# CS162 Project 2

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 will be building the network protocol to handle the interaction between a chat server and client users. The user client will connect remotely to the server and be able to send commands to the server. Our main goal is to build the protocol so that normal execution is efficient and all errors are handled gracefully.

## Specifications

For this part of the project, we're adding a ChatClient class that represents the client. Its main method will process commands from standard input and send the appropriate data over the socket to the server. For the ChatServer class, we're adding a login queue as well as the functionality to receive connections from ChatClients and send data to ChatClients.

## Considerations

Because everything is happening over the network, there is always a certain amount of uncertainty in terms of how slow things are, as well as how unpredictably events can occur. 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.
Key considerations include the following:

- 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 processed accordingly.
- Users who are connected but do not log in need to be disconnected after 5 seconds.
- Any interruption of network activity has to be handled gracefully.

## Implementation

As with project 1, we will 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 will have a thread on the server side for every client that is logged on, as well as an external thread to maintain the ChatServer. Finally, we will use a protocol according to the design described below.

## ChatServer class

The ChatServer, which handles the majority of the functionality, will now be modified to use the networking socket abstraction. This will allow 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, and we will implement a protocol that will be used by our server to properly communicate with its clients. Clients will establish a TCP connection with the server by way of the socket abstraction. Through this the client will send commands to the server to log itself on and join/create ChatGroups. The server will process the 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 will have to be modified in several ways. First is the implementation of the wait queue. The ChatServer has a limit on the number of users and any additional users must be placed on the queue. They will be allowed to connect and logon 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 5 seconds. Clients will be able to issue text commands, which will be 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**

| | |
|---|---|
| Wait Queue | //When num of users hits limit; append to wait queue |
| Int portnum | |
| ServerSocket myServerSocket | |

**Additional Methods**

| | |
|---|---|
| processPackets() | //process TCP packets for commands |
| processCommand() | //process commands and arguments |

