Nicholas Ferry

Dr. Xiaomei Zhang

CSCI 516

March 26, 2020

Lab 1 Report: A Client-Server Java Application

Introduction

The objective of the assignment was to create an efficient Internet Chat Relay(IRC) application using the Java programming language. Java is a powerful language that boasts the ability to work on any computing environment that contains a Java Virtual Machine(JVM). This highlights the incredible robust nature of the Java language. The IRC consists of both a server and client who have the same capabilities and user interface on the front-end. The main difference between the two applications is contained in the backend of the two applications. The server was responsible for establishing an open socket of a preset port number on the localhost (127.0.0.1) network that would listen for any client connection establishments. Essentially, the server application would wait and listen for a client to attempt to establish a connection on the localhost via the established port number via the server-created socket. The front-end consisted of Graphical User Interfaces(GUIs) that contained: a field for entering text desired to send to the opposing application (i.e. client -> server or server -> client), a button for sending the entered text, a display area for any text that was communicated between the two applications, a button to open a file chooser that allowed either the server or client to select a file to transfer between the two applications, a display area for displaying: the time taken to send a message and receive a reply

(more on this later), the average time taken after the 10th iteration of sent message and reply, and upon connection close the total time of the communication between the server and client application.

The Program Functionality

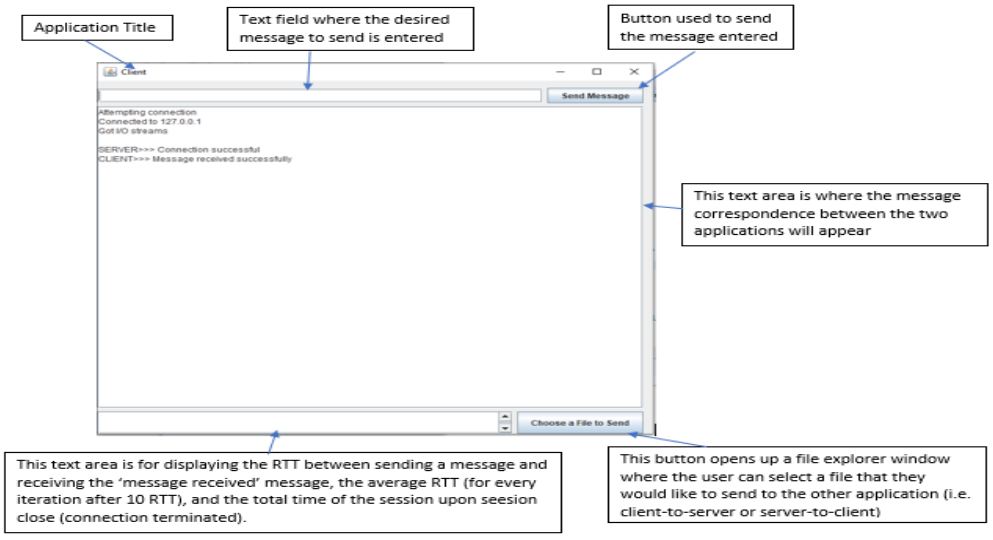
To better understand the applications the differences and similarities will be outlined below, starting with the similarities.

The Similarities

There are many similarities between the two applications. The two applications are practically mirrored images of each other in most of the functionalities.

The GUI

The Graphic User Interface is the window that represents the application graphically. The interface allows the user to execute the functionality of the application through a graphical, user-friendly way. The setup for the GUI is straightforward and apart of the many similarities that the applications share. In the image below, the GUI is shown and its interface is explained.



GUI Explained

Text Communication

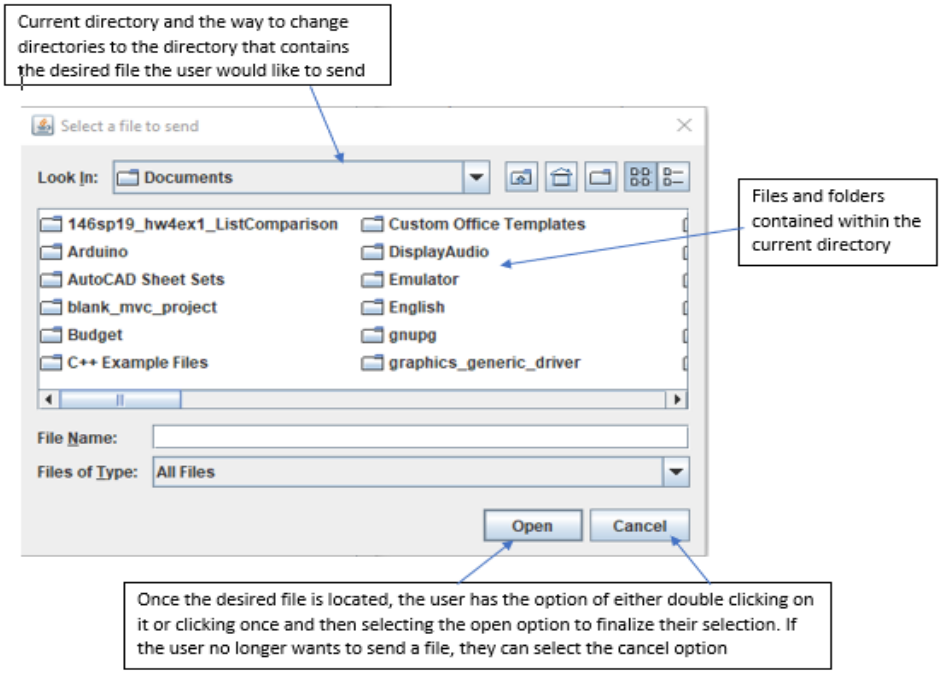
The text communication feature between the two applications is displayed in the main text area when a message is sent between the two applications. To send a message, the message must be typed into the text field at the top of the GUI. When the message that the user desires to send is typed out, the user has the option of pressing the “Enter” key or the button to the right of the text field to send the message. Immediately after receiving a message, the receiving application sends a response of “Message received successfully” back to indicate the message was received successfully. As the text area fills up, the content will move down and the window will gain a scroll bar that allows the user to view the previous content from the beginning of the communication.

