COMP2444: Networks and Scalable Architectures

Coursework 2: Using Sockets to Create a Client-Server Application

By Melissa Liau

Object Oriented Design and Threading

I have separated my program into four different files:

`ImageServer.java`, `ClientHandler.java`, `ImageClient.java`, `ImageClientUI.java`

`ImageServer.java`

The server initialises a thread pool, allowing multiple threads (each running an instance of `ClientHandler.java`) to run in the one server. This allows multiple clients to connect to the server, and creates the impression of "multitasking", as the server handles queries from each client separately and individually.

`ClientHandler.java`

`ClientHandler.java` defines a class which is initiated as an object in `ImageServer.java`. It is important for the `ClientHandler.java` to be in a different file than `ImageServer.java` because each instance of the ClientHandler class may have different characteristics. Due to the linear nature of this class, threads were not needed, because each client handler deals with only one client.

`ImageClient.java`

`ImageClient.java` is designed to be able to be run from another machine if paired with the UI. Because of this, and because it has a whole different functionality, it is its own class written in a different file, despite being only compatible with the server file. Only one client is needed per execution of the program, connecting to only one server, so threads were not needed for this.

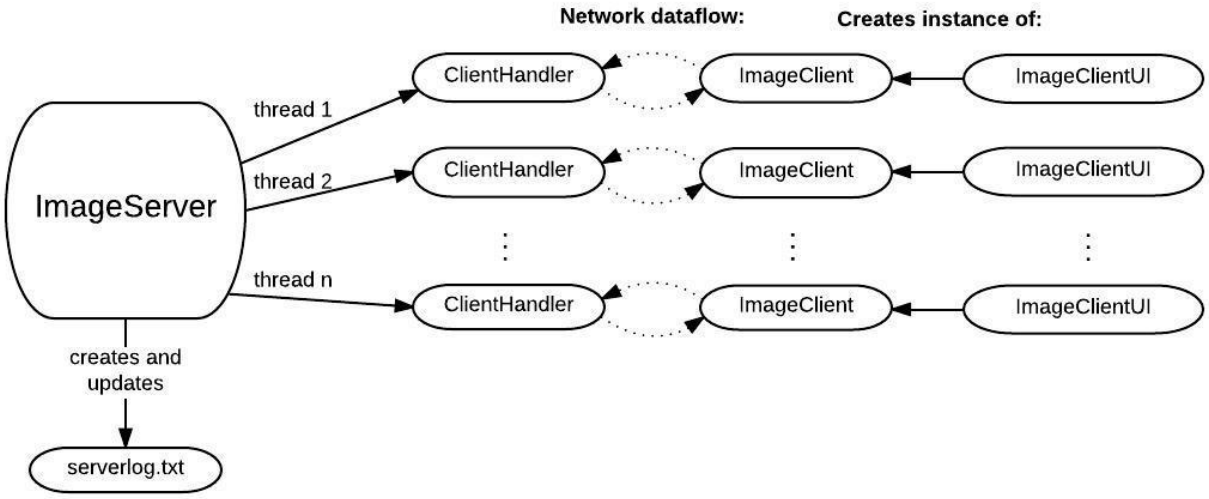
`ImageClientUI.java`

User interfaces are object oriented, therefore the code of the UI had to be object oriented. An example of where an object oriented approach was necessary is where each button needs a listener to run in parallel, while the rest of the UI deals with user interaction in real time. Only one UI is needed per execution of the program, so threads were not needed for this.

`Other Plans`

I was planning on implementing a means of timing out the client attempting to connect to the server after x amount of time. This may have required a thread to run alongside the client attempting to connect, which would tell the other thread to stop after the elapsed time. This is necessary for when the thread pool of the server is full, because the impending connection is added to a queue, and simply waits, making the application hang and blocking UI functions. There may be a method to time out a queuing thread server-side, but I did not have enough time to understand and implement this.

Diagram to show interaction of application parts



Program Structure and Features

ImageServer.java		
Class: `ImageServer`		Creates a server for `ImageClient.java` to connect to. Works with `ClientHandler.java` to handle all server to/from client functions. Has a connection limit of 10 clients and requires `ClientHandler.java`.
	`main(String[] args)`	If run from terminal, automatically opens server on port 5000, with thread limit of 10.
	`ImageServer(int port, int upperThreadLimit)`	Class initialiser. Opens a new socket to start the server on and if successful, starts a thread to run the server on.
Subclass: `ServerConnectLoop`		Creates a thread pool with the thread limit as the global variable `threadLimit`. Runs the server.
	`run()`	Listens for connections. Once a connection is accepted, runs connection on a new thread and prints out to terminal that a client has connected.

