Each Client communicates with the Server using a unique Socket, ClientConnection and ServerConnection. Clients logging into the Server provide a username and are assigned a userID; a User object is then created and stored onto the Server to represent each logged-in Client. A channel has a unique title and is started by a User; any other User can join the channel by adding themselves to the list of its members (issuing a *join* command). A channel is destroyed when all of its members have left, either by manually choosing to leave the channel or exiting the chat client. The client sends commands to the server through an instance of ClientConnection and an instance of ServerConnection. They handle the Sockets at either end of the connection between client and server. Each constantly reads in input to process, and has a queue for writing.

public class Client

The client runs on the client's computer.

```
	private ClientConnection conn;


	public Client();


	public void createConnection(String user);
```

public class Connection

A superclass of ClientConnection and ServerConnection; handles a Socket and all Client-Server communication across it. It has two Runnables, ConnectionReader and ConnectionWriter to read in and write out Messages, and a BlockingQueue of Messages to send across the socket.

```
	protected BlockingQueue<Message> messageQueue;
	protected Socket socket;
	protected User user;
	protected Thread readerThread;
	protected Thread writerThread;


	//Creates a Connection instance to handle communication across this
socket.
	public Connection(Socket socket)
```

```
//Offers a message to the Queue
public void sendMessage(Message output)

//Interrupts the current writer thread
public void closeConnection();

// Use the type of the message to determine the relevant function to
// call on the server
public void processMessage(Message message);

//Interrupts the current thread
public User getUser();

//set the current user
public void setUser(User user);

// Handles client-server requests represented by Message message.
public void processMessage(Message message);
```

public class ClientConnection extends Connection

```
private Client client;
```

A subclass of Connection; handles Client-Server communications at the Socket on the Client's end. Overrides Connection methods such as processMessage() to call Client's methods.

public class ServerConnection extends Connection

```
private Server server;
private int userId;
```

Handles incoming communication (commands) from the client by delegating any actions to be performed to the running Server instance. Each user has a single ServerConnection that handles server-side socket input/output relating to that particular user.

public class Channel

This class represents a channel in guichat, and stores a list of users and a list of messages.

The channel has functions to process client commands such as "join #channel" and "leave #channel". An instance of Channel is created when at least one person joins the Channel, and then removed when it has no more members.

```
private String name;
private User owner;
private List<User> users;
private List<Message> messages;

//Construct a Channel with name channelName, started by User owner
public Channel(String channelName, User owner);

//Returns true if this Channel has User u as one of its current members;
//false otherwise
public boolean hasUser(User u);

//Add User u to this Channel's list of members
public void addUser(User u);

//Remove User u from this Channel's list of members
public void removeUser(User u);

//Send a message to the group by
//adding Message m to this Channel's List of Messages
public void sendMessage(Message m);

//Returns the users in this Channel
public List<User> getUserCount();

//Returns the number of users in this Channel.
public int getUsers();
```

public class Server

The chat server. Stores a HashMap linking Channel names to Channels, a HashMap of nicknames and their respective User classes, and a HashMap of Users to the Thread that processes them. Also has functions to carry out User commands.

```
private ServerSocket server;
private HashMap<int, User> userMap;
private HashMap<String, Channel> ChannelMap;

// Instantiate a server on the specified port.  Create Server fields of
// a ServerSocket, a HashMap<String, User> and a HashMap<String,
Channel>
public Server(int port);

// Listen for connections on the specified port; upon accepting create a
ServerConnection to handle the Socket and a User to represent the
logged-in Client.
public void listen();

// Process a user “joining” a Channel, true if join successful
public boolean addUserToChannel(String nickname, String channelName);

// Process a user “leaving” a Channel, true if leave successful
public boolean removeUserFromChannel(int userID, String channelName);

// Process a user creating a Channel, true if creation successful
public boolean createChannel(int userID, String channelName);

// Return a Channel on the Server.
```

```
public Channel getChannel(String channelName);

// Return a User on the Server.
public User getUser(int userID);

//Remove a User that has disconnected from the Server
public void processUserDisconnect(int userID);
```