Round Trip Time (RTT), Average RTT, and Total Session RTT Calculator

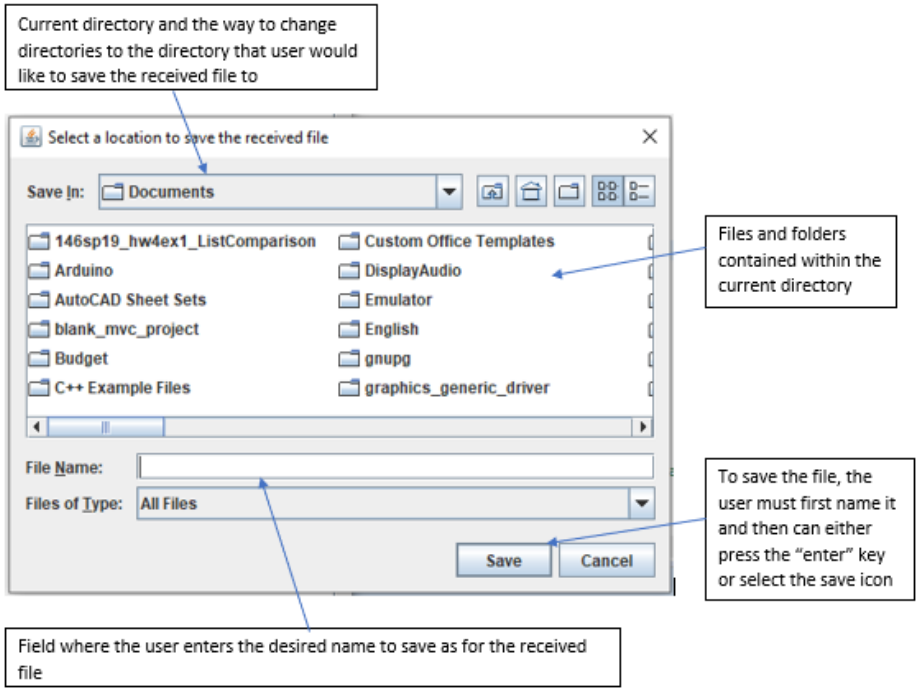
The Round Trip Time (RTT) describes the amount of time it takes for a message to be sent from one application and the “Message received successfully” response from the receiving application to occur. The respective application computes the RTT and displays the result in the bottom text area. After ten RTT’s are calculated, the application computes the average of the RTT’s and display it in the same text area. Similar to the text communications text area, the RTT’s text area will gain a scroll bar as the content fills the area up allowing the user to view previous RTT calculations.

The File-Sharing

The file-sharing feature of the applications allows for the applications to share files. Through this feature, a user can select a file to share through a convenient explorer window. When the file is sent, the receiving application can then select a location to save the file and what they would like to name it. When saving the file, the user does NOT have to append the extension to the file name. Instead, the application will automatically append the extension to the file based on the extension of the file that was sent. See below the GUI representation of this feature.



Send File GUI



Receive File GUI

The Differences

Next, the differences between the two applications will be highlighted to ensure a full understanding of the two applications.

How the Connection is Made

The primary difference between the applications is how the connection is firmed and established between the two applications. The server's responsibility is to create a socket on the localhost (127.0.0.1) and port 1007. The server then waits for any connection requests made on this socket. The client's responsibility is to then send a connection request through the localhost on the same port that the server is listening on.

