A Presence-based Messaging Application

ELEC 6861 Project

Masood Khosroshahy

m.kh@ieee.org - www.masoodkh.com

Instructor: Prof. Roch Glitho

December 12, 2009

Contents

1 Introduction

As per the specification, a presence-based messaging and file-exchange application has been designed and implemented. Here is the scenario:

- Clients connect to the server and immediately declare their presence.
- A connected client can initiate a session by sending the request to the server along with the preferred number of clients in the session.
- The server application checks the number of available clients for the session.
- The server application initiates a session between the clients, if preferred number of clients are available.
- When the session is underway, participants can exchange messages and files.
- Only the session initiator can terminate the session.

1.1 Choices

The implementation has been done in Java using Socket programming (NetBeans IDE v.6.7.1 has been used for development). Protocols for signaling (presence and session establishment/termination), message and file exchanges have been implemented on top of TCP (i.e., respective Socket type has been chosen) due to the reliability that it offers.

2 Implementation

The implementation is split into five classes and will be described briefly as follows:

A general Data object (see Figure 1), that is passed between the client and server, which carries signal, message and file depending on how it's been constructed on the originating side. On the receiver side, Data object is read directly from the Socket and analyzed/parsed according to its type. Using this method, Java API has been used to its fullest potential to reduce the manual delimiter placing/parsing.

The client (see Figures 2 and 3) is implemented in a single class which has numerous functions that are invoked based on the events triggered through the client GUI.

The server application has a main class which relies on two other classes for its operations (see Figures 4 and 5); one object per client which holds all client-specific references and one thread per client which interacts with the client as long as the client is connected. The server application creates two server Sockets (one socket for receiving, and responding to, client signals and one socket dedicated to sending data to the client) at startup. It then listens to incoming connections and will create a thread to interact with a new client.

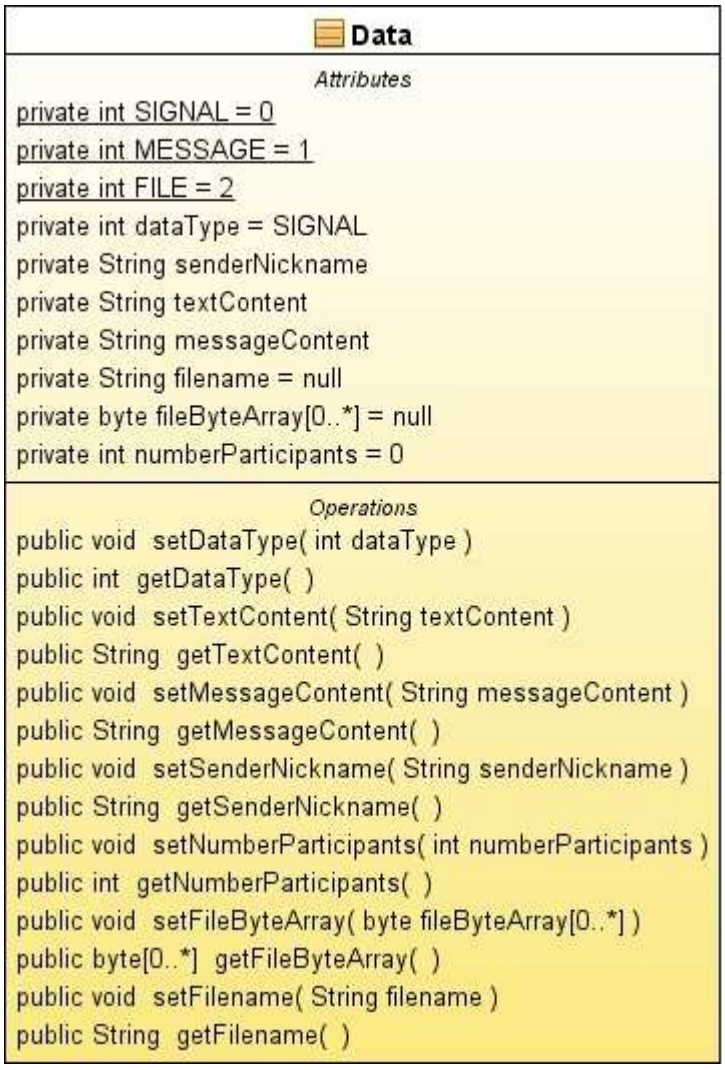

Figure 1: UML Class Diagram [Exchanged Data Object between Server and Client]

