Project Report

CS 3516: Computer Networks

Class Project: Client/Server Chat Program

Joshua McKeen

July 9, 2020

All work within this project and report is my own unless marked to the contrary.

Signed: 

Abstract

This report outlines the development of a client/server chat program written in Java. This program has been fully tested to work using Java on Windows 10. The application has a server component, which runs purely in a terminal/command line environment and allows the server operator to control the server through the use of commands. The server operator may obtain a list of connected users and stop the server (disconnecting any connected clients in the process). The server window also displays client connections/disconnections and chat messages. Additionally, there is a client component which operates in a Graphical User Interface. The GUI allows the user to connect to the server, set up their username, send general chat messages, whisper chat messages to one other specific user, and view a list of all the users connected to the chat. The server has no limit on connected clients, and all clients are required to have distinct usernames (which is enforced upon client connection to the server).

Table of Contents

Project Description

The purpose of this project was to learn how to work with network socket commands in a single-server/multiple-client model. Specific focus had to be placed on handling network socket connections and disconnections while maintaining the stability of the program. This project demonstrates how multiple clients can each connect with a client-side socket to a server-side socket and how the server can continually maintain multiple connections. It also demonstrates how data may be sent efficiently and usefully through those sockets and their corresponding input/output streams in order to facilitate an application such as a chat program, where more complex data than just "chat text" as a string needs to be communicated. Finally, the project demonstrates how multithreading can be used to facilitate simultaneous input and output on both the server and the client.

Detailed Design

Messages

The foundation of the communication between the server and the client in this project is the custom-created Message object. The use of an object to communicate between the two components instead of just a simple string is necessary to convey important details about the message. For example, if the message is intended as a private message (whisper) from one client to another, the message object will contain the details about who the sender and who the intended recipient of the message is. The message object also has flags (Boolean true/false) fields for conveying information about client username setup, retrieving user list, whispering to a nonexistent/disconnected user, and facilitating a clean disconnect/socket close between the server and the client. These message objects are sent and received through network sockets using Java's ObjectOutputString and ObjectInputStream classes.

Server

Because the server needs to be able to simultaneously handle both input and output to multiple clients, it uses a multithreaded program model. The main method starts of by establishing the server network socket to be available for clients, and then loops through a continuous while loop waiting for potential clients to connect. Once a client connects, a new ClientHandler is created and placed in its own thread. The ClientHandler is responsible for maintaining the connection with its client and responding to all incoming messages from that client. This is accomplished through a continuous while loop checking the socket input stream, similar to the loop that runs in the main method. When a user disconnects from the server, its corresponding ClientHandler thread is deleted.

The use of different threads for every client is necessary because each thread "locks up" while waiting for an incoming message. If all the client connections were handled in one thread, the server would constantly be freezing waiting for one of the clients to send a message, preventing other clients from sending messages. Because the server simply needs to respond once to each client's request, the while loop for each thread facilitates waiting for a message, properly responding to the message, then pausing and waiting for the next message.

The server maintains a list of all current ClientHandlers in its main method to facilitate communication between different ClientHandlers, which is necessary every time a user wants to send a message to a different user. The same ClientHandler list is also used to generate a user list whenever the server operator or a client requests it. The list is also consulted when a new user connects to determine if the requested username is already in use by another client.

Client

The client utilizes a Graphical User Interface (GUI) developed in Java's Swing GUI toolkit. While a little out-of-date and simplistic, the feature set in Swing is perfectly adequate to support a simple chat interface. The primary challenge of developing the Client was the integration of a real-time loop waiting for incoming messages from the server into a program that primarily deals with event-based responses, which is what GUIs utilize. If caution is not used, the GUI could potentially lock up for the user until the client receives a message from the server, since the checker for the input stream runs a continuous while loop. The client gets around this by creating a separate InputStreamReader thread. Having this process in its own thread ensures that the GUI will continue to function and be able to send messages through its event-based Action Listeners. When a message is received, the message is passed from the thread to the client GUI through a Change Listener, which triggers an action whenever the Input Stream has a "change" or incoming message.

Disconnections

Properly handling client and server disconnections was a major component of this project. Ideally, whenever a client disconnects from the server both sides have the opportunity to properly close their input and output streams and their sockets. To accomplish this, communication and coordination between the client and server is necessary. All disconnection messages are routed through the server, whether or not they originate from the server or client.

If the server requests a client disconnection (in the event the server is shutting down), the server sends a specially flagged message to the client, asking the client to close its streams and sockets and then terminate. The server, after sending this message, will also close its streams and sockets, followed by terminating the ClientHandler for that client.

If the client requests a disconnection, two disconnect messages are sent. The first is from the client to the server, asking if the client is ok to disconnect. The second is the disconnect message from the server to the client, which is the same message that is sent when the server initiates the disconnect. This path of client -> server -> client ensures all parties are informed so they can properly close their sockets. It also gives the server the opportunity to remove that client's ClientHandler from the client list, which ensures the client list is always accurate.

In the event the server or client terminates unexpectedly, the caught exceptions will ensure that the other component remains stable. If the server terminates unexpectedly, the client will automatically close its streams and sockets and terminate. If the client terminates unexpectedly, the server will automatically catch the exception and close the streams and sockets, terminate the ClientHandler

thread, and remove the client from its client list. In both cases, specially written try/catch statements which catch Java I/O exceptions ensure that the user isn't exposed to any confusing stack traces or error messages.

Feature Testing/Evaluation

Contained within this section are screenshots of the client and server in operation demonstrating the required specifications as well as the program's behavior for unusual/unexpected circumstances.

Client

Upon connection, the user is prompted by a dialog to choose a username.

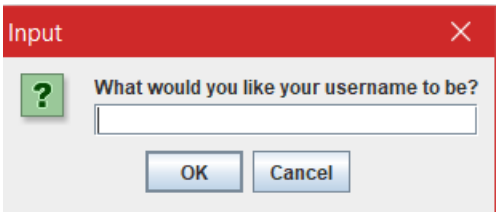

Figure 1: Username prompt

If the user presses cancel, the application automatically disconnects and closes. If the user leaves the box blank, the prompt appears again.

If the user requests a username that is already taken, they are informed and given the opportunity to choose another name.

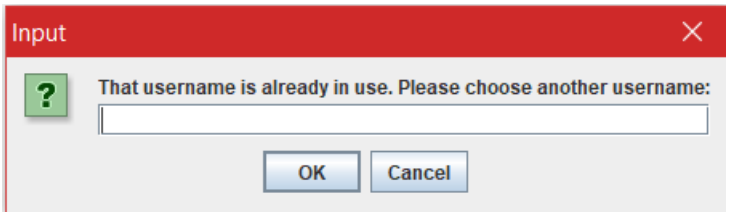

Figure 2: Username in use, prompting the user to chose another name

Again, if the user presses cancel or close, the application automatically disconnects and terminates.

