ADVANCED INTERNET PROGRAMMING
ASSIGNMENT 1

1.0 Contents

2.0 Introduction

2.1 Aims & Objectives

To enable the student to produce a Distributed Computer application, using the Java programming language utilising the TCP/IP protocol.

2.2 Brief

Design, implement and test a chat program using the Java programming language. The implementation may use a peer-to-peer model or a client-server model and my use connection or connectionless networking programming or a mixture of both. The minimum acceptable implementation is two Java applications which can send and receive lines of text across a network connection as well as the deliverables details below.

Extra marks will be given for additional features and functionality which enhances the user experience such as

- a graphical user interface
- logging of connections
- logging of messages sent and received
- a server based chat program which relays messages between clients

These extra features are only suggestions and are not exhaustive. Evidence of cross platform testing will also merit extra marks.

3.0 Design

3.1 Client Application Overview

The application will consist of two programs. The first program will be the server that will administer all connected users, log activity and relay messages to the clients. The send application will be the client messenger application that will connect to the remote server.

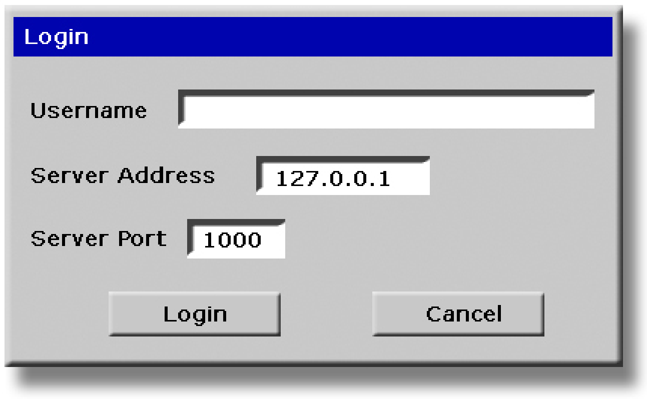
When a user loads the client application, they will be required to log into the remote server. *Figure 1.1* demonstrates the appearance of the login window. This will involve assigning a desired username to the session. This username will be used by the server to distinguish between different connections and therefore is important they remain unique. For example, two users cannot have the same username.

Additional the username field, the user can also specify the remote server host name or IP address and the port that the server is listening on. By default, this value will be *127.0.0.1* and port *1000*. Should the application go live for public use then these values would most likely be hidden from the user and the server would run on a dedicated IP address and port number.

Once the user has connected and been validated with the remote server, they will enter the main chat room. *Figure 1.2* demonstrates the appearance of the chat room. Immediately the server will inform the client of all members currently in the chat room. These values will populate a list of connected users in the main application window. The user will see two text fields, the first will display the entire conversation and system messages (for example, a user joins or leaves) for the session of the user. The user will also have a text window where they can type their own messages to the conversation window.

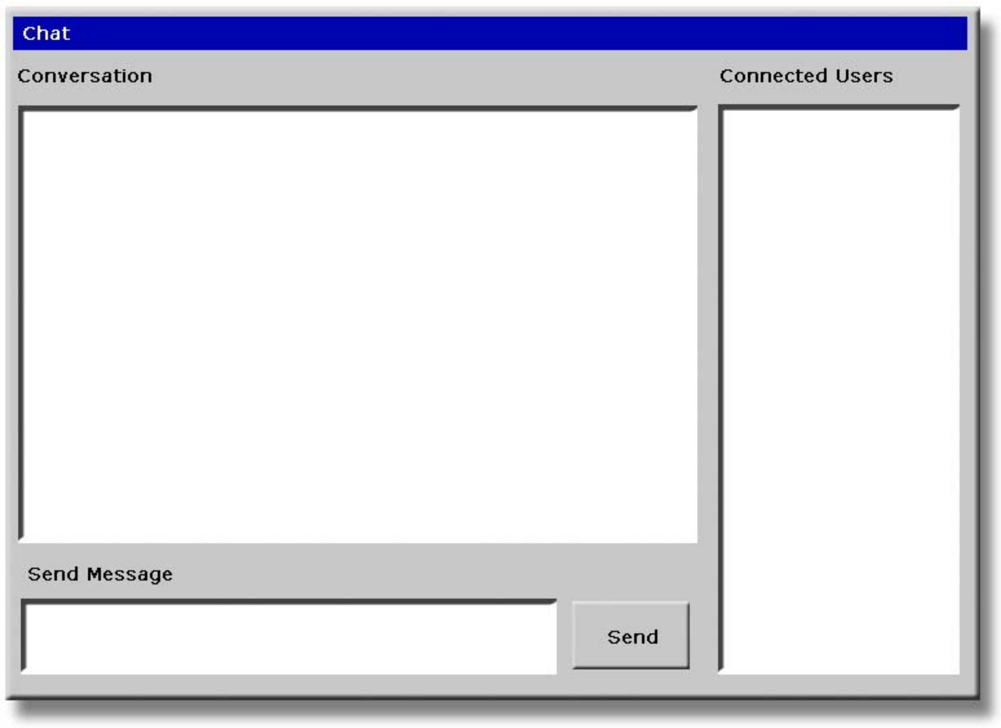
The user will also have the option to create a private message with individual members. To initiate a private message they will select a user from the user list and 'double-click' the item in the list. This will open a new chat window just for a conversation with that particular user. The user cannot open a private message window with him or her self. If a remote user establishes a private message and sends it to the user, the message will automatically be populated in the associated private message window. If there is currently no private message window open with that remote user, the application automatically opens it and displays the message.

Figure 1.1 - Client Login Screen



Server Address: 127.0.0.1
Server Port: 1000

Figure 1.2 – Main Chat Window

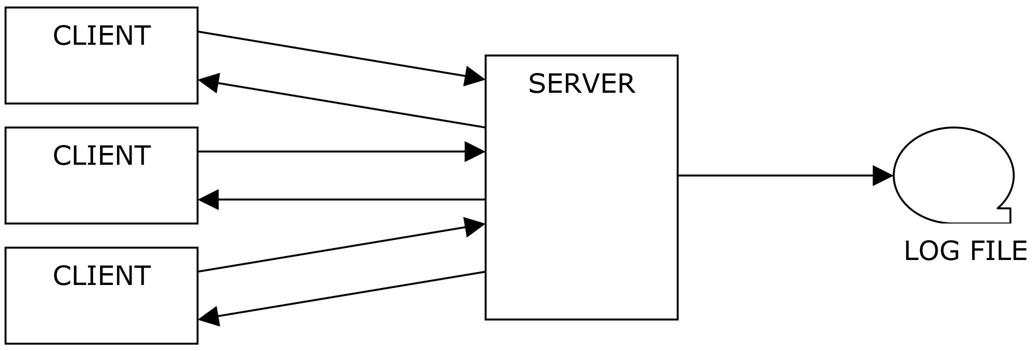


3.2 Network Communication Design

The server application will listen for any new connections from client connections on port 1000. Any new client connections will be added to an internal stack of client connections. All client applications will only communicate to the server and not to other client computers. This ensures that a client system cannot show the remote address of a particular remote user and results in improved security against other users of the application.

Figure 1.3 shows the data flow of the application between the clients and server. Each client will have a two-way communication link (send and receive) with the server. The server is responsible for relaying any messages between clients. The server may also log any events and data to a local file on the server computer.

Figure 1.3 – Data Flow of Application



3.3 Server Application Overview

When the server application is launched, the listen port number can be specified as a parameter during execution of the program. If the port number is not specified then the application uses a default port number of **1000**.

The server will run in a command window and have a text based menu system to navigate through the server commands. *Figure 1.4* shows the initial text menu when the server application is run.

Figure 1.4 – Initial Server Application Menu

```
[1] Start server
[2] Enable logging
[3] Clear log file
[4] View log file
[5] View current users

[x] Exit

Enter option: _
```

Pressing '1' starts the server listening on the designated port. If the server is listening the menu option will state 'Stop server' and will stop the server from listening and close all existing connections.