ClientHandler.java		
Class: `ClientHandler`		Client handler – a protocol to handle queries between the client and the server. Designed to work with ImageClient.java. Not a program on its own – called by ImageServer.java. Requires folder structure: • `/ImageServer.j`ava • `/images/[various images]`
	`ClientHandler(`*`Socket client`*`)`	Class initialiser. Initialises the ClientHandler object.
	`send(`*`String message`*`)`	Sends message through output stream to client and prints message to terminal.
	`writeToLog(`*`String functionRequested, String filename`*`)`	Appends to serverlog.txt in the format: `date : time : ipAddress : req`uest
	`handleQueries()`	Reads input from client which should be one of the following formats: • `requesting::[filename]` • `sending::[filename]` • `requestlist::` Calls appropriate method if input is valid.
	`readImagesFromFile()`	Updates server image file list and prints out filenames to terminal.
	`uploadToUser(`*`String fileName`*`)`	Sends file of name specified in argument to client. Prints out actions to terminal during process.
	`downloadFromUser( String fileName`*`)`	Downloads file of name specified in argument from client. Prints out actions to terminal during process.
	`sendImageList()`	Sends list of server images to client.
	`run()`	Runs the client handler and listens for messages from client. If message stream is broken, stops reading, breaking the thread.

ImageClient.java		
Class: `ImageClient`		A client which connects to ImageServer.java. Downloads and uploads images from/to the server.
	`main(String[] args)`	If run from terminal, automatically connects to localhost for bus testing, then runs `queryHandler()` to read terminal inputs.
	`ImageClient(String host, int port)`	Initiliases ImageClient class. Opens a new socket and retrieves information from the starter information from the server once connected. Calls `readImagesFromFile()` to update image list
	`requestImageList()`	Requests the list of images from the server and prints to terminal the list of images as well as assigns variable `serverImageList` to list.
	`readImagesFromFile()`	Reads the list of images in the /images folder of the client and assigns variable `imageList` to list before printing out list of images to terminal.
	`queryHandler()`	Read input from terminal which should be one of the following: • `requesting::[filename]` • `sending::[filename]` • `requestlist::` Calls appropriate method if input is valid. Does not send query to server if e.g. file does not exist.
	`sendToServer(String fileName)`	Sends file of the name specified as argument to server. Prints out actions to terminal during process.
	`downloadFromServer( String fileName)`	Downloads file of the name specified as argument to server. Prints out actions to terminal during process.

ImageClientUI.java		
Class: `ImageClientUI`		Code to display the `java.swing` interface for a simple image transfer client between the client and the server. Requires folder structure: • `/ImageClient.java` • `/images/[various images]`
	`main(String[] args)`	Runs the user interface.
	`ImageClientUI()`	Initialises the setup of the UI.
	`createComponents()`	Creates all UI components to be used.
	`createLayout()`	Sets the layout for the UI components created in `createComponents()`.
	`updateStatus(String text)`	Updates the status bar message with the string that passed to this method, and prints to terminal what the string is.
	`updateLists()`	Reads the image lists from the ImageClient object. Updates the server and user image lists, and calls `updateStatus()` to show the user what the program is doing in the status bar.
	`connectsToServer( String host, String port)`	Creates a new ImageClient object using the host and port read from the corresponding fields. Calls `updateLists()` to update the lists of the user and the server.
Class: `Connect`		Listener for the "Connect" button.
	`actionPerformed( ActionEvent a)`	Calls `connectToServer()` using text in host and port fields.
Class: `Download`		Listener for the "Download" button.
	`actionPerformed( ActionEvent a)`	Gets name of file selected in server images list, and uses ImageClient object's method `downloadFromServer()` to download from server. Updates lists afterwards.
Class: `Upload`		Listener for the "Upload" button.
	`actionPerformed( ActionEvent a)`	Gets name of file selected in own images list, and uses ImageClient object's method `sendToServer()` to send to server. Updates lists afterwards.
Class: `Update`		Listener for the "Update" button.
	`actionPerformed( ActionEvent a)`	Calls `updateLists()` method.

