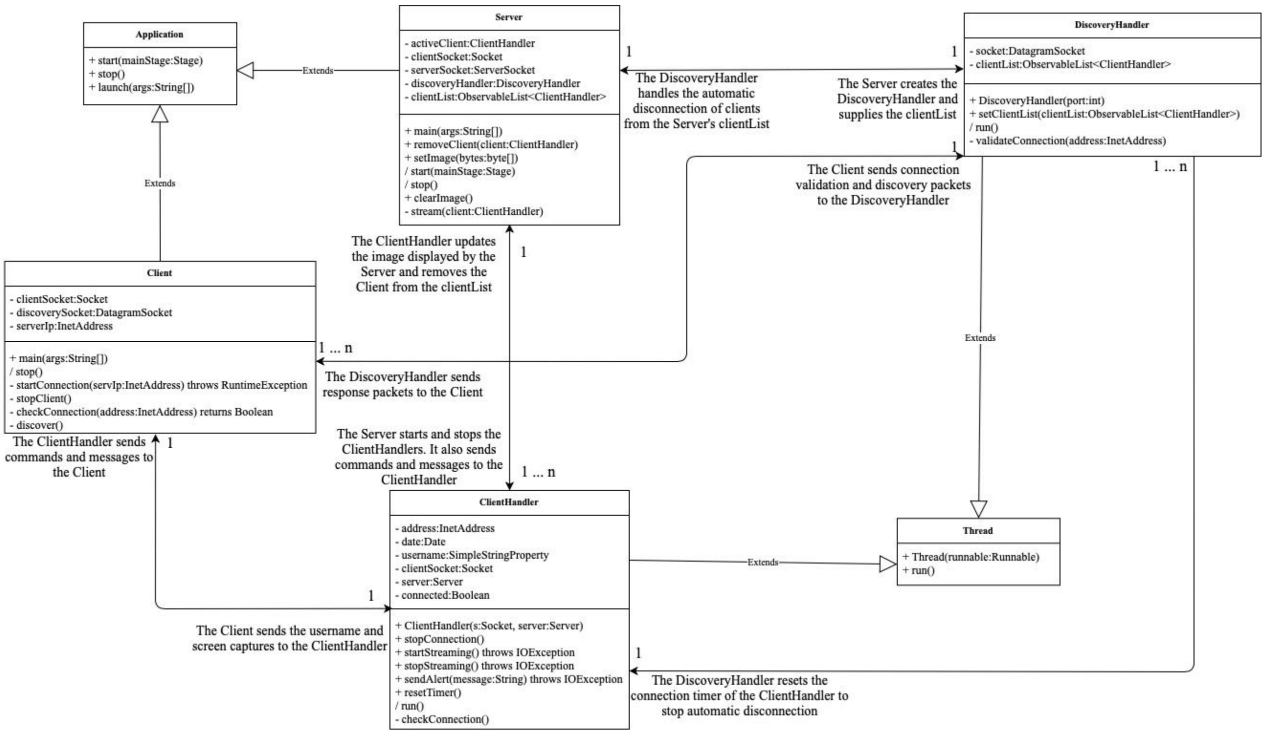
Criterion C: Development

Words: 852

Table of Contents

Object Oriented Programming

The client program only consists of the Client object, while the server program consists of the Server, DiscoveryHandler, and ClientHandler objects. Below is a UML diagram explaining their relationships.



Object oriented programming allows me to compartmentalize the code. The DiscoveryHandler manages the DatagramSocket opened by the server. The ClientHandler objects handles the connection between the server and an individual client. This allows the Server to easily manage all clients by simply iterating over all clients and calling any necessary methods. All objects run on separate threads to avoid interfering with one another. Inheritance allows me to use pre-existing classes or interfaces to define the new objects and customize their functionality.