Pressing '2' will make the server log all activities and traffic. The log will go to **JMessenger.log** for later debugging. If logging is currently enabled then the option will state 'Disable logging' and will stop logging each event.

Pressing '3' will delete the log file on the server.

Pressing '4' will show all the contents of the log file on screen.

Pressing '5' will show a list of all currently connected users on screen.

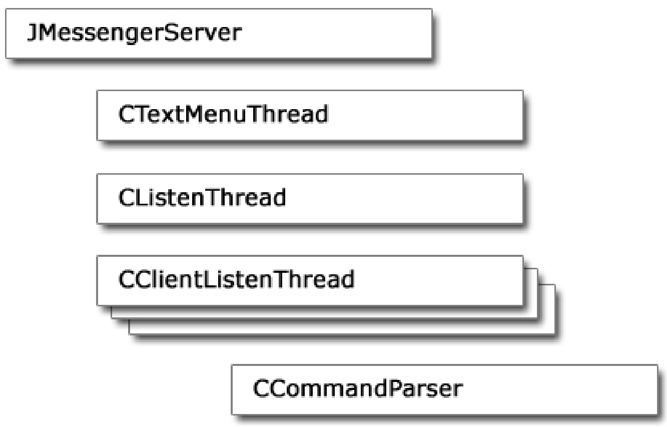
Pressing 'x' will close any active connections and terminate the server application.

4.0 Implementation

4.1 Structure of the Server Application

The server application creates an instance of a menu thread class, a listen thread class monitoring new connections and a list of clients running in their own threads tracking incoming data from individual clients. *Figure 1.5* shows the structure of the server application classes.

Figure 1.5 – Server Class Structure



JMessengerServer is the main application class that create an instance of the text menu, listen thread and stores the client threads. Each sub class uses action events to return data and user responses to the main class ready for processing.

CTextMenuThread displays the server menu options and reads the user response via the keyboard. This class calls the following action events defined in *CMenuListener* that are implemented in *JMessengerServer*:

- onServerRunning
    - Used to start or stop the server from listening to new connections and relaying messages from connected users.
- onShutdown
    - Shuts down the server application and exits back to the operating system.
- onLogging
    - Used to enable or disable server logging.
- onLogFileClear
    - Used when the user requests to clear the log file.
- onViewLogFile
    - Used when the user requests to view the log file.
- onShowConnectedUsers
    - Used when the user requests to view all connected users.

CListenThread listens for incoming connections from remote clients. The class listens for a connection for a 10th of a second and then loops. The timeout is set to a 100 milliseconds to enable the application to stop listening if required. This class calls the following action events defined in *CListenListener* that are implemented in *JMessengerServer*:

- onListen

  - Used to inform the application that the server socket is listening on the specified port.
- onClose
  - Used to inform the application that the server socked was closed.
- onNewConnection
  - Used to inform the application that a new client connection has been made.
- onListenError
  - Used to inform the application that there was an error relating to the server socket listen thread.

CClientListenThread handles all the individual operations relating to a particular client connection. This class will deal with incoming data from the client and will also send data to the client. This class calls the following action events defined in *CClientListenListener* and are implemented in *JMessengerServer*:

- onUserAuthenticate
  - Used when a user has requested to authenticate a particular username.
- onUserLeave
  - Used when a user has disconnected from the server.
- onUserSendMessage
  - Used when a user has sent a message to another particular user.
- onUserSendMessageToAll
  - Used when a user has sent a message to all connected users.

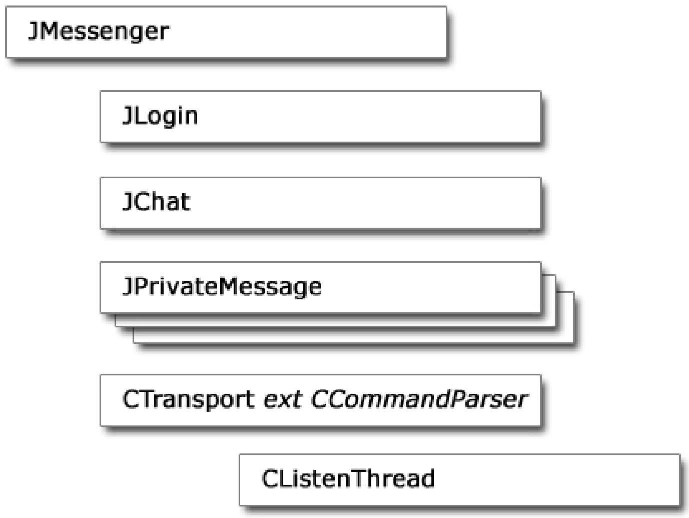
CCommandParser (shared by the server and client applications) is used to extract the data from a command send to or from the server. The commands follow the following specification.

Character	Description
0	This is a response code. It can relate to **+** for success or **–** for failure.
1-2	This is a two digit value relating to the numerical value of the command. **01** states the user is requesting authentication. **02** states that a user has joined the chat room. **03** states that a user has left the chat room. **04** states that a private message has been received. **05** states that a public message has been received. **06** states that the server is shutting down.
3..n	The remaining characters are the data characters. This can either be a username on its own, a message on its own or a combination of a username and message separated by the tilde character (~).

4.2 Structure of the Client Application

The client application creates instances of window classes (*JLogin*, *JChat* and *JPrivateMessage*) and relays information from the Graphical User Interface (GUI) and the network connection through *CTransport*. *Figure 1.6* shows the class structure of the client application.

Figure 1.6 – Client Class Structure



JMessenger is the main application that controls the visual windows and the network connection and processes data to send and data received from the server.

JLogin is the window where the user enters their desired username and specifies the server address and port. This class calls the following action events that are defined in *CLoginListener* and are implemented in *JMessenger*:

- onConnectRequest
  - Called when the user presses the login button to connect to the remote server.
- onLoginCancel
  - Called when the user presses the cancel button on the login screen. Ultimately this will close the application.

JChat is the window that displays the public conversation, displays the list of connected users and allows a user to send a message to all other users. This class calls the following action events that are defined in *CChatListener* and are implemented in *JMessenger*:

- onSendMessageToAll
  - Used when a user wishes to send a message to all other users.
- onInitiatePrivateMessage
  - Used when a user wishes to establish a private message with a remote user. Ultimately this will display a private message window if one is not already open.

JPrivateMessage is the window that displays the private conversation. This window is similar to the chat window except it does not contain a list of connected users. This class calls the following action events that are defined in *CPrivateMessageListener* and are implemented in *JMessenger*:

- onSendMessage
    - Used when the user wishes to send a message to the remote user.
- onClosePrivateMessage
    - Used when a user closes the window. This is required to ensure the main application is aware that the window no longer exists.

CTransport is used to handle the two-way communications with the client application and the remote server. This class implements the *CCommandParser,* which has been described earlier in this documentation. This class calls the following action events defined in *CTransportListener* and are implemented in *JMessenger*:

- onConnect
    - Called when a connection to the remote server has been established.
- onConnectionError
    - Called when there was an error with the connection.
- onUserValidated
    - Called when the server accepted the username.
- onUserRejected
    - Called when the server rejected the username.
- onUserJoin
    - Called when a new user has joined the chat room.
- onUserLeave
    - Called when a user has left the chat room.
- onMessageReceived
    - Called when a private message has been received from the server.
- onMessageReceivedFromAll
    - Called when a public message has been received from the server.
- onSendMessageError
    - Called when there was an error sending the message to a remote user.
- onLostConnection
    - Called when the connection to the remote server was lost.