public class User

A class representing a user

```
public int id;
public String nickname;
public Connection connection;

// Create a new User given a server, socket, and nickname
public User(String nickname, int id, Connection connection);
```

public class Message

A message is a discrete unit of communication between the client and the server. Messages are sent through a socket.

```
private Command command;
private String channelName;
private Calendar calendar;
private String messageText;
private String author;

// Create a new message given a command, Channelname, date, author and
message content
```

```
 public Message(Command command, String channelName, Calendar calendar,
String messageText, String author);

 // Returns the contents of this Message
 public String getContent();

 // Returns the date that this message was sent
 public Calendar getCalendar();

 // Returns the sender of this message
 public User getSender();

 // Returns the Channel of this message
 public Channel getChannel();
```

Client-Server Protocol

Shared grammar:
WORD ::= [A-Za-z0-9_]+
SPACE := " "
NEWLINE := "\n" //Use as end of message character?
COMMA ::= ","
PERIOD := "."

NAME := [WORD]+
nickname = CHANNELNAME = AUTHOR = NAME
ID := [0-9]+
LINE := [WORD SPACE COMMA PERIOD]*
TEXTLINE := [LINE]* NEWLINE

Client --> Server

protocol ::= command*
command ::= (login | logout | list-channels | list-users | user-channels | join | leave | post) NEWLINE
login := "login" nickname //logs in a user with that nickname
logout := "logout" //logs the current user out
list-channels := "list-channels" //list all open channels (channels with >0 users in them)
list-users := "list-users" //list all users currently online
user-channels := "user-channels" //list the channels the user is in

join := "join" CHANNELNAME
//have user join the channel with name CHANNELNAME

leave := "leave" CHANNELNAME?
//have user with name nickname leave the channel with name CHANNELNAME
//If no channel argument supplied, user leaves current channel

message := "message" CHANNELNAME TEXTLINE
// have user with name nickname post a message comprising of TEXTLINE in the channel with name CHANNELNAME