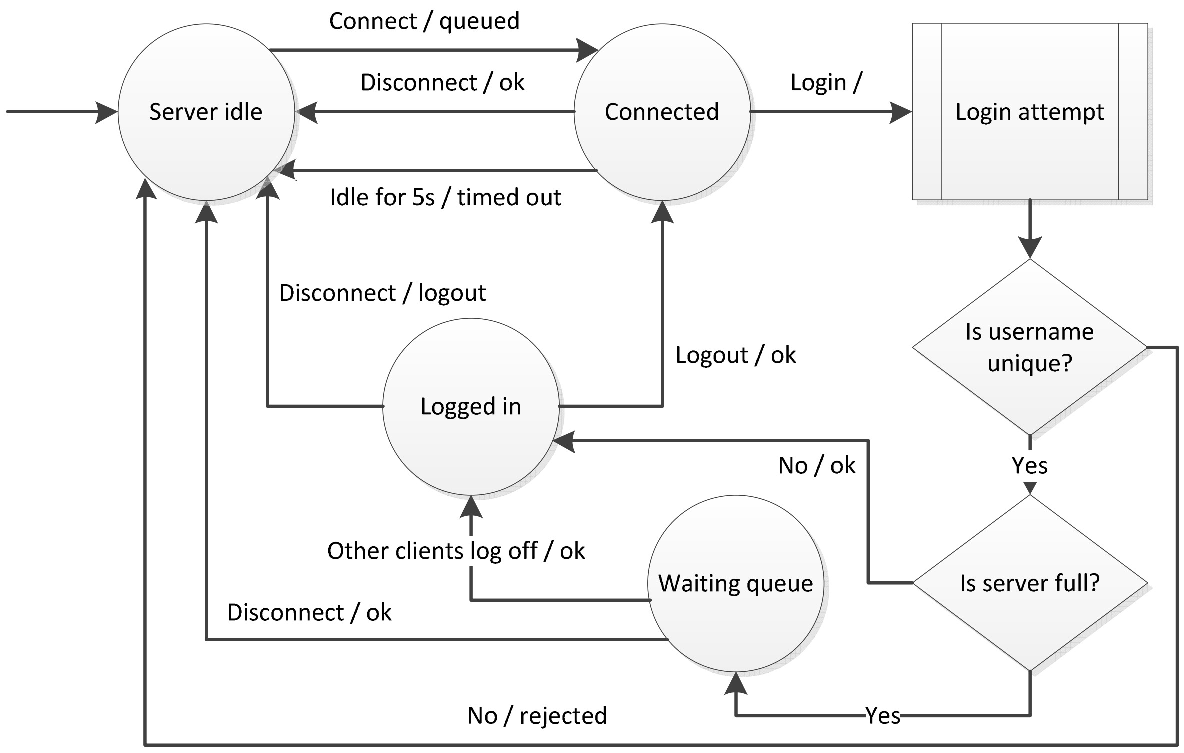

Figure 1: Server state diagram for logins/logouts

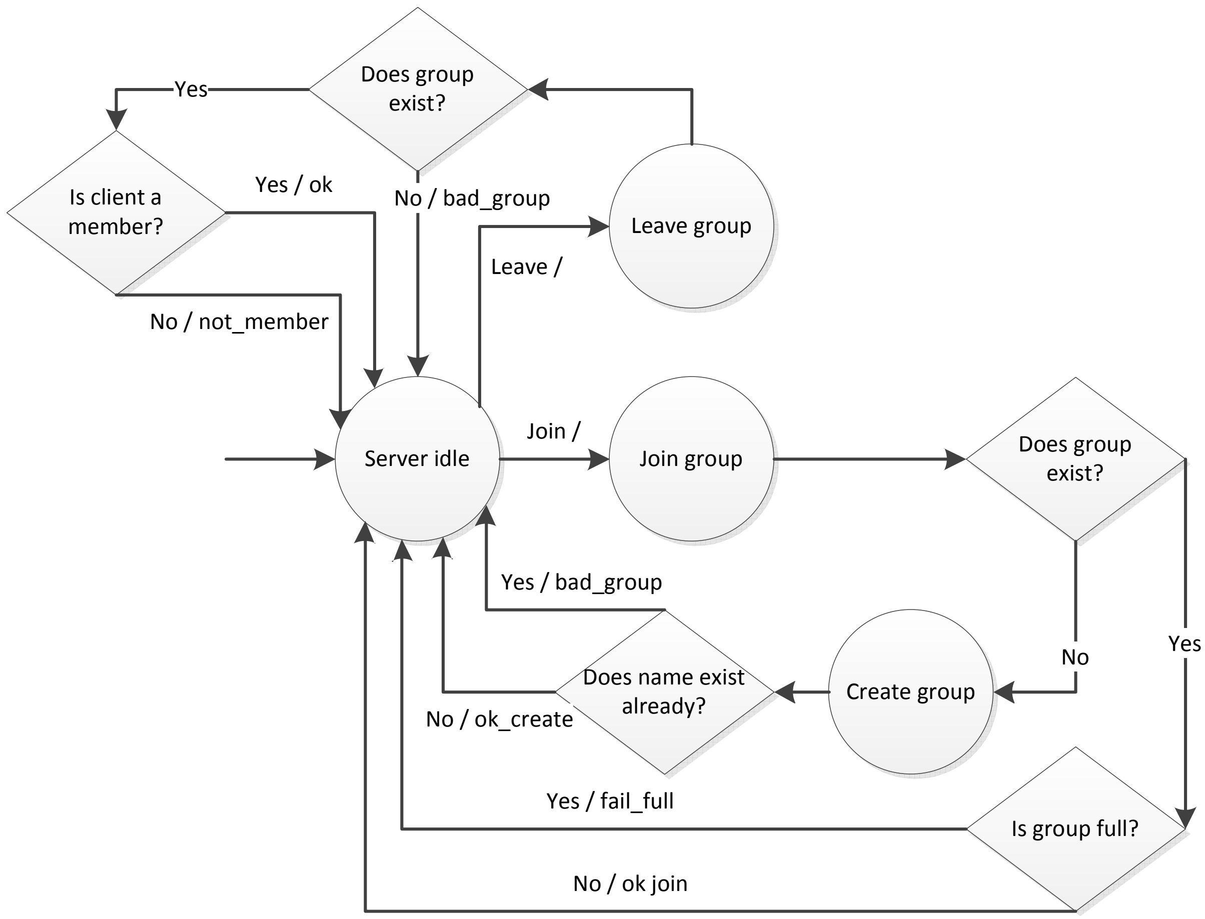

Figure 2: Server state