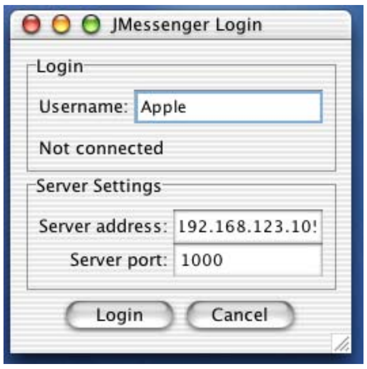
CListenThread is a dedicated thread class that listens for incoming data from the remote server. This process is independent of the rest of the application allowing the user to navigate through the GUI interface and send data to the server. This class calls the following action events defined in *CListenThreadListener* and are implemented in *CTransport*:

- onDataReceived
    - Incoming data was received and is to be send to *CTransport* for interpretation.
- onDataError
    - There was an error receiving data. This may be called when the connection to the server is lost.

5.0 Testing

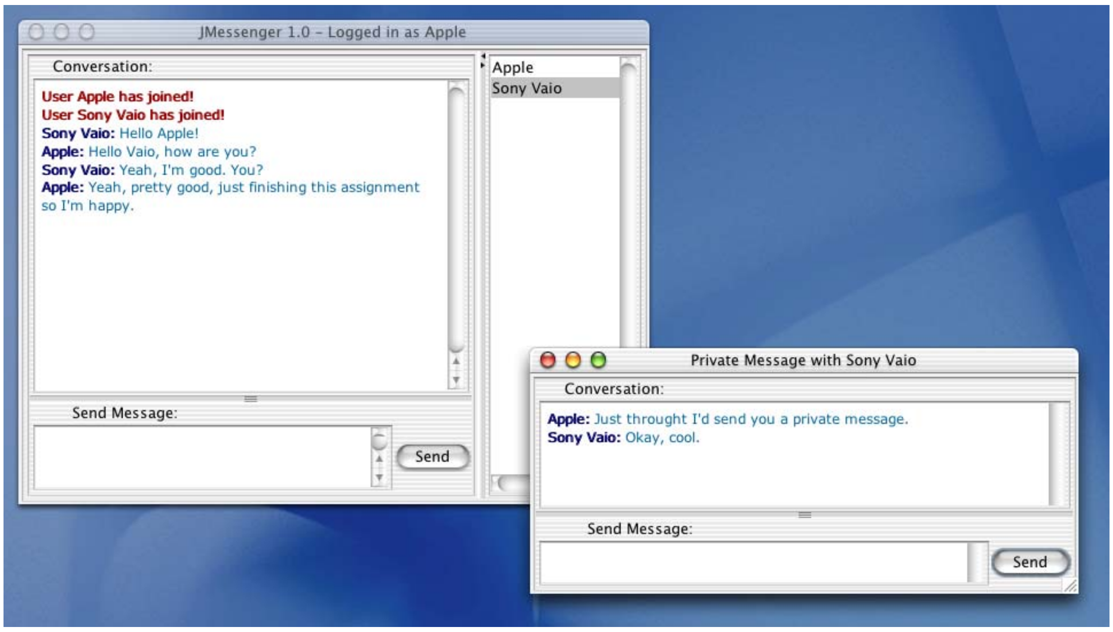
This application has been created to the Java 1.4 specification. The application will work on multiple platforms but may require a few modifications if it is to be compiled on a previous version of the Java SDK. *Figure 1.7* and *Figure 1.8* demonstrates the application working successfully on Mac OS X (Version 10.1.1) compiled with Project Builder.

Figure 1.7 – JMessenger Login Window running on Mac OS X



Login
Username: Apple
Not connected
Server Settings
Server address: 192.168.123.10!
Server port: 1000

Figure 1.8 – JMessenger Conversation Windows running on Mac OS X


JMessenger 1.0 – Logged in as Apple
Conversation:
Send Message:
Send
Private Message with Sony Vaio
Conversation:
Send Message:
Send
User Apple has joined!
User Sony Vaio has joined!
Sony Vaio: Hello Apple!
Apple: Hello Vaio, how are you?
Sony Vaio: Yeah, I'm good. You?
Apple: Yeah, pretty good, just finishing this assignment so I'm happy.

Apple
Sony Vaio

Apple: Just throught I'd send you a private message.
Sony Vaio: Okay, cool.

6.0 Conclusion

After having developed both a client and server application that allows multiple concurrent connections for both public and private messages, there are a few improvements that could be added to improve the application:

1. The server application can be made more user friendly by having a confirmation prompt before clearing the log file.
2. The client application could be improved to support transfer of files.
3. The client application could support emoticons. These a small images use to describe emotions, for example happy (☺). This would be relatively simple to implement due to the conversation window being created through the user of HTML. To display images would simply require a HTML link to the image (either a local file image or an image on a remote web site).
4. Users can create accounts with the server, thus protecting their username from being taken by other users. Connected users could send a private message to a user account. The message can either get send immediately to the user, or if they are not logged in, it can get archived until they next log in.

7.0 Appendix

7.1 Contents of JMessengerServer.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.text.*;

public class JMessengerServer
  implements CMenuListener, CListenListener, CClientListenListener
{
  public static final int DEFAULT_PORT = 1000;
  public static final String LOG_FILE = "JMessenger.log";

  private int port;
  private CMenuThread menu;
  private CListenThread listen;
  private LinkedList clients = new LinkedList();
  private boolean bLogging = false;
  private FileOutputStream log;

  public JMessengerServer(int port)
  {
    if(port == 0)
      this.port = DEFAULT_PORT;
    else
      this.port = port;

    // Create a new text menu
    menu = new CMenuThread(this);
    // Create a listen thread
    listen = new CListenThread(this, this.port);
  }

  public void onServerRunning(boolean enable)
  {
    if(!enable)
    {
      // Inform all clients that the server is stoping
      for(int i = 0; i < clients.size(); i++) {
        ((CClientListenThread)clients.get(i)).shutDown();
      }
      clients.clear();

      writeLog("Server has stopped running");
    }
    else
    {
      writeLog("Server has started running");
    }

    listen.setServerRunState(enable);

  }

  public void onShutdown()
  {
    // Inform all clients that the server is shutting down
    for(int i = 0; i < clients.size(); i++)
      ((CClientListenThread)clients.get(i)).shutDown();
    menu.shutDown();
    listen.shutDown();
    writeLog("Server shut down");
    closeLogFile();
  }
  public void onListen()
  {
    menu.setServerRunState(true);
  }

  public void onClose()
  {
    menu.setServerRunState(false);
  }

  public void onNewConnection(Socket s)
  {
    // Add this connection to the list of users
    clients.add(new CClientListenThread(this, s));
    writeLog("User connected at address " +
s.getRemoteSocketAddress().toString());
  }

  public void onListenError(String description)
  {
    System.out.println("Error: " + description);
    writeLog("Error - " + description);
  }

  public void onUserLeave(CClientListenThread c)
  {
    // Remove this user
    clients.remove(c);