Figure 2: UML Class Diagram [Client-side: Main App(Part 1)]

Figure 3: UML Class Diagram [Client-side: Main App(Part 2)]

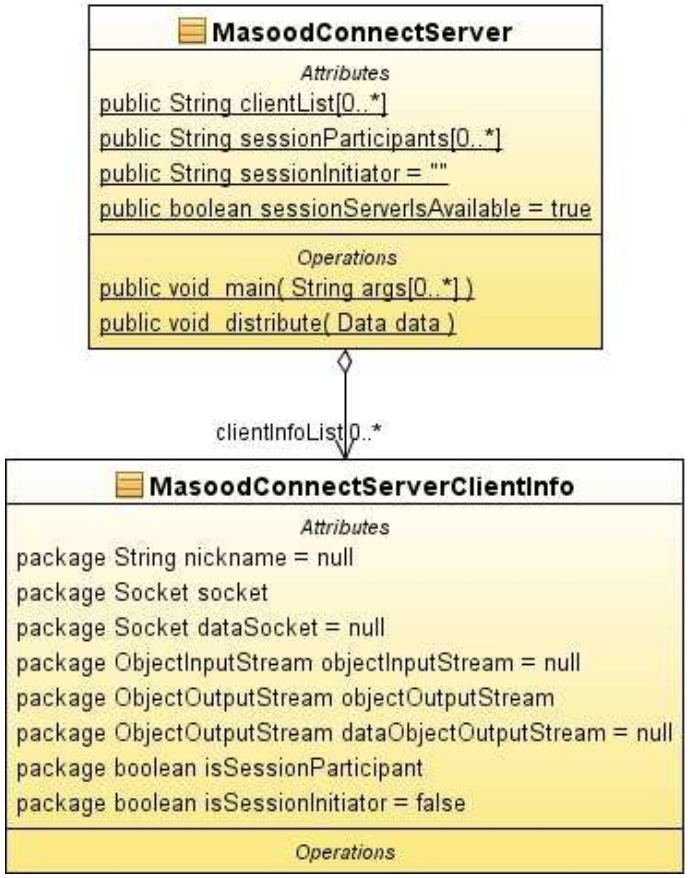

Figure 4: UML Class Diagram [Server-side: Main App]

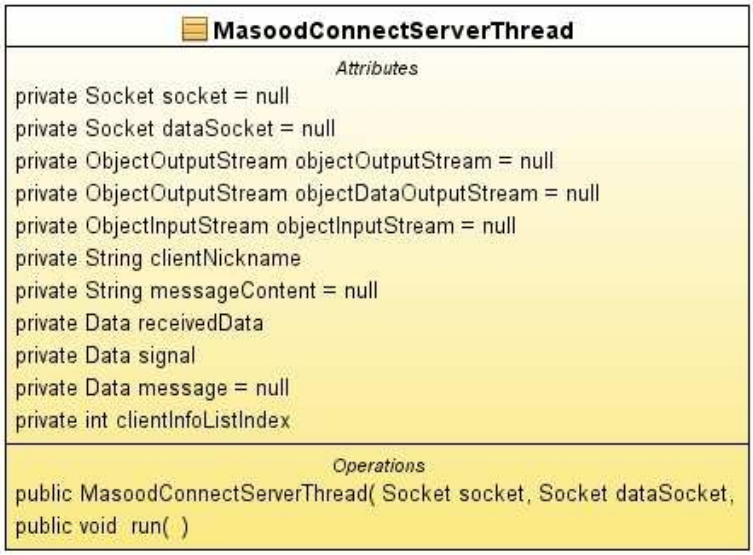

Figure 5: UML Class Diagram [Server-side: One thread per client]

3 Protocols Design: Messages and Rules

In describing the sequence of messages for different aspects of the application, it might be clearer to show the screenshots of the actual developed application, rather than using abstract sequence diagrams. That is why in what follows, the designs of simple protocols for presence, session establishment/termination, message and file exchanges are demonstrated using screenshots.

3.1 Presence

Client needs to set the server address along with signal and data ports (see Figure 6).

