Thread-Safe Chat Server

Overall Design

Our task is to implement a chat server which can accommodate multiple concurrent users. Users can create and join multiple groups. Users can send unicast messages to each other as well as broadcast messages to members of their groups.

Chat semantics are handled by a centralized `ChatServer` running on its own thread. It delegates user and group management to a child thread, `UserManager`, which handles requests to read and modify users and groups as in the Reader-Writers Problem. It delegates message sending to another child thread, `MessageDispatcher`, which receives messages to and sends messages from a message buffer as in the Producer-Consumer problem.

Interaction with the Chat Server

First of all, when we initialize the chat server, we set the maximum number of users that it can take. In our implementation, we do not put users on the wait list if the server is full (they have to keep trying).

Inside the chat server class, we have two hash tables: one for storing user objects (`BaseUser`), the other for storing group objects (`Group`), using names as their keys for both tables. Whenever a user tries to sign in, we first check if the server is full. If it is full, we show the message “sorry, the chat server is full.” Otherwise, we try to store the new user and see if there's a hit in the hash table. If there is, it means that the user name has been taken, and we have to reject user's request by showing the message "the user name already exists". If not, we store this user in to the user hash table.

When a user tries to create a group, we check if a group with the same name exists by checking the group hash table. If a group with the same name exists, we reject user’s request by saying “the group name already exists”. If not, we initialize a group object with the user added into it and specify the maximum number of users allowed in this group. Then, store the group in to the group hash table. When a user tries to join a group, we first have to check the group hash table to see if the targeting group exists. If not, show the error message “the group does not exist”. If the group exists but is full, show the message “sorry, the group is full.” If the group exists and is not full, we add the user into the group object. If a user is already in a group and request to join the same group, we show the error message “the user is already in the group”.

To differentiate between `BaseUser` and `Group`, we require the users to provide a name that starts with `u_`. For example, instead of `mike`, we require the name to be `u_mike`. And for Group names, we require the prefix `g_`.

When the user wants to see the list of users or the list of groups on the chat server, we can use

`hash.keys()` to see all the user names or group names. We can also provide users with the number of users/groups using `hash.count()`. However, before actually providing any information, we have to check if the requesting user is in our chat server system.

When a user logs out of the chat server, we remove that user object from the user hash table. And for each group in the server, we check if the group contains this user. If it does, remove the user.

Tests

- Users who try to login with a name which is already taken by a logged in user are rejected.
  - Case 1
    - A user logs in as `u_user1`.
    - Another user tries to login as `u_user1` and is rejected.
- No more than 100 users may be logged in to the server. Additional users who try to login are rejected.
  - Case 1
    - 100 users login as `u_user1` through `u_user100`.
    - Another user tries to login as `u_user101` and is rejected.
- Users who try to create a group with a name which is already taken by an existent group are rejected.
  - Case 1
    - A user logs in as `u_user1`.
    - `u_user1` creates group `g_group1`.
    - `u_user1` tries to create group `g_group1` and is rejected.
- No more than 10 users may join a chat group. Additional users who try to join are rejected.
  - Case 1
    - 11 users login as `u_user1` through `u_user11`.
    - `u_user1` creates group `g_group1`.
    - `u_user1` through `u_user10` join `g_group1`.
    - `u_user11` tries to join `g_group1` and is rejected.
- Users can join multiple groups.
  - Case 1
    - A user logs in as `u_user1`.
    - `u_user1` creates group `g_group1`.
    - `u_user1` creates group `g_group2`.
    - `u_user1` joins `g_group1`.
    - `u_user1` joins `g_group2`.
- Invalid users (not logged in) who try to request server information are rejected.
  - Case 1
    - A user (not logged in) requests server information and is rejected.
- Only one thread at a time may modify the users or groups.
  - Case 1
    - 2 users login as `u_user1` and `u_user2`.
    - `u_user1` creates group `g_group1`.
    - Concurrently:
      - `u_user1` joins `g_group1`.

- `u_user2` creates group `g_group2`.
- A user logs in as `u_user3`.

1 user is allowed to proceed, while the other users block.

