NAD Assignment Report

This report describes the procedures i followed in order to solve the assignment.

Server Application

The server application is a multi-threaded application that handles users.
It identifies users by references to their sockets, and assigns each connection to a different thread.
The server application consists of three files:

- **App.java**: defines the entry point, and its responsibility is to start the server using supplied configurations and asks JVM to create threads for each connection.
- **ClientThread.java**: defines the behavior of each thread and its responsibility is to propagate messages to all connected clients.
- **Message.java**: a small class that store some information about messages.

Client Application

The client application is a multi-threaded application that displays GUI for user to chat.
The client application consists of four files:

- **MainWindow.java**: defines the entry point, it fires up the GUI and contains the greatest percentage of programming logic.
- **Callback.java**: an interface that contains a single **execute()** method to be used.
- **DataRecievedEvent.java**: a class that calls **Callback.execute()** when data is recieved from the server.
- **Message.java**: a small class that store some information about messages.

How server application works

The **App** class creates a threadpool of default size 10 and works on port **45012**.

When a client connects, a reference to its **java.net.Socket** object, then a new thread is made and this object is passed to it.

Finally, this thread is submitted to the threadpool.

App.java

```
1  package app;
2
3  import java.net.ServerSocket;
4  import java.net.Socket;
5  import java.util.concurrent.ExecutorService;
```

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class App {
    public static void main(String[] args) throws Exception {
        int PORT_NUMBER       = 45012;
        int THREAD_POOL_SIZE  = 10;

        if (args.length == 1) {
            PORT_NUMBER = Integer.parseInt(args[0]);
        }
        else if (args.length == 2) {
            THREAD_POOL_SIZE = Integer.parseInt(args[1]);
        }

        ExecutorService pool = Executors.newFixedThreadPool(THREAD_POOL_SIZE);
        ServerSocket serverSocket = new ServerSocket(PORT_NUMBER);
        try {
            while (true) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    Runnable clientThread = new ClientThread(clientSocket);
                    pool.submit(clientThread);
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            try {
                serverSocket.close();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
            pool.shutdown();
        }
    }
}
```

I've faced a problem, when i sent a message from one GUI application it didn't show up in the other applications.

After investigating for a while, i discovered that clients must be stored inside a static ArrayList, and when the server recieve data from one client it propagates it to all other clients.

ClientThread.java

```java
package app;

import java.io.DataInputStream;
import java.io.DataOutputStream;
```

```
import java.io.IOException;
import java.net.Socket;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.ArrayList;
import com.google.gson.Gson;

class ClientThread implements Runnable {
    private static final int MESSAGE_SIZE = 1024;
    private static ArrayList<Socket> clients = new ArrayList<Socket>();
    private Socket clientSocket;

    public ClientThread(Socket clientSocket) {
        this.clientSocket = clientSocket;
        clients.add(clientSocket);
        Message message = new Message();
        message.setSender("Server");
        message.setContent("Let's welcome [" + clientSocket.getInetAddress() + ":" + clientSocket.getPort() + "] !");
        message.setTimestamp(LocalDateTime.now().toEpochSecond(ZoneOffset.ofHours(3)));
        try {
            updateClients(message);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("Client: " + clientSocket.getInetAddress() + ":" + clientSocket.getPort()
                           + " has connected at: " + LocalDateTime.now());
    }

    public void updateClients(Message msg) throws IOException {
        byte[] buffer = new Gson().toJson(msg, Message.class).getBytes();
        for (int i = 0; i < clients.size(); i++) {
            if (clients.get(i).isClosed()) {
                clients.remove(i);
                continue;
            }
            DataOutputStream os = new DataOutputStream(clients.get(i).getOutputStream());
            os.write(buffer);
            os.flush();
        }
    }

    public void run() {
        try {
            DataInputStream is;
            byte[] bytes = new byte[MESSAGE_SIZE];
            Message incomingMessage;
            Message outgoingMessage;

            while (true) {
                is = new DataInputStream(clientSocket.getInputStream());
                incomingMessage = new Message();
                outgoingMessage = new Message();

                if (is.read(bytes) == -1) {
                    break;
                }
```

```
                String validJSON = new String(bytes).trim();
                System.out.println("Recieved: (" + validJSON + ") from client: " + clientSocket.getInetAddress() + ":"
                                    + clientSocket.getPort() + " at: " + LocalDateTime.now());

                incomingMessage = new Gson().fromJson(validJSON, Message.class);

                outgoingMessage.setContent(incomingMessage.getContent());
                outgoingMessage.setSender(incomingMessage.getSender());
                outgoingMessage.setTimestamp(LocalDateTime.now().toEpochSecond(ZoneOffset.UTC));

                updateClients(outgoingMessage);
                bytes = new byte[MESSAGE_SIZE];
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            try {
                clientSocket.close();
            }
            catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

To transmit data we need some sort of serialization/deserialization process, this could have been done manually but i chose to use **GSON** library to achieve this.