Appendix 1. – ImageServer.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.*;

/**
* Creates a server for ImageClient.java to connect to. Works with
* ClientHandler.java to handle all server to/from client functions.
* Has a connection limit of 10 clients.
* Requires: /ClientHandler.java
*
* @author Melissa Liau
*/

public class ImageServer
{
  private ServerSocket serverSocket = null;
  private Integer threadLimit = null;

    /**
    * Opens a new socket the start the server on and if successful,
    * starts a thread to run the server on.
    **/
  public ImageServer(int port, int upperThreadLimit)
  {
    threadLimit = upperThreadLimit;
    try
    {  serverSocket = new ServerSocket(port);
    }
    catch (IOException e)
    {  e.printStackTrace();
    }

    Thread sclThread = new Thread(new ServerConnectLoop());
    sclThread.start();
  }

    /**
    * Creates a thread pool with the thread limit as the value of
    * threadLimit. Runs the server.
    **/
  public class ServerConnectLoop implements Runnable
  {
    ThreadPoolExecutor pool = (ThreadPoolExecutor)
Executors.newFixedThreadPool(threadLimit);

    /**
    * Listens for connections. Once a connection is accepted,
    * runs connection on a new thread and prints out to terminal
    * that a client has connected.
    **/
    public void run()
    {
      try
      {  while (true)
        {  Socket clientSocket = serverSocket.accept();
          ClientHandler c = new ClientHandler(clientSocket);
          pool.execute(c);
          System.out.println("[SERVER] Client connected. " + pool.getActiveCount() + "
client(s) connected.");
        }
      }
```

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

    /**
    * If run from terminal, automatically opens server on port 5000,
    * with thread limit of 10.
    **/
  public static void main(String[] args)
  {  ImageServer s = new ImageServer(5000, 10);
  }
}
```