Data Structures

The Client program utilizes queues for execution. This allows a thread to call a method on another thread while avoiding the creation of new threads to minimize the chance of a thread error occurring. Below is sample code from the Client object that utilizes this.

```java

    /**
     * Constantly executes any runnables in the readRequests queue
     */
    private static Thread readThread =
    new Thread(() -> {

        while(true){

            try{
                readRequests.take().run();
            } catch(InterruptedException e){
                e.printStackTrace();
            }

        }

    });

    /**
     * Constantly executes any runnables in the connectionRequests queue
     */
    private static Thread connectionThread =
    new Thread(() -> {

        while(true){

            try{
                connectionRequests.take().run();
            } catch(InterruptedException e){
                e.printStackTrace();
            }

        }

    });

```

Runnables are taken from a LinkedBlockingQueue and run, thus the thread pauses until a runnable is added.

```java
        try {
            System.out.println("Connecting...");
            clientSocket = new Socket(servIp, port);
            System.out.println("Connected to " + clientSocket.getInetAddress());
            connected = true;
            out = clientSocket.getOutputStream();
            in = clientSocket.getInputStream();
            jpgWriter = ImageIO.getImageWritersByFormatName("jpg").next();
            jpgWriteParam = jpgWriter.getDefaultWriteParam();
            jpgWriteParam.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
            jpgWriteParam.setCompressionQuality(compressionQuality);
            byte[] username = System.getProperty("user.name").getBytes();
            writeLength(out, username.length);
            out.write(username);

            readRequests.add(() -> {
                readFromConnection();
            });

            while(connected){

                if(streaming){
                    sendScreen();
                }

                Thread.sleep(100);
            }
```

A runnable containing a method is added to the LinkedBlockingQueue to be run.

Utilizing a LinkedBlockingQueue avoids creating threads, ensuring thread safety. Moreover, the reference to the executing thread is retained, so there is no chance of resource leakage.

The Server object utilizes a dynamic array to handle and display ClientHandlers. This allows ClientHandlers to be easily iterated over and modified. Furthermore, it is a JavaFX ObservableList, allowing the client list in the GUI to be updated by adding a change listener to the GUI. On the following page is sample code that utilizes this.

```java

private ObservableList<ClientHandler> clientList = FXCollections.observableArrayList();
private TableView<ClientHandler> UIclients = new TableView<ClientHandler>(clientList);
private TableColumn<ClientHandler, String> UIconnected = new TableColumn<ClientHandler, String>("Connected Computers");

```

Adds listener to GUI objects

```java

    /**
     * Handles the creation of the layout and logic of the GUI
     */
    private void createScene(){
        rootNode.setCenter(streamView);
        rootNode.setLeft(menu);
        rootNode.setBottom(msgBox);
        UIconnected.setCellValueFactory(new PropertyValueFactory<ClientHandler, String>("username"));
        UIconnected.prefWidthProperty().bind(UIclients.prefWidthProperty());
```

Links the listener to the username property of the ClientHandlers in clientList

Setting up this ObservableList simplifies the code as there is no need for the ClientHandler to call a method in the Server whenever its username updates; instead, the property only needs to be modified for the name to automatically update in the Server GUI.

A list also simplifies searching through the array. In the following code from the Server object, the list can quickly be searched through.

```java

    /**
     * Requests a ClientHandler to stream, disabling all others
     *
     * @param chosenClient The ClientHandler that should be streamed
     */
    private void stream(ClientHandler chosenClient) throws IOException {

        for(ClientHandler client : clientList){

            if(client != chosenClient){
                client.stopStreaming();
            } else if(client == chosenClient){
                client.startStreaming();
                changeText(streamControlBtn, "STOP");
                streaming = true;
            } else{
                throw new RuntimeException("This should NEVER happen.\nClient is null.");
            }

        }

    }

```

Repeats loop for each ClientHandler in clientList

By taking advantage of the features of these various data structures, the code is a lot easier to follow and requires fewer methods to implement.

Recursion

The Client program never ceases execution and must use recursion to automatically restart whenever it disconnects from the server. Below is the portion of the Client object that uses this.

```

/**
 * Recursive method that constantly attempts to discover and check the connection to the server
 */
private static void discover(){

    try{
        sendDiscoveryPackets();
        System.out.println("Now waiting for reply");
        receivePacket();

        if(connected){
            connectionRequests.add(() -> {
                attemptConnection();
            });

            while(connected){
                connected = checkConnection(discoverReceivePacket.getAddress());
                Thread.sleep(checkDelay);
            }

        }

        System.out.println("Restarting discovery");
        discover();
    } catch(SocketException e){
        System.out.println("DatagramSocket failed");
        e.printStackTrace();
    } catch(InterruptedException Ie){
        System.out.println("Packet thread interrupted");
    }

}