Server --> Client

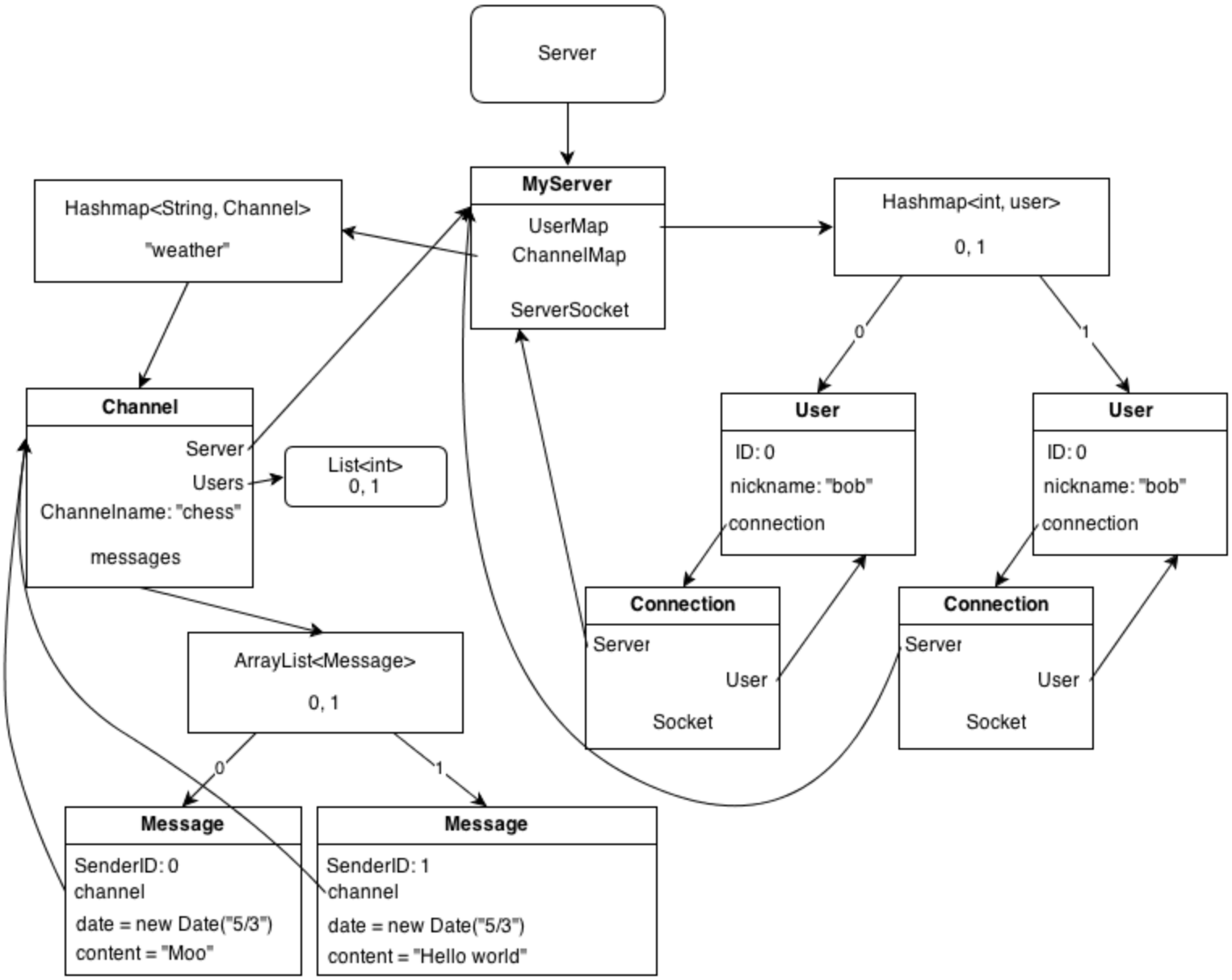
protocol ::= command*
command := success | failure | error | reply-list-channels | reply-list-users
success := "reply-success"
failure := "reply-failure" TEXTLINE* // TODO: have TEXTLINE contain helpful info about failure
error := "error" TEXTLINE // When server throws Exceptions
reply-list-channels := "reply-list-channels" CHANNELNAME*
//info message sent in response to a list-channels or user-channels request
reply-list-users := "reply-list-users" nickname*
//info message sent in response to a list-users request