The client has a simple GUI that presents the chat window, along with an input box at the bottom, and user options on the right side.

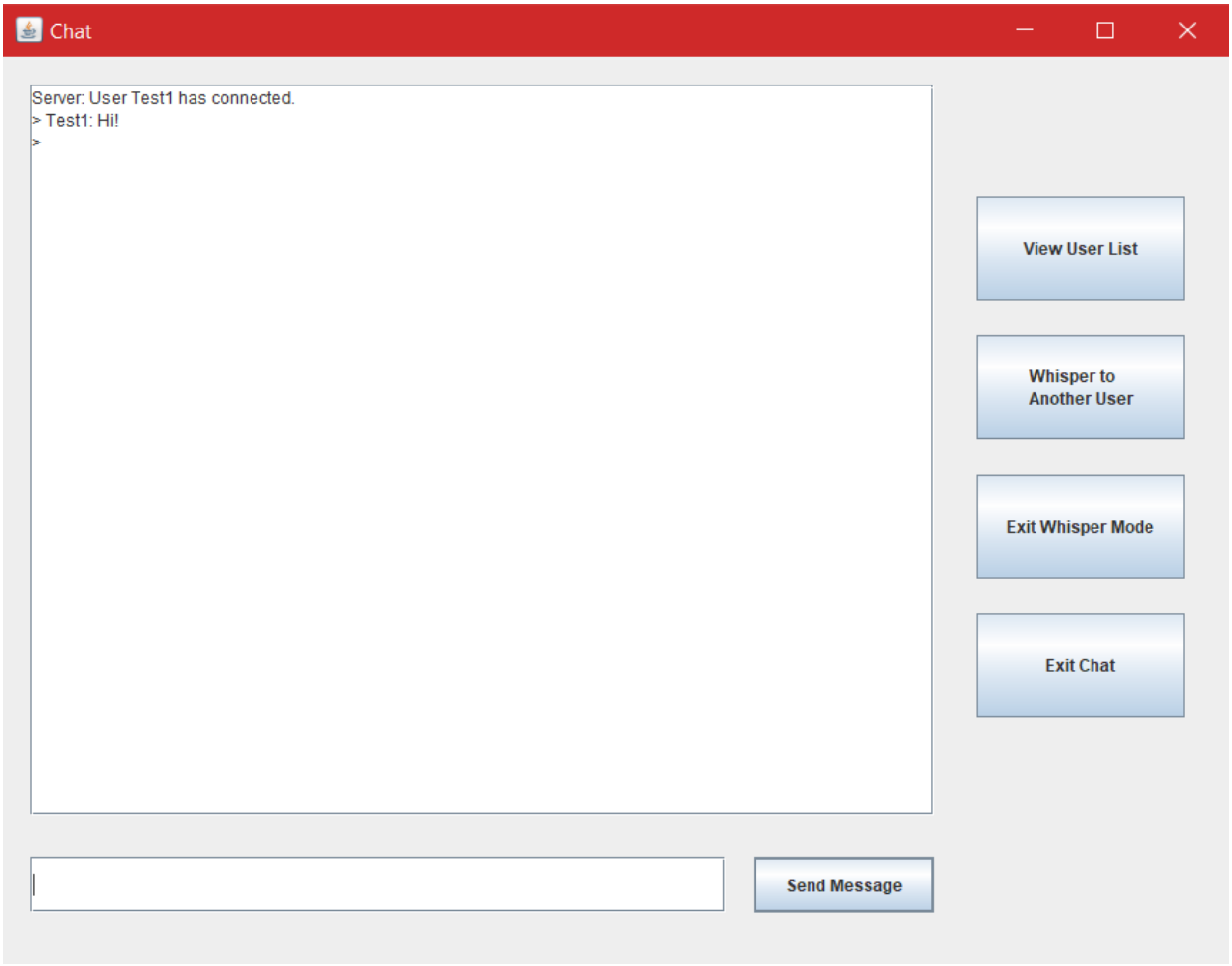

Figure 3: Client GUI

The user may begin typing a message, then press the enter key or the “send message” button to send their message.

If the user wishes to “whisper” to another user, they may press the “Whisper to another user” button

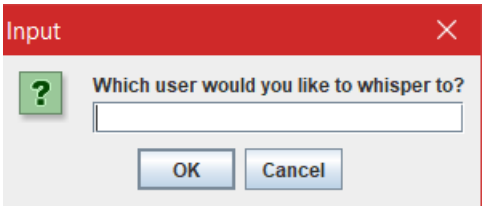

Figure 4: Whisper dialogue

At this point, the user may send a message to that particular user only, and not have it be visible to other users

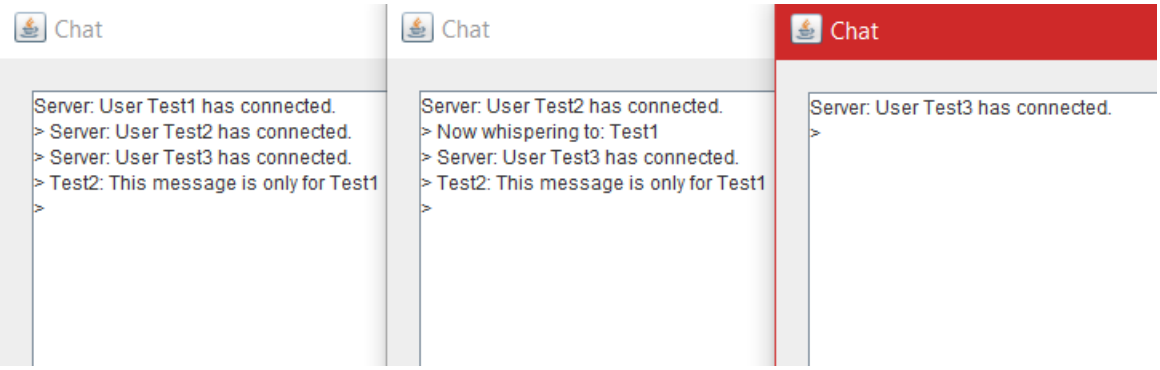

Figure 5: Whispering in action

Here, Test2 has whispered to Test1, while the message was not transmitted to Test3.

If the user attempts to whisper to a client that does not exist, they will be informed and whisper mode will automatically stop

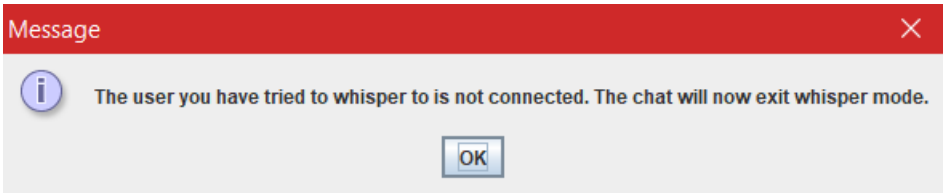

Figure 6: Whisper error, user not connected

This error was generated after Test3 tried to whisper to Test4 (who doesn't exist).