diagram for joining/leaving groups

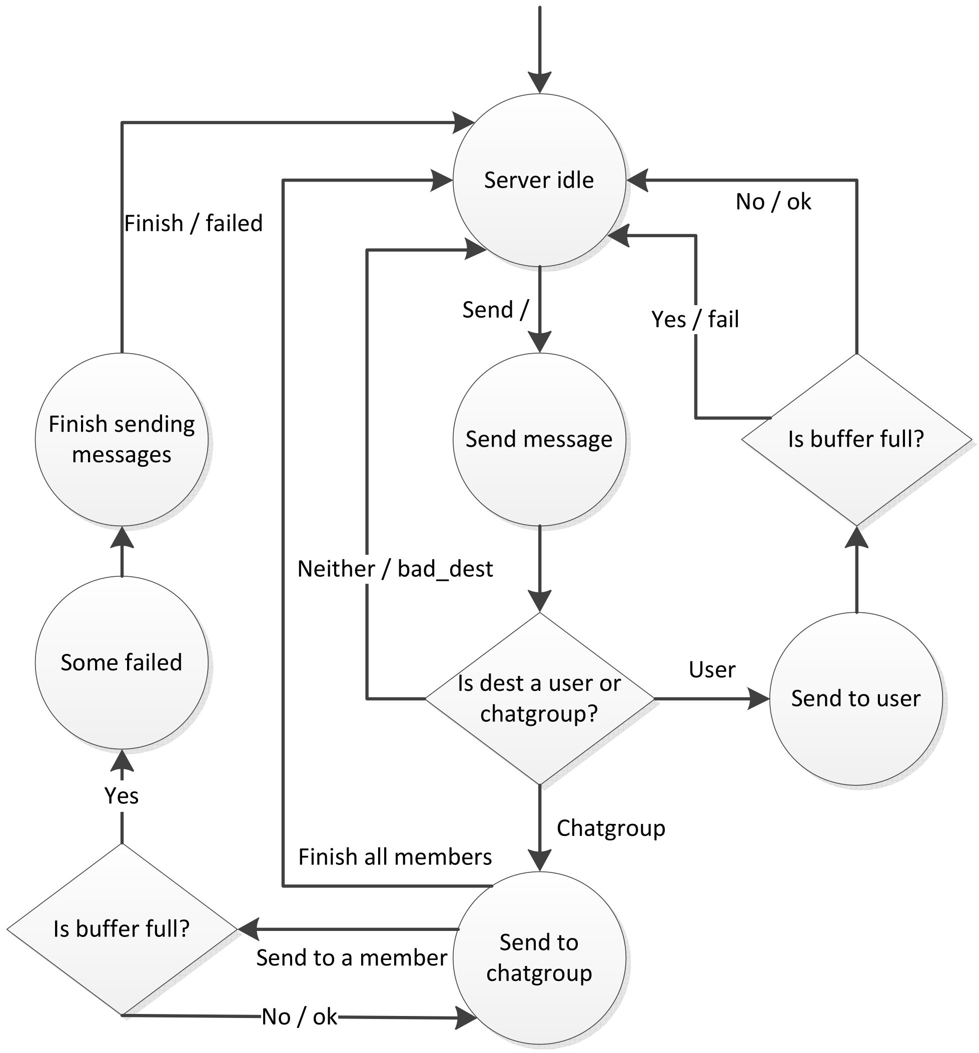

Figure 3: Server state diagram for processing messages

## 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