Appendix 2. – ClientHandler.java

```
import java.io.*;
import java.net.*;
import java.nio.file.*;
import java.util.*;
import java.awt.image.*;
import javax.imageio.*;
import java.net.Socket;
import java.text.*;

/**
* Client handler - a protocol to handle queries between the client
* and the server. Designed to work with ImageClient.java.
* Requires: /ImageServer.java
*           /images/[various images]
*
* @author Melissa Liau
*/

public class ClientHandler implements Runnable
{
  private Scanner reader = null;
  private PrintWriter writer = null;
  private PrintWriter logWriter = null;
  public Boolean nseeStatus = false;
  private String message = null;
  public ArrayList<String> imageList = null;
  public File[] fileList = null;
  private String dirPath = null;
  private InetAddress clientAddress = null;
  private Calendar calendar = null;

    /**
    * Initialises the ClientHandler object.
    **/
  public ClientHandler(Socket client)
  {
    // Gets the client's inet address
    clientAddress = client.getInetAddress();
    // Gets the directory the server is run from in order to read
    // image files.
    dirPath = System.getProperty("user.dir");
    try
    {
      // Creates input and output streams with client.
      reader = new Scanner(client.getInputStream());
      writer = new PrintWriter(client.getOutputStream(), true);
      send("\n  Please enter one of the following:");
      send("   * requesting::[filename] ");
      send("   * sending::[filename]");
      send("   * requestlist:: \n");
      send("stop::");
      System.out.println("");
    }
    catch (IOException e)
    {  e.printStackTrace();
    }
    writeToLog("client connected", "");
  }
```

```
  /**
  * Sends message through output stream to client and prints
  * message to terminal.
  **/
  private void send(String message)
  {
    this.writer.println(message);
    System.out.println("[SERVER OUTPUT] " + message);
  }

  /**
  * Appends to serverlog.txt in the format:
  * date : time : ipAddress : request
  **/
  private void writeToLog(String functionRequested, String filename)
  {
    File log = new File("serverlog.txt");

    // If serverlog.txt does not exist, create empty txt file.
    if (!log.exists())
    {
      try
      {  PrintWriter writer = new PrintWriter("serverlog.txt", "UTF-8");
        writer.write("");
        writer.close();
      }
      catch (Exception e)
      {  e.printStackTrace();
      }
    }
    // Create a new Date object and read date and time from it as strings.
    Date currentDateTime = new Date();
    DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
    DateFormat timeFormat = new SimpleDateFormat("HH:mm:ss");
    String request = functionRequested + " " + filename;
    String date = (String)dateFormat.format(currentDateTime);
    String time = (String)timeFormat.format(currentDateTime);
    String toLog = date + " : " + time + " : " + this.clientAddress + " : " + request +
"\n";
    // Write to serverlog.txt
    try
    {  Files.write((Paths.get("serverlog.txt")), toLog.getBytes(),
StandardOpenOption.APPEND);
      System.out.println("Logged event.");
    }
    catch (IOException e)
    {  e.printStackTrace();
    }
  }

  /**
  * Reads input from client, which should be one of the following
  * formats:
  * requesting::[filename]
  * sending::[filename]
  * requestlist::
  * Calls appropriate method if input is valid.
  **/
  private void handleQueries()
  {
    String fileName = "";
    // Retrieve filename from terminal query.
    int separatorLocation = this.message.lastIndexOf("::");
    if ((separatorLocation > 0) && this.message.contains("::"))
    {  if (!this.message.endsWith("::"))
      {
        fileName = this.message.substring(separatorLocation+2);
      }
```

```
    }
    // Reacts accordingly.
    if (this.message.startsWith("requesting::"))
    {  uploadToUser(fileName);
    }
    else if (this.message.startsWith("sending::"))
    {  downloadFromUser(fileName);
    }
    else if (this.message.startsWith("requestlist::"))
    {  sendImageList();
    }
    else
    {  send("Invalid query: " + this.message);
    }
  }

  /**
  * Updates server image file list and prints out filenames to
  * terminal.
  **/
  private void readImagesFromFile()
  {
    System.out.println("Updating own list...");
    File dir = new File(this.dirPath + "/images");
    this.fileList = dir.listFiles();
    this.imageList = new ArrayList<String>(this.fileList.length);
    for (File file : this.fileList)
    {  if (file.isFile())
      {  System.out.println(" adding " + file.getName());
        // adds each filename to imageList
        this.imageList.add(file.getName());
      }
    }
    System.out.println("");
  }

  /**
  * Sends file of name fileName to client. Prints out what is
  * being done to terminal during process.
  **/
  public void uploadToUser(String fileName)
  {
    try
    {
      int dotLocation = fileName.lastIndexOf(".");
      String fileType = "";
      if (dotLocation > 0)
      {  fileType = fileName.substring(dotLocation+1);
      }
      System.out.println("Detected file type: " + fileType);
      System.out.println("Reading file: /images/" + fileName);
      BufferedImage img = ImageIO.read(new File(this.dirPath + "/images/" + fileName));
      System.out.println("Converting image to byte array output stream.");
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ImageIO.write(img, fileType, baos);
      baos.flush();
      byte[] bytes = baos.toByteArray();
      baos.close();
      System.out.println("Byte array of length " + bytes.length + " created.");

      System.out.println("Opening new socket to connect to user.");
      Socket soc = new Socket(this.clientAddress, 4000);
      System.out.println("Opening streams with user.");
      OutputStream outStream = soc.getOutputStream();
      DataOutputStream dos = new DataOutputStream(outStream);

      System.out.println("Writing to user stream.");
      dos.writeInt(bytes.length);
```

```
      dos.write(bytes, 0, bytes.length);
      System.out.println("Closing streams/socket.");
      dos.close();
      outStream.close();
      soc.close();
      writeToLog("sent image", fileName);
    }

    catch (Exception e)
    {  e.printStackTrace();
    }
  }

  /**
  * Downloads file of name fileName from client. Prints out what is
  * being done to terminal during process.
  **/
  public void downloadFromUser(String fileName)
  {
    try
    {
      System.out.println("Opening new socket for client to connect...");
      ServerSocket server = new ServerSocket(4000);
      Socket socket = server.accept();
      System.out.println(" Accepted connection with client.\nRetrieving input
stream...");
      InputStream inStream = socket.getInputStream();
      DataInputStream dis = new DataInputStream(inStream);

      int dataLength = dis.readInt();
      byte[] data = new byte[dataLength];
      dis.readFully(data);
      dis.close();
      inStream.close();
      System.out.println(" Finished receiving input stream.\nConverting to file...");

      InputStream bais = new ByteArrayInputStream(data);
      String filePath = this.dirPath + "/images/" + fileName;
      OutputStream toFile = new FileOutputStream(filePath);
      byte[] buffer = new byte[1024];
      int bytesRead;
      while ((bytesRead = bais.read(buffer)) != -1)
      {  System.out.println(" Bytes read of length: " + bytesRead);
        toFile.write(buffer, 0, bytesRead);
      }
      bais.close();
      toFile.flush();
      toFile.close();
      server.close();
      System.out.println(" ...Finished!\n");
      writeToLog("received image", fileName);

      readImagesFromFile();
    }

    catch (Exception e)
    {  e.printStackTrace();
    }
  }

  /**
  * Sends list of images on server to client.
  **/
  public void sendImageList()
  {
    readImagesFromFile();
    send(Integer.toString(imageList.size()));
    for (int i = 0; i < this.imageList.size(); i++)
```

```
    {  send(imageList.get(i));
    }
    send("stop::");
  }

  /**
  * Runs the client handler and listens for messages from client. If
  * message stream is broken, stops reading, breaking the thread.
  **/
  public void run()
  {
    while (true)
    {  try
      {
        this.message = reader.nextLine();
        while (this.message != null)
        {
          System.out.println("Server read: "+ this.message + "\n");
          handleQueries();
          System.out.println("Listening for message...");
          this.message = reader.nextLine();
        }
      }
      catch (NoSuchElementException nsee)
      {  this.nseeStatus = true;
        reader.close();
        writer.close();
        System.out.println("Client disconnected.");
        writeToLog("client disconnected", "");
        break;
      }
    }
  }
}
```