The user may stop whispering at any time by clicking "Exit whisper mode" and may then send a message to everyone

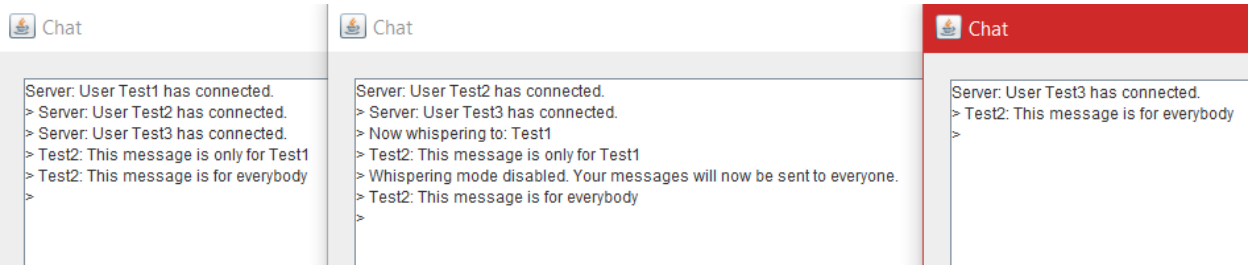

Figure 7: Whisper mode enabled then disabled

The user may also change which user they are whispering to without exiting whisper mode

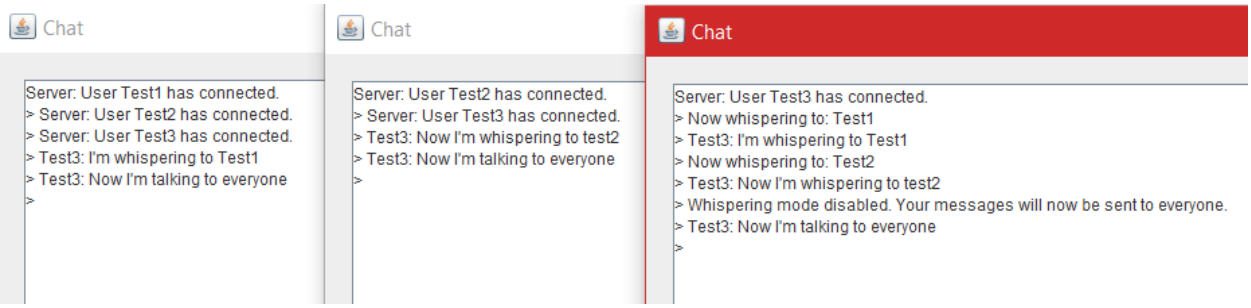

Figure 8: Test3 whispers to Test1, then switches directly to whispering to Test2, then sends a message to everyone

The user may request a user list at any time by pressing the “view user list” button

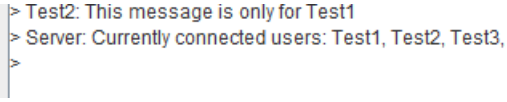

Figure 9: User list

If the client attempts to connect and the server is not available or running, the message will be shown in the command line window from which the application was executed. The GUI will not start and the program terminates

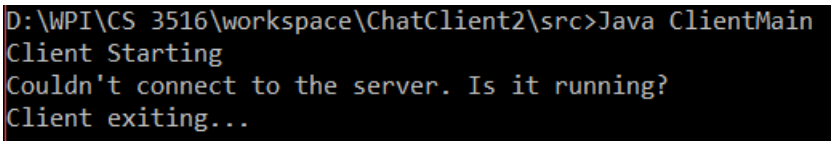

Figure 10: Server is not available or not running

The user is alerted to all other client connections and disconnections for the duration of time they are connected. After a connection/disconnection they may request an updated user list if they desire.

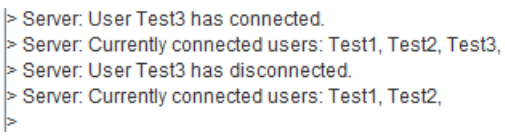

Figure 11: Connection and disconnection notifications and updated user lists

The user may disconnect from the server by clicking the “exit chat” button. This automatically disconnects the streams and socket and exits the program.

If the server unexpectedly shuts down or crashes, the client will automatically terminate. There is unfortunately no adequate way of showing this functionality via screenshot.

The client supports receiving messages while the user is typing a message.

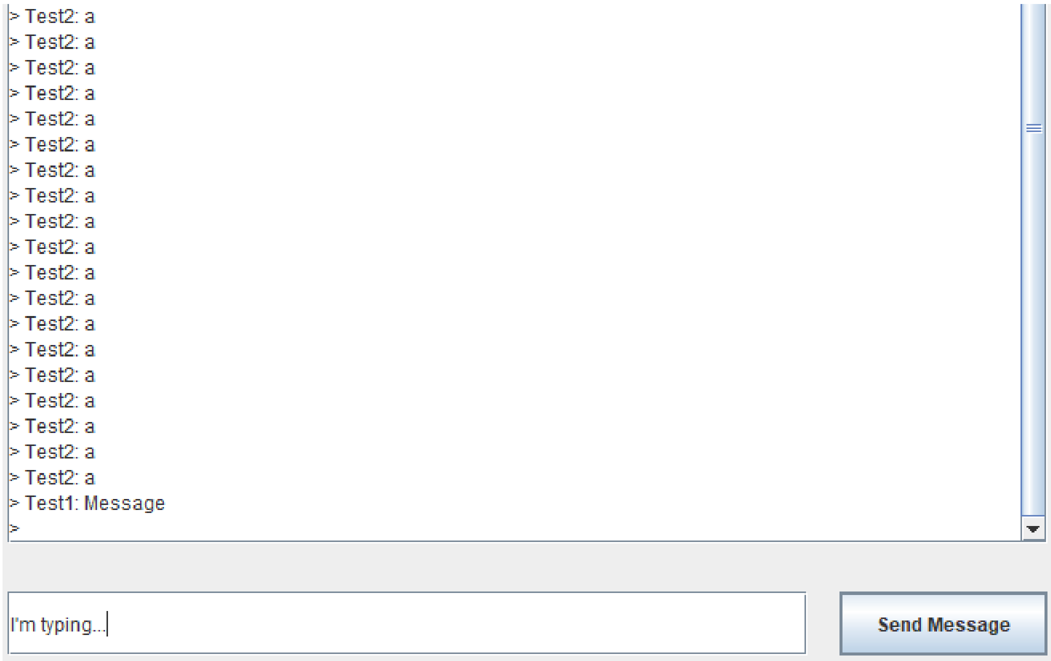

Figure 12: Showing an incoming message from client Test1 while Test2 is typing a message. The message from Test1 arrived while Test2 was typing.