```

To avoid resource leakage, no new objects are declared

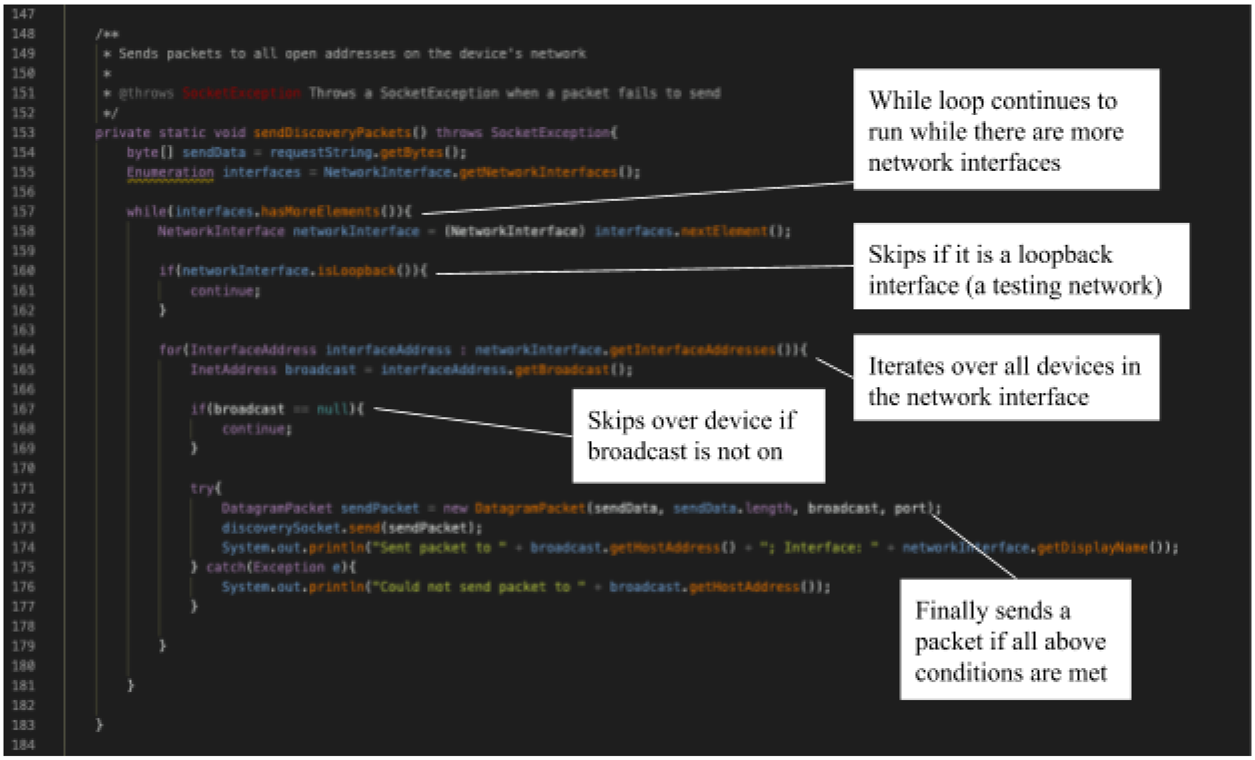
Loops while the program is connected to the server

At the end, this method is re-called, effectively restarting the program

Through recursion, the Client program never stops running, ensuring the Client readily connects to the server. Avoiding the declaration of new objects or variables in the recursive method ensures there is no slow-down on the student’s computer due to resource leakage as the program runs.

Complex Loops

Many components of the Server and Client programs must iterate over enumerations or lists, so it requires complex loops to execute properly. These include nested if statements, for loops, while loops, and other statements. This can be seen in the following code from the Client object.



By utilizing complex loops, I can sort through any enumeration or array and run code only when certain conditions are met. In the sample code above, packets are sent to devices in all non-loopback interfaces that have a broadcast on.

