Criterion A: Problem Specification

Problem Scenario

A friend of mine has noticed that he cannot find a chatting application that allows one to chat with people in a way that makes the chat truly private and not saved on the servers of big corporations, and that all of the popular online chatting softwares that he used save chats even when they are deleted. This realization of his sparked my interest and made me talk to some of my other friends about it. They all said that the chatting apps they currently use suffice all of their needs, however, they are all centralized and are not the safest to use regarding private information, as all of the chats are saved even upon deletion, and they are afraid that their account might get hacked. This is why I have decided to create a decentralized software solution that will allow anyone to create, join and host a chat room, the chats of which will stay private to the people in the chat, and where messages will be purged upon closing the server, in order to assure maximal privacy and confidentiality.

Computational Context

For the purpose of this software, I will be using Java. The reason for this is because it features two powerful libraries inside of its standard library, Swing and Sockets. Swing will allow me to easily develop the GUI for my application with no major issues, and Java's sockets will allow me to develop a networking system that will be able to create servers and clients that will have bidirectional communications, allowing for an easy but safe chatting experience.

Success Criteria

1. Any user can create a chat room.
2. The chat room will be hosted on the computer of the host, with their IP address and port.
3. Users can connect without the need for an account, but rather by a simple username.
4. The chats will be purged upon the closing of a chat room to ensure confidentiality.
5. A user can connect to an existing chat room only if they have received the connection IP from the chat room host.
6. The app will be multithreaded in order to ensure safe networking.

