**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 two message queues, one for sending and one for receiving. Pending messages are placed on the send queue, and similarly with the receive queue.
  + Upon wakeup, User threads examine their queues to see if any messages are waiting.
    - If there are any to send, the User will forward them appropriately.
    - If there are any to receive, the User will log them appropriately.
* 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 delivery will silently fail.
* Another situation in which this task fails is when either the send queue of the sender or the receive queue of the recipient is full. If so, then the Message will be dropped, and the sender will be notified appropriately.

**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: 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.

Joining groups (Fig. 2): Whenever a User wishes to join a ChatGroup, the ChatServer must check if it 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 2: Joining a group**

Leaving groups (Fig. 3): 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 3: Leaving a group**

Sending messages (Fig. 4): 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. Now when its 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 msgReceived() function, which just places the Message on his receive queue. 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 places the Message on everyone’s receive queue. If the Message is successfully enqueued 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 4: Sending a message**

Receiving messages (Fig. 5): Receiving messages is relatively simpler than sending them. When the User thread wakes up and sees that his receive queue is not empty, he will simply pop and log it. In logging the message, we first check if it comes from a ChatGroup. If so, then we reference the chatlog by the group name; otherwise, we reference it by the sender (since it’s a 1-on-1 conversation).

It is possible that the ChatLog already exists for this particular reference, in which case it is simply the continuation of a previous conversation. Otherwise, we create a new ChatLog corresponding to the reference, and add it to the User’s list of ChatLogs. Then we just tack on the new Message onto the end of the ChatLog, since it is just a list of Messages.



**Figure 5: Receiving messages**

Shutting down: 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.

**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 two ReentrantReadWriteLocks: recvLock and sendLock. As their names suggest, each one places a lock on the respective queue when something is being enqueued or dequeued. That way, we prevent any other process from messing with the queues 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.

**ChatServer class**

    The ChatServer object maintains lists of users and groups on the server. In addition, it forwards Messages from Users to the appropriate destination user or group, asking the former to receive and the latter to broadcast the created Message object. It also handles any illegal actions by users.  
    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 plan to 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**  
private final static int MAX\_USERS //max number of users allowed to be logged in (100)  
private HashMap<String, User> users //maps usernames to Users  
private HashMap<String, ChatGroup> groups //maps existing chat group names to ChatGroups  
private Set<String> allNames //set of all names that are in use.  
private boolean isDown //true if server's shutdown() method is called  
private ReentrantReadWriteLock lock //protects reads and writes from users, groups, and allNames.  
  
This 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**  
public BaseUser getUser(String username) - Returns User with username if it exists in user, or null.  
public List<String> getUsers() - Returns a list of users’ names that are logged in.           
public ChatGroup getGroup(String name) – Returns group with given name, or null.  
public List<String> getGroups() - Returns a list of names of existing groups.  
public int getNumUsers() - Returns number of users.  
public int getNumGroups() - Returns number of groups.  
public void shutdown()   
public LoginError login(String username)  
public boolean logoff(String username)   
public boolean joinGroup(BaseUser user, String groupName)   
public boolean leaveGroup(BaseUser user, String groupName)  
public MsgSendError processMessage(String source, String destination, String message)  
  
    The above first five methods are simply getters to access info from users and groups. They all acquire the read lock to run, allowing other readers at the same time to concurrently access them.  
  
public void shutdown()

Shuts down the ChatServer and removes all users and groups.  
We acquire the write lock and set IsDown to true. All Messages in transit before this is called should have a read lock and would therefore be completed before shutdown completes.  
  
public LoginError login(String username)

If username is not in allNames and 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, too full, or username taken already.

    Because multiple Users may try to login at the same time, concurrency errors of 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.  
  
Acquires write lock  
    # of users < MAX\_USERS?  
        Releases write lock  
        Returns USER\_DROPPED  
    Name exists already?  
        Releases write lock  
        Returns USER\_REJECTED  
    Adds user

Have the user connect

Releases write lock  
Return USER\_ACCEPTED  
  
public boolean logoff(String username)

If 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.  
    The write lock is acquired, as we must make changes to allNames and users. After the deletion is completed, the lock is downgraded to a read lock. 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.  
  