Message Delivery

To address the feature of message delivery, we have created a Message class. The message class has four instance variables:

String receiver, String sender, String text, TimeStamp timestamp, and Boolean error

While receiver, sender and text will be initialized with constructor's parameters, the constructor will initialize the timestamp instance variable to the time the message was created and set the boolean variable error to be false. The error boolean variable, when set to true will signal that this message was not successfully received.

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, or if we used an array, we need to keep track of the meaning of the indexes. This is very inconvenient.

A user will also keep a ArrayList of Messages. This ArrayList is used as a log to keep track of all the conversations that a user has.

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.

The send method will support both sending messages to users and groups, the dest string can be either the name of the user or the name of the group. We will be using prefix on the name to distinguish. A user will have a prefix u_ on the name. Once we check that dest is a user, we will instantiate a new message object. The new message object will incorporate the source as the user itself, receiver's name and the message. we will put this message object into the ArrayList of Messages, so that the user will have a log of the conversation. The user will also print the message in the format: "<source name>\t<destination>\t<sequence number>\t<message>" to stdout.

The User will use the chatServer handle to send the message to the chat server and let the server to resolve message delivery. When the chat server starts, the chat server will call the start method of the MessageDispatcher. This will keep the MessageDispatcher in a loop, so that it will continue to deliver messages. Once the message object is sent to the server, the MessageDispatcher class is responsible for putting the messages onto the message queue by the enqueue. We will be using the java data structure queue to implement the message. Because queue is FIFO, messages can be dispatched in order they received, but we need to place a lock around the enqueue method to make sure multiple users can't try to enqueue at the same time. If the message queue is full, MessageDispatcher should use the source

from the message object to retrieve the sender and send message to let the sender know that the server is busy and message is not sent.

To deliver a message, MessageDispatcher will dequeue a message object from the message queue in FIFO order, and examine this message object to get the name of the receiving user and the msg. MessageDispatcher will construct a new string in the format: "<source name>\t<destination>\t<sequence number>\t<message>". To get the user who will receive the message, MessageDispatcher will use the receiver's name to get the user object from the UserManager. GetUserByName 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. Inside the msgReceived method, the receiver should parse the msg and reconstruct a new Message object and put the new Message object in the log of conversation. MsgReceived will also print the message to stdout.

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. MessageDispatcher will also send a message to the sender saying that message was not received. If both users log out at the same time, MessageDispatcher will do nothing.

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 send messages to a group, the send method will check if dest has a group prefix g_. If dest is a groupname, the user will use the chatServer handle to call the UserManaer to find out a list of users in the group. The next step is very similar to send messages to single user. We will put the message to the log of conversation. We will iterate through the list of users' names. For each user name except the user itself (because he/she is in the group will be in the list as well), we will call getUserByName from UserManager to get the user object. we will call the send method for each user object. Because sending messages to a 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. MessageDispatcher will also send a message to the sender saying that message was not received. We will not try to call listUsers() if getUserGroup returns null.

Tests

- Users are able to send and receive unicast messages to and from each other.
  Case 1
  2 users login as `u_user1` and `u_user2`.

u_user1 sends a unicast message to u_user2.
u_user2 receives the unicast message from u_user1.

- Users are able to send broadcast messages to groups to which they belong.
  Case 1
  3 users login as u_user1 through u_user3.
  u_user1 creates group g_group1.
  u_user1 through u_user3 join g_group1.
  u_user1 sends a broadcast message to g_group1.
  u_user1 through u_user3 receive the broadcast message from u_user1.
- Users who try to broadcast messages to groups to which they do not belong are rejected.
  Case 1
  A user logs in as u_user1.
  u_user1 creates group g_group1.
  u_user1 sends a broadcast message to g_group1 and is rejected.
- Only one thread at a time may add a message to the message buffer.
  Case 1
  3 users login as u_user1 through u_user3.
  Concurrently:
  - u_user1 sends a unicast message to u_user2.
  - u_user2 sends a unicast message to u_user3.
  - u_user3 sends a unicast message to u_user1.