Criterion B: Planning

Decomposition

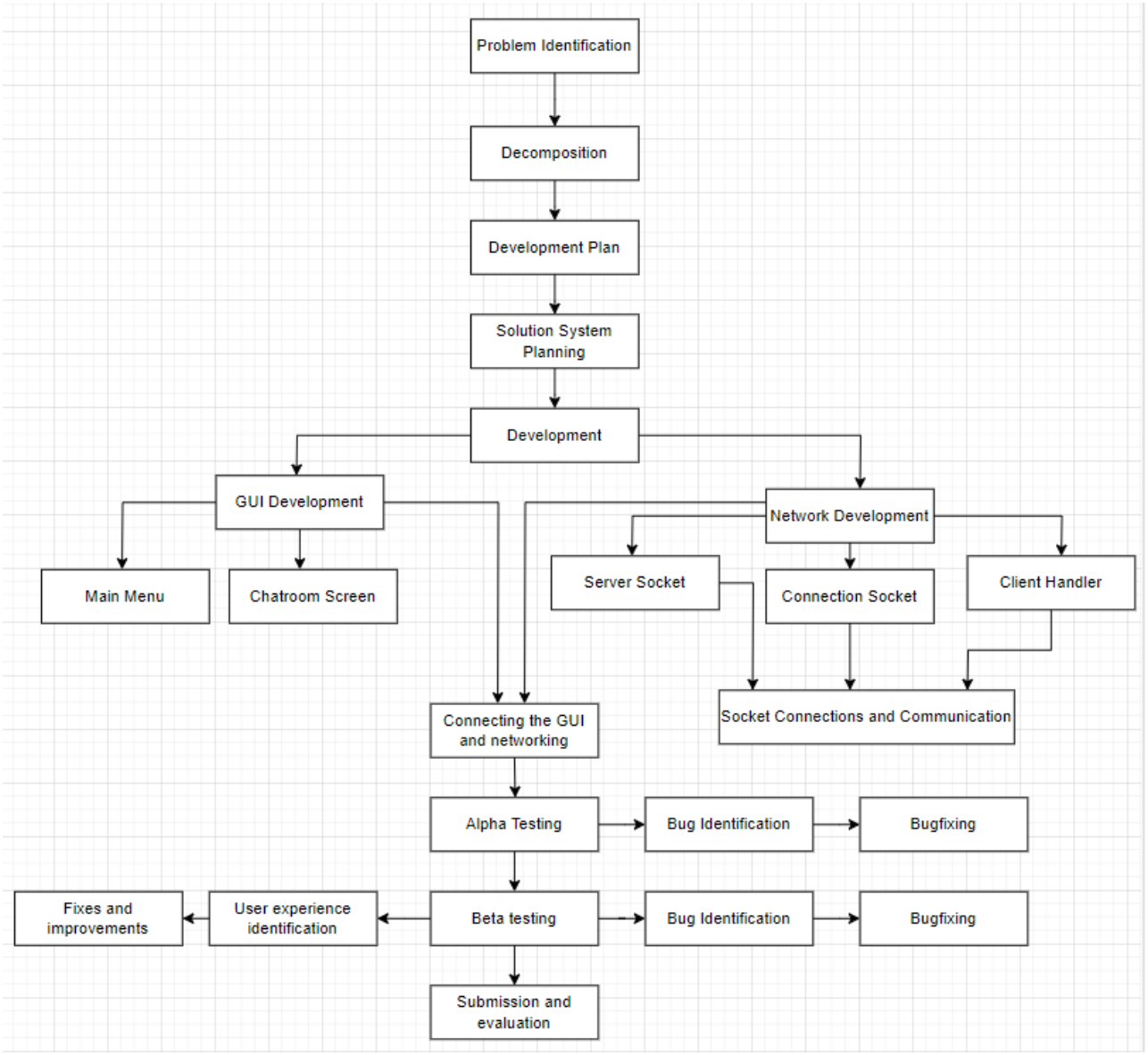

Image 1: Decomposition of all the tasks in development, drawn using draw.io

Plan

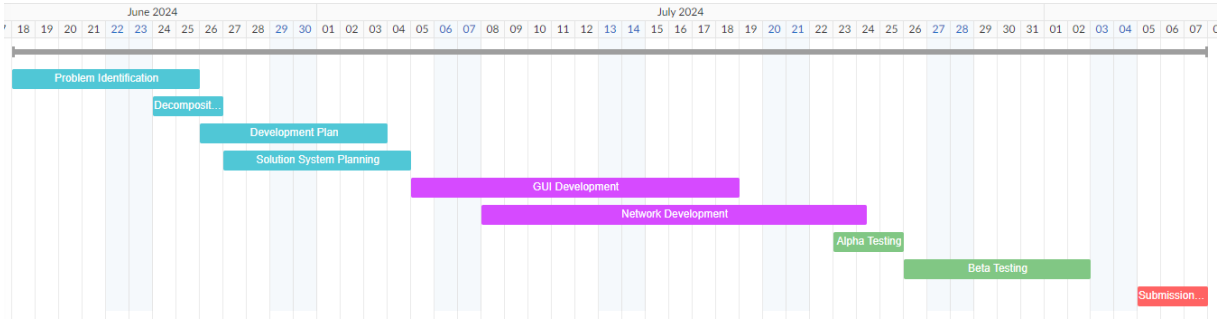

Image 2: GANTT Chart of the plan

Phase 1 (The Plan)	Problem Specification, Decomposition, Development and System Solution Plan	2 weeks
Phase 2 (Development)	GUI and Network Development	2-3 weeks
Phase 3 (Testing and fixing)	Alpha and Beta Testing, including fixing of bugs and UX	1 week
Phase 4 (Submission)	Submission, Evaluation and Documentation-writing	3 days

For further information on each subsection of each part of the process, reference the decomposition.