Appendix 3. – ImageClient.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.image.*;
import javax.imageio.*;
import java.net.Socket;

/**
* A client which connects to ImageServer.java. Downloads and uploads
* images from/to the server.
* Requires arguments: host, port
*
* @author Melissa Liau
*/

public class ImageClient
{
  private Scanner socketIn = null;
  private PrintWriter socketOut = null;
  private Scanner keyboardIn = null;
  private String messageIn = null;
  public ArrayList<String> serverImageList = null;
  public ArrayList<String> imageList = null;
  public File[] fileList = null;
  private String dirPath = null;
  private String serverHost = null;

  /**
  * Initialises ImageClient class. Opens a new socket and retrieves
  * information from the starter information from the server once
  * connected. Calls readImagesFromFile() to update image list.
  **/
  public ImageClient(String host, int port)
  {
    serverHost = host;
    dirPath = System.getProperty("user.dir");

    try
    {
          Socket socket = new Socket(host, port);
      socketIn = new Scanner(socket.getInputStream());
      socketOut = new PrintWriter(socket.getOutputStream(), true);
      keyboardIn = new Scanner(System.in);
      String intro = socketIn.nextLine();
      while (true)
      {  System.out.println(intro);
        intro = socketIn.nextLine();
        if (intro.equals("stop::"))
        {  break;
        }
      }
      requestImageList();
      readImagesFromFile();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
```

```
  }

  /**
  * Requests the list of images from the server and prints to
  * terminal the list of images as well as assigns variable
  * serverImageList to list.
  **/
  public void requestImageList()
  {
    socketOut.println("requestlist::");
    int length = Integer.parseInt(socketIn.nextLine());
    this.serverImageList = new ArrayList<String>(length);
    this.messageIn = socketIn.nextLine();
    System.out.println("  Available images to download:");
    while (!this.messageIn.equals("stop::"))
      {
        System.out.println("    - " + this.messageIn);
        this.serverImageList.add(this.messageIn);
        this.messageIn = socketIn.nextLine();
      }
    System.out.println("");
  }

  /**
  * Reads the list of images in the /images folder of the client
  * and assigns variable imageList to list, before printing out
  * list of images to terminal.
  **/
  private void readImagesFromFile()
  {
    File dir = new File(this.dirPath + "/images");
    this.fileList = dir.listFiles();
    this.imageList = new ArrayList<String>(this.fileList.length);
    System.out.println("Finding files in /images directory...");
    boolean found = false;
    for (File file : this.fileList)
    {  if (file.isFile())
      {  System.out.println(" Found: " + file.getName());
        found = true;
        this.imageList.add(file.getName());
      }
    }
    if (!found)
    {
      System.out.println("No files found!");
    }
    System.out.println("");
  }

  /**
  * Reads input from terminal, which should be one of the following:
  * requesting::[filename], sending::[filename], requestlist::
  * Calls appropriate method if input is valid. Does not send
  * query to server if e.g. file does not exist.
  **/
  private void queryHandler()
  {
    String messageOut;
    while ((messageOut = keyboardIn.nextLine()) != null)
    {
      String fileName = "";
```