  1 user is allowed to add his message to the message buffer, while the other users block.
  Case 2
  3 users login as u_user1 through u_user3.
  u_user1 creates group g_group1.
  u_user1 through u_user3 join g_group1.
  Concurrently:
  - u_user1 sends a unicast message to u_user2.
  - u_user2 sends a broadcast message to g_group1.
  - u_user3 sends a broadcast message to g_group1.

  1 user is allowed to add his message to the message buffer, while the other users block.

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 lock the enqueue action. Therefore, at any given one time, only one message can be enqueued to the message queue. Because

only the MessageDispatcher will dequeue messages, only one message will be delivered at any give one time, so we will not need to lock the dequeue method.

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

We will be using a HashTable in the group class to store all user objects.
Because we actually use a group class 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. This user should not receive the message. To ensure this, we need to synchronize the HashTable, so that all the method calls on the HashTable will be kept in order. 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).

Because only the user object itself will be calling the send method, we don't have concurrent issue with sending messages. Messages will always appear in the order as they are sent. We don't have to worry about messages received because only one message will be delivered by the MessageDispatcher.

A user cannot see a message that was posted before she joined the group is secured because we have put a synchronized key word to the HashTable in the group class. This ensures that we will get the most up to date list of users and message will not be sent to a user before she joined the group. When the user leaves the group, MessageDispatcher loses the handle to the user object. Therefore, we will not be able to send the message.

* Correctness Constraints:

1. Why are the last two constraints important for correctness?

If we don't put the constraint that only the current members of the group can consume messages posted to the group, then we are implying that users who's logged out and who haven't logged in will all be able to see the messages. This is not correct.

If the order of events and message posts are not consistent, then a user can see a message before he joins the group or after he left the group. Both situations are not correct.

2. Can a message be lost if these constraints are not maintained?

Yes, a message can be lost if these the order of messages are not preserved. Imagine if a message is not synchronized and was pushed onto the message queue until much later, then user could be logged out by the time when message is delivered.

Tests

- For unicast messages between two users, messages are delivered in the order in which the chat server receives them.

Case 1

2 users login as `u_user1` and `u_user2`.
`ChatServer` receives a sequence of messages from/to `u_user1` and `u_user1`.
`ChatServer` delivers messages to their respective recipients in the order received.

Case 2

2 users login as `u_user1` and `u_user2`.
Concurrently:

- `ChatServer` receives a sequence of messages from/to `u_user1` and `u_user1`.
- Other users send messages and do other activities.

`ChatServer` delivers messages to their respective recipients (`u_user1` and `u_user2`) in the order received.

- For broadcast messages between members of a group, messages are delivered to each member in the order in which the chat server receives them.

Case 1

2 users login as `u_user1` and `u_user2`.
`u_user1` creates `g_group1`.
`u_user1` and `u_user2` joins `g_group1`.
`ChatServer` receives a sequence of broadcast messages from `u_user1` and `u_user1` to `g_group1`.
`ChatServer` delivers messages to both `u_user1` and `u_user2` in the order received.

Case 2

2 users login as `u_user1` and `u_user2`.
`u_user1` creates `g_group1`.
`u_user1` and `u_user2` joins `g_group1`.
Concurrently:

- `ChatServer` receives a sequence of broadcast messages from `u_user1` and `u_user1` to `g_group1`.
- Other users send messages and do other activities.

`ChatServer` delivers messages to both `u_user1` and `u_user2` in the order received.

- At any time, only group members receive group broadcast messages. In particular, a member should not receive messages received by the chat server before he joins. Also, a member should not receive messages received by the chat server after he leaves.