The Code

Server Class

```
import java.io.*;
import java.net.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.lang.Thread;

/**
 * Class implementation of TextingServerJF that extends JFrame
 *
 * @author nferry@email.sc.edu
 *
 */
public class TextingServerJF /* extends Thread */ {
    // Class-wide fields declarations
    private JFrame frame;
    private JPanel panel;
    private GroupLayout layout;
    private JTextField enterField;
    private JButton sendTextButton;
    private JTextArea displayArea;
    private JScrollPane displayScroll;
```

```
    private JTextArea infoField;
    private JScrollPane infoScroll;
    private JButton sendFileButton;
    private ObjectOutputStream output;
    private ObjectInputStream input;
    private ServerSocket server;
    private Socket connection;
    private int counter = 1;
    private long time;
    private long rtt;
    private ArrayList<Long> timeAverage = new ArrayList<Long>();
    private int timeCount;
    private JFileChooser chooser;
    private JFileChooser receiver;
    // private boolean isRunning;

    /**
     * Constructor for the TextingServerJF Class.
     */
    public TextingServerJF(/* Socket socket, ObjectInputStream input,
ObjectOutputStream output */) {

        /*
         * For the multithreading failure this.connection = socket;
this.input = input;
         * this.output = output;
         */
        setupGUI();
    } // end TextingServerJF constructor

    /**
     * This method sets up the window and all of the necessary components
needed to
     * create the GUI
     */
    public void setupGUI() {

        frame = new JFrame("Server");
        frame.setSize(600, 700);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        panel = new JPanel();

        /*
         * Create a new editable JTextField. We set the editing ability of
the field to
         * false in the beginning so that the field does not expect an input
at the
         * initial runtime of the program. Then we attach an event listener
to the text
         * field that waits for an enter key to be pressed. When the enter
key is
         * pressed, the action listener takes whatever characters are in the
field and
         * sends them to the sendData method. After the data is sent the
field is set to
         * an empty field again.
```

```
         */

        enterField = new JTextField();
        enterField.setEditable(false);
        enterField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                sendData(event.getActionCommand());
                enterField.setText("");
            } // end actionPerformed
        } // end class ActionListener
        ); // end addActionListener

        // Set the location of the enter field to the north section of the
border layout
        // panel.add(enterField, BorderLayout.NORTH);

        /*
         * Create a new JTextarea for displaying text and set it to the
center location
         * of the border as well as making it scalable through scrolling. We
then set
         * the size of the entire window (and all of its contents) and set
the
         * visibility to true
         */
        displayArea = new JTextArea();
        displayArea.setColumns(20);
        displayArea.setRows(5);
        displayArea.setEditable(false);
        displayScroll = new JScrollPane();
        displayScroll.setViewportView(displayArea);

        infoField = new JTextArea();
        infoField.setColumns(20);
        infoField.setRows(5);
        infoField.setEditable(false);
        infoScroll = new JScrollPane();
        infoScroll.setViewportView(infoField);

        chooser = new JFileChooser();
        chooser.setDialogTitle("Select a file to send");
        chooser.setCurrentDirectory(new
File("C:\\Users\\compu\\OneDrive\\Old\\Documents\\"));

        receiver = new JFileChooser();
        receiver.setDialogTitle("Select a location to save the received
file");

        sendFileButton = new JButton("Choose a File to Send");
        sendFileButton.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                sendData("data sent");
                setupFileChooser();
            }// end actionPerformed
        }// end ActionListener
```

```
        ); // end addActionListener

        sendTextButton = new JButton("Send Message");
        sendTextButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                sendData(enterField.getText());
                enterField.setText("");
            } // end actionPerformed
        } // end class ActionListener
        ); // end addActionListener

        layout = new GroupLayout(panel);
        panel.setLayout(layout);

layout.setHorizontalGroup(layout.createParallelGroup(GroupLayout.Alignment.L
EADING).addGroup(layout
                .createSequentialGroup()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                        .addGroup(GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
                                .addComponent(infoScroll,
GroupLayout.DEFAULT_SIZE, 511, Short.MAX_VALUE)

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED).addComponent(sendFi
leButton)) // end

// inner

// sequential

// group
                        .addComponent(displayScroll,
GroupLayout.Alignment.TRAILING)

.addGroup(layout.createSequentialGroup().addComponent(enterField)

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED).addComponent(sendTe
xtButton)) // end

// inner

// sequential

// group
                ) // end inner parallel group
                .addContainerGap()) // end sequential group
        ); // end horizontal group


layout.setVerticalGroup(layout.createParallelGroup(GroupLayout.Alignment.LEA
DING)
                .addGroup(GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup().addContainerGap()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
```

```
                                    .addComponent(enterField,
GroupLayout.PREFERRED_SIZE, 23, GroupLayout.PREFERRED_SIZE)
                                    .addComponent(sendTextButton,
GroupLayout.DEFAULT_SIZE, GroupLayout.DEFAULT_SIZE,
                                                Short.MAX_VALUE))

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                            .addComponent(displayScroll,
GroupLayout.DEFAULT_SIZE, 516, Short.MAX_VALUE)

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING, false)
                                    .addComponent(infoScroll,
GroupLayout.PREFERRED_SIZE, 0, Short.MAX_VALUE)
                                    .addComponent(sendFileButton,
GroupLayout.DEFAULT_SIZE, 38, Short.MAX_VALUE)) // end

// inner

// parallel

// group
                ) // end inner sequential group
        ); // end vertical group

        // add the panel to the frame, 'pack' everything in, and set it to
be visible
