Network Programming

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

Overview

In this part of the project, we are using the interface that we built from phase 1 to make a transition to a server-client model of the chat server. More specifically, we are using the tools provided by the Java API to use an underlying network protocol to handle the interaction between a chat server and client users on separate machines. Thus, we do not need to worry about how the TCP protocol works, but rather how we deal with the abstracted layer above.

The user client will connect remotely to the server, possibly from a different machine than the one that the server is on, and send commands to the server. The latter will then receive these commands and process them as specified, making the correct modifications and carrying out the right actions on the server side. Afterward, the server will usually send an acknowledgement back to the client, either saying that the command was successful or that something went wrong. Our main goal is to build the protocol so that normal execution is efficient, subtle concurrency bugs are prevented, and all errors are handled gracefully.

Specifications

One of the main new classes that we've added for this project is the ChatClient class that represents the client. Its main method will process commands from standard input; all valid commands are specified, while any other commands are rejected. Upon receiving the command and parameters, the client then sends the appropriate data over the socket to the server. The new ChatServer now takes care of listening from its socket for any new messages that have arrived.

In the ChatServer class, we're now adding a login queue. This will keep track of all new user logins when the server has reached maximum capacity. More importantly, we have added the functionality to receive connections from ChatClients and send data to ChatClients. We also now use our old User class to map connected users on the server to their respective clients.

One special case on the server side is that we have to keep track of a timer for each user who has connected but has not yet logged in. The specification is that any user who has connected but has been not logged in for 20 seconds is automatically timed out and disconnected. The effect of this is to discard any state corresponding to this user from the server and to let the client know that he has timed out.

We also deal with both intended and unintended disconnects. The correct way for the client to leave is to first log out and then disconnect. But the user may also disconnect without doing the former. In that case, the server must handle the log out for the user. In addition, the client may also disconnect unintentionally, without even sending the server a disconnect command.

Finally, in addition to our usual functionalities from last phase, the client is also able to sleep. In this state, he just sleeps for the specified amount of time and doesn't take any command (but he does continue to receive messages).

Considerations

Because everything is happening over a network, there is now an additional source of uncertainty other than concurrency: the network may not always operate as expected. For example, things can run slowly for no reason, or things may have been lost along the way. An important thing to build into the design is the isolation of clients from each other. What happens to one client should not affect other clients. More abstractly, each client should be able to interact with the server as though he were the only one in the network.

- All specifications from phase 1 must still hold here.
- Users can disconnect asynchronously or be slow to respond, and the server must handle these situations appropriately.
- If a client logs out while he has messages in transit, those have to be gracefully.
- Users who are connected but do not log in need to be disconnected after 20 seconds.
- Any interruption of network activity has to be handled gracefully.
- Since we have been sending both messages in the form of ACKs and objects (Messages) among different users, we now have to ensure that they are TCP-safe.

Implementation

As with project 1, we have two main classes again. The ChatServer class will carry over, while we now have a new ChatClient class that encapsulates our old User class. We also have a new TransportObject class to encapsulate all commands and messages going along the network. This implements the Serialized interface to ensure that all messages are written and read correctly on both ends. This also ensures that nothing is corrupted or lost over the channel.

The ChatClient is first initialized with two threads. One processes any send jobs, while the other solely listens for replies from the server. For the ChatServer, we simply have one thread to keep it alive. This is the state of both sides before any interaction occurs.

When a client connects, we immediately instantiate a thread to keep track of its countdown state. If the client does not log in before 20 seconds has passed, this thread dies and the client is effectively disconnected from the server. Otherwise, if the client does log in, whether successfully or queued, we instantiate a User object for this client, and within this object we make three new threads: one for receiving commands from the client, one for sending replies back, and one to process any local activity within the server relating to the client.

If the client logs out, then we kill two of its threads and leave the third one to run the timer again. As before, if the client does not do anything before 20 seconds pass, the server automatically disconnects him. Otherwise, the client can log in again, and the process continues as before.

Upon connection, two sockets on either side, client and server, are established and connected to each other. Any subsequent interactions will occur via this channel. The client may want to log

in, send a message, or join/leave a group. To do so, the Client class makes a TransportObject that encapsulates the appropriate command along with the necessary parameters. This is then placed on an output stream that the socket grabs from to send to its equal on the server.

Eventually, the server side socket and client listener thread will see something on its input stream and read an object from it. This is recast as a TransportObject again, and we can then parse its parameters to see what kind of command it was. Assuming it was a valid command, the server then processes it as it did back in phase 1. All the necessary changes are then made to the User object corresponding to the client.

Once the job is done, the server then provides an ACK signaling either success or failure. This goes on the output stream of the user in the chat server, mirroring the one on the client side. The sender thread of the user then picks this up and sends it over via the sockets back to the client. Finally, the client listener thread sees this and processes with the TransportObject.