    // Inform all the other users that this user has left provided they
    // were authenticated
    if(!c.username.equals(""))
      for(int i = 0; i < clients.size(); i++)
        ((CClientListenThread)clients.get(i)).userLeft(c.username);
    writeLog("User " + c.username + " has left");
  }

  public void onUserAuthenticate(CClientListenThread c, String username)
  {
    boolean valid = true;
    // The user has request authentication so make sure the username
    // isn't already taken
    for(int i = 0; i < clients.size(); i++)

if(((CClientListenThread)clients.get(i)).username.equalsIgnoreCase(username)
)
        valid = false;

    if(valid)
    {
      c.authenticate(username);
      // Inform all the users that this user has joined
      for(int i = 0; i < clients.size(); i++)
      {
        // Let users who isn't the new one know of the new member

if(!((CClientListenThread)clients.get(i)).username.equalsIgnoreCase(username
))
          ((CClientListenThread)clients.get(i)).userJoined(username);
        // Let the new user know who everyone else is
        c.userJoined(((CClientListenThread)clients.get(i)).username);
      }
      writeLog("user " + username + " has joined");
    }
    else
    {
      c.reject("Username already taken");
      // Remove the client from the list
      clients.remove(c);
      writeLog("User " + username + " was rejected because the name was
already taken");
    }
  }

  public void onUserSendMessage(CClientListenThread c, String to, String
message)
  {
    // Loop through each client until we have the correct user
    for(int i = 0; i < clients.size(); i++)
    {

if(((CClientListenThread)clients.get(i)).username.equalsIgnoreCase(to))
      {
        ((CClientListenThread)clients.get(i)).sendMessage(c.username,
message);
      }
    }
    writeLog(c.username + " sent message to " + to + "\r\n" + message);
  }

  public void onUserSendMessageToAll(CClientListenThread c, String message)
  {
    // Loop through each client and send the message
    for(int i = 0; i < clients.size(); i++)
      ((CClientListenThread)clients.get(i)).sendMessageToAll(c.username,
message);
    writeLog(c.username + " sent to all\r\n" + message);
  }

  public void onLogging(boolean enable)
  {
    bLogging = enable;
    if(bLogging)
    {
      // Open the log file and append contents
      try
      {
        log = new FileOutputStream(LOG_FILE, true);
      }
      catch (IOException e) {
        bLogging = false;
      }
    }
    else
    {
      closeLogFile();
    }
  }

  public void onLogFileClear()
  {
    File f;

    // Close the log file if it is open
    if(bLogging)
      closeLogFile();

    f = new File(LOG_FILE);
    f.delete();

    // Reopen the log file if necessary
    if(bLogging)
    {
      try {
        log = new FileOutputStream(LOG_FILE, true);
      }
      catch (IOException e)
      {
        bLogging = false;
      }
    }
  }

  public void onViewLogFile()
  {
    try
    {
      File f = new File(LOG_FILE);
      int size = (int)f.length();
      FileInputStream file = new FileInputStream(f);
      for (int i = 0; i < size; i++)
        System.out.print((char)file.read());
      file.close();
    }
    catch (IOException e) {
      System.out.println("Could not read " + LOG_FILE);
    }

    menu.anyKeyContinue();
  }

  public void onShowConnectedUsers()
  {
    if(clients.size() == 0)
    {
      System.out.println("There are no users connected");
    }
    else
    {
      // Loop through each client and output their details
      for(int i = 0; i < clients.size(); i++)
      {
        System.out.println(((CClientListenThread)clients.get(i)).username);
      }
    }

    menu.anyKeyContinue();

  }

  private void writeLog(String message)
  {
    if (bLogging)
    {
      String output;
      Date time = new Date();
      output = DateFormat.getDateInstance().format(time) + " " +
               DateFormat.getTimeInstance().format(time) +
               ": " + message + "\r\n";

      try
      {
        log.write(output.getBytes());
      } catch (IOException e) {}
    }
  }

  private void closeLogFile()
  {
    if(bLogging)
    {
      // Close the log file
      try
      {
        log.close();
      }
      catch (IOException e)
      {
        bLogging = true;
      }
    }
  }


  public static void main (String[] args)
  {
    int port = 0;
    if (args.length == 1)
    {
      try
      {
        port = Integer.parseInt (args[0]);
      }
      catch (NumberFormatException e)
      {
        port = 0;
      }
    }

    new JMessengerServer (port);
  }
}
```

7.2 Contents of CMenuThread.java

```
import java.io.*;

class CMenuThread extends Thread
{
  protected CMenuListener listener;

  protected boolean shutdown;
  protected boolean running;
  protected boolean logging;
  protected boolean anykey;

  public CMenuThread(CMenuListener l)
  {
    listener = l;
    shutdown = false;
    running = false;
    logging = false;
    anykey= false;

    this.start();
  }

  public void run()
  {
    String dataIn;
    BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));

    // Keep listening until the user exits
    while(!shutdown)
    {
      if(anykey)
        showPressAnyKey();
      else
        showMenuOptions();
      try
      {
        dataIn = stdIn.readLine();

        if(anykey)
        {
          anykey = false;
        }
        else
        {
          if(dataIn.equals("1"))
            listener.onServerRunning(!running);

          if(dataIn.equals("2"))
          {
            logging = !logging;
            listener.onLogging(logging);
          }

          if(dataIn.equals("3"))
            listener.onLogFileClear();

          if(dataIn.equals("4"))
            listener.onViewLogFile();

          if(dataIn.equals("5"))
            listener.onShowConnectedUsers();

          if(dataIn.equals("x"))
            listener.onShutdown();
        }
      }
      catch (IOException e)
      {
        System.err.println(e);
      }
    }
  }

  public void anyKeyContinue()
  {
    anykey = true;
  }

  private void showPressAnyKey()
  {
    System.out.println("");
    System.out.print("Press enter to continue");
  }

  private void showMenuOptions()
  {
    int i;
    for (i = 1; i < 20; i++)
      System.out.println("");
    System.out.println("JMessengerServer 1.0");
    System.out.println("----------------------------------");

    if(running)
      System.out.println("[1] Stop server");
    else
      System.out.println("[1] Start server");

    if(logging)
      System.out.println("[2] Disable logging");
    else
      System.out.println("[2] Enable logging");

    System.out.println("[3] Clear log file");
    System.out.println("[4] View log file");
    System.out.println("[5] View currently connected users");
    System.out.println("");
    System.out.println("[x] Shut down and exit");

    System.out.println("");
    System.out.print("Enter option: ");
  }


  public void shutDown()
  {
    shutdown = true;
  }

  public void setServerRunState(boolean running)
  {
    this.running = running;
  }
}
```

7.3 Contents of CMenuThreadListener.java

```
public interface CMenuListener
{
  public void onServerRunning(boolean enable);
  public void onShutdown();
  public void onLogging(boolean enable);
  public void onLogFileClear();
  public void onViewLogFile();
  public void onShowConnectedUsers();
}
```

7.4 Contents of CListenThread.java

```
import java.net.*;
import java.io.*;

class CListenThread extends Thread
{
  protected CListenListener listener;

  protected boolean shutdown;
  protected boolean running;
  protected ServerSocket listen_socket;
  protected int port;

  public CListenThread(CListenListener l, int port)
  {
    listener = l;
    shutdown = false;
    running = false;
    this.port = port;

    this.start();
  }

  public void run()
  {
    Socket new_client = null;

    // Keep listening until the user exits
    while(!shutdown)
    {
      if(running)
      {
        // See if there are any new connections
        try
        {
          new_client = listen_socket.accept();
        }
        catch (IOException e)
        {
          new_client = null;
        }
        if(new_client != null)
        {
          // Create the new connection event
          listener.onNewConnection(new_client);
        }
      }
    }
  }

  public void shutDown()
  {
    shutdown = true;
    // Disable the server listen state
    setServerRunState(false);
  }

  public void setServerRunState(boolean running)
  {
    // Only do something if the state has changed
    if(this.running != running)
    {
      if(running)
      {
        // Create the listen socket
        try
        {
          listen_socket = new ServerSocket(port);
        }
        catch(IOException e)
        {
          // There was an error to raise an event
          listener.onListenError(e.getMessage());
        }
        // Set the listen timeout to a 10th of a second
        try
        {
          listen_socket.setSoTimeout(100);
          // This was successful, call the success event
          listener.onListen();
        }
        catch (IOException e)
        {
          // Raise an error event
          listener.onListenError(e.getMessage());
        }
      }
      else
      {
        // Close the listen socket
        try
        {
          listen_socket.close();
          listen_socket = null;
          listener.onClose();
        }
        catch (IOException e)
        {
          // Raise the error
          listener.onListenError(e.getMessage());
        }
      }
      this.running = running;
    }
  }
}
```

7.5 Contents of CListenListener.java