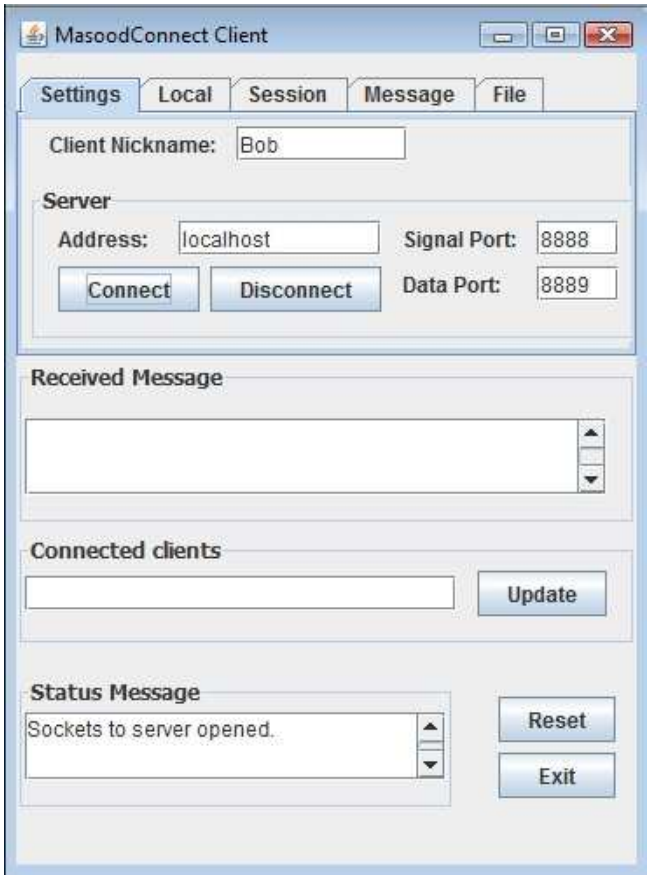

Figure 6: Presence: Client Connects to Server

Client sets its Nickname which should be unique; otherwise, it will not be accepted by the server. When two sockets are created, client immediately sends its nickname. After receiving the nickname, server creates the Output Streams, starts a thread, passes the relevant information to the thread and keeps all the references in an instance of "MasoodConnectServerClientInfo" Class (see Figures 7, 8 and 9).

```
private void ConnectActionPerformed(java.awt.event.ActionEvent evt) {

    try {
        if (!isConnected) {
            if (ServerAddress.getText().isEmpty() || ServerPort.getText().isEmpty()) {
                StatusMessage.setText("Server address or port not set correctly.");
            } else {
                isConnected = true;
                socket = new Socket(ServerAddress.getText(), Integer.parseInt(ServerPort
                objectOutputStream = new ObjectOutputStream(socket.getOutputStream());
                objectInputStream = new ObjectInputStream(socket.getInputStream());

                dataSocket = new Socket(ServerAddress.getText(), Integer.parseInt(server
                dataObjectInputStream = new ObjectInputStream(dataSocket.getInputStream(

                StatusMessage.setText("Sockets to server opened.");

                signal = new Data();
                signal.setTextContent(nickname.getText());
                objectOutputStream.writeObject(signal);
```

Figure 7: Presence: Client Creates the Sockets and Sends its Nickname

```
Administrator: Command Prompt - java -jar MasoodConnectServer.jar 8888 8889
C:\MasoodConnectApp\Server>java -jar MasoodConnectServer.jar 8888 8889
Bob Connected.
Alice Connected.
John Connected.
```

Figure 8: Presence: Server Receives Client's Nickname Upon Connection - Output

```
if (isNicknameUnique) {
    System.err.println(clientNickname + " Connected.");
    clientList.add(clientNickname);

    new MasoodConnectServerThread(socket, dataSocket, threadObjectInputStream, threadObjectOutp

    MasoodConnectServerClientInfo clientInfo = new MasoodConnectServerClientInfo();
    clientInfo.nickname = clientNickname;
    clientInfo.socket = socket;
    clientInfo.dataSocket = dataSocket;
    clientInfo.objectInputStream = threadObjectInputStream;
    clientInfo.objectOutputStream = threadObjectOutputStream;
    clientInfo.dataObjectOutputStream = threadDataObjectOutputStream;
    clientInfoList.add(clientInfo);
```

Figure 9: Presence: Server Receives Client's Nickname Upon Connection - Code

3.2 Session setup and termination

After connecting to the server, a client can request a session. The client sends to the server the number of desired participants. If the server is not busy and that number of clients are available, server picks randomly from the connected clients to form a session of the desired number (see Figures 10, 11 and 12).