Acquires write lock  
    User exists?  
        Removes user  
        Have user leave all its groups  
        Release write lock  
        Returns true  
Releases write lock  
Returns false  
  
public boolean joinGroup(BaseUser user, String groupName)

Fetches ChatGroup from groups and have the group add the user. If the group doesn't exist, add the group first by adding a mapping between groupName and the ChatGroup in groups. If User is already in the ChatGroup, or if there's no more room in the group, returns false. Otherwise returns true.  
    Because it is possible that the ChatGroup doesn't exist yet, we first acquire the write lock in case we have to add to groups. Once it is determined that the ChatGroup  exists, we downgrade from a write lock to a read lock. The write lock is no longer necessary as we won't be adding to groups. But the read lock is necessary in case the user joins but the group gets deleted.  
    The issue of joining groups and having too many users in a group because of concurrent joining is handled in the ChatGroup class.  
  
Acquires write lock  
    Group exists?  
        Acquires read lock  
        Releases write lock  
        Join group  
        Releases read lock  
        Return joined successful?  
    Add group  
    Acquires read lock  
    Releases write lock  
    Join group  
    Releases read lock  
    Returns joined successful?  
  
public 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 doesn't exist in 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. We downgrade to a read lock if it turns out that the User won't be the last one leaving.  
  
Acquires write lock  
    Group exists?  
        Users in group is 1?  
            Leave successful?  
                Delete group  
                Releases write lock  
            else  
                Releases write lock  
                Returns false  
        else  
            Acquires read lock  
            Releases write lock  
            Leave successful?  
                Releases read lock  
                Returns true  
            else  
                Releases read lock  
                Returns false  
Releases write lock  
Returns false  
  
public MsgSendError processMessage(String source, String destination, String message) - Returns success if the destination 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 receive or broadcast. Otherwise, we return an appropriate failure message if either the source or destination is not valid.  
  
Acquires read lock  
    source is a user?  
        destination is a user?  
            Creates message object  
            Have destination user receive  
        destination is group?  
            Creates message object  
            Have group broadcast  
            Failed?  
                Releases read lock  
                Returns failure message - NOT\_IN\_GROUP  
        else  
            Releases read lock  
            Returns failure message - INVALID\_DESTINATION  
    else  
        Release read lock  
        Return failure message - INVALID\_SOURCE  
    Release read lock  
    Returns success message - MESSAGE\_SENT  
  
The above method processMessage uses the following MsgSendError enum.  
public enum MsgSendError{  
    MESSAGE\_SENT  
    INVALID\_DEST  
    INVALID\_SOURCE  
    NOT\_IN\_GROUP }

**User class**  
  
The User class represents an individual who can interact with other Users via the ChatServer. All Users have the ability to login and logoff, join and leave groups, and send and receive messages, either with other Users or with ChatGroups. Each user also maintains its own chat log of each conversation it participates in, starting from the time it joins the conversation to the time it leaves.  
  
**Description of Fields**  
String username  
ChatServer server //used to access server methods  
HashMap<String, ChatLog> chatlogs //used to maintain user chat logs, keyed by destination (which is either destination username or group name)  
List groupsJoined //used so server can find out which groups user belongs to, when the user logs off  
  
As with the ChatServer’s user and group lists, we chose to use a HashMap on source name for  differentiating ChatLogs. This way we can efficiently decide which chat log to write a Message to.  
  
**Description of Methods**  
public void connected() // starts user thread  
public MsgSendError send(String dest, String message) // passes message to chat server  
public void msgReceived(Message message) // adds message to correct chat log  
public void msgReceived(String msg) // used by BaseUser for testing  
private boolean joinGroup(String groupname) // puts request to chat server to join specified group  
public boolean leaveGroup(String groupname) // requests to chat server to be removed from group  
public boolean logoff() // request to chat server to be removed from user list  
public List<String> getGroups() // returns a list of groupnames in which the user is a member  
  
MsgSendError send

The user passes a message and the destination to the ChatServer by calling the latter’s processMessage method, which then takes care of wrapping the message and passing it along. Its return type is a MsgSendError, exactly the same as the ChatServer’s processMessage.  
  
void msgReceived

The user receives the message object and hashes the source field to obtain the correct ChatLog. A ChatLog is just a linked list of Message objects, so the new Message is simply appended to the end to signify a successful logging. If the ChatLog does not yet exist for the User who sent the Message, a new ChatLog is created.  
  
boolean joinGroup

This method is called for both creating and joining groups. It calls the ChatServer’s joinGroup method; if it returns true, then the User will create a new ChatLog for the group and add the group’s name to the list of groups that it participates in. Otherwise, the method simply returns false.  
  
boolean leaveGroup

This method calls the ChatServer and asks to be removed from the ChatGroup. The server will then respond with either a success or failure message. If successful, the User can also remove the group from its list of ChatGroups.

