Thread-Safe Chat Server

Tony Dear, Jason Jee, Harry Jiang, Dustin Kwong, Paul Ruan

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.

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 will keep track of how many Users and ChatGroups currently exist on the server. This information should be publicly viewable via getter methods. Users are expected to log in with a unique name, as well as log out when finished. They can also create ChatGroups (which also require a unique name), as well as join or 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? It is necessary to prevent any User from reading the wrong information while the ChatServer is busy dealing with someone else.

- 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.
    - 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.

- 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 can 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.

Implementation

As the specifications above imply, we will design one class for each of the above types of objects. Thus, at any given time there are three main types of threads running: ChatServer, User, and ChatGroup. Messages and ChatLogs also exist, but only for logging purposes. A message goes through the ChatServer, which then wraps it into a Message object. The ChatServer then fowards the object to the receiver, whether User or ChatGroup. If it is the latter, the ChatGroup broadcasts the Message to all its Users. The recipient then add the Message to his ChatLog.

Concurrency is taken into account wherever a function needs to run atomically. To ensure that their execution runs in a critical section, we will apply Locks where appropriate, as described in the code explanation below.

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.

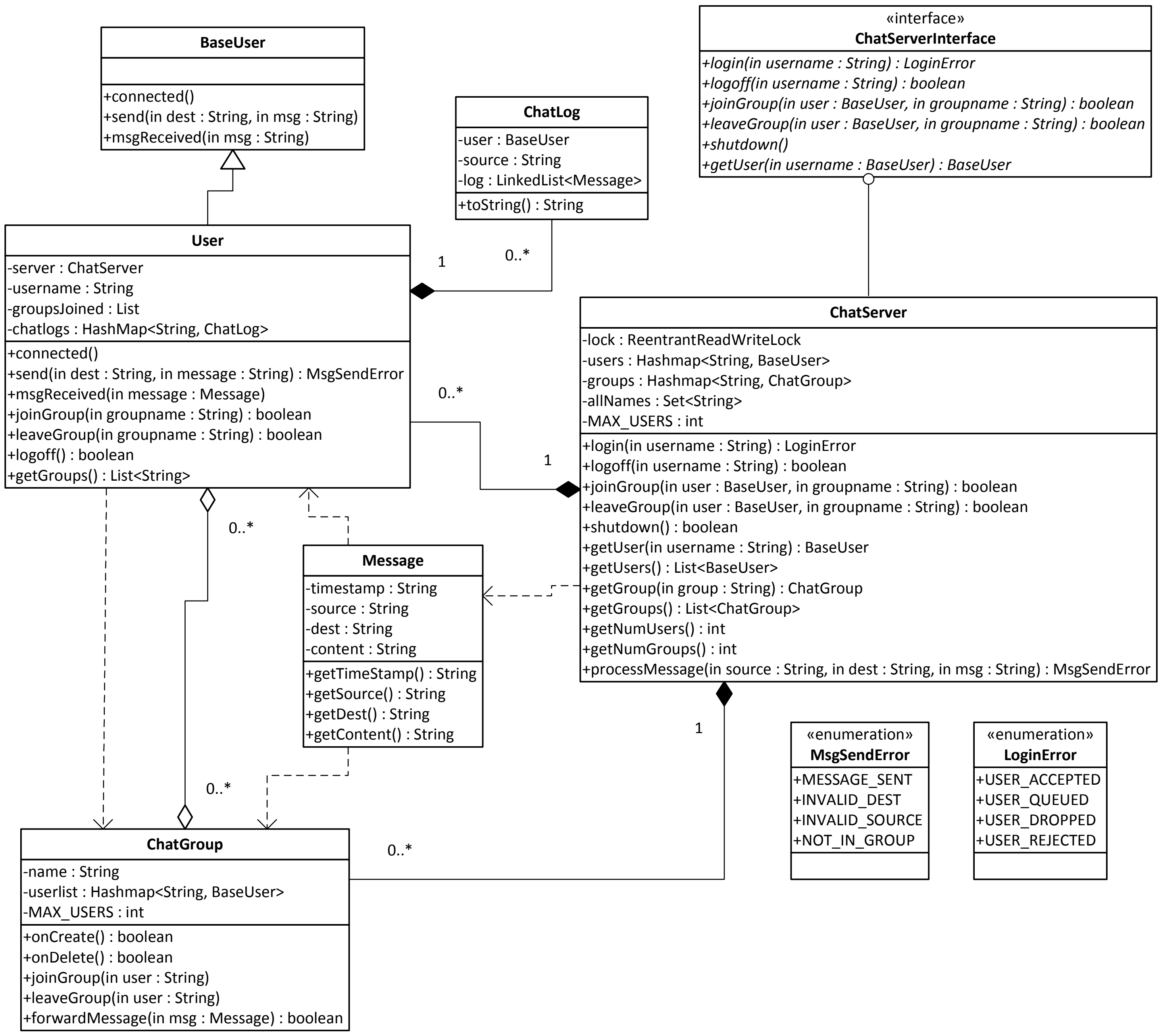
g. Have a User send a message to a ChatGroup and check that it is written to each User’s chatLog correctly.
h. Have the ChatServer attempt to log in a User when it is full and check that it is handled correctly.

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.

Class Diagram


BaseUser
+connected()
+send(in dest : String, in msg : String)
+msgReceived(in msg : String)
ChatLog
-user : BaseUser
-source : String
-log : LinkedList<Message>
+toString() : String
«interface»
ChatServerInterface
+login(in username : String) : LoginError
+logoff(in username : String) : boolean
+joinGroup(in user : BaseUser, in groupname : String) : boolean
+leaveGroup(in user : BaseUser, in groupname : String) : boolean
+shutdown()
+getUser(in username : BaseUser) : BaseUser
User
-server : ChatServer
-username : String
-groupsJoined : List
-chatlogs : HashMap<String, ChatLog>
+connected()
+send(in dest : String, in message : String) : MsgSendError
+msgReceived(in message : Message)
+joinGroup(in groupname : String) : boolean
+leaveGroup(in groupname : String) : boolean
+logoff() : boolean
+getGroups() : List<String>
1
0..*
ChatServer
-lock : ReentrantReadWriteLock
-users : Hashmap<String, BaseUser>
-groups : Hashmap<String, ChatGroup>
-allNames : Set<String>
-MAX_USERS : int
+login(in username : String) : LoginError
+logoff(in username : String) : boolean
+joinGroup(in user : BaseUser, in groupname : String) : boolean
+leaveGroup(in user : BaseUser, in groupname : String) : boolean
+shutdown() : boolean
+getUser(in username : String) : BaseUser
+getUsers() : List<BaseUser>
+getGroup(in group : String) : ChatGroup
+getGroups() : List<ChatGroup>
+getNumUsers() : int
+getNumGroups() : int
+processMessage(in source : String, in dest : String, in msg : String) : MsgSendError
0..*
1
0..*
Message
-timestamp : String
-source : String
-dest : String
-content : String
+getTimeStamp() : String
+getSource() : String
+getDest() : String
+getContent() : String
1
«enumeration»
MsgSendError
+MESSAGE_SENT
+INVALID_DEST
+INVALID_SOURCE
+NOT_IN_GROUP
«enumeration»
LoginError
+USER_ACCEPTED
+USER_QUEUED
+USER_DROPPED
+USER_REJECTED
0..*
0..*
ChatGroup
-name : String
-userlist : Hashmap<String, BaseUser>
-MAX_USERS : int
+onCreate() : boolean
+onDelete() : boolean
+joinGroup(in user : String)
+leaveGroup(in user : String)
+forwardMessage(in msg : Message) : boolean