```
import java.net.*;

public interface CListenListener
{
  public void onListen();
  public void onClose();
  public void onNewConnection(Socket s);
  public void onListenError(String description);
}
```

7.6 Contents of CClientListenThread.java

```
import java.net.*;
import java.io.*;

class CClientListenThread extends Thread
{
  // Constants
  public static final int CMD_AUTHENTICATE         = 1;
  public static final int CMD_USER_JOINED          = 2;
  public static final int CMD_USER_LEFT            = 3;
  public static final int CMD_MESSAGE_RECEIVED     = 4;
  public static final int CMD_ALL_MESSAGE_RECEIVED = 5;
  public static final int CMD_SERVER_SHUTDOWN      = 6;

  protected CClientListenListener listener;
  protected Socket conn;
  public String username;
  protected PrintStream out;
  protected BufferedReader in;
  protected boolean running;
  protected CCommandParser cmd = new CCommandParser();

  public CClientListenThread(CClientListenListener l, Socket s)
  {
    cmd.clientProtocol(false);
    listener = l;
    conn   = s;
    running = true;
    username = "";
    // Create an output stream
    try
    {
      in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
      out = new PrintStream(conn.getOutputStream());
    }
    catch(IOException e)
    {
      try
      {
        conn.close();
      }
      catch (IOException e2) {}
      System.err.println("Exception while getting socket streams");
      return;
    }

    this.start();
  }

  public void run()
  {
    String buffer = "";

    while(running)
    {
      try
      {
        buffer = in.readLine();

        // Make sure the connection wasn't dropped
        if(buffer == null)
        {
          try
          {
            conn.close();
          }
          catch (IOException e3) {}
          conn = null;
          // Call the user left event
          listener.onUserLeave(this);
          return;
        }
        // Process the data received
        cmd.parseData(buffer);
        switch(cmd.getCommand())
        {
          case CMD_AUTHENTICATE:
          {
            listener.onUserAuthenticate(this, cmd.getUser());
            break;
          }
          case CMD_MESSAGE_RECEIVED:
          {
            listener.onUserSendMessage(this, cmd.getUser(), cmd.getData());
            break;
          }
          case CMD_ALL_MESSAGE_RECEIVED:
          {
            listener.onUserSendMessageToAll(this, cmd.getData());
            break;
          }
        }
      }
      catch (IOException e)
      {
        // There was an error so close the connection
        // to this client
        try
        {
          if(conn != null)
            conn.close();
        }
        catch (IOException e2) {}
        conn = null;
        // Call the user left event
       if(running)
          listener.onUserLeave(this);
        running = false;
      }
    }
  }

  public void sendMessage(String from, String message)
  {
    sendCommand(true, CMD_MESSAGE_RECEIVED, from + "~" + message);
  }

  public void sendMessageToAll(String from, String message)
  {
    sendCommand(true, CMD_ALL_MESSAGE_RECEIVED, from + "~" + message);
  }

  public void sendCommand(boolean valid, int id, String data)
  {
    String command;

    if(valid)
      command = "+";
    else
      command = "-";

    if(id < 10)
      command += "0" + id;
    else
      command += id;
    command += data;

    out.println(command);

  }

  public void authenticate(String username)
  {
    this.username = username;
    sendCommand(true, CMD_AUTHENTICATE, "");
  }

  public void reject(String reason)
  {
    sendCommand(false, CMD_AUTHENTICATE, reason);
  }

  public void userJoined(String username)
  {
    sendCommand(true, CMD_USER_JOINED, username);
  }

  public void userLeft(String username)
  {
    sendCommand(true, CMD_USER_LEFT, username);
  }

  public void shutDown()
  {
    // Inform the client of the shutdown
    running = false;
    sendCommand(true, CMD_SERVER_SHUTDOWN, "");
    // Close the connection
    try
    {
      conn.close();
    }
    catch (IOException e) {}
    conn = null;
  }
}
```

7.7 Contents of CClientListenListener.java

```
public interface CClientListenListener
{
  public void onUserLeave(CClientListenThread c);
  public void onUserAuthenticate(CClientListenThread c, String username);
  public void onUserSendMessage(CClientListenThread c, String to, String
message);
  public void onUserSendMessageToAll(CClientListenThread c, String message);
}
```

7.8 Contents of CCommandParser.java

```
class CCommandParser
{
  // Constants
  public static final int CMD_AUTHENTICATE         = 1;
  public static final int CMD_USER_JOINED          = 2;
  public static final int CMD_USER_LEFT            = 3;
  public static final int CMD_MESSAGE_RECEIVED     = 4;
  public static final int CMD_ALL_MESSAGE_RECEIVED = 5;
  public static final int CMD_SERVER_SHUTDOWN      = 6;

  private boolean bFromServer;
  private boolean bValid;
  private int iCommand;
  private String sUsername;
  private String sData;

  public void clientProtocol(boolean isClient)
  {
    bFromServer = isClient;
  }

  public void parseData(String sData)
  {
    boolean bGetUser, bGetData;

    bGetUser = false;
    bGetData = false;

    // Clear the variables
    bValid = false;
    iCommand = 0;
    sUsername = "";
    this.sData = "";

    // First get the valid code (+ or -)
    bValid = sData.substring(0,1).equalsIgnoreCase("+");

    // Now get the code
    iCommand = Integer.parseInt(sData.substring(1,3));

    // Test the command and see what data to extract
    if(!bFromServer && iCommand == CMD_AUTHENTICATE)
      bGetUser = true;
    if(bFromServer && !bValid && iCommand == CMD_AUTHENTICATE)
      bGetData = true;
    if(bFromServer && bValid && iCommand == CMD_USER_JOINED)
      bGetUser = true;
    if(bFromServer && bValid && iCommand == CMD_USER_LEFT)
      bGetUser = true;
    if(iCommand == CMD_MESSAGE_RECEIVED ||
       (iCommand == CMD_ALL_MESSAGE_RECEIVED && bFromServer))
    {
      bGetUser = true;
      bGetData = true;
    }
    if(iCommand == CMD_ALL_MESSAGE_RECEIVED && ! bFromServer)
      bGetData = true;

  // If we only want the user then extract the user details
    if(bGetUser && !bGetData)
    {
      sUsername = sData.substring(3, sData.length());
    }
    // If we only want the data then extract the data
    if(!bGetUser && bGetData)
    {
      this.sData = sData.substring(3, sData.length());
    }

    // If we want both the username and data then get them both
    if(bGetUser && bGetData)
    {
      int iPos;
      iPos = sData.indexOf("~");
      if(iPos > 0)
      {
        sUsername = sData.substring(3, iPos);
        this.sData = sData.substring(iPos + 1, sData.length());
      }
    }
  }

  public boolean isValid()
  {
    return bValid;
  }

  public int getCommand()
  {
    return iCommand;
  }

  public String getUser()
  {
    return sUsername;
  }

  public String getData()
  {
    return sData;
  }
}
```

7.9 Contents of JMessenger.java

```
import javax.swing.*;
import java.util.*;

public class JMessenger implements
  CTransportListener, CChatListener, CLoginListener,
  CPrivateMessageListener
{
  private CTransport conn;
  private JLogin login = null;
  private String username;
  private JChat chat = null;
  private LinkedList privateMessages = new LinkedList();

  public JMessenger()
  {
    // Create a new connection
    conn = new CTransport(this);
    login = new JLogin(this, "JMessenger Login");
    chat = new JChat(this, "JMessenger 1.0");