Appendix A: Class API

ChatServer API

```
package edu.berkeley.cs.cs162

public class ChatServer
extends Thread
implements ChatServerInterface, Runnable
```

```
Constructor Detail
       public ChatServer()
              Creates and initializes an instance of ChatServer

Method Detail
       public LoginError login(String username)
              Logs a new user into the chat server, subject to
              the constraint that user name
              shall be unique and max number of user is not reached.
              Specified by:
                     login in interface ChatServerInterface
              Parameters:
                     username - the user name of the User to be logged in.
              Returns:
                     a LoginError message describing success or one of the failure
modes.

       public boolean logoff(String username)
              Logs a specified user out of the chat server.
              Specified by:
                     logoff in interface ChatServerInterface
              Parameters:
                     username - the user name of the User to be logged off.
              Returns:
                     true if the logoff was successful.

       public synchronized boolean joinGroup(BaseUser user,
                                 String groupname)
              Adds user into a chat group named groupname, subject to the
              constraint that each group has a maximum capacity. Creates
              new group if no such group named groupname exists.
              Specified by:
                     joinGroup in interface ChatServerInterface
              Parameters:
                     user - the object representing the User that wants
                            to join the group.
                     groupname - the name of the group.
              Returns:
                     true if the user was successfully added into group.

       public synchronized boolean leaveGroup(BaseUser user, String groupname)
              Removes user from chat group.  Deletes the group if group
              is empty afterwards.
              Specified by:
                     leaveGroup in interface ChatServerInterface
              Parameters:
                     user - the object representing the User that wants
                            to leave the group.
                     groupname - the name of the group.
              Returns:
                     true if the user was successfully removed from group.
```

```
	public void shutdown()
		Shuts down the chat server and kills all subserver threads.
		Specified by:
			shutdown in interface ChatServerInterface

	public synchronized BaseUser getUser(String username)
		Fetches the user with username username from the pool of users.
		Specified by:
			getUser in interface ChatServerInterface
		Returns:
			the user with username username, or null if no such user exists.

	public void start()
		Starts up the chat server and spawns the MessageDispatcher and
UserManager.
		Overrides:
			start in class Thread

	public void run()
		runs the chat server thread.
		Specified by:
			run in interface Runnable

	private void createGroup(String groupname)
		Creates a new chat group.
		Parameters:
			groupname - the name assigned to the group after group creation.

	public synchronized void destroyGroup(String groupname)
		Destroys a chat group.
		Parameters:
			groupname - the name of the group.

	public void send(Message message)
		Receives a message and passes it to the MessageDispatcher.
		Parameters:
			message - the Message object to be passed to the
MessageDispatcher.
```

ChatUser API

```
package edu.berkeley.cs.cs162

public class ChatUser
extends BaseUser, Thread
implements Runnable

Constructor Detail
	public ChatUser(ChatServer handle)
		Creates and initializes an instance of ChatUser with a specified
```

```
            handle to the ChatServer it will be communicating through.
            Parameters:
                  handle - a handle to a ChatServer

Method Detail
      public void connected()
            Starts the thread that represents this particular user. Called
            when the user successfully connect to the server.
            Overrides:
                  connected in ChatUser

      public void send(String dest, String msg)
            Sends messages from the user to a destination by injecting them
            into the ChatServer
            Overrides:
                  send in BaseUser
            Parameters:
                  dest - Destination of the message.  Can be a user or group.
                  msg - Message to be sent.

      public synchronized void msgReceived(String msg)
            Called by a MessageDispatcher when a message is received by the thread
from        another user.
            Overrides:
                  msgReceived in BaseUser
            Parameters:
                  msg - The received message.  Format is:
                        <source name>\t<destination>\t<sequence number>\t<message>
```

Group API

```
package edu.berkeley.cs.cs162

public class Group

Constructor Detail
      public Group()
            Cover method, using 10 as the default max number of users
      public Group(int maxNumUsers)
            Creates and initializes an instance of Group with a specified
            max number of users.
            Parameters:
                  maxNumUsers - the max number of users allowed in the group.

Field Detail
      public final NAME
            The name of the group.

Method Detail
```

```
	public synchronized boolean addUser(BaseUser user)
		Adds a new user into group.
		Parameters:
			user - the user object to be added into the group.
		Returns:
			true, if the user was successfully added into the group.

	public synchronized void removeUser(BaseUser user)
		Adds a new user into group.
		Parameters:
			user - the user object to be removed from the group.

	public synchronized String listUsers()
		Lists the users in the group.
		Returns:
			a list of users by name and including description.

	public synchronized int userCount()
		Returns the number of users in the group.
		Returns:
			the number of users in the group.
```

