isSuperLogOf(ChatLog chatLog) //Checks if the argument is a sublog of this one (testing).

ChatGroup class

We have decided to use a ChatGroup class to ensure some of our operational constraints. In particular, all Users belonging to a ChatGroup will receive the same sequence of messages. In addition, the number of Users in any ChatGroup is limited to ten.

When a User has been authenticated by the ChatServer, the user will be allowed to either join or create a ChatGroup. On creation, the creator is automatically added as a member of the ChatGroup. As a member of the ChatGroup, the user is allowed to send messages to the rest of the ChatGroup. Messages sent by a valid User are forwarded to the ChatGroup from the ChatServer, at which point the ChatGroup will broadcast the message to all current members. This will ensure that all users of the ChatGroup receive a consistent sequence of messages from the ChatGroup. Users may join and leave the ChatGroup as they please as long as no more than ten users are in the ChatGroup at once. Finally, the ChatGroup is deleted only after every user has left the ChatGroup.

Description of Fields

```
int MAX_USERS
String name                          //Name of this ChatGroup.
Hashmap<String, User> userlist       //All members of this ChatGroup.
```

Description of Methods

```
Hashmap<String, User> getUserList()                    //Return all members
int getNumUsers()                                      //Return number of members
String getName()                                       //Return ChatGroup name
boolean joinGroup(String user, BaseUser userObj)       //Adds user to userlist, if valid
boolean leaveGroup(String user)                        //Removes user, if valid
boolean Synchronized forwardMessage(Message msg)       //Forwards msg to all members
```

The methods for joinGroup and leaveGroup operate similarly. Both check if the User is actually a member before proceeding. Joining a group also invokes a check on the number of current members in the group so that we are not over the limit. If all tests pass, then we can either add or remove the User to or from the member list.

Synchronized is used for forwardMessage, since it deals with the ChatGroup's userlist, which includes all the User objects in there. After checking that the sender is valid, we iterate through the members list and one by one have call acceptMsg() on them. Having this method be Synchronized allows for a universal lock on all the Users, so that none of them conflict with other Messages coming in at the same time.

Unit Tests

ChatGroupTest

- User joins an empty ChatGroup. Check that it is in the ChatServer and that its size is 1.
- User joins a ChatGroup. Check that both Users are members and that its size is 2.
- User tries to join a full ChatGroup. Check that group size is still 10 and that the join returns false.
- User tries to leave a ChatGroup of size 2. Check that the size is 1 after leaving.
- User tires to leave a ChatGroup of size 1. Check that the ChatGroup is destroyed.
- User tries to leave a nonexistent ChatGroup. Check that it should return false.

- User joins and leaves a ChatGroup, then he tries to send a Message to the group. Check that it returns a MsgSendError: NOT_IN_GROUP.
- User sends to a ChatGroup of three users. Check that the Message shows in the ChatLog of all the members.
- User logs off the ChatServer and tries to join a ChatGroup. Check that the operation returns false.

ChatServerTest

- User logs and tries to process a message from a User who is not logged in. Check that a MsgSendError is thrown: INVALID_SOURCE.
- User logs in and attempts to send to a User who is not logged in. Check that a MsgSendError is thrown: INVALID_DEST.
- Two Users log in and send a Message to each other. Check that a MsgSendError is thrown: MESSAGE_SENT.

LoginTest

- User logs into the ChatServer. Check that a LoginError is thrown: USER_ACCEPTED.
- User logs in with a non-unique username. Check that a LoginError is thrown: USER_REJECTED.
- User tries to join a full ChatServer. Check that a LoginError is thrown: USER_DROPPED.

LogoutTest

- User logs in and then logs out. Check that the operation returns true.
- Try to log a User off who does not exist in the ChatServer. Check that the operation returns false.
- User joins three different ChatGroups, two of which existing Users. Then the User logs off. Check that the ChatGroup that he created is destroyed and that he is erased from the userlist of the other two groups.

OneToOneTest

- Two users participate in a 1-to-1 conversation. One sends a Message to the other. Check that both Users log the Message correctly.

Behavioral Tests

TestChatGroups

- Two Users join the same ChatGroup. A number of MessageDeliveryTasks send Messages to the group. Check that both Users have the same ChatLog for the group.

TestChatLogs

- Same test as above for 1-on-1 conversations.

TestLeaveGroup

- A number of MessageDeliveryTasks associated with a ChatGroup are started. A User in a ChatGroup leaves the group, and then more MessageDeliveryTasks for that ChatGroup start up. Check that the User receives all the Messages that were posted to the group before and that all other ones are dropped.

TestLogOff

- A number of MessageDeliveryTasks associated with a ChatGroup are started. A User in a ChatGroup logs off, and then more MessageDeliveryTasks for that ChatGroup start up. Check

that the User receives all the Messages that were posted to the group before and that all other ones are dropped.

TestMessageOrder

- Log twenty Users into the ChatServer. Create four ChatGroups and assign five Users to each. Have all members of each ChatGroup randomly send and receive Messages to and from each other. Check that all ChatLogs match up, and that all Messages have been received.
- Also, have Users randomly send to different ChatGroups. Check that the Messages that are not destined to groups of which the User is a member are dropped.

TestOneToOneLogOff

- A number of MessageDeliveryTasks intended for a particular User are started. This User logs off the ChatServer, and then more MessageDeliveryTasks for that ChatGroup start up. Check that the User receives all the Messages that were started before and that all other ones are dropped.

Conclusions

Our implementation passed all the above tests. To do all this, we use tools covered in class and provided by the Java API, such as the concepts of locks, atomization, and synchronization. In addition, the design also takes into account possible consistency issues in terms of serialization and concurrency. Based on these results, we believe that we have successfully designed a ChatServer that meets all specifications and runs in a thread-safe manner.

Class Diagram