This is also implemented to search through an array. The following example comes from the DiscoveryHandler object.

```java

    /**
     * Resets the timer of a certain client
     *
     * @param address The InetAddress of the client who sent a heartbeat packet
     */
    private void validateConnection(InetAddress address){
        ClientHandler currentClient;

        for(int i = clientList.size() - 1; i >= 0; i--){
            currentClient = clientList.get(i);

            if(currentClient != null && currentClient.getAddress().equals(address)){
                currentClient.resetTimer();
                return;
            }

        }

    }

```

Resets heartbeat timer only if the address matches the address the heartbeat came from

Iterates backwards as it should maintain a connection with the most recently connected clients, avoiding duplication of client connections.

This implementation of the for loop and a nested if statement allows me to customize how the program searches through the array. In this case it iterates backwards instead of forwards to avoid validating the connection of dead sockets.

Error Handling

Due to the program’s reliance on networking, there are many places where errors may occur. However, many of these errors are also discardable, as they are the result of streams being closed after the socket disconnects. Very specific errors are thrown in very specific cases, such as in the following code from the Client object that handles a SocketTimeoutException.

```java

/**
 * Checks the connection by sending and then receiving a packet
 *
 * @param address The InetAddress to which the packet should be sent
 * @return Returns whether the correct response was received from the server in time
 */
private static Boolean checkConnection(InetAddress address){

    try{
        byte[] sendMsg = checkString.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendMsg, sendMsg.length, address, port);
        discoverySocket.send(sendPacket);

        byte[] recvBuf = new byte[15000];
        DatagramPacket receivePacket = new DatagramPacket(recvBuf, recvBuf.length);
        discoverySocket.receive(receivePacket);

        String message = new String(receivePacket.getData()).trim();

        if(message.equals(connectedString)){
            return true;
        } else{
            return false;
        }

    } catch(SocketTimeoutException e){
        System.out.println("Reply not received, disconnected");
        return false;
    } catch(IOException e){
        e.printStackTrace();
        return false;
    }

}

```

Throwing a SocketTimeoutException indicates no packet has been received within the timeout duration of 10 seconds, thus it can be assumed the Client has been disconnected from the Server

Rare exceptions such as this IOException are largely ignored, but it can be assumed something has happened to the connection between the Client and Server, so false is returned

A similar practice occurs in the Server object code below.

```java

    /**
     * Constantly accepts and processes new clients
     */
    private Thread acceptThread = new Thread(() -> {

        try{

            while(true){
                clientSocket = serverSocket.accept();
                tryAdd(clientSocket);
            }

        } catch(IOException e){
            showError("Server suddenly stopped");
        }

    });

```

When the ServerSocket fails to function, something has went wrong, and so the error is displayed to the user

By handling very specific errors, I can shut down the program if it is a fatal error, take appropriate action to correct it, or utilize it as information about what is happening in the program.

Threading

Networking requires many simultaneous processes, so threads are important for these programs. Implemented thread-related features include cross-thread communication and thread safety. To utilize threads, both the ClientHandler and DiscoveryHandler extend the Thread object, as seen in the following code.

```java

/**
 * This is the class that handles the DatagramSocket for server discovery and client connection validation (heartbeat)
 *
 * @author Jonathan Zhao
 * @version 1.0
 */
public class DiscoveryHandler extends Thread {
    private DatagramSocket socket;
    private int port;

```

```java

/**
 * This is the class that handles connections and communication to clients
 *
 * @author Jonathan Zhao
 * @version 1.0
 */
public class ClientHandler extends Thread {
    private InetAddress address;
    private Date date;
    private SimpleStringProperty username;

```

This means both the DiscoveryHandler and ClientHandler's processes do not interfere with the main thread of the Server, allowing it to run without significant pauses. Various other processes are also threaded, which are outlined on the following page.

```java

    /**
     * Constantly accepts and processes new clients
     */
    private Thread acceptThread = new Thread(() -> {

        try{

            while(true){
                clientSocket = serverSocket.accept();
                tryAdd(clientSocket);
            }

        } catch(IOException e){
            showError("Server suddenly stopped");
        }

    });

```

This allows the Server to continue accepting clients through the ServerSocket without pausing any other processes in the application thread, which handles GUI. This is important since serverSocket.accept() blocks the thread until a socket is accepted.

```java

    /**
     * Constantly checks the connection with the client
     */
    private Thread checkThread = new Thread(() -> {

        try{
            Thread.sleep(1000);

            while(!clientSocket.isClosed()){
                checkConnection();
                Thread.sleep(100);
            }

        } catch(InterruptedException interruptEx){
            System.out.println("Check thread interrupted");
        }

    });