As before, concurrency is a problem. This especially applies to the input and output streams for the User objects, since multiple threads may be writing to one. Not only the send thread for a User has access to it, but so do other User threads when placing messages on the receive queue, as well as the ChatServer. Thus, we implement blocking queues on a layer above the streams. These queues come with the Java guarantee that concurrent modification is protected. Thus, when the output streams are actually written to, TranportObjects are done one at a time.

Finally, we had to handle unintentional disconnects. This was solved with the broader solution of catching SocketExceptions. Essentially, whenever we found that a socket was closed, we would get exceptions on our input and output streams. Thus, we would just disconnect the client on the server side and clean up any leftover user state corresponding to the client.

A small aside is that we have another helper class (like TransportObject) called SocketParams. This helps us to avoid passing way too many parameters when calling methods dealing with sockets. This helper class encapsulates necessary fields, including an input stream, and output stream, and the socket itself.

ChatServer class

The ChatServer, which handles the majority of the functionality, is now modified to use the networking socket abstraction. This allows remote clients to connect to the server over a network to issue commands to the ChatServer and communicate with other client users.

The ChatServer will receive commands and messages through the socket abstraction, specifically through TransportObjects used by our server to properly communicate with its clients. Clients will also establish a TCP connection with the server by way of this abstraction. Through this the client will send commands to the server. The server then processes these packets to extract commands from the client. From there the server can call the appropriate methods to log on or off users, deliver messages to individuals or groups, etc. Throughout this process, the connection between the client and the server may be ended, both intentionally and unintentionally.

Our current ChatServer has been modified in several ways. First is the implementation of the wait queue. The ChatServer has a limit on the number of users that can be logged on at any onoe time. Any additional users must be placed on the queue, which also has a finite capacity. They will be allowed to connect and log on in order once other users have logged off and disconnected. In addition, clients that have connected but not logged on will be timed out and disconnected after 20 seconds. Clients are able to issue text commands, which are then encoded in our protocol. The ChatServer must extract from the packets the correct commands issued by the client and then call the appropriate methods to handle the command.

New Fields

BlockingQueue<User> waiting_users	//Blocking queue to keep track of queued users
int MAX_WAITING_USERS	//Max number of users on the waiting queue
int TIMEOUT	//Time that a client gets before timeout
ServerSocket mySocket	//Socket that the server keeps for itself
ExecutorService pool	//Used for timer as described below

The first three fields are self-explanatory. We need mySocket to call accept(). This is done whenever a new client connects and we need to make a new socket to correspond to the one on the client side.

Modified Methods

We have modified most of our methods to include the TransportObject protocol. This includes logins, logoffs, joins, and leaves. The latter two are taken care of in the two new methods mentioned below. For the former two, we simply modified the methods to construct new TransportObjects corresponding to the correct success or fail message. At this point, we simply place this object inside the output stream so that the socket will eventually pick it up and send.

New Methods

joinAck(User, String gname, ServerReply)	//Constructs TransportObject ACK for join group
leaveAck(User, String gname, ServerReply)	//Constructs TransportObject ACK for leave group
startNewTimer(SocketParams)	//Starts a new countdown timer with given socket
run()	//Main method running while server is alive

startNewTimer() does what its name suggests—every time a client connects, we create a new timer associated with that client to keep track of how much time has passed. To do so, we call the invokeAll() command with the pool, telling it to timeout at 20 seconds. If this does happen, we just construct a new TransportObject telling the client that he timed out.

While not technically a new method, run() now actually performs some functionality as opposed to before. As the server thread listens on its socket, it will create a new socket for any incoming connections. In addition, it creates a new thread to keep track of the timer that we create for this connected client. This is essentially done via pool, an ExecutorService that provides us this functionality. The task that we pass into this timer is an object of the class Handler, which we define below. The server then invokes this Handler “task” until the client times out, which is checked in the run() method.

Subclasses

Handler
This helper class implements the functionality described above. As it is a simple encapsulation of a login "task," all it has to keep track of is the socket in question, as well as the input and output streams. It implements Callable, so that the timer task can actually "execute" it. This is done via the call() method. This method simply loops on the empty input stream (in other words, we are just waiting for a login command from the client).

To get this task to actually execute, we also create a new thread that has a handle to this Handler. This thread has some special properties, so we decided to create a second subclass as described below. In the run() method of ChatServer, we start up this thread, which then starts up the Handler task. In essence, this means that the call() method of the Handler object is executed, which just loops on the empty input stream.