Also shown in Figure 12: If the chat window becomes too long, a vertical scroll bar will automatically appear, as shown on the right side.

A horizontal scroll bar will appear if a message is very long

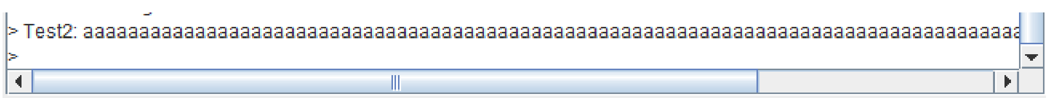

Figure 13: Horizontal scroll bar for long messages

Server

The interface with the server is completely command-line based. All server activity is logged in the console window for the server operator to see.

Upon server start, instructions are given to the console operator, as well as notification that the server is listening for new connections.

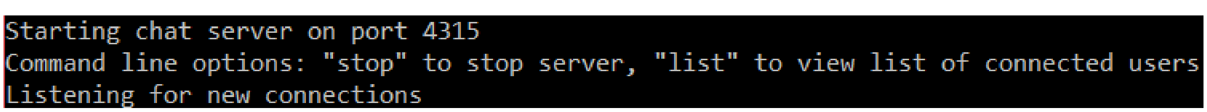

Figure 14: Server start display and instructions to operator

The console will display messages when users connect and disconnect.

```
Establishing new ClientHandler thread
Connection with username Test1 estabished.
Listening for new connections
Establishing new ClientHandler thread
Connection with username Test2 estabished.
User Test1 is requesting to disconnect
Connection with user Test1 has been terminated. Thread for this client will stop now
User Test2 is requesting to disconnect
Connection with user Test2 has been terminated. Thread for this client will stop now
```

Figure 15: Notifications for client connections and disconnections from the server.

Additionally, if a client unexpectedly terminates, a disconnection message will also be displayed and the client socket will be removed of properly.

```
Socket for user Test1 closed.
Connection with user Test1 has been terminated. Thread for this client will stop now
```

Figure 16: Notification of socket for Test 1 unexpectedly closing

Note that there is no impact on other server operations when this happens.

The server display a notification when a user has chosen an invalid username.

```
Establishing new ClientHandler thread
User needs to choose another name
Connection with username Test2 estabished.
```

Figure 17: User tries to connect with a username that's already taken.

In this case, Test1 was already connected, and the Test2 client tried to connect with the name Test1. After it changed its username to Test2, connection was established with the server.

If the user unexpectedly terminates their client before they've chosen a valid username, the messages will still be displayed, but the username field will be blank.

```
Establishing new ClientHandler thread
User  is requesting to disconnect
Connection with user  has been terminated. Thread for this client will stop now
```

Figure 18: Client disconnecting before username has been established.

The console “stop” command automatically and safely disconnects all the clients and then terminates the program.

```
stop
Disconnecting all currently connected clients
Connection with user Test1 has been terminated. Thread for this client will stop now
Socket for user Test1 closed.
Socket for user Test2 closed.
Connection with user Test2 has been terminated. Thread for this client will stop now
Connection with user Test2 has been terminated. Thread for this client will stop now
Server shutting down
Socket Closed
```

Figure 19: Console stop command disconnecting all clients

This was tested both for established users and for users with no username yet. If the user does not have a username, the name field is blank in the console.

The console operator can use the “list” command to print a list of all connected clients.

```
list
Currently connected users: Test1, Test2,
```

Figure 20: Console list command

If any user has not established their username yet, the user will show as blank in the user list.

```
list
Currently connected users: Test1, Test2, ,
```

Figure 21: Console list with client that does not have an established username.

All broadcast chat messages are placed in the server console for the operator to view.

```
Test1: Hey, how's it going?
Test2: Pretty good, how about you?
```

Figure 22: Console with chat messages

All whispered messages are also displayed.

```
User Test3 whispered to Test2 message: Pssst, hey!
```

Figure 23: Console with whispered message

If a user tries to whisper to a nonexistent user, a message is displayed.

```
Test2 tried to whisper to nonexistent user Test4
```

Figure 24: Whispering to a nonexistent user

Finally, pressing the Windows title bar [X] button for either the server or the client will result in an "unexpected" shutdown, and will result in the corresponding server reacting in the manner described in the previous figures for "unexpected shutdown." As demonstrated above, unexpected shutdowns of either the client or the server have no ill effects on the operation of the other component.

I have tried to be as thorough as possible in regards to testing the above situations, and I believe I have covered every possible action that can be accomplished with both the client and the server, and that just about every line of code has been executed.

Future Development

While this client/server chat implementation is quite stable and works well for sending basic text, there are still more features that could be added to make the chat more feature-rich and easier to use. These are just a few of the features I think the program would benefit from:

- Support for attaching pictures
- Support for emoji characters
- Ability to see when someone else is typing
- Streamlined interface for whispering (perhaps some sort of drop down instead of typing the username)
- GUI with more modern, fluid design
    - More modern look and feel
    - Resizable window
- Support for colored text
- "Chat bubble" like user interface, instead of just lines of text
- Ability for server console operator to "moderate" chat: send messages, and remove specific users from the server

All of these features are certainly achievable using Java and socket commands, but some of them would require a substantial rewrite of certain parts of this program in order to be able to achieve them.

Conclusion

This report has described the successful implementation of a multiple-client/single-server chat program using Java's network sockets and I/O streams. The project was a great experience to learn about how network sockets work, how to design a program to use them, and how to handle input/output streams. In addition, it was also a good lesson for me in Java GUI development, since I had never built a GUI before. Other important lessons learned were the difference between linear-based console programming and event-based GUI programming and how to utilize multithreading in Java to separate key tasks and prevent them from interfering with each other.

Appendix – program code

I have also attached the .java files and class files in a .zip archive in the Canvas assignment for the project, which may be a better code viewing experience.