```
      boolean error = true;
      if (messageOut.contains("::"))
      {
        int separatorLocation = messageOut.lastIndexOf("::");
        if (!messageOut.endsWith("::"))
        {
          fileName = messageOut.substring(separatorLocation+2);
        }

        if (messageOut.equals("requestlist::"))
        {
          error = false;
          System.out.println("[CLIENT OUTPUT] " + messageOut + "\n");
          System.out.println("  Available images to download:");
          requestImageList();
        }

        if (messageOut.startsWith("requesting::") &&
serverImageList.contains(fileName))
        {
          error = false;
          System.out.println("[CLIENT OUTPUT] " + messageOut);
          downloadFromServer(fileName);
        }

        if (messageOut.startsWith("sending::") && imageList.contains(fileName))
        {
          error = false;
          System.out.println("Not implemented yet");
          sendToServer(fileName);
        }
      }

      if (error)
      {
        System.out.println("\n  Invalid requests. Please enter one of the
following:");
        System.out.println("    * requesting::[filename] ");
        System.out.println("    * sending::[filename]");
        System.out.println("    * requestlist::");
      }
    }
  }

  /**
  * Sends file of name fileName to server. Prints out what is
  * being done to terminal during process.
  **/
  public void sendToServer(String fileName)
  {
    socketOut.println("sending::" + fileName);
    try
    {
      int dot = fileName.lastIndexOf(".");
      String fileType = "";
      if (dot > 0)
      {  fileType = fileName.substring(dot+1);
      }
      System.out.println("Detected file type: " + fileType);
      System.out.println("Reading file: /images/" + fileName);
```

```
      BufferedImage img = ImageIO.read(new File(this.dirPath + "/images/" +
fileName));
      System.out.println("Converting image to byte array output stream.");
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ImageIO.write(img, fileType, baos);
      baos.flush();
      byte[] bytes = baos.toByteArray();
      baos.close();
      System.out.println("Byte array of length " + bytes.length + " created.");

      System.out.println("Opening new socket to connect to server.");
      Socket soc = new Socket(this.serverHost, 4000);
      System.out.println("Opening streams with server.");
      OutputStream outStream = soc.getOutputStream();
      DataOutputStream dos = new DataOutputStream(outStream);

      System.out.println("Writing to server stream.");
      dos.writeInt(bytes.length);
      dos.write(bytes, 0, bytes.length);
      System.out.println("Closing streams/socket.");
      dos.close();
      outStream.close();
      soc.close();
    }

    catch (Exception e)
    { System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
    }
  }

  /**
  * Downloads file of name fileName from server. Prints out what is
  * being done to terminal during process.
  **/
  public void downloadFromServer(String fileName)
  {
    socketOut.println("requesting::" + fileName);
    try
    {
      System.out.println("Opening new socket for server to connect...");
      ServerSocket server = new ServerSocket(4000);
      Socket socket = server.accept();
      System.out.println(" Accepted connection with server.\nRetrieving input
stream...");
      InputStream inStream = socket.getInputStream();
      DataInputStream dis = new DataInputStream(inStream);

      int dataLength = dis.readInt();
      byte[] data = new byte[dataLength];
      dis.readFully(data);
      dis.close();
      inStream.close();
      System.out.println(" Finished receiving input stream.\nConverting to
file...");

      InputStream bais = new ByteArrayInputStream(data);
      String filePath = this.dirPath + "/images/" + fileName;
      OutputStream toFile = new FileOutputStream(filePath);
      byte[] buffer = new byte[1024];
      int bytesRead = 0;
```

```
      while ((bytesRead = bais.read(buffer)) != -1)
      { System.out.println(" Bytes read of length: " + bytesRead);
        toFile.write(buffer, 0, bytesRead);
      }
      bais.close();
      toFile.flush();
      toFile.close();
      server.close();
      System.out.println(" ...Finished!\n");

      readImagesFromFile();
    }

    catch (Exception e)
    { System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
    }
  }