    // Show the login window
    login.setVisible(true);
  }

  public void onConnect()
  {
    conn.authenticateUser(username);
  }

  public void onConnectError(int id, String description)
  {
    login.setStatus(description);
  }

  public void onUserValidated()
  {
    // Show the conversation window
    login.setVisible(false);
    chat.setTitle("JMessenger 1.0 - Logged in as " + username);
    chat.showChat();
  }

  public void onUserRejected(String reason)
  {
    login.setStatus(reason);
  }

  public void onUserJoin(String username)
  {
    chat.userJoined(username);
  }

  public void onUserLeave(String username)
  {
    chat.userLeft(username);
    // See if there were any private messages to this
    // user to let them know the user left
    for(int i = 0; i < privateMessages.size(); i++)
    {

if(((JPrivateMessage)privateMessages.get(i)).remoteUsername.equalsIgnoreCase
(username))
      {
        ((JPrivateMessage)privateMessages.get(i)).userLeft();
        break;
      }
    }
  }

  public void onMessageReceived(String from, String message)
  {
    JPrivateMessage pm;

    boolean found = false;
    // See if there is already a private message to this user
    // and if so post the message to the window
    for(int i = 0; i < privateMessages.size(); i++)
    {

if(((JPrivateMessage)privateMessages.get(i)).remoteUsername.equalsIgnoreCase
(from))
      {
        found = true;
        ((JPrivateMessage)privateMessages.get(i)).messageReceived(message);
      }
    }

    if(!found)
    {
      // Create a new private message window and post the message

      pm = new JPrivateMessage(this, "Private Message with " + from,
this.username, from);
      pm.messageReceived(message);
      privateMessages.add(pm);
    }

  }

  public void onMessageReceivedFromAll(String from, String message)
  {
    chat.messageReceived(from, message);
  }

  public void onSendMessageError(int id, String description)
  {
    // Used for debug purposes
  }

  public void onLostConnection()
  {
    // We have lost the connection to the server, hide any visible windows
    // and show the login screen again
    chat.setVisible(false);
    // Close all private message windows
    for(int i = 1; i < privateMessages.size(); i++)
      ((JPrivateMessage)privateMessages.get(i)).dispose();
    privateMessages.clear();

    login.setVisible(true);
    login.setStatus("Lost server connection");
 }

  public void onSendMessageToAll(String message)
  {
    conn.sendMessageToAll(message);
  }

  public void onSendMessage(String to, String message)
  {
    conn.sendMessage(to, message);
  }

  public void onInitiatePrivateMessage(String username)
  {
    // Make sure the user isn't trying to talk to themselves
    if(!username.equalsIgnoreCase(this.username))
    {
      boolean found = false;
      // See if there is already a private message to this user
      // and if so, bring it to the front
      for(int i = 0; i < privateMessages.size(); i++)
      {

if(((JPrivateMessage)privateMessages.get(i)).remoteUsername.equalsIgnoreCase
(username))
        {
          found = true;
          ((JPrivateMessage)privateMessages.get(i)).toFront();
        }
      }

      if(!found)
      {
        // Create a new private message window
        privateMessages.add(new JPrivateMessage(this, "Private Message with
" + username, this.username, username));
      }
    }
  }

  public void onClosePrivateMessage(String remoteUsername)
  {
    // Remote the private message
    for(int i = 0; i < privateMessages.size(); i++)
    {
      // See if this is the one to remove

if(((JPrivateMessage)privateMessages.get(i)).remoteUsername.equalsIgnoreCase
(remoteUsername))
      {
        // Dispose of the window and remove it
        ((JPrivateMessage)privateMessages.get(i)).dispose();
        privateMessages.remove(i);
        break;
      }
    }  }

  public void onConnectRequest(String host, int port, String username)
  {
    login.setStatus("Connecting...");
    this.username = username;
    conn.connect(host, port);
  }

  public void onLoginCancel()
  {
    System.exit(0);
  }

  public static void main(String[] args)
  {
    // Create the main application
    JMessenger main = new JMessenger();
  }
}
```

7.10 Contents of JLogin.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.net.*;
import java.io.*;

public class JLogin extends JFrame implements ActionListener
{
  JTextField txtUsername, txtServer, txtPort;
  JLabel     lblStatus;
  JButton    btnLogin, btnCancel;
  Socket     sckConnect;
  boolean    bConnected = false;
  private    CLoginListener listener;

  public JLogin(CLoginListener l, String title)
  {
    super(title);

    listener = l;
    screenSetup();

    pack();

    // Treat pressing the [x] as pressing cancel
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        listener.onLoginCancel();
      }
    });
  }

  public void screenSetup()
  {
    JLabel lblUsername = new JLabel("Username: ");
    txtUsername = new JTextField(10);
    lblUsername.setLabelFor(txtUsername);
    lblStatus = new JLabel("Not connected");
    lblStatus.setBorder(BorderFactory.createEmptyBorder(10,0,0,0));

    // Create a frame pane for the login details
    JPanel loginControlsPane = new JPanel();
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();

    loginControlsPane.setLayout(gridbag);

    JLabel[] labels = {lblUsername};
    JTextField[] textFields = {txtUsername};
    addLabelTextRows(labels, textFields, gridbag, loginControlsPane);

    c.gridwidth = GridBagConstraints.REMAINDER;
    c.anchor = GridBagConstraints.WEST;
    c.weightx = 1.0;
    gridbag.setConstraints(lblStatus, c);
    loginControlsPane.add(lblStatus);
    loginControlsPane.setBorder(
      BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder("Login"),
        BorderFactory.createEmptyBorder(5,5,5,5)));

    // Create the server settings controls
    JLabel lblServer = new JLabel("Server address: ");
    txtServer = new JTextField("127.0.0.1", 10);
    JLabel lblPort = new JLabel("Server port: ");
    txtPort = new JTextField("1000", 5);
    lblServer.setLabelFor(txtServer);
    lblPort.setLabelFor(txtPort);

    // Create a frame pane for the server details
    JPanel serverControlsPane = new JPanel();
    GridBagLayout gridbag2 = new GridBagLayout();
    GridBagConstraints c2 = new GridBagConstraints();

    serverControlsPane.setLayout(gridbag2);

    JLabel[] labels2 = {lblServer, lblPort};
    JTextField[] textFields2 = {txtServer, txtPort};
    addLabelTextRows(labels2, textFields2, gridbag2, serverControlsPane);

    c2.gridwidth = GridBagConstraints.REMAINDER;
    c2.anchor = GridBagConstraints.WEST;
    c2.weightx = 1.0;
    serverControlsPane.setBorder(
      BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder("Server Settings"),
        BorderFactory.createEmptyBorder(5,5,5,5)));

    // Create the login and cancel buttons
    btnLogin = new JButton("Login");
    btnLogin.addActionListener(this);
    btnCancel = new JButton("Cancel");
    btnCancel.addActionListener(this);

    // Create a pane and add the buttons next to each other
    JPanel buttonPane = new JPanel();
    buttonPane.add(btnLogin);
    buttonPane.add(btnCancel);

    JPanel pane = new JPanel();
    BoxLayout mainbox = new BoxLayout(pane, BoxLayout.Y_AXIS);
    pane.setLayout(mainbox);
    pane.add(loginControlsPane);
    pane.add(serverControlsPane);
    pane.add(buttonPane);
    pane.setBorder(
      BorderFactory.createCompoundBorder(
        BorderFactory.createEmptyBorder(10,10,10,10),
        BorderFactory.createEmptyBorder(0,0,0,0)));
    setContentPane(pane);
  }

  private void addLabelTextRows(JLabel[] labels,
                                JTextField[] textFields,
                                GridBagLayout gridbag,
                                Container container)
  {
    GridBagConstraints c = new GridBagConstraints();
    c.anchor = GridBagConstraints.EAST;
    int numLabels = labels.length;

    for (int i = 0; i < numLabels; i++)
    {
      c.gridwidth = GridBagConstraints.RELATIVE;
      c.fill = GridBagConstraints.NONE;
      c.weightx = 0.0;
      gridbag.setConstraints(labels[i], c);
      container.add(labels[i]);

      c.gridwidth = GridBagConstraints.REMAINDER;
      c.fill = GridBagConstraints.HORIZONTAL;
      c.weightx = 1.0;
      gridbag.setConstraints(textFields[i], c);
      container.add(textFields[i]);
    }
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == btnCancel)
    {
      listener.onLoginCancel();
    }
    if(e.getSource() == btnLogin)
    {
      int port = 0;
      // Make sure there is a username
      if(txtUsername.getText().equals(""))
      {
        JOptionPane.showMessageDialog(null, "Username required", "Login
error", JOptionPane.ERROR_MESSAGE);
      }
      else
      {
        try
        {
          port = Integer.parseInt(txtPort.getText());
          listener.onConnectRequest(txtServer.getText(), port,
txtUsername.getText());
        }
        catch(NumberFormatException err)
        {
          JOptionPane.showMessageDialog(null, "Invalid port number", "Login
error", JOptionPane.ERROR_MESSAGE);
        }
      }
    }
  }

  public void setStatus(String status)
  {
    lblStatus.setText(status);
  }
}
```