Figure 10: Session: Bob requests a session of 3

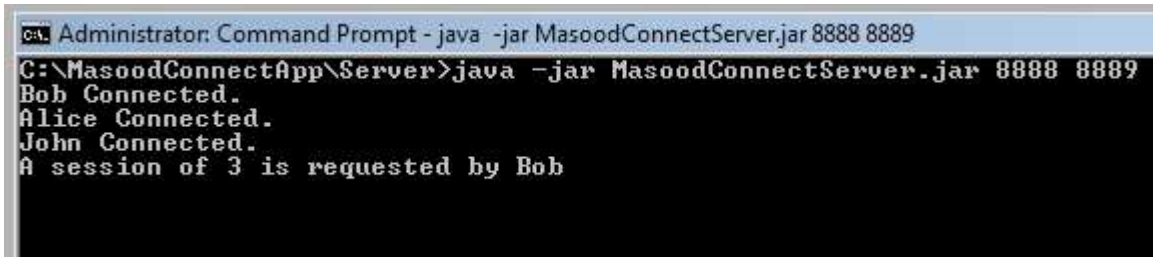

Figure 11: Session: Server receives the request and forms the session.

Figure 12: Session: Server confirms the session formation to Bob and mentions the participants.

A session can be canceled only by its initiator. If another session participant tries to cancel the session, the server sends a signal informing the client that its request has been ignored (see Figures 13, 14, 15 and 16).

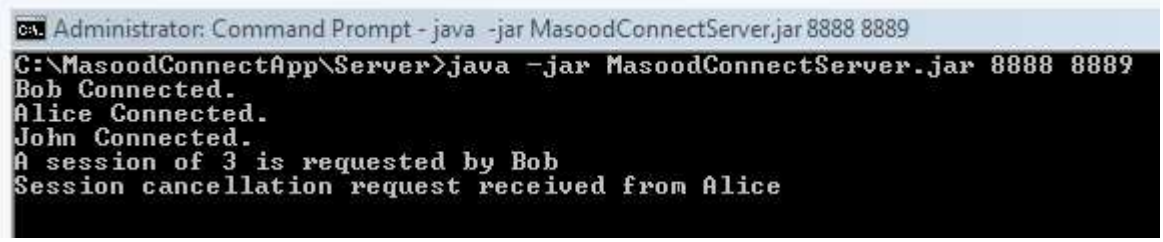

Figure 13: Session: Alice sends a session cancellation signal to server

Figure 14: Session: Alice is notified that her request has been ignored since she is not the session initiator.

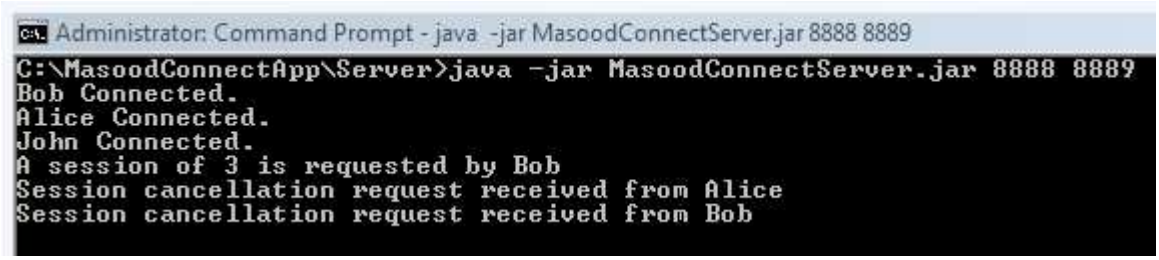

Figure 15: Session: Bob sends a session cancellation signal to server

Figure 16: Session: Server cancels the session created by Bob.

3.3 Messaging

When a session is underway, participants can send message to all session participants. To send a message, a client sends a signal to the server. The type of signal is "DistributeMessage" and its content is the message itself. When the server receives such a signal, it first checks whether the sender is part of the session. If so, the server distributes the message to all session participants (see Figures 17, 18, 19 and 20).

Figure 17: Messaging: Bob sends a message to the session (he receives his own message as well).