.
```

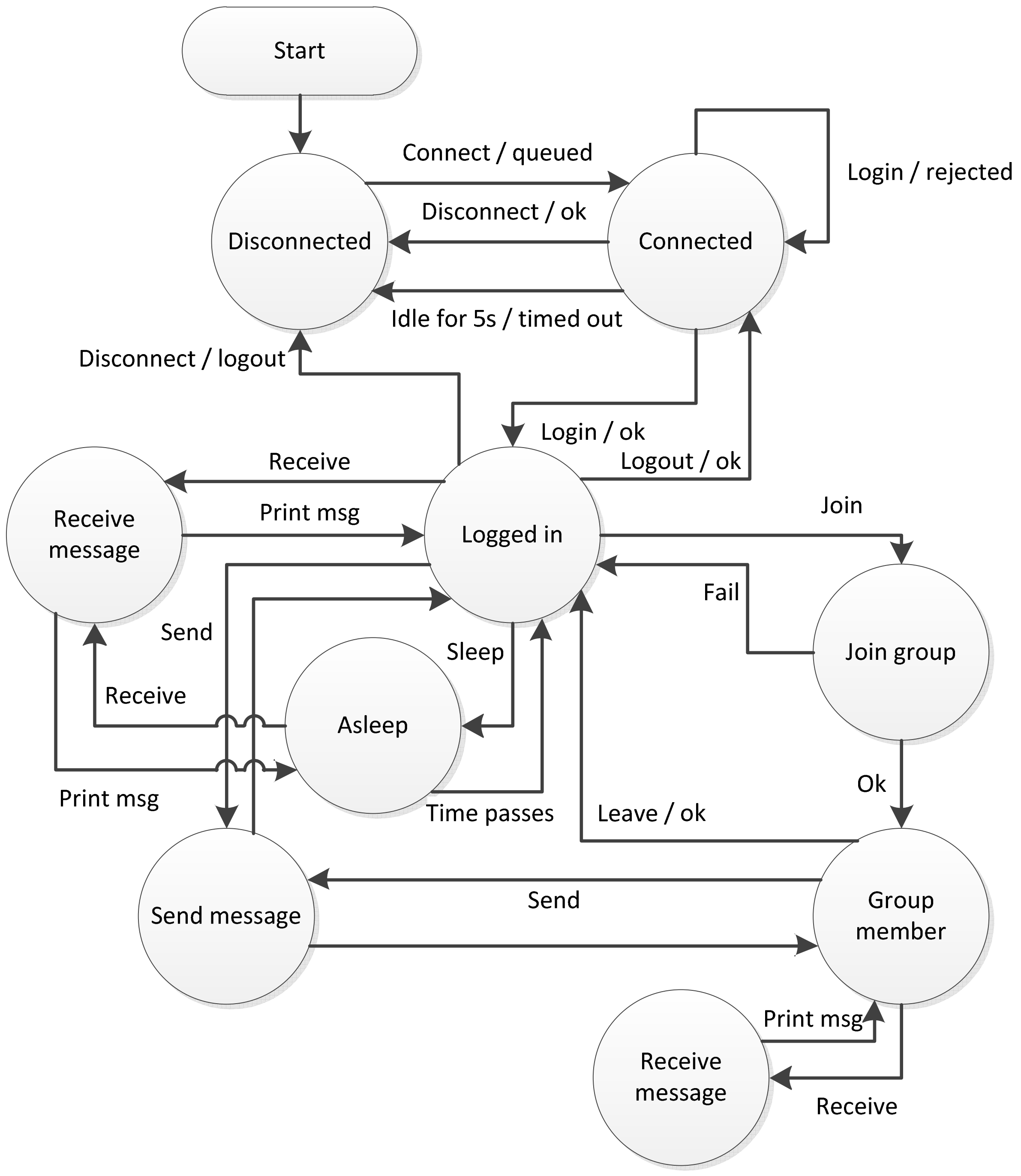

Figure 4: Client state diagram

## 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.