Client Application:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.io.Serializable;
import java.net.Socket;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

//Chat application client program - by Joshua McKeen - jdmckeen@wpi.edu

public class ClientMain {

	public static void main(String[] args) {

		System.out.println("Client Starting");

		//ESTABLISH CONNECTION WITH SERVER
		Socket clientSocket = null;
		try {
			clientSocket = new Socket("localhost", 4315);  //try to establish connection on port 4315
		}
		catch(Exception e) {  //if it fails, is server running?
			System.out.println("Couldn't connect to the server. Is it running?");
			System.out.println("Client exiting...");
			System.exit(0);  //exit client
		}
```

```
		ObjectOutputStream out = null;
		try {
			out = new ObjectOutputStream(clientSocket.getOutputStream());  //establish output stream
		} catch (IOException e1) {
			e1.printStackTrace();
		}
		ObjectInputStream in = null;
		try {
			in = new ObjectInputStream(clientSocket.getInputStream());  //establish input stream
		} catch (IOException e1) {
			e1.printStackTrace();
		}
		System.out.println("Established input and output streams");

		InputStreamReader isr = new InputStreamReader(in);  //establish new inputstreamreader
		isr.start();  //start the input stream reader

		//create a new GUI object
		new GUI(out, isr);  //the GUI has access to the out and in streams

		try {
			isr.join();  //wait for receiver thread to terminate before continuing with disconnect/shutdown
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		System.out.println("Disconnecting streams and socket");
		try {
			in.close();     //disconnect all streams and close socket
			out.close();
			clientSocket.close();
		} catch (IOException e) {

			e.printStackTrace();
		}
		System.out.println("Connection closed. Client terminating");
		System.exit(0);

	}

}
```

```
class GUI {
	//GUI Elements
	private JFrame frame;  //frame
	private JButton list;  //list button
	private JButton whisper;  //whisper button
	private JButton eWhisper;  //exit whisper button
	private JButton quit;   //quit button
	private JTextField chatIn;  //chat box input
	private JButton send;    //chat send button
	private JTextArea chat;   //chat box (incoming messages log)
	private JScrollPane chatScroll;  //scroll frame for chat

	//Event Listeners for GUI elements
	private sendListener sL;
	private listListener lL;
	private whisperListener wL;
	private eWhisperListener eWL;
	private quitListener qL;
	private messageListener mL;

	//input and output streams from main method
	private ObjectOutputStream out;

	private InputStreamReader isr; //input stream reader from main method

	private String whisperingTo = "";  //keeping track of who the client is whispering to
	private String username = "";  //keeping track of this client's username

	GUI(ObjectOutputStream o, InputStreamReader r){  //THIS IS THE GUI CONSTRUCTOR

		out = o;
		isr = r;

		mL = new messageListener();  //create a new message listener
		isr.setListener(mL);    //set the input stream reader to feed to the message listener

		//Establishing the four control buttons, list, whisper, exit whisper, and quit. Establishing sizes and locations
		//Buttons start disabled so the user doesn't try to click them while they're still establishing their username with
the server
		list = new JButton("View User List");
		list.setSize(150, 75);
		list.setLocation(700, 100);
		lL = new listListener();
```

```
list.addActionListener(lL);  //list action listener
whisper = new JButton("<html>Whisper to<br>Another User</html>");
whisper.setSize(150, 75);
whisper.setLocation(700, 200);
wL = new whisperListener();
whisper.addActionListener(wL); //whisper action listener
eWhisper = new JButton("Exit Whisper Mode");
eWhisper.setSize(150, 75);
eWhisper.setLocation(700, 300);
eWL = new eWhisperListener();
eWhisper.addActionListener(eWL);  //eWhisper action listener
quit = new JButton("Exit Chat");
quit.setSize(150, 75);
quit.setLocation(700, 400);
qL = new quitListener();
quit.addActionListener(qL);  //quit action listener

//Chat input for the client to send messages to the server
chatIn = new JTextField();
chatIn.setSize(500, 40);
chatIn.setLocation(20, 575);
//button for sending messages
send = new JButton("Send Message");
send.setSize(130, 40);
send.setLocation(540, 575);
sL = new sendListener();
send.addActionListener(sL);  //send action listener

//chat "window" or "log" where all incoming messages are displayed
chat = new JTextArea();
chat.setEditable(false);

//creating new PrintStream to redirect System.out to the chat window GUI
PrintStream GUIStream = new PrintStream(new CustomOutputStream(chat));
System.setOut(GUIStream);
System.setErr(GUIStream);

//scroll wrapper for chat window so it's scrollable when there are many messages
chatScroll = new JScrollPane(chat);
chatScroll.setSize(650, 525);
chatScroll.setLocation(20, 20);

//frame, adding elements above
```

```
        frame = new JFrame();
        frame.add(list);
        frame.add(whisper);
        frame.add(eWhisper);
        frame.add(quit);
        frame.add(chatIn);
        frame.add(send);
        frame.add(chatScroll);
        frame.getRootPane().setDefaultButton(send);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //exit program if [X] button is clicked
        frame.setLayout(null);
        frame.setSize(900, 700);  //overall frame size
        frame.setTitle("Chat");  //program name
        frame.setVisible(true);

        setUpUsername();


}
//listener class for send button
private class sendListener implements ActionListener{
        @Override
        public void actionPerformed(ActionEvent arg0) {
                String chatString = chatIn.getText();   //get the text from the chat input
                chatIn.setText("");  //reset the chat input box to blank

                Message msg = new Message(username, whisperingTo, chatString);  //construct regular message object
                sendMessage (msg);

        }

}

//listener class for list button
private class listListener implements ActionListener{
        @Override
        public void actionPerformed(ActionEvent arg0) {
                Message msg = new Message("", "", "", false, false, true, false, false); //request server to send user list
                sendMessage(msg);
        }

}
```

```
	//listener class for whisper button
	private class whisperListener implements ActionListener{
		@Override
		public void actionPerformed(ActionEvent arg0) {
			String desiredWhisper = JOptionPane.showInputDialog(frame, "Which user would you like to whisper to?");
			if(desiredWhisper==null) {  //if user hits cancel button, don't change anything
				//do nothing
			}
			else if(desiredWhisper.isEmpty()) {  //if user enters blank whisper, set whisper to blank
				whisperingTo="";
			}
			else {
				whisperingTo = desiredWhisper;  //if user enters a username, set whisperingTo
			}
			System.out.println("Now whispering to: " + whisperingTo); //confirmation message in chat window
			System.out.print("> ");

		}

	}

	//listener class for eWhisper button
	private class eWhisperListener implements ActionListener{
		@Override
		public void actionPerformed(ActionEvent arg0) {
			whisperingTo = "";  //set whisperingTo to be blank (broadcast message)
			System.out.println("Whispering mode disabled. Your messages will now be sent to everyone.");  //confirmation in
chat window
			System.out.print("> ");
		}

	}

	//listener class for quit button
	private class quitListener implements ActionListener{
		@Override
		public void actionPerformed(ActionEvent arg0) {
			Message msg = new Message("", "", "", false, false, false, false, true); //request to stop connection
			sendMessage(msg);
		}

	}
```

```
	//listener class for incoming messages
	private class messageListener implements ChangeListener{

		@Override
		public void stateChanged(ChangeEvent e) {
			Message msg = (Message)e.getSource();  //pull the message from the listener
			if(msg.getNeedAnotherName()) {  //if the server reports the user needs to choose another username
				setUpUsernameAgain();
			}
			else if(msg.getWhisperNotExist()) {
				JOptionPane.showMessageDialog(frame, "The user you have tried to whisper to is not connected. The chat
will now exit whisper mode.");
				whisperingTo="";
				System.out.println("Whispering mode disabled. Your messages will now be sent to everyone.");
//confirmation in chat window
				System.out.print("> ");
			}
			else if(!(msg.getChatMessage().isEmpty())) {  //if there is a message body
				System.out.println(msg.getFromUsername() + ": " + msg.getChatMessage());
				System.out.print("> ");
			}
		}