message := "message" CHANNELNAME AUTHOR TEXTLINE

Server --> Client

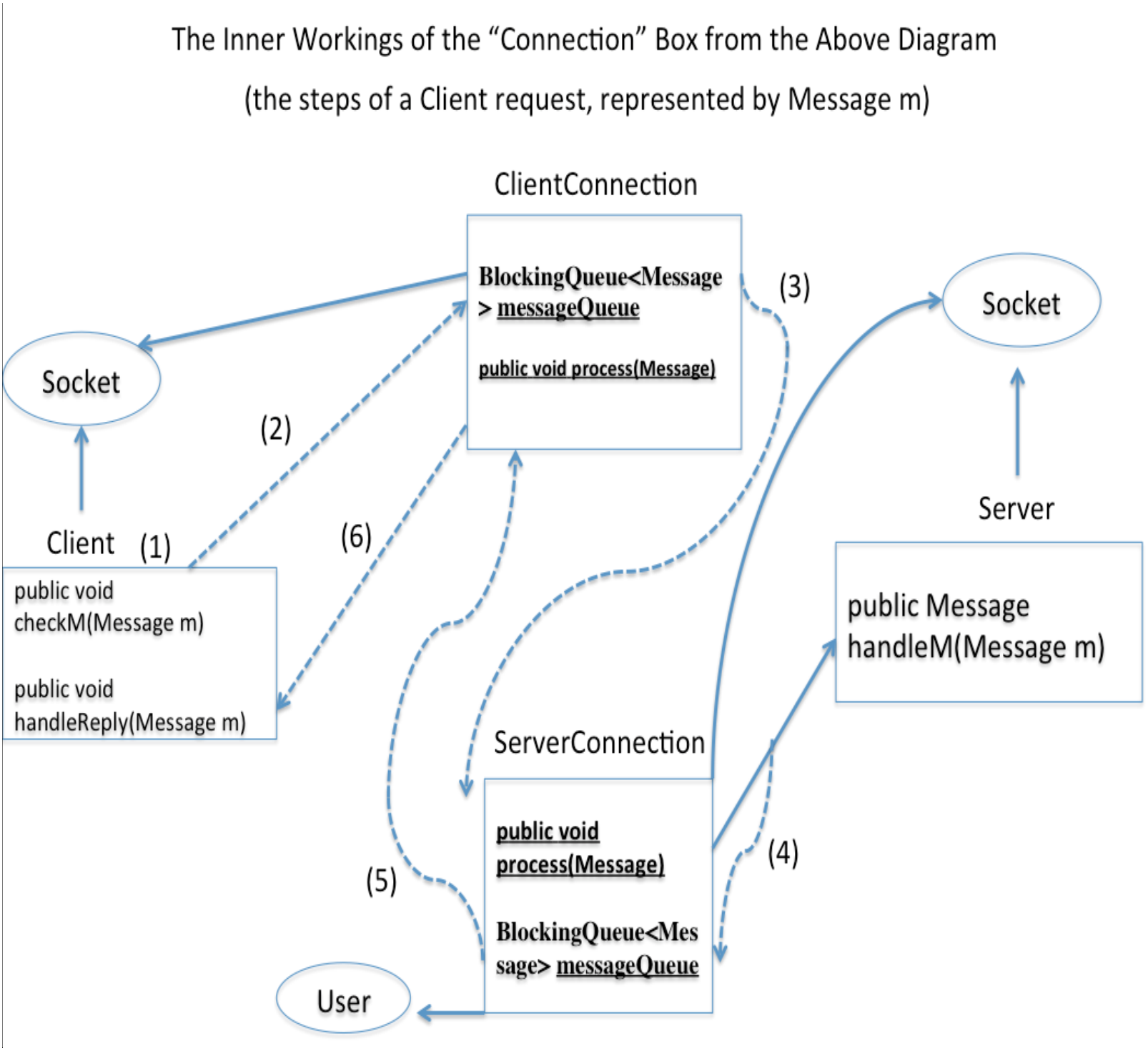
protocol ::= command*
command := success | failure | error | reply-list-channels | reply-list-users
success := "reply-success"
failure := "reply-failure" TEXTLINE* // TODO: have TEXTLINE contain helpful info about failure
error := "error" TEXTLINE // When server throws Exceptions
reply-list-channels := "reply-list-channels" CHANNELNAME*

Snapshot Diagram



** in this diagram, we treat the Connection as a 'black box'. See below in-depth diagram for details on what the Connection box is actually representing.

In-depth Snapshot Diagram



(1) The request m is checked for proper protocol grammar using Client's checkM() method.
(2) If properly formatted, m is sent to the Client's ClientConnection's queue of Messages waiting to be sent across the socket connection.

(3) m is taken from the queue in ClientConnection and sent to the ServerConnection at the 'other end' of the socket connection.
(4) Immediately after arriving at the ServerConnection, m is processed using a method in ServerConnection which calls on a handleM() method in the Server. handleM() returns a Message r, representing the Server's response to m. r is now put into the ServerConnection's queue.
(5) r is taken from the ServerConnection's queue and sent back to ClientConnection for processing.
(6) Immediately after arriving at the ClientConnection, r is processed using a method in ClientConnection which calls on a handleReply() method in the Client.