Once call() finally gets a TransportObject from our input stream and checks that it is indeed a login command, we extract the username parameter from the object and check perform the usual checks from phase 1, as shown in Figure 1 below. This effectively concludes the limbo state that the client remains in while he is connected but not logged in. Alternatively, the client may also time out, in which case the thread will deal with it as described below.

FirstThread
This is just a simple way for us to start up the initial server thread that keeps things up and running. We decided to make a new class for said thread so that it can keep its own state in the form of a Handler as described above. In the run() method, we invoke the Handler task for 20 seconds (in other words, we listen for that amount of time). When we time out, we simply construct a new TransportObject indicating timeout and send this back to the user via the socket and streams. Then we just close the socket, severing ties with the client in question.

Other Functionalities
Figure 2 below also shows the state of the server while processing join and leave after the client has successfully logged in. They are largely the same as from phase 1.

When joining and leaving groups, we again perform the same sequence of checks for this user and the requested group. First, we need to check if this group exists. If it does, then we need to check if we are already a member of said group. If we're leaving, then we can leave successfully; if we are joining, then we should not join it again. If the group exists and we're not a member, then we can join successfully if there is room in the group. Otherwise, we create a new group with the name requested, but to do so we also check that the name is valid in that it has not already been taken by a user.

Sending a message is also much along the same lines. Logging out and disconnecting are shown in Figure 1 as well, but all of the aforementioned functionalities will be implemented in the User class instead. As a reminder, this is the class on the server side that we use to keep track of each individual client that connects to the server.