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

    } // end method setupGUI

    /**
     * This method creates a new socket for the server in a try block.
Within the
     * try block, there is a while block that runs throughout the length of
the
     * program. Within the while block there is a try block that waits for a
client
     * connection, retrieves the input/output streams and then processes the
     * connection streams from the client. Finally, the connection is closed
and the
     * client counter is increased.
     */

    public void runServer() {

        try {
            server = new ServerSocket(10007, 100);

            while (true) {
                try {
                    waitForConnection();
                    getStreams();
```

```
                    time = System.nanoTime(); // setup the intital time with
the start of the application
                    processConnection();
                } catch (EOFException eofException) {
                    displayMessage("\nServer terminated connection");
                } catch (IOException e) {
                    e.printStackTrace();
                } finally {
                    closeConnection();
                    ++counter;
                } // end inner try block
            } // end while

        } catch (IOException ioException) {
            ioException.printStackTrace();
        } // end outer try block

    } // end runServer()

    /**
     * This method accepts connection(s) from clients and then displays
"connection
     * i received from client address and hostname. If no connection is
received, an
     * IOException is thrown
     *
     * @throws IOException
     */

    private void waitForConnection() throws IOException {
        displayMessage("Waiting for connection\n");
        connection = server.accept();
        displayMessage("Connection " + counter + " received from " +
connection.getInetAddress().getHostName());
    } // waitForConnection

    /**
     * This method retrieves the input and output streams from the
     * connection/client. The method allows the server to know what the
input and
     * output are of the client. The flush method forces anything in the
output
     * stream to be forced out. If no streams are retrieved then an
IOException is
     * thrown.
     *
     * @throws IOException
     */

    private void getStreams() throws IOException {
        output = new ObjectOutputStream(connection.getOutputStream());
        output.flush();

        input = new ObjectInputStream(connection.getInputStream());

        displayMessage("\nGot I/O streams\n");
```

```
    } // end method getStreams

    /**
     * The processConnection method first states that the connection was
successful
     * (as it is if we reached this method in the runServer() try block) and
then
     * inside of a try block (that is inside of a do-while block) detects
the input
     * from the client and calls the displayMessage method to display the
input on
     * the displayArea. This is done until the client sends a "TERMINATE"
message.
     * If an incorrect input is found, an IOException is thrown.
     *
     * @throws IOException
     */
    private void processConnection() throws IOException {
        String message = "Connection successful";
        sendData(message);
        setTextFieldEditable(true);

        do {
            try {
                /*
                 * Code to enter the "receive file mode" if the client sends
a file
                 */
                if (message.equals("CLIENT>>> data sent")) {
                    receiveFile();
                } // end if

                message = (String) input.readObject();

                displayMessage("\n" + message);

                /*
                 * Code for step 1 of the lab -- Code looks for a
                 * "Message received successfully" from the client and if it
gets one, it does
                 * nothing. Otherwise the message is sent from the server
that IT received the
                 * message successfully. Essentially the server is echoing
that it received the
                 * message if the client sends one.
                 */

                if (message.equals("CLIENT>>> Message received
successfully")) {
                    // Do nothing to avoid an infinite loop for step 1
                    // Below is for step 2
                    displayTime(getTime(System.nanoTime()));
                    setTimeAverage(rtt);
                    timeCount++;
                    time = 0;
                    if (timeCount > 10) {
```

```
                        displayAverage(getTimeAverage(timeAverage,
timeCount), timeCount);
                    } // end inner if
                    // end step 2
                } else {
                    sendData("Message received successfully");
                } // end if/else

            }
            catch (ClassNotFoundException classNotFoundException) {
                displayMessage("\nUnknown object type received");
            } // end try block
        } while (!message.equals("CLIENT>>> TERMINATE")); // end do-while
    } // end method processConnection

    /**
     * This method is called when the connection is terminated. The method
first
     * displays that the connection is being terminated, sets the text field
     * editable to false, closes the IO streams, and finally closes the
connection
     * to the client. If something goes wrong, an IOException is thrown and
the
     * printStackTrace method is called
     */
    private void closeConnection() {
        displayTotalTime(getTotalTime(timeAverage)); // For step 3
        displayMessage("\nTerminating connection\n");
        setTextFieldEditable(false);

        try {
            output.close();
            input.close();
            connection.close();
        } catch (IOException ioException) {
            ioException.printStackTrace();
        } // end try block
    } // end method closeConnection

    /**
     * The sendData method writes the parameter message to the output
stream. It
     * then calls the displayMessage method to display the message on the
JTextArea
     * displayArea. If an error is found, an IOException is thrown.
     *
     * @param message
     */
    private void sendData(String message) {
        try {
            output.writeObject("SERVER>>> " + message);
            output.flush();

            // Used for computing the RTT required by step 2
            time = System.nanoTime();

            displayMessage("\nSERVER>>> " + message);
```

```
        } catch (IOException ioException) {
            displayArea.append("\nError Writing object");
        } // end try block

    } // end method sendData

    /**
     * This method is adding the message to display to the Event Dispatching
Thread
