**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.  
  
**Additional Fields**  
BlockingQueue<User> waiting\_users //Blocking queue to keep track of queued users

MAX\_WAITING\_USERS //Max number of users on the waiting queue

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.  
  
**Additional Methods**  
joinAck(User, String, ServerReply) //Constructs TransportObject ACK for join group  
leaveAck(User, String, ServerReply) //Constructs TransportObject ACK for leave group

startNewTimer(SocketParams)

run()

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.

**Miscellaneous**

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**



Figure 3: Server state diagram for processing messages

**ChatClient class**  
The ChatClient class provides a way for users to interact with the ChatServer and other users connected to the ChatServer. We will use two threads for a ChatClient object. One thread processes commands issued by the user. The other thread, Receiver, processes messages received from the server. For synchronized commands we must wait for completion, or a reply from the server. In such cases, we would have the main thread, the command processing one, wait. Once the ChatClient receives a possible candidate message it is waiting for in the Receiver thread, it can signal the main thread.  
  
**Description of Fields**  
private Socket mySocket        //Used to communicate with the server  
private Map<ChatLog> logs        //Maps group name or user name with each log  
private InputStream commands    //Stream of commands from standard input  
private InputStream received        //Stream of data received via mySocket  
private Thread receiver        //Calls receive() in run  
private boolean connected        //True if connected  
private String reply            //Reply from server if ChatClient is waiting for one  
private volatile boolean isWaiting    //True if client is waiting for a reply from server.  
private int sqn                //sequence number  
  
If client is waiting for specific reply from server, the client would check the value of reply when it receives from the server.  
  
**Description of Methods**  
private boolean connect(String hostname, int port)

//Tries to connect to server with hostname on port. Waits for completion.

//Prints: login [OK, REJECTED]

private void disconnect()            //Disconnects from server. Prints: disconnect OK  
private boolean login(String username)    //Sends server login request. Waits for completion.

//Prints: login [OK, REJECTED]

private void logout()            //Sends server logout request. Waits for completion.

//Prints: logout OK

private boolean join(String gname)    //Sends server request to join group. Waits for completion.  
    //Prints: join gname [OK\_CREATE, OK\_JOIN, BAD\_GROUP, FAIL\_FULL]  
private boolean leave(String gname)

//Sends server request to leave group gnome. Waits for completion.

//Prints: leave gname [OK, BAD\_GROUP, NOT\_MEMBER]

private void send(String dest, int sqn, String msg)  
    //Sends server request to send to dest msg. Increments sqn.

//Prints: send sqn [OK, BAD\_DEST, FAIL]

private void receive()  
    //Reads from received stream, and process.  
    //Calls signalReceived() if ChatClient waiting for reply from server

//Prints where appropriate: sendack sqn FAILED, receive sender dest "msg"

private void sleep(long time)        //Does nothing for time milliseconds. Prints: sleep time OK  
public Map<ChatLog> getLogs()    //Returns logs  
private synchronized void processCommands()  
    //Reads from commands stream. Parses and executes each command.  
private synchronized void signalReceived(String msg)  
    //Sets reply to msg and wakes up command processing thread.



Figure 4: Client state diagram

**Protocol**  
  
**Syntax**  
Primary elements are command, param count and first parameter reserved for the message. There are also optional additional parameters for more specific commands.  
  
(command):(parameter count) - (parameter1 byte-length):(parameter1) - (param2 byte len):(param2)...  
  
**Command Enumeration**  
- LOGIN    //client request to log in to server  
- LOGOUT    //client request to log out of server  
- JOIN        //client request to join a group  
- LEAVE    //client request to leave a group  
- SEND        //client command to send msg  
- RECEIVE    //server command to client to receive message  
- CONNECT     //connect to server  
- DISCONNECT //disconnect from server  
- SLEEP     //sleep user on server  
- CONFIRM    //used when client receives server response to command

//can also be used for various other commands

- ERROR     //errors encountered for client’s request  
  
Examples:  
"SEND:3 - 3:bob - 5:hello - 2:15" //send example - send “hello” to bob with sqn=15  
“RECEIVE:3 - 4:john - 5:hello - 10:1300158203” //receive example - server to client recv msg from john   
“LOGIN:1 - 5:alice” //login example - login as “alice”  
“LOGOUT:0” //logout example - log client out  
“JOIN:1 - 8:newgroup” //join example - sends request to join group “newgroup”  
“LEAVE:1 - 8:newgroup” //leave example - sends request to leave group “newgroup”  
“CONFIRM:1 - 14:connect QUEUED” //confirm example - queued during logging attempt  
“ERROR:1 - 12:send 15 FAIL” //error example - dropped message with sqn=15  
  
**Semantics**  
The client class and chatserver will have methods that encode and decode strings that conform to the protocol. These strings, which are streams of bytes, are then sent from client to server or server to client through the Java socket interface. After receiving these streams, the client and/or server thread will parse the streams and use the decode method to recover the commands to the server or responses to the client. The protocol will include a message parameter that will contain one of the following: a message from the client, a message to the client, confirmation info, or error info. Other parameters will include the length of the message and an enumeration of the different commands that can be sent between client and server. Additional parameters may be specified for logging info, like timestamp and sqn number.  
  
Depending on the command, additional parameters will be expected. For example, a send command will require an additional destination parameter that will specify the destination user or group the client intends to send the message to. The client will encode the command, message, and destination into a protocol-specified stream that is sent to the server. The server will then decode the stream, and check if the operation requested by the client is valid. If valid, a confirmation is returned to the client. Otherwise, an appropriate error message is then encoded and sent to the client.

**Test Plan Overview**  
Our plan is to continue to use the JUnit testing framework that we used in the first part of the project. Again, these tests will give us confidence that our code works the way it should as well as for regression testing when we add features or change existing code.  
1)      Unit testing  
Examples:  
a)      A client tries to connect to the server, test that the response is correct for OK and QUEUED situations.  
b)      A client tries to disconnect, test that the response is OK and handled appropriately by the server.  
c)       A client tries to log in, test that the response is correct for the two situations of REJECTED and OK.  
d)      A client tries to log out, test that the server handles it appropriately.  
e)      A client tries to join an existing group, test that the output is correct in each possible scenario.  
f)       A client tries to send a message to a group, test that the output is correct in each possible scenario.  
2)      Integration testing  
Examples:  
a)      A few users connect to the server, two of them send valid messages to each other, check that everything was logged correctly.  
b)      A few users connect to the server, three of them create and join a group, send valid messages to each other, all three leave the group, then log off. Check that the server handled everything correctly and that messages were logged correctly.  
c)       A user connects the server, doesn’t log in, and is disconnected, then reconnects and logs in. Check that the disconnect happened correctly and that the user successfully reconnects and logs in.  
d)      A user connects to the server, logs in, and has another user send him messages, and logs out while the messages are in transit.