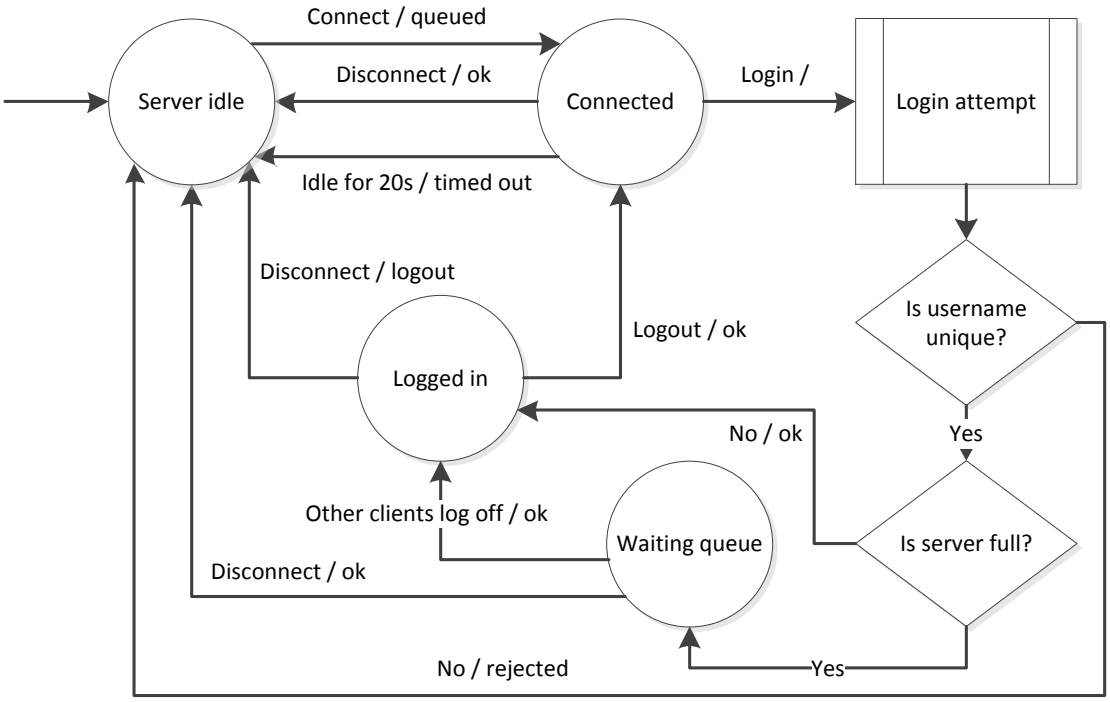

Figure 1: Server state diagram for logins/logouts

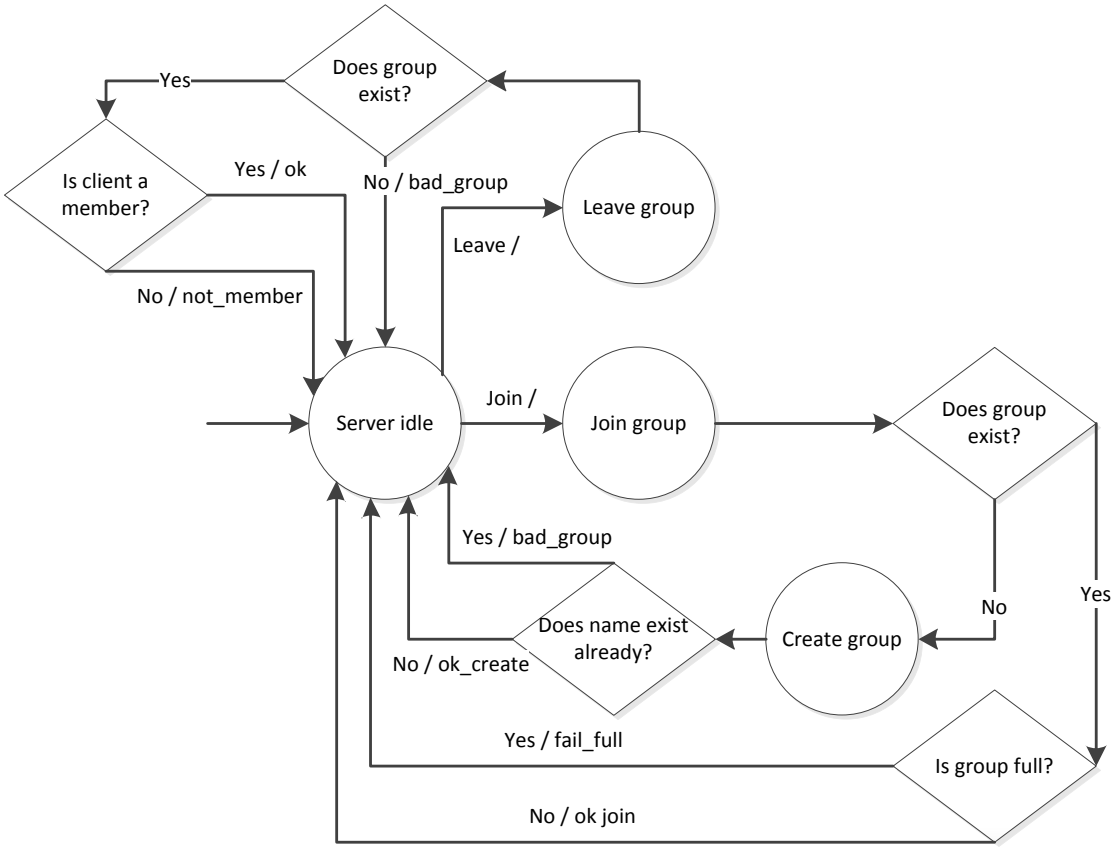

Figure 2: Server state diagram for joining/leaving groups

User class

As with the ChatServer class, the User class has remained largely the same from phase 1, except for the modifications that we have added so that it will work with the new protocol. In particular, each user now has to keep track of its own socket to the respective client. Going along with this are also the input and output streams that the client reads from and writes to. We also throw in a couple of new threads to handle the listening and sending to the client on the remote end.

New Fields

```
Socket mySocket                                  //Handle to socket
ObjectInputStream received                       //Stream to read from
ObjectOutputStream sent                          //Stream to write to
Thread receiver                                  //Listener thread
Thread sender                                    //Writer thread
BlockingQueue<TransportObject> queuedServerReplies
boolean pendingLogoff
```

While we have mentioned the input and output streams several times before, here we make them concrete as to how we are implementing them. We chose to use Java's built-in ObjectStreams for their simplicity and ease of use. More importantly, they only support Serializable objects, which meets the specifications that any objects that are sent to and fro must be preserved. By using both the ObjectInputStream and ObjectOutputStream, we can be confident that our data is unharmed. We demonstrate how to use both of these objects in the methods below.

Here, we have also added to handles to two new threads that were previously nonexistent. The receiver thread is the simpler one; it simply runs the method processCommand() continuously. This just listens for new arrivals of TransportObjects from the server.

Conversely, the sender thread polls from queuedServerReplies. This is the layer of abstraction that we provide above the output stream to protect against concurrency. Because multiple threads may be writing to the output stream at any one time, it is possible for concurrency bugs to arise. By first placing this BlockingQueue in the middle, it ensures that TransportObjects to be sent are written to the queue one at a time. Then the sender thread can just poll from the queue with the guarantee that everything it reads from the queue is complete and correct.

The last new field is the Boolean for pendingLogoff. This is in addition to our old one, which is a plain loggedOff variable. We use pendingLogoff as an intermediary. In order to carry out any other commands, the user must not be pending logoff. This is because the field is set to true when logoff() is called. Here, we first set the new field to be true and send back an ACK saying that the server has received the request and will process it shortly. Between this time and when the sever actually finishes logging the client out, the client has not yet been fully logged out; it is simply pending while the server cleans up its state.