This library serialize the contents of **Message** class into a JSON string, and the other side deserialize it.

Message.java

```
package app;

public class Message {
    private String sender;
    private String content;
    private long timestamp;

    public String getSender() {
        return this.sender;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }

    public String getContent() {
        return this.content;
```

```
    }

    public void setContent(String content) {
        this.content = content;
    }

    public long getTimestamp() {
        return this.timestamp;
    }

    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }
}
```

How client application works

The **MainWindow** drives the client GUI application.

Before the client connect to the server, the application has to do some validations first, then it connects to the server and attach a thread that continuously recieve updates from the server and updates the log.

```
connectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        if (connectionLock) {
            //  If client already connected, show warning message and do nothing
            JOptionPane.showMessageDialog(frmChatApplication, "You're already connected to a host.\nPlease disconnect first the
                JOptionPane.WARNING_MESSAGE);
        }
        else {
            String hostname = hostnameField.getText().trim();
            int port = -1;
            username = usernameField.getText().trim();
            String usernamePattern = "[A-Za-z0-9_]{6,20}";  //  Username pattern

            boolean isValid = true; //  Validation flag
            String validationError = "";    //  Error message accumulation

            //  Instead of a full-blown check, we'll just check if the hostname field isn't empty
            if (hostname.length() == 0) {
                hostnameField.setBackground(new Color(255, 0, 127));    //  Change field background color to pink to notify us
                validationError += "Hostname cannot be empty\n";
                isValid = false;
            }

            //  Parse the port and check it's validity
            try {
                port = Integer.parseInt(portField.getText().trim());
            }
            catch (NumberFormatException e) {
                portField.setBackground(new Color(255, 0, 127));
                validationError += "The value in port field must be an integer\n";
```

```
            validationError += "The value in port field must be an integer\n";
            isValid = false;
        }

        //  A port cannot be negative
        if (portField.getText().startsWith("-")) {
            portField.setBackground(new Color(255, 0, 127));
            validationError += "The value in port field must be a positive integer\n";
            isValid = false;
        }

        if (!Pattern.matches(usernamePattern, username)) {
            usernameField.setBackground(new Color(255, 0, 127));
            validationError += "The username must contain only alphanumeric characters (A-Z and 0-9) or underscore (_) and
                characters in length\n";
            isValid = false;
        }

        if (!isValid) {
            String validationMessage = String.format("The connection couldn't be established due to the following validati
                );
            JOptionPane.showMessageDialog(frmChatApplication, validationMessage, "Validation error", JOptionPane.ERROR_MES
            return;
        }


        //  Let's try to establish the connection
        try {
            clientSocket = new Socket(hostname, port);
            os = new DataOutputStream(clientSocket.getOutputStream());
            is = new DataInputStream(clientSocket.getInputStream());
            logUpdater = new DataRecievedEvent(is);
            try {
                logUpdater.attachCallback(new Callback() {
                    @Override
                    public void execute(Message message) {
                        DateTimeFormatter pattern = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
                        String time = LocalDateTime.ofInstant(Instant.ofEpochMilli(message.getTimestamp()), ZoneOffset.UTC
                        String result = String.format("[INFO] %s [%s] Says: %s\n", time, message.getSender(), message.getC
                        logField.append(result);
                    }
                });
            }
            catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            new Thread(logUpdater).start();
            connectionLock = true;  //  We're connected now
            connectionStatusLabel.setText("Connected");
            connectionStatusLabel.setForeground(Color.GREEN);
        }
        catch (IOException e) {
            String errorMessage = String.format("The connection to %s:%d couldn't be established due to the following erro
                .getMessage());
            JOptionPane.showMessageDialog(frmChatApplication, errorMessage, "Fatal error", JOptionPane.ERROR_MESSAGE);
            connectionLock = false; // But if something happens, we will release the lock
```

```
                connectionLock = false; // But if something happens, we will release the lock
                connectionStatusLabel.setText("Disonnected");
                connectionStatusLabel.setForeground(Color.RED);
                username = null;
                logUpdater = null;
                try {
                    is.close();
                    os.close();
                }
                catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
});
```