	}

	private void sendMessage(Message msg) {
		try {
			out.writeObject(msg);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	private void setUpUsername() {
		String desiredName = "";
		boolean nameChosen = false;
		while (nameChosen == false) {   //while loop to make sure user actually enters a username
			desiredName = JOptionPane.showInputDialog(frame, "What would you like your username to be?");
			if(desiredName == null) {
				Message msg = new Message("", "", "", false, false, false, false, true); //request to stop connection
				sendMessage(msg);
				nameChosen = true;  //to exit the while loop
			}
			else if(desiredName.isEmpty()) {
```

```
                        //do nothing, nameChosen stays false
                }
                else {
                        nameChosen = true;   //user did input a name
                }
        }
        Message msg = new Message(desiredName, "", "", true, false, false, false, false);  //construct username check message
        sendMessage(msg);
        username = desiredName;   //this client knows the username as well
    }

    private void setUpUsernameAgain() {  //to be used if username was already taken and need to ask user again
        String desiredName = "";
        boolean nameChosen = false;
        while (nameChosen == false) {   //while loop to make sure user actually enters a username
            desiredName = JOptionPane.showInputDialog(frame, "That username is already in use. Please choose another
username:");
            if(desiredName == null) {
                Message msg = new Message("", "", "", false, false, false, false, true); //request to stop connection
                sendMessage(msg);
                nameChosen = true;  //to exit the while loop
            }
            else if(desiredName.isEmpty()) {
                //do nothing, nameChosen stays false
            }
            else {
                nameChosen = true;   //user did input a name
            }
        }
        Message msg = new Message(desiredName, "", "", true, false, false, false, false);  //construct username check message
        sendMessage(msg);
        username = desiredName;   //this client knows the username as well
    }
}

class CustomOutputStream extends OutputStream{
    private JTextArea chat;

    //constructor takes in JTextArea from GUI
    CustomOutputStream(JTextArea jta){
        chat = jta;
    }
```

```
	@Override
	public void write(int arg0) throws IOException {
		//send data to GUI textArea
		chat.append(String.valueOf((char)arg0));
		//move to next line
		chat.setCaretPosition(chat.getDocument().getLength());

	}
}

class InputStreamReader extends Thread{  //this thread purely responsible for receiving incoming messages from the server

	private ObjectInputStream in;   //input stream
	private ChangeListener listener;  //GUI change listener, triggers when message arrives

	InputStreamReader(ObjectInputStream i){
		in = i;
		listener = null;
	}

	@Override
	public void run() {
		boolean stillRunning = true; //running set to true upon construction
		while(stillRunning && !(this.isInterrupted())) {
			Message msg = null;
			try {
				msg = (Message)in.readObject();   //retrieve message from readObject
			} catch (ClassNotFoundException e) {
				e.printStackTrace();
			} catch (IOException e) {  //connection with the server has failed (aka server terminated unexpectedly)
				stillRunning = false;
				System.out.println("Server was disconnected unexpectedly!");
				break;  //break out of while loop
			}

			if(msg.getStopConnection()) {   //if stopconnection is requested either by the client (via the server) or
directly by the server
				stillRunning = false;
				System.out.println("Disconnecting from server...");
			}

			ChangeEvent event = new ChangeEvent(msg);  //create new change event
			listener.stateChanged(event);   //trigger the listener with the event
```

```
			}

		}

		public ChangeListener getListener() {  //method so others can have access to the listener
			return listener;
		}

		public void setListener(ChangeListener cl) {  //method to put in listener
			listener = cl;
		}

}

//Message class is the foundation of information sent between server and clients
//All communication between server and clients is via a Message object
class Message implements Serializable{
		private static final long serialVersionUID = 1L;
		private String fromUsername;
		private String toUsername;  //blank if chatMessage is intended to be broadcasted to everyone
		private String chatMessage;

		//internal communication parameters
		private boolean usernameCheck;    //send upon initial client connection to check if username is ok
		private boolean needAnotherName;  //server sends back true if another username is needed
		private boolean usersList;        //requesting a list of users
		private boolean whisperNotExist;  //sent by server in true if client tries to whisper a nonexistant user
		private boolean stopConnection;   //sent when client/server wants the other component to close their end of the connection

		Message(String f, String t, String c){  //regular chat message
			fromUsername = f;
			toUsername = t;
			chatMessage = c;
			usernameCheck = false;
			needAnotherName = false;
			usersList = false;
			whisperNotExist = false;
			stopConnection = false;
		}

		Message(String f, String t, String c, boolean u, boolean n, boolean l, boolean w, boolean s){  //for internal boolean flags
			fromUsername = f;
			toUsername = t;
```

```
		chatMessage = c;
		usernameCheck = u;
		needAnotherName = n;
		usersList = l;
		whisperNotExist = w;
		stopConnection = s;
	}

	//SETTERS
	public void setFromUsername(String u) {
		fromUsername = u;
	}
	public void setToUsername(String u) {
		toUsername = u;
	}
	public void setChatMessage(String c) {
		chatMessage = c;
	}
	public void setUsernameCheck(boolean b) {
		usernameCheck = b;
	}
	public void setNeedAnotherName(boolean b) {
		needAnotherName = b;
	}
	public void setUsersList(boolean b) {
		usersList = b;
	}
	public void setWhisperNotExist(boolean b) {
		whisperNotExist = b;
	}
	public void setStopConnection(boolean b) {
		stopConnection = b;
	}

	//GETTERS
	public String getFromUsername() {
		return fromUsername;
	}
	public String getToUsername() {
		return toUsername;
	}
	public String getChatMessage() {
		return chatMessage;
```