Of course, we have not removed any of the fields that have carried over from phase 1. The user still needs to keep track of its username and chatLogs, as well as the ChatGroups it has joined and its send message queue.

Modified Methods

acceptMsg(Message)

From the first phase, acceptMsg simply logged the message as having been received. For this project, we now have to actually send the message back to the recipient client, rather than just logging it. To do so, we simply create a new TransportObject parameterized with fields such as the source, destination, and message as listed in the specifications.

run()—We now have to also include a call to sendClientAck() whenever we send a message from the user's send queue. This may or may not be successful. As described below, the ack method will place an appropriate response back onto the user's blocking queue.

New Methods

queueReply(TransportObject) //Adds the given TransportObject to the blocking queue
processCommand() //Receives a TransportObject and parses it
setSocket(Socket, ObjectInputStream, ObjectOutputStream)
sendClientAck(MsgSendError, MessageJob) //Prepares and queues TransportObject
logoffAck() //As above
logoff()
disconnect()

The processCommand() method forms the core of our receive protocol. What it essentially does is try to read TransportObjects from the received queue, all of which come from the server. It then does a series of attempted matchups to determine what kind it is. Firstly, if we get a null object, this means that the client has disconnected unexpectedly, at which point we call disconnect() to clean up. We actually do the same thing if the command is a legitimate disconnect; for our server then, both types of disconnect are done in the same way.

If we read a logout command, then we first call the logoff() method. Then we must again set the timer on, so that if the user does not log in again after 20 seconds he is kicked off the server. All other commands are relatively straightforward; depending on whether it is a join, leave, or send, we simply call the appropriate methods with the appropriate parameters. All of these are just extracted from the TransportObject; they are just fields, as explained below in the section about the TransportObject class.

The setSocket method has three parameters—all three are passed in by the external caller so that we can set the respective fields for this user. In this case, it is the ChatServer who passes these parameters in. This is because the server has already created these objects for us to use when the user first logs in; it simply passes these to us so that the user has the correct handles to them.

Both sendClientAck and logoffAck perform similar functions to the ChatServer's joinAck and leaveAck methods. They simply let the client know whether their desired effect was successful or not. The reason we chose to have the former in User and the latter in ChatServer was because these were their respective roles; the ChatServer takes care of group memberships, while the User has to worry about sending messages and logging itself off (hence the new methods). In effect, these methods all create appropriate new TransportObjects and then queues them up.

Both logoff() and disconnect() are new, since we did not have them for phase 1. This is because we had the ChatServer take care of this functionality; however, here we decided that it was easier on the server for each user to take care of its own cleanup when leaving. The latter method actually just calls the former and then closes the socket after sending over TransportObject ACK.

The boolean field pendingLogoff is useful for logoff(). This is because we must set it to true so that we know not to accept any new commands that come in. In the meantime, we must clear out our blocking queue to send back any ACKs or messages that have not yet been received by the client. Once we finish this, we can set loggedOff to be true and finish the job.

Other Functionalities

As before, the User class mainly takes care of the sending and receiving of messages. Nothing has changed for this protocol; its state diagram is shown below.