Criterion C: System Overview

System Model

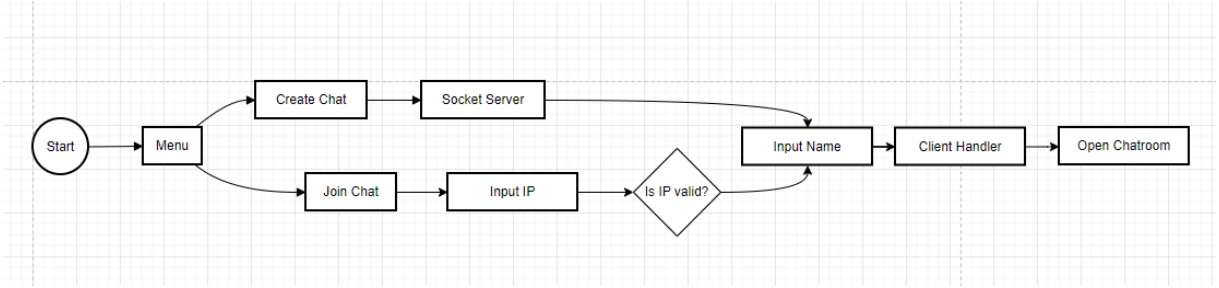

Image 3: The app usage process

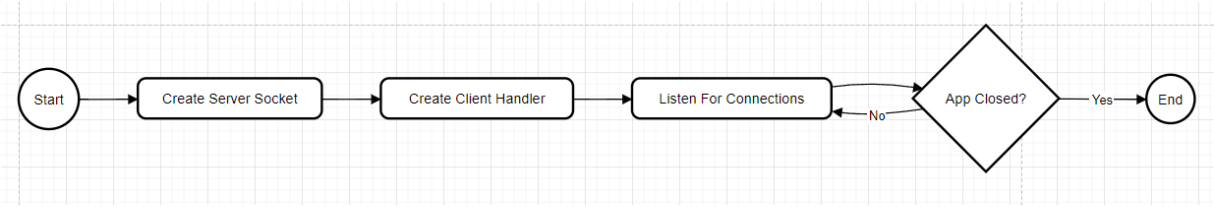

Image 4: The socket network loop

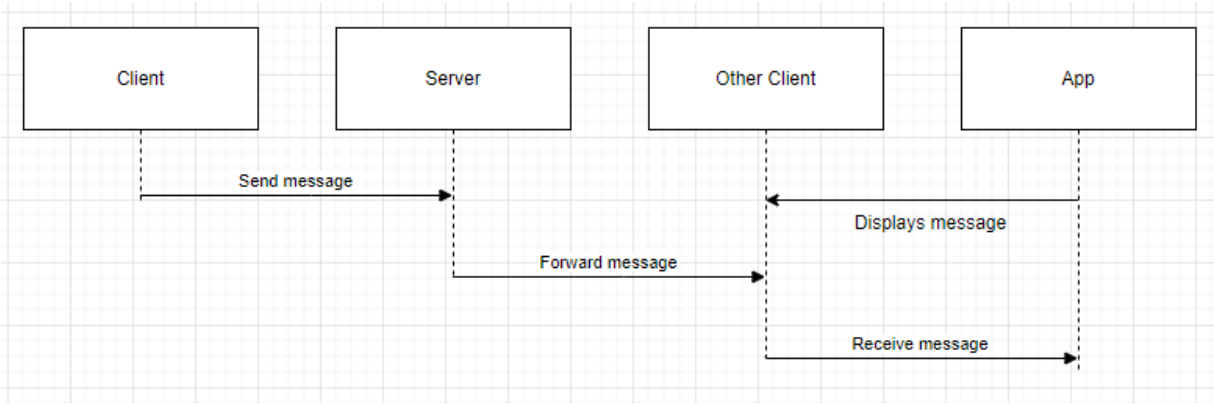

Image 5: Message sending

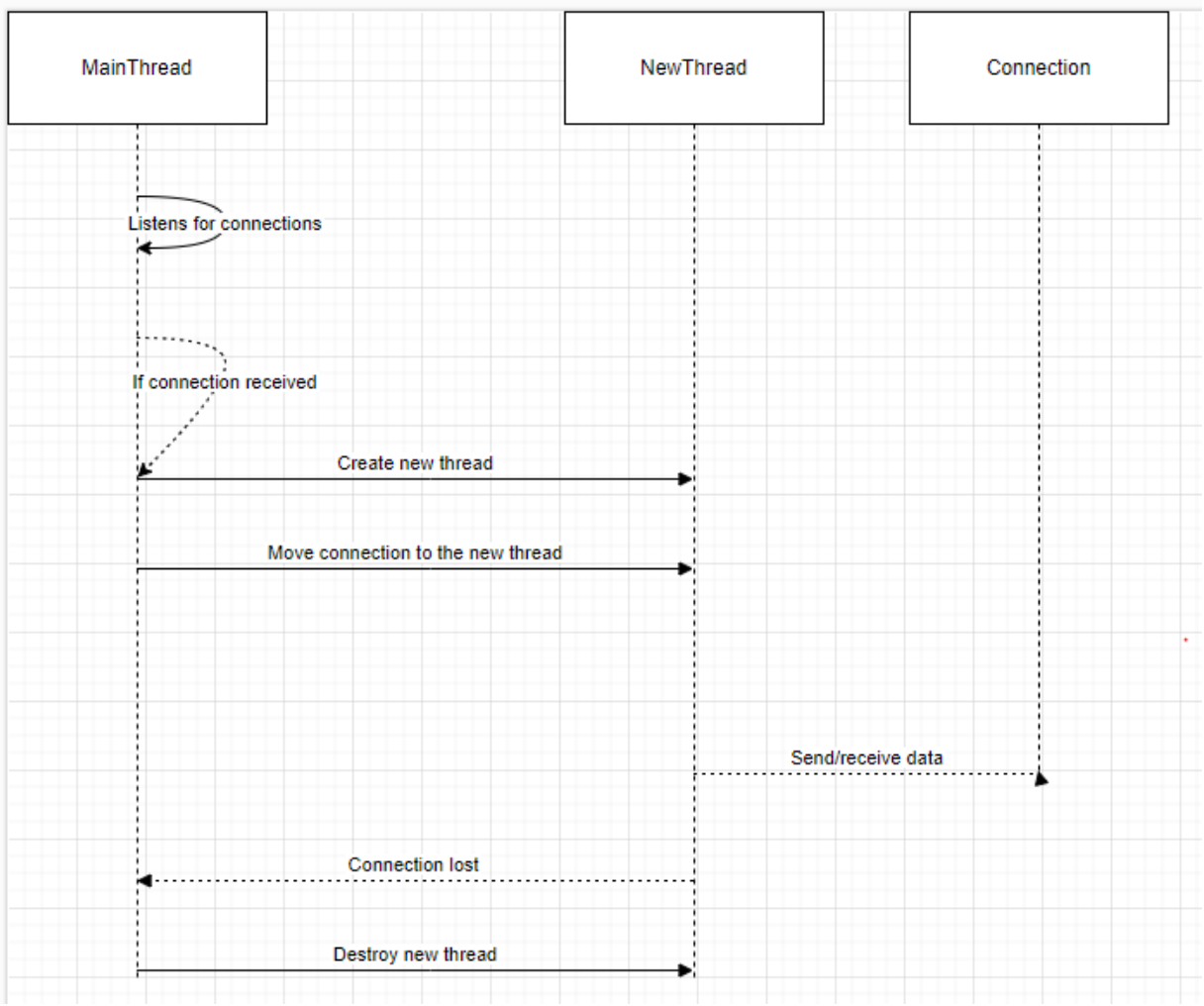

Image 6: Multithreading

Proposed Algorithms

- New server socket algorithm

```
function startServer() throws IOException:
    serverSocket = createServerSocket(12345)
    ipAddress = getPublicIpAddress()
    if ipAddress is not null:
        appendToChatArea("Server started at " + ipAddress + ":" +
serverSocket.getLocalPort() + "\n")
    else:
        appendToChatArea("Could not retrieve public IP address.\n")
    while true:
        clientSocket = serverSocket.accept()
        appendToChatArea("Client connected: " + clientSocket.getInetAddress())
        writer = createPrintWriter(clientSocket.getOutputStream(), true)
        addToClientWriters(writer)
        new ClientHandler(clientSocket, chatArea, clientWriters).start()
```

- Listen to messages algorithm

```
function run():
    message = readLineFromInput()
    while message is not null:
        appendToChatArea(message + "\n")
        broadcastMessage(message)
        message = readLineFromInput()
    handleException(IOException e):
        appendToChatArea(e.getMessage() + "\n")
```

- Connect to existing server algorithm

```
function connectToServer() throws IOException:
    socket = createSocket(host, port)
    out = createPrintWriter(socket.getOutputStream(), true)
    in = reader(streamreader(socket.getInputStream()))
    appendToChatArea("Connected to server at " + serverAddress + "\n")
    createThread():
        message = readLineFromInput()
        while message is not null:
            appendToChatArea(message + "\n")
            message = readLineFromInput()
        handleException(IOException e):
            showMessageDialog(this, "Server has disconnected", ERROR_MESSAGE)
            exit(0)
```

- Send message to all clients algorithm

```
sendButton.addActionListener(e -> {
    message = trim(inputField.getText())
    if not message.isEmpty():
        appendToChatArea(userName + ": " + message + "\n")
        inputField.setText("")
        for writer in clientWriters:
            writer.println(message)
})
```

Testing Strategy

Success Criteria	Testing Process	Success Requirement
1	Attempt to create a chat room upon opening the software.	After correctly inputting a username, if there is internet connection, a chat room is successfully created.
2, 3, 5	Connect with multiple different app instances to the same chat room.	After successfully inputting a username and a correct IP with a specified port, the users are able to stay in the same chat room and communicate with each other and see each other's messages.
4	Reopen the application. Check the app's files.	When the hosts closes the application, all clients are disconnected and the chat is not saved anywhere to ensure safety and decentralization.
6	Test the algorithms in the IDE directly using the Java Thread library and debug the application in the IDE to check which threads are running which part of the software.	Every client is connected to a different thread so that the clients are not blocking each other's threads whilst using the application, handled by the ClientHandler class.

Criterion D: Development

REST API usage for IP retrieval

Since Java's standard library doesn't offer a possibility to acquire the host's public IP address, I used an external REST API by Amazon to obtain the IP address of the host so that they may share it for other people to connect, if they are unaware of their IP address.

```
private String getPublicIpAddress() {
    try {
        URL url = new URL( spec: "https://checkip.amazonaws.com");
        BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));
        return in.readLine();
    } catch (IOException e) {
        return null;
    }
}
```

Initially, a URL instance is created with the link to the API's endpoint. A stream is opened to connect to the endpoint, and a BufferedReader is created from the respective InputStreamReader in order to read the HTTP request response. The response text is returned as a String. In order to ensure exception handling, the block of code is surrounded in a try-catch statement. The advantage of this approach is that it is quick and efficient and returns the public IP, however, a better approach than returning null upon catching an exception would have been benefitial.