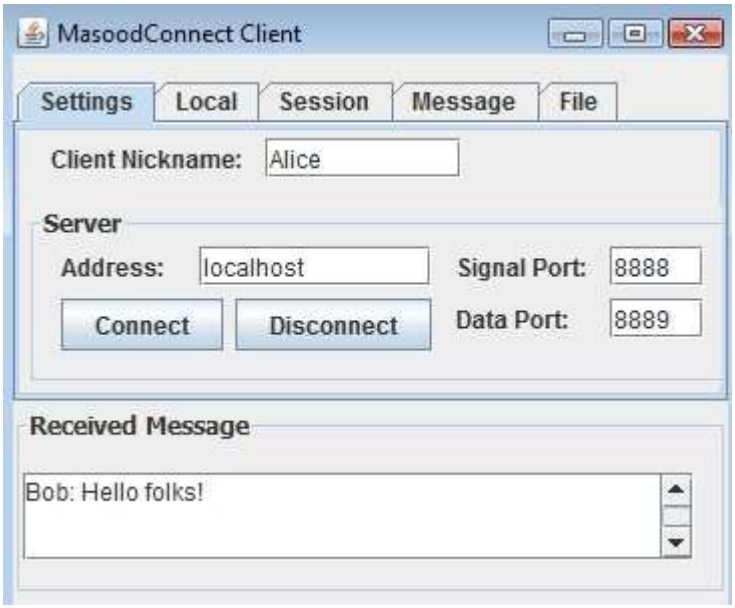

Figure 18: Messaging: Alice receives Bob's message

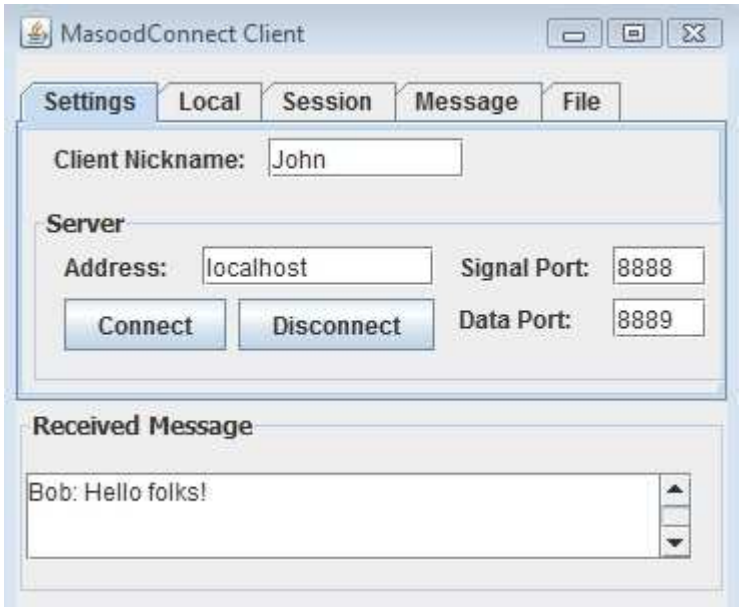

Figure 19: Messaging: John also receives Bob's message

```
private void SendMessageActionPerformed(java.awt.event.ActionEvent evt) {
    String fromUser = null;
    try {
        fromUser = Message.getText();
        if (fromUser != null) {
            signal = new Data();
            signal.setSenderNickname(nickname.getText());
            signal.setTextContent("DistributeMessage");
            signal.setMessageContent(fromUser);
            objectOutputStream.writeObject(signal);
        }
```

Figure 20: Messaging: Client sends a signal to the server which contains the message.

3.4 Exchanging files

In much the same way as messaging, session participants can exchange files. A client selects a file and creates a Data object containing a Byte Array which in turn contains the Bytes read from the file. The client then sends a signal to the server. The type of signal (Data object) is "DistributeFile" and its content is the aforementioned Byte Array. When the server receives the signal, it checks whether the sender is part of the session. If so, it distributes the Data object to all session participants. Each of the session participants saves the file to disk from the Byte Array as soon as they receive the Data object from the server.