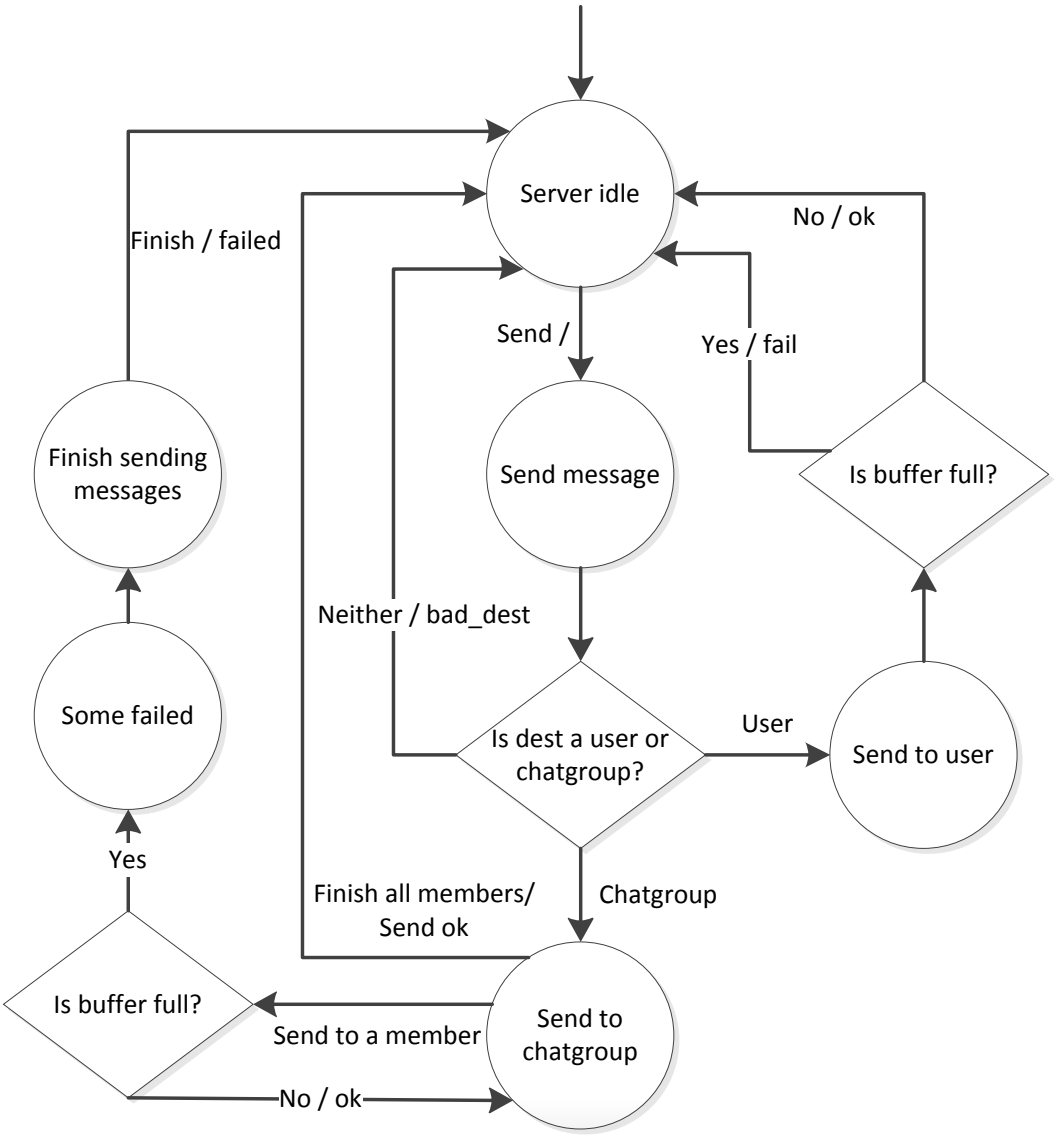

Figure 3: Server state diagram for processing messages

ChatClient class

The ChatClient is the main new class here, providing a way for users to interact with the ChatServer and other users connected to it. Any ChatClient object utilizes two threads. One thread processes commands issued by the user; this can come via the command line or a file provided at the command line. The other thread processes TransportObjects received from the server; they can be ACKs for previous commands or messages sent by other clients. Such organization thus mirrors that of the server, with separate threads for separate functionalities.

Most commands in the spec are synchronized. This means that we must wait for an ACK from the server before we can continue; this tells us that the server has received our request and has processed it (whether it was successful or is thus indicated here). For such commands, we would have the main thread wait. It cannot process new ones until the receiver sees a corresponding ACK from the server. At this point, it can signal the main thread to continue.

A few replies that come from the server are asynchronous, in that they can arrive at any time and that we do not expect them. We also handle these in a similar manner.

Description of Fields

```
Socket mySocket                          //Handle to socket
ObjectInputStream received               //Stream to read from
ObjectOutputStream sent                  //Stream to write to
Thread  receiver                         //Handle to receiver thread
BufferedReader commands                  //A reader from which to take in commands
boolean connected, isLoggedIn, isQueued  //Booleans describing the client's state
boolean isWaiting                        //True if client is waiting for ACK
Command reply                            //Reply to look for if isWaiting is true
```

The above fields are fairly straightforward. The first four are identical to that of the ChatServer, except that they are now implemented in the ChatClient as well. Each one performs exactly the same function as its counterpart in the server. One consideration is that we only need one separate receiver thread, since the main thread of the client also acts as the sender.

The BufferedReader simply reads from System.in, which is where the commands are coming from. These are of course read as strings, which we then parse in the methods below.

Each of the booleans simply describe what they are named. The last one, isWaiting, is set whenever the client has sent a request to the server and is waiting for an ACK back. Whenever this happens, we set isWaiting to be true and the reply to be whatever Command that we expect to hear get back. For example, if we requested a join group command, then we need to make sure that whatever ACK we get back is for the join Command.

As an aside, Command is simply an enum that we have defined to make these operations easier. It will be described later along with the ServerReply enum, which has a similar advantage. TransportObjects include both enums as fields, assisting in the encapsulation and decapsulation process.