  /**
  * If run from terminal, automatically connects to localhost
  * for bug testing, then runs queryHandler() to read terminal
  * inputs.
  **/
  public static void main(String[] args)
  {
    String host = "127.0.0.1";          //args[0]
    int port = 5000;
    ImageClient ic = new ImageClient(host, port);
    ic.queryHandler();
  }
}
```

Appendix 4. - ImageClientUI.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

/**
* Code to display the java.swing interface for a simple image
* transfer client between the client and the server.
* Requires: /ImageClient.java
*           /images/[various images]
*
* @author Melissa Liau
*/

public class ImageClientUI extends JFrame
{
  private static final long serialVersionUID = 1L;

  // Global java swing variables.
  private JButton connectButton = null;
  private JButton downloadButton = null;
  private JButton uploadButton = null;
  private JButton updateButton = null;
  private JLabel hostLabel = null;
  private JLabel portLabel = null;
  private JLabel ownLabel = null;
  private JLabel serverLabel = null;
  private JLabel status = null;
  private JList<String> ownImageList = null;
  private JList<String> serverImageList = null;
  private JPanel mainPanel = null;
  private JPanel statusBar = null;
  private JScrollPane serverListScrollPane = null;
  private JScrollPane ownListScrollPane = null;
  private JSplitPane imagesSplitPane = null;
  private JTextField hostField = null;
  private JTextField portField = null;
  private ImageClient ic = null;
  private String[] ownList = null;
  private String[] serverList = null;

  /**
  * Initialises the setup of the UI.
  **/
  public ImageClientUI()
  {
    setLayout(new BorderLayout());
    setPreferredSize(new Dimension(450, 600));
    setTitle("Image Download and Upload Client");

    createComponents();
    createLayout();
    add(mainPanel);
    add(statusBar, BorderLayout.SOUTH);

    pack();
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
  }
```

```
/**
* Creates all UI components to be used.
**/
private void createComponents()
{
  this.connectButton = new JButton("Connect");
  this.connectButton.addActionListener(new Connect());
  this.downloadButton = new JButton("Download");
  this.downloadButton.addActionListener(new Download());
  this.downloadButton.setEnabled(false);
  this.uploadButton = new JButton("Upload");
  this.uploadButton.addActionListener(new Upload());
  this.uploadButton.setEnabled(false);
  this.updateButton = new JButton("Update Lists");
  this.updateButton.addActionListener(new Update());
  this.updateButton.setEnabled(false);

  this.hostLabel = new JLabel("Host: ");
  this.hostField = new JTextField("127.0.0.1", 20);
  this.portLabel = new JLabel("Port: ");
  this.portField = new JTextField("5000", 15);

  this.ownLabel = new JLabel("My image list:");
  this.ownImageList = new JList<String>();
  this.ownListScrollPane = new JScrollPane();
  this.ownListScrollPane.getViewport().add(ownImageList);
  this.ownListScrollPane.setPreferredSize(new Dimension(10000, 9000));

  this.serverLabel = new JLabel("Server image list:");
  this.serverImageList = new JList<String>();
  this.serverListScrollPane = new JScrollPane();
  this.serverListScrollPane.getViewport().add(serverImageList);
  this.serverListScrollPane.setPreferredSize(new Dimension(10000, 900));

  this.status = new JLabel("Ready to connect!");
  this.status.setHorizontalAlignment(SwingConstants.LEFT);

}

/**
* Sets the layout for the UI components created in createComponents().
**/
private void createLayout()
{
  Box connectBox = new Box(BoxLayout.LINE_AXIS);
  connectBox.add(this.hostLabel);
  connectBox.add(this.hostField);
  connectBox.add(Box.createHorizontalStrut(10));
  connectBox.add(this.portLabel);
  connectBox.add(this.portField);
  connectBox.add(Box.createHorizontalStrut(10));
  connectBox.add(this.connectButton);
  connectBox.add(Box.createHorizontalStrut(10));
  connectBox.add(this.updateButton);

  Box ownListBox = new Box(BoxLayout.PAGE_AXIS);
  ownListBox.add(this.ownLabel);
  ownListBox.add(Box.createVerticalStrut(5));
  ownListBox.add(this.ownListScrollPane);
  ownListBox.add(Box.createVerticalStrut(5));
```

```
    ownListBox.add(this.uploadButton);
    ownListBox.add(Box.createVerticalStrut(5));

    Box serverListBox = new Box(BoxLayout.PAGE_AXIS);
    serverListBox.add(this.serverLabel);
    serverListBox.add(Box.createVerticalStrut(5));
    serverListBox.add(this.serverListScrollPane);
    serverListBox.add(Box.createVerticalStrut(5));
    serverListBox.add(this.downloadButton);
    serverListBox.add(Box.createVerticalStrut(5));

    Box imageListBox = new Box(BoxLayout.LINE_AXIS);
    imageListBox.add(ownListBox);
    imageListBox.add(Box.createHorizontalStrut(5));
    imageListBox.add(new JSeparator(SwingConstants.VERTICAL));
    imageListBox.add(Box.createHorizontalStrut(5));
    imageListBox.add(serverListBox);