7.11 Contents of CLoginListener.java

```
public interface CLoginListener
{
  public void onConnectRequest(String host, int port, String username);
  public void onLoginCancel();
}
```

7.12 Contents of JChat.java

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class JChat extends JFrame implements ActionListener
{
  private JEditorPane   txtConversation = new JEditorPane();
  private JTextArea     txtSend = new JTextArea();
  private JButton       btnSend = new JButton("Send");
  private JList         lstUsers;
  private CChatListener listener;
  private JScrollPane   scrlConversation;
  private DefaultListModel lstUserList = new DefaultListModel();

  // Menu items
  private JMenuBar topMenu;
  private JMenu loginMenu, helpMenu;
  private JMenuItem menuItemLogin, menuItemExit, menuItemAbout;

  // Message Data items
  private String messageHeader = "<html><body bgcolor=\"#FFFFFF\"
text=\"#000000\" " +
                                 "leftmargin=\"5\" topmargin=\"5\"
marginwidth=\"5\" marginheight=\"5\">";

  private String messageFooter = "</body></html>";
  private String messages = "";

  public JChat(CChatListener l, String title)
  {
    super(title);

    listener = l;

    // Setup the controls
    controlsSetup();
    // Setup the menu
    menuSetup();
    pack();

    // Treat pressing the [x] as logging out
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
  }

  private void controlsSetup()
  {
    // Create the conversation label
    JLabel lblConversation = new JLabel("Conversation:");

    // Create the conversation text area
    txtConversation.setEditable(false);
    txtConversation.setContentType("text/html");
    txtConversation.setText(messageHeader + messageFooter);
    // Create the conversation scroll pane
    scrlConversation = new JScrollPane(txtConversation);
    scrlConversation.setVerticalScrollBarPolicy(
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    scrlConversation.setPreferredSize(new Dimension(400,300));
    scrlConversation.setMinimumSize(new Dimension(10, 10));
    // Create the lefttop pane
    JPanel lefttopPane = new JPanel();
    BoxLayout lefttopBox = new BoxLayout(lefttopPane, BoxLayout.Y_AXIS);
    lefttopPane.setLayout(lefttopBox);
    lefttopPane.add(lblConversation);
    lefttopPane.add(scrlConversation);

    // Create the send label
    JLabel lblSend = new JLabel("Send Message:");

    // Create the send text scroll pane and the send button
    JScrollPane scrlSend = new JScrollPane(txtSend);
    scrlSend.setVerticalScrollBarPolicy(
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    scrlSend.setPreferredSize(new Dimension(330, 30));
    scrlSend.setMinimumSize(new Dimension(10, 10));

    // Set the send button properties
    btnSend.setPreferredSize(new Dimension(70,30));
    btnSend.setMinimumSize(new Dimension(10, 10));
    btnSend.addActionListener(this);

    // Set the send pane
    JPanel sendPane = new JPanel();
    BoxLayout sendBox = new BoxLayout(sendPane, BoxLayout.X_AXIS);
    sendPane.setLayout(sendBox);
    sendPane.add(scrlSend);
    sendPane.add(btnSend);

    // Create the leftbottom pane
    JPanel leftbottomPane = new JPanel();
    BoxLayout leftBottomBox = new BoxLayout(leftbottomPane,
BoxLayout.Y_AXIS);
    leftbottomPane.setLayout(leftBottomBox);
    leftbottomPane.add(lblSend);
    leftbottomPane.add(sendPane);

    // Create the left split pane
    JSplitPane leftSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                                              lefttopPane,
                                              leftbottomPane);
    leftSplitPane.setOneTouchExpandable(false);
    leftSplitPane.setDividerLocation(300);
    leftSplitPane.setPreferredSize(new Dimension(400, 400));

    // Create the list on the right hand side
    lstUsers = new JList(lstUserList);
    lstUsers.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JScrollPane scrlContacts = new JScrollPane(lstUsers);
    scrlContacts.setVerticalScrollBarPolicy(
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    scrlContacts.setHorizontalScrollBarPolicy(
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
    scrlContacts.setPreferredSize(new Dimension(150, 400));
    scrlContacts.setMinimumSize(new Dimension(10, 10));

    // Create a split pane between the conversation and the list
    JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                          leftSplitPane,
                                          scrlContacts);
    splitPane.setOneTouchExpandable(true);
    splitPane.setDividerLocation(400);
    splitPane.setPreferredSize(new Dimension(550, 400));

    setContentPane(splitPane);

    // Setup a mouse listener to trap double clicks
    // on a list item
    MouseListener mouseListener = new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        if (e.getClickCount() == 2) {
          int index = lstUsers.locationToIndex(e.getPoint());
          listener.onInitiatePrivateMessage((String)lstUserList.get(index));
        }
      }
   };
   lstUsers.addMouseListener(mouseListener);

  }

  public void menuSetup()
  {
    topMenu = new JMenuBar();
    setJMenuBar(topMenu);
    loginMenu = new JMenu("Login");
    menuItemExit = new JMenuItem("Exit");
    loginMenu.add(menuItemExit);
    menuItemExit.addActionListener(this);
    topMenu.add(loginMenu);
    helpMenu = new JMenu("Help");
    menuItemAbout = new JMenuItem("About...");
    menuItemAbout.addActionListener(this);
    helpMenu.add(menuItemAbout);
    topMenu.add(helpMenu);
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == btnSend)
    {
      // Replace all new lines with <br> tags when we send the data
      listener.onSendMessageToAll(txtSend.getText().replaceAll("\n",
"<br>"));
      txtSend.setText("");
    }
    if(e.getSource() == menuItemExit)
    {
      System.exit(0);
    }
    if(e.getSource() == menuItemAbout)
    {
      JOptionPane.showMessageDialog(null, "Created by Martin Adams",
"JMessenger", JOptionPane.INFORMATION_MESSAGE);
    }
  }

  public void showChat()
  {
    // Clear the list and reset the message header
    lstUserList.clear();
    messages = "";
    txtConversation.setText(messageHeader + messageFooter);
    txtSend.setText("");

    setVisible(true);
  }

  public void messageReceived(String from, String message)
  {
   addMessage("<b><font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#000066\">" +
               from + ": </font></b>" +
              "<font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#006699\">" +
              message + "</font><br>");
  }

  private void addMessage(String message)
  {
    messages += message;
    txtConversation.setText(messageHeader + messages + messageFooter);
    txtConversation.selectAll();

txtConversation.setCaretPosition(txtConversation.getSelectedText().length())
;
  }

  public void userJoined(String username)
  {
    lstUserList.addElement(username);
   addMessage("<b><font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#990000\">" +
               "User " + username + " has joined!</font></b><br>");
  }

  public void userLeft(String username)
  {
    for(int i = 0; i < lstUserList.getSize(); i++)
    {
      if(((String)lstUserList.get(i)).equalsIgnoreCase(username))
      {
        lstUserList.remove(i);
        break;
      }
    }
   addMessage("<b><font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#990000\">" +
               "User " + username + " has left!</font></b><br>");
  }
}
```

7.13 Contents of CChatListener.java