Description of Methods

output(String out)	//Sends the given string to standard out
connect()	
disconnect()	
login(String username)	
logout()	
join(String gname)	
leave(String gname)	
send(String dest, int sqn, String msg)	
sleep(int time)	
receive()	//Process any received TransportObjects
processCommands()	//Process any inputted commands
run()	//Constantly calls processCommands() above

Most of these methods operate the same way and do as their name suggests. In particular, every single method from disconnect() through send() work the same way. They first check the state of the client—whether he is connected and/or logged in, depending on which command we want. Next we create a TransportObject corresponding to the Command we want along with any parameters. We then write this object to our sent stream so that it will eventually go to the server. In the meantime, since we assume all command to be synchronous, **including the send command**, we set isWaiting to be true and the reply to be the command of interest.

The sleep() and connect() methods work a bit differently. If we want to sleep for the specified period of time, we simply call Thread.sleep(time) to achieve this. Here we require no interaction with the server, since the client is just sleeping locally so as to ignore any requests from the command line during this time.

On the other hand, for connect() we actually need to start up an interaction with the server. After first checking that we are not already connected, we end up creating several new objects—the socket and input/output streams associated with that socket. If we have gotten to this point, that means we have been successful so far, and we set connected to true, and finish by starting up the receiver thread. Otherwise, if we have not been successful, then we will probably end up throwing an exception, which is caught and leads to an equivalent output of a rejection by the server. Unlike other ACKs, we actually have to generate these ourselves on the client side, since the server has no way of sending anything to us before we are connected.

receive() and processCommands() run in much the same way, in that they both read something and then parse it to call the right method. The former reads the input stream coming from the server. Whenever it reads a non-null object, we first check if we are even waiting for a ACK at all. If we are, then we process it to make sure that it matches the one we expect; then we reset isWaiting and signal the main thread to continue to process more commands from standard in. If the TransportObject comes in asynchronously, we just output it without resetting anything.

processCommands() reads its input from the BufferedReader called commands. Here we use a somewhat more tedious approach, tokenizing the input on whitespace and separating the command and parameters that way. Once we have the right command, we just call the method

associated with it, such as send() or leave(). One subtle point is the fact that we parse the send command such that we can have multi-word messages. Thus, the actual message sent can consist of many words, rather than just one.