```

This allows the ClientHandler to check the connection with the Client without interrupting continuous processes such as streaming the client's screen.

```java

    /**
     * Constantly tries to maintain a connection with the server
     */
    private static Thread discoveryThread =
    new Thread(() -> {

        try{
            discoverySocket = new DatagramSocket();
            discoverySocket.setBroadcast(true);
            discoverySocket.setSoTimeout(timeoutDelay);
            discover();
        } catch(IOException ioE){
            System.out.println("Failed to create datagram socket");
        }

    });

```

This allows the Client to call discover(), which handles discovery and connection validation, without blocking continuous processes such as streaming.

```java

    /**
     * Constantly executes any runnables in the readRequests queue
     */
    private static Thread readThread =
    new Thread(() -> {

        while(true){

            try{
                readRequests.take().run();
            } catch(InterruptedException e){
                e.printStackTrace();
            }

        }

    });

```

This maintains a reference to the thread in the Client object that reads commands sent from the server, both ensuring thread safety and allowing the Client to constantly read from the input stream and parse commands without blocking processes such as streaming.

```java
    /**
     * Constantly executes any runnables in the connectionRequests queue
     */
    private static Thread connectionThread =
    new Thread(() -> {

        while(true){

            try{
                connectionRequests.take().run();
            } catch(InterruptedException e){
                e.printStackTrace();
            }

        }

    });
```

This allows the Client to perform continuous processes such as streaming without being affected by the main application thread, which handles displaying the alert.

By threading these various processes, there is no interruption of important continuous operations such as reading from a stream, writing to a stream, or blocking methods.

There is also cross-thread communication which is achieved by the use of volatile variables queues, and a JavaFX timeline. These are all displayed below.

```java
    /**
     * Shows an alert on the JavaFX application thread
     */
    private static Timeline alertTimeline =
    new Timeline(new KeyFrame(Duration.millis(1), e -> {
        System.out.println("Sending message");
        alert.setContentText(alertMessage);
        alert.show();
    }));
```

Allows the alert to be displayed on the application thread of the Client, even if it is called from another thread. This is important since displaying an alert blocks the thread.

```java
    private static LinkedBlockingQueue<Runnable> connectionRequests = new LinkedBlockingQueue<Runnable>();
    private static LinkedBlockingQueue<Runnable> readRequests = new LinkedBlockingQueue<Runnable>();
    private static volatile Boolean streaming = false;
    private static volatile Boolean connected = false;
    private static volatile InetAddress serverIp = null;
```

```java

                readRequests.add(() -> {
                    readFromConnection();
                });

                while(connected){

                    if(streaming){
                        sendScreen();
                    }

                    Thread.sleep(100);
                }

```

When connected is set to false, this thread in the Client object quickly exits this loop without attempting to stream to a disconnected server.

As described in the data structures section, queues allow cross-thread communication by placing runnables inside which are then called by another thread. Volatile booleans act as flags across threads, as their values are constantly checked before being used.

Cross-thread communication allows all these simultaneous processes to occur as well as affect one another such that they can perform blocking operations without blocking the original thread or cease execution when a certain condition is detected by another thread.

Bibliography

Mey, Michiel De. “Network Discovery Using UDP Broadcast (Java).” Michiel De Mey's Blog, 28 Aug. 2012, michieldemey.be/blog/network-discovery-using-udp-broadcast/.

Minh, Nam Ha. “How to Capture Screenshot Programmatically in Java.” CodeJava, 10 Aug. 2019, www.codejava.net/java-se/graphics/how-to-capture-screenshot-programmatically-in-java.

Pallavi, Priya. “How to Convert Image to Byte Array in Java.” TutorialsPoint, 9 Jan. 2018, www.tutorialspoint.com/How-to-convert-Image-to-Byte-Array-in-java.

“TableColumn (JavaFX 8).” JavaDocs, Oracle, 10 Feb. 2015, docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TableColumn.html.

Tsagklis, Ilias. “Compress a JPEG File.” Java Code Geeks, 11 Nov. 2012, examples.javacodegeeks.com/desktop-java/imageio/compress-a-jpeg-file/.