     * so that the message gets displayed in the JTextArea displayArea.
     *
     * @param messageToDisplay
     */
    private void displayMessage(final String messageToDisplay) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                displayArea.append(messageToDisplay);
            } // end run method
        } // end anonymous Runnable class call
        ); // end invokeLater method
    } // end method isplayMessage

    /**
     * This method allows the user to write a message in the editable text
field so
     * that the user can send a message to the client.
     *
     * @param editable
     */
    private void setTextFieldEditable(final boolean editable) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                enterField.setEditable(editable);
            } // end method run
        } // end Anonymous class call to Runnable
        ); // end invokeLater method
    } // end setTextFieldEitable method

    /**
     * This method computes the RTT from message send to echo received by
subtracting the
     * time from the input
     * @param time
     */
    private long getTime(long time) {
        this.rtt = time - this.time;
        return this.rtt;
    } // end method setTime

    /**
     * This method adds the display of the RTT to the Event Dispatch Thread
and is
     * displayed in the infoField area at the bottom of the JFrame
     *
     * @param time
     */
```

```
    private void displayTime(long time) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                infoField.append("The RTT for the last message is: " +
Long.toString(time) + " ns\n");
            } // end run method
        } // end anonymous implementation of the Runnable class
        ); // end invokeLater method
    } // end method displayTime

    /**
     * This method adds the individual RTTs to an array list
     *
     * @param time
     */
    private void setTimeAverage(long time) {
        timeAverage.add(time);
    } // end method setTimeAverage

    /**
     * This method computes the average time by adding up all of the items
in the
     * array list and dividing that number by the timeCount
     *
     * @param timeAverage
     * @param timeCount
     * @return average
     */
    private long getTimeAverage(ArrayList<Long> timeAverage, int timeCount)
{
        long average = 0;

        for (int i = 0; i < timeCount; i++) {
            average += timeAverage.get(i);
        } // end for

        average = average / timeCount;

        return average;
    } // end getTimeAverage

    /**
     * This method adds the display of the average RTT to the Event Dispatch
Thread
     * and is displayed in the infoField area at the bottom of the JFrame
     *
     * @param average
     * @param timeCount
     */
    private void displayAverage(long average, int timeCount) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                infoField.append(
                        "The average RTT for the RTT count " + timeCount + "
is " + Long.toString(average) + " ns\n");
            } // end run method
        } // end anonymous implementation of the Runnable class
```

```
        ); // end invokeLater method
    } // end method displayAverage

    /**
     * This method computes the total time for the session by adding
together all of
     * the elements inside the timeAverage array (which contains each RTT)
     *
     * @param timeAverage
     * @return
     */
    private long getTotalTime(ArrayList<Long> timeAverage) {
        long totalTime = 0;

        for (int i = 0; i < timeAverage.size(); i++) {
            totalTime += timeAverage.get(i);
        } // end for

        return totalTime;
    } // end getTotalTime method

    /**
     * This method adds the display of the total RTT to the Event Dispatch
Thread
     * and is displayed in the infoField area at the bottom of the JFrame
     *
     * @param totalTime
     */
    private void displayTotalTime(long totalTime) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                infoField.append("The total RTT for this session is: " +
Long.toString(totalTime) + " ns\n");
            } // end run method
        } // end anonymous implementation of the Runnable class
        ); // end invokeLater method
    } // end method displayTotalTime

    /**
     * This method writes a file to the designated path from bytes received
within
     * the input stream based on the user selected path and filename
     *
     * @throws ClassNotFoundException
     * @throws IOException
     */
    public void receiveFile() throws ClassNotFoundException, IOException {

        String extension = (String) input.readObject();
        byte[] fileContent = null;
        fileContent = (byte[]) input.readObject();

        int userSaveFile = receiver.showSaveDialog(new JFrame());

        if (userSaveFile == JFileChooser.APPROVE_OPTION) {
            File directory = receiver.getCurrentDirectory();
            String filename = receiver.getSelectedFile().getName();
```

```
            File fileToSave = new File(directory + "\\" + filename +
extension);
            System.out.println(receiver.getCurrentDirectory());
            System.out.println(filename);
            Files.write(fileToSave.toPath(), fileContent);
        } // end if

    } // end method receiveFile

    /**
     * This method sends the file from the input parameter and its extension
by
     * converting it to bytes and sending it out through the output stream.
     *
     * @param file
     */
    private void sendFile(File file) {
        byte[] fileContent;
        String fileExtension = getFileExtension(file);

        try {
            fileContent = Files.readAllBytes(file.toPath());
            output.writeObject(fileExtension);
            output.flush();
            output.writeObject(fileContent);
            output.flush();
        } catch (IOException e) {
            e.printStackTrace();
        } // end try block
    } // end method sendFile

    /**
     * This method opens a new JFileChooser JFrame that allows a user to
send a
     * desired file to send to the server.
     */
    public void setupFileChooser() {
        int result = chooser.showOpenDialog(new JFrame());
        if (result == JFileChooser.APPROVE_OPTION) {
            File selectedFile = chooser.getSelectedFile();
            sendFile(selectedFile);
        } // end if

    } // end method setupFileChooser

    /**
     * This method retrieves the file extension of the file in the input
parameter
     *
     * @param file
     * @return
     */
    public String getFileExtension(File file) {
        String fileExtension = file.getName();
        int lastIndexOf = fileExtension.lastIndexOf('.');
        return fileExtension.substring(lastIndexOf);
    } // end method getFileExtension
```

```
} // end class TextingServerJF
```