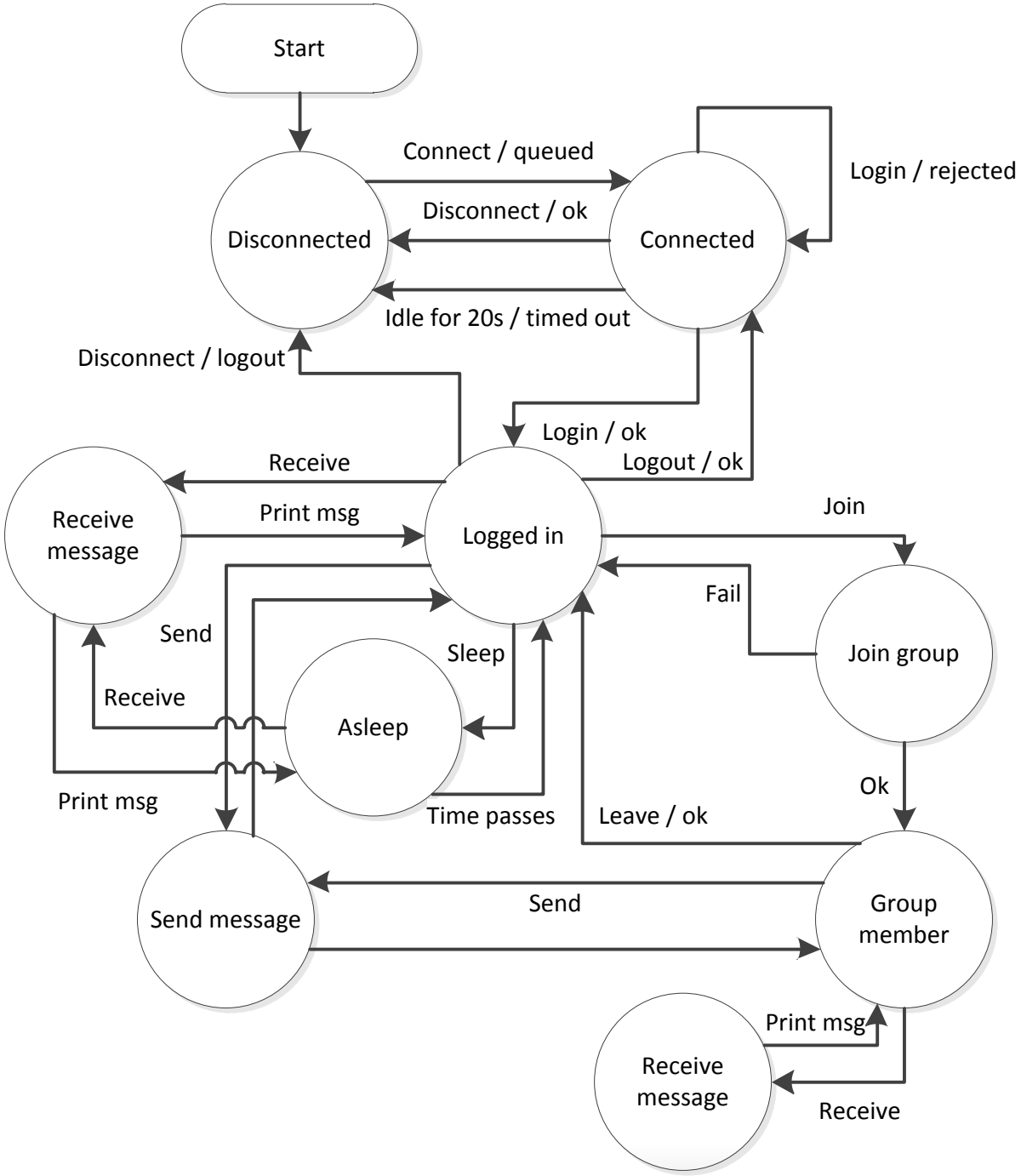

Figure 4: Client state diagram

TransportObject class (Serialized)

As has been mentioned many times already, we have actually created our own class to implement the protocol that use. In such a scheme, all requests sent back and forth between the server and client are equivalent. That is, from an outsider's perspective there is no differentiation among simple join/leave requests, more complex send message requests, and even ACKs from the server. This is possible because we package every possible thing that can be sent over the network into one class that can encapsulate everything. As a carrier class, it does not support any actual methods (other than getters) and simply exists to be Serialized.

Description of fields

Command command
String username
String gname
String sender
String dest
int sqn
String msg
int time
ServerReply reply

Most of these are obvious and do not require explanation. For any one transport object, many of these fields are null. That is because a TransportObject usually only supports one type of command, and each command only has a few relevant parameters. Leaving a group does not require anything but the gname, while sending a message will actually need most of the fields (except for maybe gname). Again, we note that both Command and ServerReply are enums, as described below.

SocketParams class

This class only exists to make our code cleaner and easier to use. In particular, when we make a new Handler in the ChatServer or restart the timer after logoff, we need to pass the respective methods three things: the socket, the input stream, and the output stream. Instead of always passing these items back and forth, we decided to just encapsulate these three objects in a class.

Command enum

This enum lists all the possible commands that the client can receive from the command line: connect, disconnect, login, logout, join, leave send. This is a important field in a TransportObject, since it often determines what kind of request that it was meant to be.

ServerReply enum

Similar to the Command enum, this lists all the possible replies that can come from the server that are not already covered by Command. It also includes any asynchronous replies, such as timeout or receive. Like Command, these also play a large role in a TransportObject, since they tell us the result of the server trying to carry out the job.

Testing and Conclusions

Because we were now working over a network, there was not much benefit that unit tests provided us as they did in phase 1. In addition, we already had most of this code written, so there was no point in doing unit tests on them again. To thoroughly test our code, we developed an extensive testing suite as described below.

Description of test cases:

A) Connecting, disconnecting, logging in and logging out
- Attempt to execute commands before connecting and check that nothing happens.
- Attempt to execute commands after connecting but before logging in and checking that nothing happens.
- Connect and disconnect and check correctness of output.
- Connect and log in, then log out and disconnect and check correctness of output.
- Connect and log in, then disconnect without logging out first and check that log out is still outputted.
- Test multiple people being connected at the same time.
- Test trying to connect multiple times.

B) Login queue
- Connect and login to a full server and check that it becomes login queued. Then test a user leaving the server and user on the queue being logged in.
- Connect and try to log in to a full server with a full login queue and expect login rejected.

C) Timeout
- Test connecting and check timeout.
- Test connecting, logging in and out and then timeout.

D) Groups
- Test connecting, logging in and creating a group.
- Test leaving a group.
- Test logging out while in a group.
- Test disconnecting while in a group.
- Test joining the same group twice.
- Test leaving a group the user is not a member of.

E) Sending messages
- Test sending a message to a group.
- Test sending a message to a specific user.
- Test sending to a group that the user is not a member of.

We believe that this covers a lot of border cases that would have given us the most trouble. To do all this, we had one of our machines be the server, and then connected multiple clients to that machine via other ones. Because our tests have passed all of the above cases, we can have high confidence in the integrity and functionality of our chat server. The application that we have built should be able to handle multiple users performing different activities, all the while operating safely atop an unpredictable layer of networking and concurrency.