Testing Strategy

Automated Testing

Our automated testing will focus on ensuring our back-end modules, including the Server, the Client, and the Connection in addition to our Model classes (such as Message, Channel) function as expected. We will usually do so by using a dummy server, then simulating a client connecting to this dummy server and processing different actions.

We will first test the methods that the Server and Client use to respond to messages. For example, in the Server we will use JUnit tests to test logging in/out clients; creating/deleting channels; adding a client to/deleting a client from a channel; posting messages; getting a channel's history; seeing other clients and channels, all of which may or may not exist when the method is called.

Then, we test that upon a single Client logging in, followed by one or more Clients logging in, the created Client Connection and ServerConnection instances successfully pass Messages in-between them. For this step we may simply have the Client send along dummy messages, which the Server will respond to with another dummy message, which the Client will finally print out. We will test for correct Messages and behavior given interaction between users, such as having one user trying to respond to a conversation that another starts. We may also check the order in which Messages are processed.

Next we test to ensure that ClientConnection and ServerConnection will always pass properly formatted Message instances to each other; therefore, there will be no problems when it comes time to process the messages. This checking will occur in our Client, Server, ClientConnection or ServerConnection classes. In general, if a client inputs a request not adhering to the grammar, we will treat it as an undocumented case and unexpected behavior might occur.

Finally, as an integrated test we will simulate a lengthy chat session (by manually forcing certain messages into the Connection), involving at least two Clients who log in at different times, but always making protocol grammar-adherent requests. At specified points in this session we check the fields of the Server and the Client to ensure everything is being processed correctly. Eventually, we want to test all commands in our Client-Server protocol, and ensure that the messages sent back to the client from the server are correct.

Manual Testing

We will manually test our GUI, using System.out.println() statements to double check the states and values of back-end objects. All aspects and displays in our GUI should appear as expected. We will also check the functionality of all our Listeners; that is, inputted action commands should produce the appropriate response. We can check by either further interacting with the GUI, and may also add some debugging lines to our code (such as printing out the Message sent from a Connection's queue). In general, our testing GUI testing strategy will consist of testing some boundary cases and certain paths in our back-end code (glass box testing) as well as some random conversations between multiple Clients, ensuring that in general the GUI will work (black box testing). Finally, the GUI should not 'hang' when we do not want it to.

Particular GUI tests include joining rooms, switching to a room and seeing the messages received therein (if the user had been active in that room but minimized it), typing messages, and seeing all the users in a room, and seeing all the users.

In addition, we will to the best of our abilities manually test our program's concurrency. Two or more Clients will attempt to send commands at the same time which require proper concurrency to execute correctly (for example, one leaves a Channel in which it was the only member thus deleting the Channel from the server, while the other attempts to join that Channel). We also want to check the multithreaded status of our program - that is, with multiple Clients accessing a single Server, as long as their requests don't both call synchronized methods, then all Clients should be able to receive responses simultaneously. However, it is reasonable to have a slower speed due to a possible bottleneck if the requests do both call upon synchronized methods.

Concurrency Strategy

Each connection (Socket) made between a Client and a Server is managed by a ClientConnection and a ServerConnection; both are subclasses of the Connection class. Both

ClientConnection and ServerConnection are constantly waiting for incoming Messages, which they then immediately process. Thus, each incoming Message is processed one at a time (in the same Thread), so we do not have to worry about concurrency issues there. Each also has a blocking queue (the “writing” queue, consisting of Messages to be written to the Socket output stream); the elements of the queue are popped off and written using a separate Thread.

We run into problems with concurrency when two requests from different Clients (through separate Connections) both attempt to mutate the same object on the Server. Since we only mutate objects on the Server by calling Server’s methods, we will make our Server threadsafe by synchronizing methods which mutate Server objects (or alternatively, using a BlockingQueue to store commands to be executed by the Server). Then only one request may mutate the Server’s objects at a time. A bottleneck may then be formed but this is acceptable.