Message API

```
package edu.berkeley.cs.cs162

public class Message

Constructor Detail
	public Message(String sender,
		       String receiver,
		       String text)
	Creates and initializes a new Message instance and time-stamps it.
	Parameters:
		sender - the name of the user that sent the message.
		receiver - the name of the user that is to receive the message.
		text - the message body.

Field Detail
	public final String RECEIVER
		The name of the user that is to receive the message.

	public final String SENDER
		The name of the user that sent the message.

	public final String TEXT
		The message body.

	public final TimeStamp TIMESTAMP
		The time the message was created.
```

MessageDispatcher API

```
package edu.berkeley.cs.cs162

public class MessageDispatcher
extends Thread
implements Runnable

Constructor Detail
    public MessageDispatcher()

Method Detail
       public void start()
              Starts up the message dispatcher thread.
              Overrides:
                     start in class Thread
       public void run()
              Runs the message dispatcher thread.
              Specified by:
                     run in interface Runnable

       public void stop()
              Terminates the message dispatcher.
              Overrides:
                     stop in class Thread

       public synchronized void enqueue(Message message)
              Enqueues an incoming message for the message dispatcher to deliver
later.
              Parameters:
                     message - the Message object to be delivered later.

       private synchronized void deliver()
              Delivers the messages waiting in queue in FIFO order.
```

UserManager API

```
package edu.berkeley.cs.cs162

public class UserManager
extends Thread
implements Runnable

Constructor Detail
       public UserManager()
              Creates and initializes an instance of UserManager

Method Detail
       public synchronized boolean addUser(String username)
              Adds a new user to the chat server.
              Parameters:
                     username - the user name of the User to be
                                created and added into the chat server.
              Returns:
                     true, if the user was successfully added to chat server.
```

```
public synchronized boolean addUserToGroup(BaseUser user,
                                String groupname)
       Adds a user to a chat group.
       Parameters:
              user - the object representing the User that
                     wants to join the group.
              groupname - the name of the group.
       Returns:
              true, if the user was successfully added to the chat group.

public synchronized void removeUserFromGroup(BaseUser,
                                 String groupname)
       Adds a user to a chat group.
       Parameters:
              user - the object representing the User that wants to
                     leave the group.
              groupname - the name of the group.

public synchronized void listGroups()
       Lists the groups present on chat server.
       Returns:
              a list of groups, along with their descriptions and attributes.

public synchronized void createGroup(String groupname)
       Creates a new chat group in the chat server.
       Parameters:
              groupname - the name assigned to the group after group creation.


public synchronized void deleteGroup(String groupname)
       Deletes a chat group from the chat server.
       Parameters:
              groupname - the name of the group.

public synchronized String listUsersOfGroup(String groupname)
       Lists the users in a group.
       Parameters:
              groupname - the name of the group.
       Returns:
              a list of users that belong in the group, with additional
              descriptions and attributes, or null if no such group with
              the specified group name is found.

public synchronized void removeUser(String username)
       Removes a user from chat server
       Parameters:
              username - the name of the user.

public synchronized BaseUser getUserByName(String username)
       Finds and returns a user by username
       Parameters:
              username - the name of the user.
```

```
		Returns:
			a BaseUser, or null if no user with the specified username is
found.

	public synchronized Group getGroupByName(String groupname)
		Finds and returns a group by its name
		Parameters:
			groupname - the name of the group.
		Returns:
			a Group, or null if no group with the specified group name is
found.

	public synchronized int getNumberOfUsers()
		Returns the number of users logged into the server.
		Returns:
			the number of users logged into the server.

	public synchronized ArrayList getListOfUsers()
		Returns a list of users logged into the server.
		Returns:
			an ArrayList of all BaseUsers logged into the server.

	public void start()
		Starts up the user manager.
		Overrides:
			start in class Thread

	public void run()
		Runs the user manager thread.
		Specified by:
			run in interface Runnable

	public void stop()
		Terminates the message dispatcher.
		Overrides:
			stop in class Thread
```