Server Test Implementation

```
/**
 * Test class implementation of the Server
 * @author nferry@email.sc.edu
 */
public class TextingServerTest
{
    /**
     * Main method for the test class
     * @param args
     */
    public static void main(String[] args)
    {
        // Create a new TextingServerJF object
        TextingServerJF application = new TextingServerJF();
        //ClientHandler application = new ClientHandler();

        // Call the runServer method contained within the newly created
application object
        application.runServer();

    } // end main method



} // end class TextingServerTest
```

Client Class

```
import java.io.*;
import java.net.*;
import java.nio.file.Files;
import java.awt.*;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;

/**
 * Class decleration for the TextingClientJF class that extends JFrame
 * @author nferry@email.sc.edu
 */
public class TextingClientJF
{
        // Class-wide fields declarations
        private JFrame frame;
        private JPanel panel;
```

```
        private GroupLayout layout;
        private JTextField enterField;
        private JTextArea displayArea;
        private JScrollPane displayScroll;
        private JButton sendTextButton;
        private JTextArea infoField;
        private JScrollPane infoScroll;
        private JButton sendFileButton;
        private ObjectOutputStream output;
        private ObjectInputStream input;
        private Socket client;
        private String message="";
        private String chatServer; // host server
        private long time;
        private long rtt;
        private ArrayList<Long> timeAverage = new ArrayList<Long>();
        private int timeCount;
        File file = new
File("C:\\Users\\compu\\OneDrive\\Old\\Documents\\testImage.jpg");
        private JFileChooser chooser;
        private JFileChooser receiver;
        //private int port;

        /**
         * Constructor for the texting client app. The constructor first
sets the chatserver
         * to the host provided by the parameter when the class is called.
         * @param host
         */
        public TextingClientJF(String host)
        {
            // Sets the chatServer to the host input from the parameter when
the class is called
            chatServer = host;
            //this.port = port;
            setupGUI();

        } // end Server()

        /**
         * This method sets up the window and all of the necessary
components needed to create the GUI
         */
        public void setupGUI() {

            frame = new JFrame("Client");
            frame.setSize(600, 700);
            frame.setVisible(true);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            panel = new JPanel();

            /*
            * Create a new editable JTextField. We set the editing ability
of the field
            * to false in the beginning so that the field does not expect an
input at the
```

```
            * initial runtime of the program. Then we attach an event
listener to the text
            * field that waits for an enter key to be pressed. When the
enter key is pressed,
            * the action listener takes whatever characters are in the field
and sends them to
            * the sendData method. After the data is sent the field is set
to an empty field again
            */

            enterField = new JTextField();
            enterField.setEditable(false);
            enterField.addActionListener(
                new ActionListener()
                {
                    public void actionPerformed (ActionEvent event)
                    {
                        sendData( event.getActionCommand() );
                        enterField.setText("");
                    } // end actionPerformed
                } // end class ActionListener
            );  // end addActionListener

            // Set the location of the enter field to the north section of
the border layout
            //panel.add(enterField, BorderLayout.NORTH);

            /*
            * Create a new JTextarea for displaying text and set it to the
center location
            * of the border as well as making it scalable through scrolling.
We then set the
            * size of the entire window (and all of its contents) and set
the visibility to true
            */
            displayArea = new JTextArea();
            displayArea.setColumns(20);
            displayArea.setRows(5);
            displayArea.setEditable(false);
            displayScroll = new JScrollPane();
            displayScroll.setViewportView(displayArea);

            infoField = new JTextArea();
            infoField.setColumns(20);
            infoField.setRows(5);
            infoField.setEditable(false);
            infoScroll = new JScrollPane();
            infoScroll.setViewportView(infoField);

            chooser = new JFileChooser();
            chooser.setDialogTitle("Select a file to send");
            chooser.setCurrentDirectory(new
File("C:\\Users\\compu\\OneDrive\\Old\\Documents\\"));

            receiver = new JFileChooser();
            receiver.setDialogTitle("Select a location to save the received
file");
```

```
            sendFileButton = new JButton("Choose a File to Send");
            sendFileButton.addActionListener(
                    new ActionListener() {

                        @Override
                        public void actionPerformed(ActionEvent e)
                        {
                            sendData("data sent");
                            setupFileChooser();
                        }// end actionPerformed
                    }// end ActionListener
            ); // end addActionListener



            sendTextButton = new JButton("Send Message");
            sendTextButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed (ActionEvent e)
                {
                    sendData( enterField.getText() );
                    enterField.setText("");
                } // end actionPerformed
            } // end class ActionListener
        );  // end addActionListener

            layout = new GroupLayout(panel);
            panel.setLayout(layout);
            layout.setHorizontalGroup(

layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                    .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                            .addGroup(GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
                                    .addComponent(infoScroll,
GroupLayout.DEFAULT_SIZE, 511, Short.MAX_VALUE)

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                                    .addComponent(sendFileButton)
                                     ) // end inner sequential group
                            .addComponent(displayScroll,
GroupLayout.Alignment.TRAILING)
                            .addGroup(layout.createSequentialGroup()
                                    .addComponent(enterField)

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                                    .addComponent(sendTextButton)
                            ) // end inner sequential group
                        ) // end inner parallel group
                        .addContainerGap()
                    ) // end sequential group
            ); // end horizontal group
```

```
            layout.setVerticalGroup(

layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                    .addGroup(GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
                        .addContainerGap()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                            .addComponent(enterField,
GroupLayout.PREFERRED_SIZE, 23, GroupLayout.PREFERRED_SIZE)
                            .addComponent(sendTextButton,
GroupLayout.DEFAULT_SIZE, GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(displayScroll,
GroupLayout.DEFAULT_SIZE, 516, Short.MAX_VALUE)

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING, false)
                            .addComponent(infoScroll,
GroupLayout.PREFERRED_SIZE, 0, Short.MAX_VALUE)
                            .addComponent(sendFileButton,
GroupLayout.DEFAULT_SIZE, 38, Short.MAX_VALUE)