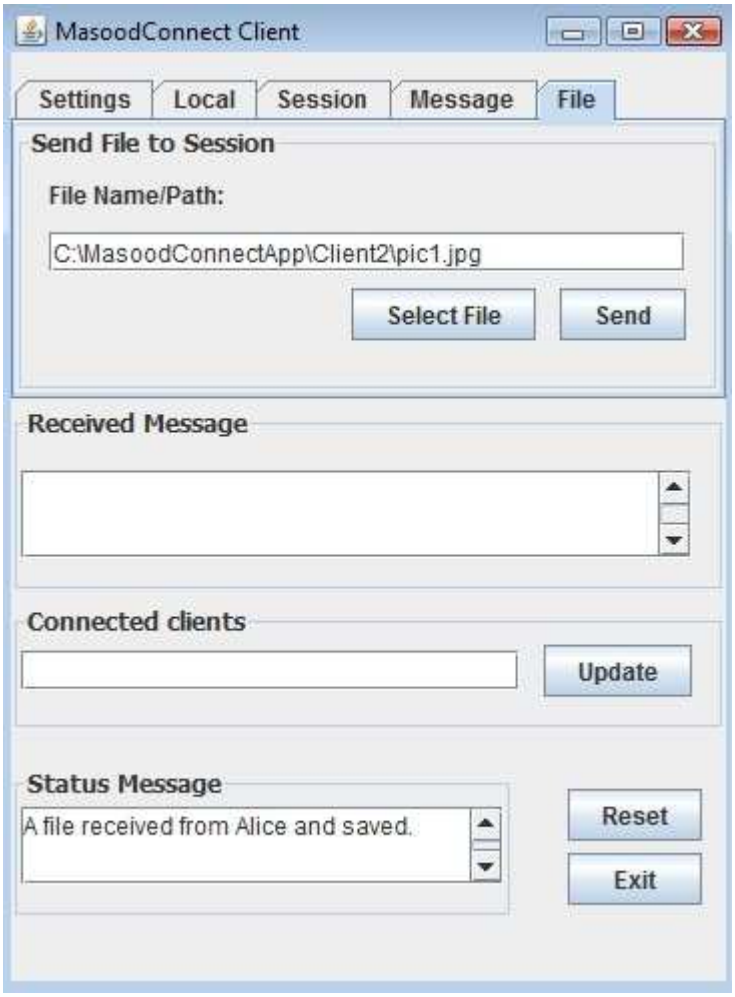

Figure 21: Exchanging files: Alice sends a file to the session (she receives her own file as well).

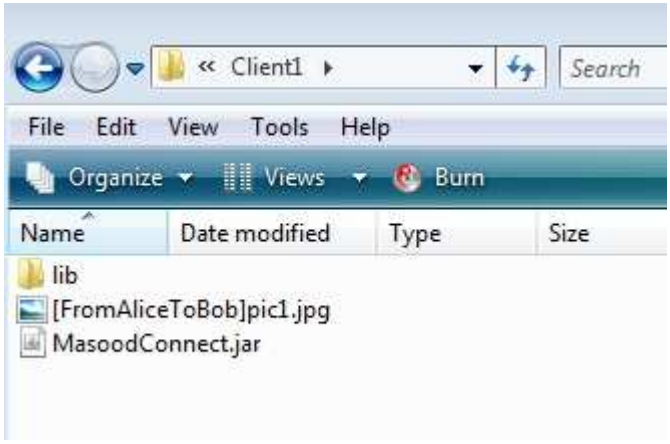

Figure 22: Exchanging files: Bob receives the file sent by Alice.

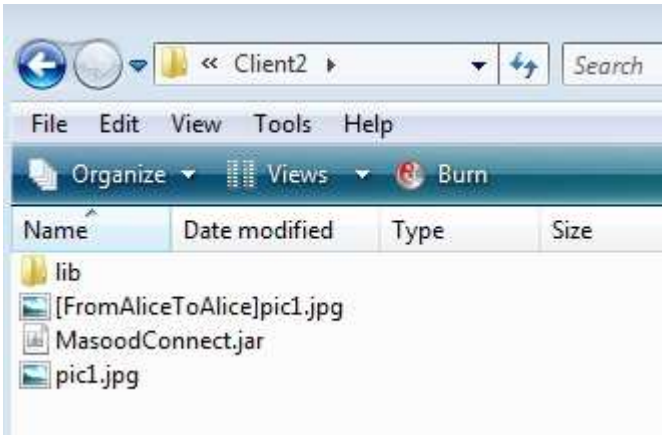

Figure 23: Exchanging files: Alice receives her own file as well.

Figure 24: Exchanging files: John receives the file sent by Alice as well.