**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, so that Users receive Messages in the correct order. 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  
  
**Description of Methods**  
getSource() // getter method to retrieve source  
getDest() // getter method to retrieve dest  
getTimestamp() // getter method to retrieve dest  
getContent() // getter method to retrieve content  
  
  
**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.  
  
**Description of Fields**  
LinkedList<Message> log  
BaseUser user // user that owns log  
String source // group or user that messages are from  
  
**Description of Methods**  
String toString() // returns all messages printed in string form

**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**  
String name // unique to all other groups and users  
Hashmap<String, User> userlist //ensures maximum of ten users in the group and allows broadcast to members

int MAX\_USERS // maximum number of allowed users: default of ten

    A Hashmap was chosen so that it can be quickly determined who belongs to a certain group.  A Hashmap also returns an iterator which can be used to iterate over all the Users to broadcast the incoming message.  
  
**Description of Methods**  
boolean onCreate()

Creator of group automatically becomes a member of the group

boolean onDelete()

Once every user has left a group, the group is destroyed

boolean synchronized joinGroup(String user)  
Called when user wishes to join group; adds entry to hashmap  
boolean synchronized leaveGroup(String user)

Called when user wishes to leave group; deletes from hashmap and ensures that user will no longer receive messages from this group

boolean Synchronized forwardMessage(Message msg)

Synchronized because multiple users may attempt to send a message at the same time and so allows concurrent access to the chat group; broadcast of the message to all current user

    Synchronized is used in several of ChatGroup’s methods, since there will be reads and writes to the ChatGroup’s userlist.  joinGroup() and leaveGroup() must be synchronized because multiple joins and leaves may occur simultaneously in the same group; if the userlist is not properly locked, some of the updates may be lost.  forwardMessage() must be synchronized because multiple Users may attempt to send Messages to the ChatGroup simultaneously, and the ChatGroup must be able to handle that gracefully.  This forces the Messages to be processed one at a time and removes the problem of clashes.  
  
**Test Plan Overview**  
Our plan is to use the JUnit testing framework to incrementally build up a test suite that will give us confidence that our code works the way we intended, as well as for regression testing as we add more features or change existing code.

Our test cases will be divided into three categories as follows:

1)      Unit testing to test individual methods

a. Have the server log a user in and make sure that the User object was correctly created and added to the userList of the ChatServer.  
b.      Have a User join a nonexistent ChatGroup and check that a new one is correctly added to the groupList of the ChatServer.  
c.       Have a user join an existent ChatGroup and check that the user is correctly added to the userList of the ChatGroup that the User joined.  
d.      Have a User send a message to another User and check that the Message is written to the recipient’s chatLog.  
e.      Have a User try to send a message to a nonexistent User and check that it is handled correctly.  
f.       Have the ChatServer attempt to log in a User with a name that’s already taken and check that it is handled correctly.  
g.      Have a User send a message to a ChatGroup and check that it is written to each User’s chatLog correctly.  
h.      Have the ChatServer attempt to log in a User when it is full and check that it is handled correctly.  
  
2)      Behavioral testing to test that our code behaves the way we intend it to, semantically.  
a.       Have two Users log in and have a simulated conversation, check that the logs match the conversation and that they match each other, and have both Users log off and check that the state of the ChatServer is correct afterwards.  
b.      Have three Users log in, one create a ChatGroup and the other two join and have a simulated conversation. Then check that the chatLog for each User matches the actual conversation and that they match each other.  
c.       Fill up the ChatServer with Users and have a new User try to join; then have a User leave the ChatServer and check that the new User is able to join the ChatServer.  
d.      Fill up a ChatGroup with Users and have a new User try to join; then have a User leave the ChatGroup and check that the new User is able to join the ChatGroup.  
  
3)      Specific testing against possible concurrency issues  
a.       Have two Users log off at the same time.  
b.       Have two Users send a message to the same User at the same time (check chatlog consistency).  
c.       Have multiple Users in a ChatGroup send a message at the same time (also check message order).  
d.       Have multiple Users try to join an empty ChatGroup at the same time  
e.       Have multiple Users try to join a ChatGroup that has only one more available slot at the same time.  
f.       Have two Users send each other messages at the same time and check that message order is consistent.  
  
For all three categories, we plan to make sure to test corner cases.