                            ) // end inner parallel group
                        ) // end outer sequential group
            ); // end vertical group

            // add the panel to the frame, 'pack' everything in, and set it
to be visible
            frame.add(panel);
            frame.pack();
            frame.setVisible(true);

        } // end method setupGUI

        /**
         * Within the try block, there is a while block that runs throughout
the length of the program.
         * Within the while block there is a try block that waits for a
server connection,
         * retrieves the input/output streams and then processes the
connection streams from the client.
         * Finally, the connection is closed.
         */
        public void runClient()
        {
            try
            {
                connectToServer();
                getStreams();
                processConnection();
            }
            catch (EOFException eofException)
            {
                displayMessage("\nServer terminated connection");
            }
```

```
            catch (IOException ioException)
            {
                ioException.printStackTrace();
            }
            finally
            {
                closeConnection();
            } // end try block

        } //end runClient()

        /**
         * This method opens a new socket for the client based on the
designated host within the
         * chatServer variable. An exception is thrown if there is an
unknown host or if the
         * IO streams cannot be found.
         * @throws IOException
         */
        private void connectToServer() throws IOException
        {
            try{
                displayMessage("Attempting connection \n");
                client = new Socket( chatServer, 10007);
                displayMessage("Connected to " +
client.getInetAddress().getHostName());
                }
            catch (UnknownHostException e) {
                System.err.println("Don't know about host: " + chatServer);
                System.exit(1);
                }
            catch (IOException e) {
                System.err.println("Couldn't get I/O for " + "the connection
to: " + chatServer);
                System.exit(1);
            } // end try block
        } // end method connectToServer

        /**
         * This method retrieves the input and output streams from the
connection/server. The method
         * allows the server to know what the input and output are of the
server. The flush method
         * forces anything in the output stream to be forced out. If no
streams are retrieved then
         * an IOException is thrown.
         * @throws IOException
         */
        private void getStreams() throws IOException
        {
            output =  new ObjectOutputStream(client.getOutputStream());
            output.flush();

            input = new ObjectInputStream( client.getInputStream() );

            displayMessage( "\nGot I/O streams\n");
        } // end method getStreams
```

```
        /**
         * The processConnection method first states that the connection was
successful (as it
         * is if we reached this method in the runServer() try block) and
then inside of a try block
         * (that is inside of a do-while block) detects the input from the
server and calls
         * the displayMessage method to display the input on the
displayArea. This is done until
         * the server sends a "TERMINATE" message. If an incorrect input is
found, an IOException
         * is thrown.
         * @throws IOException
         */
        private void processConnection() throws IOException
        {
            setTextFieldEditable(true);

            do
            {
                try
                {
                    /*
                     * Code to enter the "receive file mode" if the server
sends a file
                     */
                    if (message.equals("SERVER>>> data sent")) {
                        receiveFile();
                    } // end if

                    message = (String) input.readObject();
                    displayMessage("\n"+ message);

                    /*
                     * Code for step 1 of the lab -- Code looks for a
"Message received successfully" from the server
                     * and if it gets one, it does nothing. Otherwise the
message is sent from the server that IT received
                     * the message successfully. Essentially the client is
echoing that it received the message if the server
                     * sends one.
                     */
                    if (message.equals("SERVER>>> Message received
successfully")) {
                        // Do nothing to avoid infinite loop for step 1
                        // Below is for step 2
                        displayTime(getTime(System.nanoTime()));
                        setTimeAverage(rtt);
                        timeCount++;
                        time = 0;
                        if (timeCount > 10) {
                            displayAverage(getTimeAverage(timeAverage,
timeCount), timeCount);
                        } // end inner if
                    } else {
                        sendData("Message received successfully");
```

```
                    } // end if/else


                }
                catch (ClassNotFoundException classNotFoundException)
                {
                    displayMessage("\nUnknown object type received");
                } // end try block
            }while (!message.equals("SERVER>>> TERMINATE")); // end do-while
        } // end processConnection method

        /**
         * This method is called when the connection is terminated. The
method first displays
         * that the connection is being terminated, sets the text field
editable to false, closes the
         * IO streams, and finally closes the connection to the server. If
something goes wrong, an
         * IOException is thrown and the printStackTrace method is called
         */
        private void closeConnection()
        {
            displayTotalTime(getTotalTime(timeAverage)); // For step 3
            displayMessage("\nTerminating connection\n");
            setTextFieldEditable(false);

            try
            {
                output.close();
                input.close();
                client.close();
            }
            catch (IOException ioException)
            {
                ioException.printStackTrace();
            } // end try block
        } // end closeConnection method


        /**
         * The sendData method writes the parameter message to the output
stream. It then calls
         * the displayMessage method to display the message on the JTextArea
displayArea. If
         * an error is found, an IOException is thrown.
         * @param message
         */
        private void sendData (String message)
        {
            try
            {
                output.writeObject("CLIENT>>> " + message);
                output.flush();

                time = System.nanoTime();   // part of step 2
```

```
                displayMessage("\nCLIENT>>> " + message);


            }
            catch (IOException ioException)
            {
                displayArea.append("\nError Writing object");
            } // end try block
        } // end sendData method

        /**
         * This method is adding the message to display to the Event
Dispatching Thread so that
         * the message gets displayed in the JTextArea displayArea.
         * @param messageToDisplay
         */
        private void displayMessage(final String messageToDisplay)
        {
            SwingUtilities.invokeLater( new Runnable()
                {
                    public void run()
                    {
                        displayArea.append( messageToDisplay);
                    } // end run method
                } // end anonymous implementation of the Runnable class
            ); // end invokeLater method
        } // end method displayMessage

        /**
         * This method allows the user to write a message in the editable
text field so that the user