    this.mainPanel = new JPanel();

    this.mainPanel.setBorder(BorderFactory.createEmptyBorder( 5,5,5,5));
    this.mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.PAGE_AXIS));
    this.mainPanel.add(connectBox);
    this.mainPanel.add(Box.createVerticalStrut(5));
    this.mainPanel.add(new JSeparator(SwingConstants.HORIZONTAL));
    this.mainPanel.add(Box.createVerticalStrut(5));
    this.mainPanel.add(imageListBox);

    this.statusBar = new JPanel();
    this.statusBar.setBorder(new BevelBorder(BevelBorder.LOWERED));
    this.statusBar.setPreferredSize(new Dimension(this.getWidth(), 25));
    this.statusBar.setLayout(new BoxLayout(statusBar, BoxLayout.LINE_AXIS));
    this.statusBar.add(this.status);
  }

  /**
  * Updates the status bar message with the string that passed to this method,
  * and prints to terminal what the string is.
  **/
  private void updateStatus(String statusMessage)
  {
    System.out.println("[CLIENT STATUS] " + statusMessage);
    this.status.setText(statusMessage);
  }

  /**
  * Reads the image lists from the ImageClient object.
  * Updates the server and user image lists, and calls updateStatus() to show
  * the user what the program is doing in the status bar.
  **/
  private void updateLists()
  {
    this.ic.requestImageList();
    updateStatus("Retrieving list of own images.");
    this.ownList = this.ic.imageList.toArray(new String[ic.imageList.size()-1]);
    this.ownImageList = new JList<String>(this.ownList);
    updateStatus("List compiled! Retrieving server image list.");
    this.serverList = this.ic.serverImageList.toArray(new
String[ic.imageList.size()-1]);
    this.serverImageList = new JList<String>(this.serverList);
    updateStatus("Finished reading from server.");
```

```
    this.ownImageList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    this.ownImageList.setSelectedIndex(0);
    this.ownListScrollPane.getViewport().add(this.ownImageList);

    this.serverImageList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    this.serverImageList.setSelectedIndex(0);
    this.serverListScrollPane.getViewport().add(this.serverImageList);
  }


  /**
  * Creates a new ImageClient object using the host and port read from the
corresponding
  * fields. Calls updateLists() to update the lists of the user and the server.
  **/
  private void connectToServer(String host, String port)
  {
    try
    {   updateStatus("Creating ImageClient object.");
      this.ic = new ImageClient(host, Integer.parseInt(port));
      updateLists();
      uploadButton.setEnabled(true);
      downloadButton.setEnabled(true);
      updateButton.setEnabled(true);
      updateStatus("Connected to server and updated image lists.");
    }

    catch (Exception e)
    {   System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
      updateStatus("Error connecting to server! Is server running?");
    }
  }

  /**
  * Listener for the "Connect" button.
  **/
  private class Connect implements ActionListener
  {
    /**
    * Calls connectToServer() using text in host and port fields.
    **/
     public void actionPerformed(ActionEvent a)
     {
      connectToServer(hostField.getText(), portField.getText());
     }
  }

  /**
  * Listener for the "Download" button.
  **/
  private class Download implements ActionListener
  {
    /**
    * Gets name of file selected in server images list, and uses ImageClient
object's
    * method downloadFromServer() to download from server. Updates lists
afterwards.
    **/
    public void actionPerformed(ActionEvent a)
```

```
    {
      String fileName = serverList[serverImageList.getSelectedIndex()];
      ic.downloadFromServer(fileName);
      updateLists();
      updateStatus("Downloaded " + fileName + " and updated lists.");
    }
  }

  /**
  * Listener for the "Upload" button.
  **/
  private class Upload implements ActionListener
  {
    /**
    * Gets name of file selected in own images list, and uses ImageClient object's
    * method sendToServer() to send to server. Updates lists afterwards.
    **/
    public void actionPerformed(ActionEvent a)
    {
      String fileName = ownList[ownImageList.getSelectedIndex()];
      ic.sendToServer(fileName);
      updateLists();
      updateStatus("Uploaded " + fileName + " and updated lists.");
    }
  }

  /**
  * Listener for the "Update Lists" button.
  **/
  private class Update implements ActionListener
  {
    /**
    * Calls updateLists() method.
    **/
    public void actionPerformed(ActionEvent a)
    {
      updateLists();
      updateStatus("Updated lists.");
    }
  }

  /**
  * Main of the ImageClientUI class; runs the user interface.
  **/
  public static void main(String[] args)
  {
    SwingUtilities.invokeLater(new Runnable()
    {
      @Override
      public void run()
      {
        ImageClientUI icui = new ImageClientUI();
        icui.setVisible(true);
      }
    });
  }
}
```