```
	}
	public boolean getUsernameCheck() {
		return usernameCheck;
	}
	public boolean getNeedAnotherName() {
		return needAnotherName;
	}
	public boolean getUsersList() {
		return usersList;
	}
	public boolean getWhisperNotExist() {
		return whisperNotExist;
	}
	public boolean getStopConnection() {
		return stopConnection;
	}
}
```

Server Application:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

//Chat application server program - by Joshua McKeen - jdmckeen@wpi.edu

public class ServerMain extends Thread{

	private ServerSocket serverSocket;
	private int port;
	//	private boolean running = false;  //true while listening for new connections
	private List<ClientHandler> clients;
```

```
public ServerMain(int port) {  //constructor for ServerMain
	this.port = port;
}

public void startServer() {
	try {
		serverSocket = new ServerSocket(port);  //create new ServerSocket port 4315
		this.start();   //EXECUTES run()
	}catch (IOException e) {
		e.printStackTrace();
	}
}

public void stopServer() {
	System.out.println("Disconnecting all currently connected clients");
	//disconnect all the clients
	synchronized(clients) {
		Iterator<ClientHandler> itr = clients.iterator();  //use iterator to loop through list
		while(itr.hasNext()) {
			ClientHandler c = itr.next();
			c.shutDownClientFromServer();  //modified version of shutDownClient for accommodating iterator here
			itr.remove();   //remove client from the list here (instead of in shutDownClient)
		}
	}

	this.interrupt();  //stop main loop
	System.out.println("Server shutting down");
	try {
		serverSocket.close();
	} catch (IOException e) {
		e.printStackTrace();
	}
}

public void run() {
	//create the synchronized list to have a list of ClientHandlers
	clients = Collections.synchronizedList(new ArrayList<ClientHandler>());
	//		running = true;
	while (!(this.isInterrupted())) {
		try {
			System.out.println("Listening for new connections");
			Socket socket = serverSocket.accept();  //accept the next connection
```

```
                        ClientHandler clientHandler = new ClientHandler(socket, clients);  //pass new connection to new
clientHandler
                        clientHandler.start();
                        clients.add(clientHandler);

                }catch(IOException e) {
                        System.out.println("Socket Closed");
                }
        }
    }

    public void getClientList() {
        StringBuilder ulsb = new StringBuilder();  //start a StringBuilder to create a string with the list
        ulsb.append("Currently connected users: ");
        synchronized(clients) {
            for(ClientHandler client: clients) {  //loop through all connected clients
                String name = client.getUsername();  //get usernames from each ClientHandler
                ulsb.append(name);  //add username to the StringBuilder
                ulsb.append(", ");
            }
        }
        String userlist = ulsb.toString();   //turn StringBuilder into regular String
        System.out.println(userlist);
    }

    public static void main(String[] args) {
        int port = 4315;
        boolean keepMenuRunning = true;
        System.out.println("Starting chat server on port " + port);

        BufferedReader console = new BufferedReader(new InputStreamReader(System.in));

        ServerMain server = new ServerMain(port);
        server.startServer();

        System.out.println("Command line options: \"stop\" to stop server, \"list\" to view list of connected users");

        String option = "";

        while (keepMenuRunning) {
            try {
                option = console.readLine();
            } catch (IOException e) {
```

```
                        e.printStackTrace();
                }
                if(option.equals("stop")) {
                        server.stopServer();
                        keepMenuRunning = false;
                }
                else if(option.equals("list")) {
                        server.getClientList();
                        option = "";
                }else {
                        System.out.println("Not a recognized command.");
                        System.out.println("Command line options: \"stop\" to stop server, \"list\" to view list of connected
users");
                }
        }
    }
}

class ClientHandler extends Thread{   //a new ClientHandler thread is created for every client
    private Socket socket;    //this client's socket
    private String username;  //this client's username
    private boolean connectionRunning;  //run loop control for this client
    private List<ClientHandler> allClients; //reference to client list

    private ObjectInputStream in;  //input stream
    private ObjectOutputStream out;  //output stream

    ClientHandler (Socket s, List<ClientHandler> l){    //constructor for ClientHandler
        socket = s;                         //socket passed in from ServerSocket.accept() via ServerMain.run()
        username = "";                  //username starts off blank, to be established later with the client
        connectionRunning=true;         //loop control starts off true
        allClients = l;                 //client list reference passed in from ServerMain
        in = null;  //in and out streams created upon execution of run()
        out = null;
    }

    public void run() {
        System.out.println("Establishing new ClientHandler thread");

        //set up input and output streams for this client
        try {
            out = new ObjectOutputStream(socket.getOutputStream());
            in = new ObjectInputStream(socket.getInputStream());
```

```
		} catch (IOException e) {
			e.printStackTrace();
		}

		//THIS IS THE MAIN LOOP
		while(connectionRunning) {
			Message msg = null;
			try {
				msg = (Message)in.readObject();   //retrieve message from readObject
			} catch (ClassNotFoundException e) {
				e.printStackTrace();
			} catch (IOException e) {
				System.out.println("Socket for user " + username + " closed.");
				if(connectionRunning) {  //this is true if exception was triggered by an unexpected disconnect, false if
triggered by shutDownClient
					this.removeClient();
				}
				break;
			}
			//at this point we have a message from the client
			//work through message possibilities below

			//client is trying to set up username:
			if(msg.getUsernameCheck()) {
				String desiredName = msg.getFromUsername();  //get desired username from message
				if(allClients.size() == 1) {   //if there are no other clients
					username = desiredName;    //username is automatically accepted
					Message response = new Message("", desiredName, "", false, false, false, false, false);
					this.sendMessage(response);
					//log in the server console
					System.out.println("Connection with username " + username + " estabished.");
					//tell all the connected clients
					Message broadcast = new Message("Server", "", ("User " + username + " has connected."));
					synchronized(allClients) {
						for(ClientHandler client: allClients) {
							client.sendMessage(broadcast);
						}
					}
				}else {
					boolean nameOk = true;  //gets set to false if name is duplicate of another client's name
					synchronized(allClients) {
						for (ClientHandler client: allClients) {
```

```
                                        if(client.getUsername().equals(desiredName)) {  //if desired name is in use by
another client
                                                nameOk = false;  //name is not ok
                                        }
                                }
                        }
                        if(nameOk == true) {   //if username passed the test (not taken) construct and send response to
client
                                username = desiredName;   //username is accepted
                                Message response = new Message("", desiredName, "", false, false, false, false, false);
                                this.sendMessage(response);
                                //log in the server console
                                System.out.println("Connection with username " + username + " estabished.");
                                //tell all the clients
                                Message broadcast = new Message("Server", "", ("User " + username + " has connected."));
                                synchronized(allClients) {
                                        for(ClientHandler client: allClients) {
                                                client.sendMessage(broadcast);
                                        }
                                }
                        }else {   //otherwise tell client they need to pick another name
                                System.out.println("User needs to choose another name");  //for console log
                                Message response = new Message("", desiredName, "", false, true, false, false, false);
                                this.sendMessage(response);
                        }
                }
        }

        //client is asking for a user list
        else if(msg.getUsersList()) {
                System.out.println("User " + username + " is requesting a list of all users");
                StringBuilder ulsb = new StringBuilder();  //start a StringBuilder to create a string with the list
                ulsb.append("Currently connected users: ");
                synchronized(allClients) {
                        for(ClientHandler client: allClients) {  //loop through all connected clients
                                String name = client.getUsername();  //get usernames from each ClientHandler