         * can send a message to the server.
         * @param editable
         */
        private void setTextFieldEditable( final boolean editable)
        {
            SwingUtilities.invokeLater( new Runnable()
                {
                    public void run()
                    {
                        enterField.setEditable (editable);
                    } // end run method
                } // end anonymous call to the Runnable class
            ); // end invokeLater method
        } // end method setTextFieldEditable

        /**
         * This method computes the RTT from message send to echo received
         * @param time
         */
        private long getTime(long time) {
            this.rtt = time - this.time;
            return this.rtt;
        } // end method setTime

        /**
```

```
         * This method adds the display of the RTT to the Event Dispatch
Thread and is displayed in the
         * infoField area at the bottom of the JFrame
         * @param time
         */
        private void displayTime(long time)
        {
            SwingUtilities.invokeLater( new Runnable()
                {
                    public void run()
                    {
                        infoField.append("The RTT for the last message is: "
+ Long.toString(time) + " ns\n");
                    } // end run method
                } // end anonymous implementation of the Runnable class
            ); // end invokeLater method
        } // end method displayTime

        /**
         * This method adds the individual RTTs to an array list
         * @param time
         */
        private void setTimeAverage(long time) {
            timeAverage.add(time);
        } // end method setTimeAverage

        /**
         * This method computes the average time by adding up all of the
items in the array list and
         * dividing that number by the timeCount
         * @param timeAverage
         * @param timeCount
         * @return average
         */
        private long getTimeAverage(ArrayList<Long> timeAverage, int
timeCount) {
            long average = 0;

            for (int i = 0; i < timeCount; i++ ) {
                average +=  timeAverage.get(i);
            } // end for

            average = average / timeCount;

            return average;
        } // end getTimeAverage

        /**
         * This method adds the display of the average RTT to the Event
Dispatch Thread and is displayed in
         * the infoField area at the bottom of the JFrame
         * @param average
         * @param timeCount
         */
        private void displayAverage(long average, int timeCount)
        {
            SwingUtilities.invokeLater( new Runnable()
```

```
                {
                    public void run()
                    {
                        infoField.append("The average RTT for the RTT count
" + timeCount +
                                         " is " + Long.toString(average) + " ns\n");
                    } // end run method
                } // end anonymous implementation of the Runnable class
            ); // end invokeLater method
        } // end method displayAverage

        /**
         * This method computes the total time for the session by adding
together all of the elements inside
         * the timeAverage array (which contains each RTT)
         * @param timeAverage
         * @return
         */
        private long getTotalTime(ArrayList<Long> timeAverage) {
            long totalTime = 0;

            for (int i = 0; i < timeAverage.size(); i++) {
                totalTime += timeAverage.get(i);
            } // end for

            return totalTime;
        } // end getTotalTime method

        /**
         * This method adds the display of the total RTT to the Event
Dispatch Thread and is displayed in
         * the infoField area at the bottom of the JFrame
         * @param totalTime
         */
        private void displayTotalTime(long totalTime)
        {
            SwingUtilities.invokeLater( new Runnable()
                {
                    public void run()
                    {
                        infoField.append("The total RTT for this session is:
" + Long.toString(totalTime) + " ms\n");
                    } // end run method
                } // end anonymous implementation of the Runnable class
            ); // end invokeLater method
        } // end method displayTotalTime

        /**
         * This method writes a file to the designated path from bytes
received within the
         * input stream based on the user selected path and filename
         *
         * @throws ClassNotFoundException
         * @throws IOException
         */
        public void receiveFile() throws ClassNotFoundException,
IOException{
```

```
            String extension = (String) input.readObject();
            byte[] fileContent = null;
            fileContent = (byte[]) input.readObject();
            Files.write(file.toPath(), fileContent);

            int userSaveFile = receiver.showSaveDialog(new JFrame());

            if (userSaveFile == JFileChooser.APPROVE_OPTION) {
                File directory = receiver.getCurrentDirectory();
                String filename = receiver.getSelectedFile().getName();
                File fileToSave = new File(directory + "\\" + filename +
extension);
                System.out.println(receiver.getCurrentDirectory());
                System.out.println(filename);
                Files.write(fileToSave.toPath(), fileContent);
            } // end if

        } // end method receiveFile

        /**
         * This method sends the file from the input parameter and its
extension by converting it to bytes
         * and sending it out through the output stream.
         * @param file
         */
        private void sendFile(File file) {
            byte[] fileContent;
            String fileExtension = getFileExtension(file);

            try {
                fileContent = Files.readAllBytes(file.toPath());
                output.writeObject(fileExtension);
                output.flush();
                output.writeObject(fileContent);
                output.flush();
            } catch (IOException e) {
                e.printStackTrace();
            } // end try block
        } // end method sendFile

        /**
         * This method opens a new JFileChooser JFrame that allows a user to
send a desired file to send
         * to the server.
         */
        public void setupFileChooser() {
            int result = this.chooser.showOpenDialog(new JFrame());
            if (result == JFileChooser.APPROVE_OPTION) {
                File selectedFile = this.chooser.getSelectedFile();
                sendFile(selectedFile);
            } // end if

        } // end method setupFileChooser

        /**
         * This method retrieves the file extension of the file in the input
parameter
```

```
         * @param file
         * @return
         */
        public String getFileExtension(File file) {
            String fileExtension = file.getName();
            int lastIndexOf = fileExtension.lastIndexOf('.');
            return fileExtension.substring(lastIndexOf);
        } // end method getFileExtension

    } // end class TextingClient
```

Client Implementation

```
/**
 * Test class implementation for the client class
 * @author nferry@email.sc.edu
 */
public class TextingClientTest
{
    /**
     * Main method for the test class
     * @param args
     */
    public static void main(String[] args)
    {
        // Create a TextingClientJF object
        TextingClientJF application;

        /*
         * Create a new client on the local host address if the args length
is 0, else
         * create the client on the host address stored at the 0 index of
args
         */
        if (args.length == 0)
            application = new TextingClientJF( "127.0.0.1");
        else
            application = new TextingClientJF( args[0]);

        // Call the runClient method contained within the newly created
application object
        application.runClient();


    } // end main method

} // end TextingClientTest class
```