The testing strategy effectively tests this technique and algorithm, as by testing the creation of new chat rooms also outputs the public IP of the chat room created, showcasing whether or not this HTTP request was successfully performed.

Connection to a socket server

```
private void connectToServer() throws IOException {
    String[] addressParts = serverAddress.split( regex: ":");
    String host = addressParts[0];
    int port = Integer.parseInt(addressParts[1]);

    socket = new Socket(host, port);
    out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

    chatArea.append("Connected to server at " + serverAddress + "\n");

    new Thread(() -> {
        try {
            String message;
            while ((message = in.readLine()) != null) {
                chatArea.append(message + "\n");
            }
        } catch (IOException e) {
            JOptionPane.showMessageDialog( parentComponent: this, message: "Server has disconnected", title: "Server Disconnected", JOptionPane.ERROR_MESSAGE);
            System.exit( status: 0);
        }
    }).start();
}
```

Upon inputting a username and an address (in the form IP:port), the address is split into the port and IP respectively, and a socket instace is created, along with a reader and writer to send data to the server and receive data from it. The chat area (which is a Swing component) informs the user that they have successfully connected, after which a new thread is made to listen to messages from the server. If an exception is caught, it means that the server closed the connection and the user is informed that the chat room is closed.

A major advantage of this approach is that it does not block the main thread due to multithreading and it allows the user to both send and receive messages, however, it is therefore hard to debug.

The testing strategy effectively tests this algorithm, as by testing connections to a chat room and receiving messages from other clients, the functionality of this algorithm is also tested, and can be tested repeatedly and quickly.

Creation of a socket server

```
private void startServer() throws IOException {
    serverSocket = new ServerSocket( port: 12345);

    String ipAddress = getPublicIpAddress();
    if (ipAddress != null) {
        chatArea.append("Server started at " + ipAddress + ":" + serverSocket.getLocalPort() + "\n");
    } else {
        chatArea.append("Could not retrieve public IP address.\n");
    }

    while (true) {
        Socket clientSocket = serverSocket.accept();
        chatArea.append("Client connected: " + clientSocket.getInetAddress() + "\n");

        PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), autoFlush: true);
        clientWriters.add(writer);

        new ClientHandler(clientSocket, chatArea, clientWriters).start();
    }
}
```

When a user attempts to create a chat room, a ServerSocket instance is created at port 12345 and the public IP is retrieved, and, if successful, the chat room area informs the user of their designated chat room address. After this, an infinite loop is created that listens to connections of clients, when a connection is made, a ClientHandler thread is created and started.

An advantage of this approach is that it is simple and allows for effective functioning of the app, as well as the fact that ClientHandler inherits from the Thread class, meaning that multithreading is ensured.

The testing strategy effectively tests this algorithm, as by testing success criterion 1, this algorithm is responsible for achieving the tests, meaning that it is fully tested for functionality.

Criterion E: Evaluation

Success Criteria Evaluation

Since the users are able to create a chat room, success criterion 1 has been met.

The socket server is hosted on the computer of the chat room's creator, meaning that success criterion 2 has been met, however, a better approach could have been to not have the host to be both a client and a server.

The only requirement to create a chat room is a username, and the only requirements to join an existing one is a username and the address of the room. Success criterion 3 is met.

There is no saving of the chats on purpose as a requirement in the problem scenario, success criterion 4 is met.

To connect to a room, a user must have the IP address and port of the room, however, there is no full guarantee that this was provided by the host. Success criterion 5 is mostly met.

The app handles clients and listens to messages and connections all on separate threads, meaning that multithreading and success criterion 6 have been achieved.

Justifications for future improvements

One of the most notable issues with the app is that in order for anyone other than the host's local area network users to connect to the chat room, the host must have the port 12345 opened on their network, either via UPnP or opening it directly. This can be improved by embedding a UPnP into the software directly if necessary.

Another notable improvement that should be added is a nicer and more approachable user interface, since the current one focuses on functionality rather than looks.

Finally, a good improvement would be to add a possibility for the host to accept or reject someone's access to the chat, as obtaining the IP address and port of the chat room would allow one to access the chat room regardless of it they were shared by the host or not.