                                ulsb.append(name);  //add username to the StringBuilder
                                ulsb.append(", ");
                        }
                }
                String userlist = ulsb.toString();   //turn StringBuilder into regular String
                Message listResponse = new Message("Server", "", userlist);  //build message
                this.sendMessage(listResponse);   //send the message to the client for THIS thread
```

```
                        System.out.println("User list was sent to " + username);
                }

                //client is asking to disconnect
                else if(msg.getStopConnection()) {
                        System.out.println("User " + username + " is requesting to disconnect");
                        this.shutDownClientConnection();
                }

                //client wants to send a chat message
                else {
                        if(!(msg.getToUsername().isEmpty())) {  //if user wants to whisper a message to a particular user
                                ClientHandler targetClient = null;  //this reference to be used for target client
                                synchronized(allClients) {
                                        for(ClientHandler client: allClients) {  //loop through client list
                                                if(client.getUsername().equals(msg.getToUsername())) {  //and find the client with
the desired username
                                                        targetClient = client;  //set the target client to this client
                                                }
                                        }
                                }
                                if(targetClient == null) {  //if the target client was not found (no user with desired username)
                                        this.sendMessage(new Message("", "", "", false, false, false, true, false)); //tell the
client the target user isn't online
                                        System.out.println(username + " tried to whisper to nonexistent user " +
msg.getToUsername());
                                }else {
                                        targetClient.sendMessage(msg);  //pass the message along to the target client
                                        this.sendMessage(msg);          //let the sending user see the message as well (in their
chat window)
                                        System.out.println("User " + username + " whispered to " + msg.getToUsername() + " message:
" + msg.getChatMessage());
                                }
                        }
                        else { //user wants to send a regular broadcast chat message
                                synchronized(allClients) {
                                        for(ClientHandler client: allClients) {       //send message to all connected clients
(including this one)
                                                client.sendMessage(msg);
                                        }
                                }
                                System.out.println(msg.getFromUsername() + ": " + msg.getChatMessage());
                        }
```

```
                }
            }
        }

        //this method responsible for sending all messages to this client (from this ClientHandler)
        //may be called by THIS client's run() method, another client's run() method, or the ServerMain run() method
        public void sendMessage(Message m) {
            try {
                out.writeObject(m);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        //method for server and other ClientHandlers to get username for this client
        public String getUsername() {
            return username;
        }

        //to shut down this client's connection
        public void shutDownClientConnection() {
            //send disconnect message to client
            Message msg = new Message("", "", "", false, false, false, false, true);
            this.sendMessage(msg);

            //close streams and socket
            try {
                in.close();
                out.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            //removing this client from the client list
            synchronized(allClients) {
                Iterator<ClientHandler> itr = allClients.iterator();  //use iterator to loop through list
                while(itr.hasNext()) {
                    ClientHandler c = itr.next();
                    if(c.getUsername().equals(username)) {  //if item in list has same username (aka is the same
clientHandler)
                        itr.remove();               //remove the clientHandler from the list
                    }
```

```
                }
            }

            System.out.println("Connection with user " + username + " has been terminated. Thread for this client will stop
now");
            Message broadcast = new Message("Server", "", ("User " + username + " has disconnected."));
            synchronized(allClients) {
                for(ClientHandler client: allClients) {
                    client.sendMessage(broadcast);
                }
            }
            connectionRunning = false;
        }
        //to shut down this client's connection, modified for server shutdown sequence
        public void shutDownClientFromServer() {
            //send disconnect message to client
            Message msg = new Message("", "", "", false, false, false, false, true);
            this.sendMessage(msg);

            //close streams and socket
            try {
                in.close();
                out.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            System.out.println("Connection with user " + username + " has been terminated. Thread for this client will stop
now");

            connectionRunning = false;
        }

        public void removeClient() {  //this method is used when a user has unexpectedly disconnected and needs to be removed from
the client list
            //removing this client from the client list
            synchronized(allClients) {
                Iterator<ClientHandler> itr = allClients.iterator();  //use iterator to loop through list
                while(itr.hasNext()) {
                    ClientHandler c = itr.next();
                    if(c.getUsername().equals(username)) {  //if item in list has same username (aka is the same
clientHandler)
```

```
                                itr.remove();               //remove the clientHandler from the list
                        }
                }
        }

        System.out.println("Connection with user " + username + " has been terminated. Thread for this client will stop
now");
        Message broadcast = new Message("Server", "", ("User " + username + " has disconnected."));
        synchronized(allClients) {
                for(ClientHandler client: allClients) {
                        client.sendMessage(broadcast);
                }
        }
        connectionRunning = false;
    }

}

//Message class is the foundation of information sent between server and clients
//All communication between server and clients is via a Message object
class Message implements Serializable{
    private static final long serialVersionUID = 1L;
    private String fromUsername;
    private String toUsername;  //blank if chatMessage is intended to be broadcasted to everyone
    private String chatMessage;

    //internal communication parameters
    private boolean usernameCheck;    //send upon initial client connection to check if username is ok
    private boolean needAnotherName;  //server sends back true if another username is needed
    private boolean usersList;        //requesting a list of users
    private boolean whisperNotExist;  //sent by server in true if client tries to whisper a nonexistant user
    private boolean stopConnection;   //sent when client/server wants the other component to close their end of the connection

    Message(String f, String t, String c){  //regular chat message
        fromUsername = f;
        toUsername = t;
        chatMessage = c;
        usernameCheck = false;
        needAnotherName = false;
        usersList = false;
        whisperNotExist = false;
        stopConnection = false;
    }
```

```
Message(String f, String t, String c, boolean u, boolean n, boolean l, boolean w, boolean s){  //for internal boolean flags
	fromUsername = f;
	toUsername = t;
	chatMessage = c;
	usernameCheck = u;
	needAnotherName = n;
	usersList = l;
	whisperNotExist = w;
	stopConnection = s;
}

//SETTERS
public void setFromUsername(String u) {
	fromUsername = u;
}
public void setToUsername(String u) {
	toUsername = u;
}
public void setChatMessage(String c) {
	chatMessage = c;
}
public void setUsernameCheck(boolean b) {
	usernameCheck = b;
}
public void setNeedAnotherName(boolean b) {
	needAnotherName = b;
}
public void setUsersList(boolean b) {
	usersList = b;
}
public void setWhisperNotExist(boolean b) {
	whisperNotExist = b;
}
public void setStopConnection(boolean b) {
	stopConnection = b;
}

//GETTERS
public String getFromUsername() {
	return fromUsername;
}
public String getToUsername() {
```

```
		return toUsername;
	}
	public String getChatMessage() {
		return chatMessage;
	}
	public boolean getUsernameCheck() {
		return usernameCheck;
	}
	public boolean getNeedAnotherName() {
		return needAnotherName;
	}
	public boolean getUsersList() {
		return usersList;
	}
	public boolean getWhisperNotExist() {
		return whisperNotExist;
	}
	public boolean getStopConnection() {
		return stopConnection;
	}
}
```