```
public interface CChatListener
{
  public void onSendMessageToAll(String message);
  public void onInitiatePrivateMessage(String username);
}
```

7.14 Contents of JPrivateMessage.java

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class JPrivateMessage extends JFrame implements ActionListener
{
  private JEditorPane   txtConversation = new JEditorPane();
  private JTextArea     txtSend = new JTextArea();
  private JButton       btnSend = new JButton("Send");
  private CPrivateMessageListener listener;
  private JScrollPane   scrlConversation;
  public  String localUsername;
  public  String remoteUsername;

  // Message Data items
  private String messageHeader = "<html><body bgcolor=\"#FFFFFF\"
text=\"#000000\" " +
                                 "leftmargin=\"5\" topmargin=\"5\"
marginwidth=\"5\" marginheight=\"5\">";

  private String messageFooter = "</body></html>";
  private String messages = "";


  public JPrivateMessage(CPrivateMessageListener l, String title, String
localUser, String remoteUser)
  {
    super(title);

    listener = l;
    localUsername = localUser;
    remoteUsername = remoteUser;

    // Setup the controls
    controlsSetup();
    pack();

    // Catch pressing the [x] and call the close event
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        listener.onClosePrivateMessage(remoteUsername);
      }
    });

    setVisible(true);
  }

  private void controlsSetup()
  {
    // Create the conversation label
    JLabel lblConversation = new JLabel("Conversation:");

    // Create the conversation text area
    txtConversation.setEditable(false);
    txtConversation.setContentType("text/html");
    txtConversation.setText(messageHeader + messageFooter);
    // Create the conversation scroll pane
    scrlConversation = new JScrollPane(txtConversation);
    scrlConversation.setVerticalScrollBarPolicy(
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    scrlConversation.setPreferredSize(new Dimension(400,300));
    scrlConversation.setMinimumSize(new Dimension(10, 10));
    // Create the lefttop pane
    JPanel lefttopPane = new JPanel();
    BoxLayout lefttopBox = new BoxLayout(lefttopPane, BoxLayout.Y_AXIS);
    lefttopPane.setLayout(lefttopBox);
    lefttopPane.add(lblConversation);
    lefttopPane.add(scrlConversation);

    // Create the send label
    JLabel lblSend = new JLabel("Send Message:");

    // Create the send text scroll pane and the send button
    JScrollPane scrlSend = new JScrollPane(txtSend);
    scrlSend.setVerticalScrollBarPolicy(
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    scrlSend.setPreferredSize(new Dimension(330, 30));
    scrlSend.setMinimumSize(new Dimension(10, 10));

    // Set the send button properties
    btnSend.setPreferredSize(new Dimension(70,30));
    btnSend.setMinimumSize(new Dimension(10, 10));
    btnSend.addActionListener(this);

    // Set the send pane
    JPanel sendPane = new JPanel();
    BoxLayout sendBox = new BoxLayout(sendPane, BoxLayout.X_AXIS);
    sendPane.setLayout(sendBox);
    sendPane.add(scrlSend);
    sendPane.add(btnSend);

    // Create the leftbottom pane
    JPanel leftbottomPane = new JPanel();
    BoxLayout leftBottomBox = new BoxLayout(leftbottomPane,
BoxLayout.Y_AXIS);
    leftbottomPane.setLayout(leftBottomBox);
    leftbottomPane.add(lblSend);
    leftbottomPane.add(sendPane);

    // Create the left split pane
    JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                                          lefttopPane,
                                          leftbottomPane);
    splitPane.setOneTouchExpandable(false);
    splitPane.setDividerLocation(300);
    splitPane.setPreferredSize(new Dimension(400, 400));

    setContentPane(splitPane);
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == btnSend)
    {
      String message = txtSend.getText().replaceAll("\n", "<br>");
      // Replace all new lines with <br> tags when we send the data
      listener.onSendMessage(remoteUsername, message);
      // Add the local message to the conversation window
      addMessage("<b><font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#000066\">" +
               localUsername + ": </font></b>" +
              "<font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#006699\">" +
              message + "</font><br>");

      txtSend.setText("");
    }
  }

  public void messageReceived(String message)
  {
   addMessage("<b><font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#000066\">" +
               remoteUsername + ": </font></b>" +
              "<font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#006699\">" +
              message + "</font><br>");
  }

  public void userLeft()
  {
   addMessage("<b><font face=\"Verdana, Arial, Helvetica, sans-serif\"
size=\"2\" color=\"#990000\">" +
               "User " + remoteUsername + " has left!</font></b><br>");
    // Disable the send button
    btnSend.setEnabled(false);
  }

  private void addMessage(String message)
  {
    messages += message;
    txtConversation.setText(messageHeader + messages + messageFooter);
    txtConversation.selectAll();

txtConversation.setCaretPosition(txtConversation.getSelectedText().length())
;
 }
}
```

7.15 Contents of CPrivateMessageListener.java

```
public interface CPrivateMessageListener
{
  public void onSendMessage(String to, String message);
  public void onClosePrivateMessage(String remoteUsername);
}
```

7.16 Contents of CTransport.java

```
import java.net.*;
import java.io.*;

public class CTransport extends CCommandParser implements
CListenThreadListener
{
  private String username;

  // Listerner variables
  private CTransportListener listener;

  private CListenThread listen_thread = null;
  private Socket conn = null;
  private PrintStream out;

  CTransport( CTransportListener l )
  {
    clientProtocol(true);
    listener = l;
  }

  public void connect(String remoteHost, int port)
  {
    // Connect to the remote client
    try
    {
      conn = new Socket(remoteHost, port);
      // Create a listen thread
      listen_thread = new CListenThread(this, new
InputStreamReader(conn.getInputStream()));
      // Create an output stream
      out = new PrintStream(conn.getOutputStream());
      listener.onConnect();
    }
    catch(IOException e)
    {
      listener.onConnectError(1, e.getMessage());
    }
  }

  public void authenticateUser (String username)
  {
    this.username = username;
    sendCommand(CMD_AUTHENTICATE, username);
  }

  public void sendCommand(int id, String data)
  {
    if(id < 10)
      out.println("+0" + id + data);
    else
      out.println("+" + id + data);
  }

  public void sendMessage(String username, String message)
  {
    sendCommand(CMD_MESSAGE_RECEIVED, username + "~" + message);
  }

  public void sendMessageToAll(String message)
  {
    sendCommand(CMD_ALL_MESSAGE_RECEIVED, message);
  }

  public void onDataReceived(String data)
  {
    parseData(data);
    switch(getCommand())
    {
      case CMD_AUTHENTICATE:
      {
        if(isValid())
        {
          listener.onUserValidated();
        }
        else
        {
          // The user was rejected so close the
          // connection
          try
          {
            conn.close();
          }
          catch(IOException e) { }
          listener.onUserRejected(getData());
        }
        break;
      }
      case CMD_USER_JOINED:
      {
        listener.onUserJoin(getUser());
        break;
      }
      case CMD_USER_LEFT:
      {
        listener.onUserLeave(getUser());
        break;
      }
      case CMD_MESSAGE_RECEIVED:
      {
        listener.onMessageReceived(getUser(), getData());
        break;
      }
      case CMD_ALL_MESSAGE_RECEIVED:
      {
        listener.onMessageReceivedFromAll(getUser(), getData());
        break;
      }
      case CMD_SERVER_SHUTDOWN:
      {
        listener.onLostConnection();
        break;
      }
    }
  }

  public void onDataError(int id, String description)
  {
    listener.onLostConnection();
  }
}
```

7.17 Contents of CTransportListener.java

```
public interface CTransportListener
{
  public void onConnect();
  public void onConnectError(int id, String description);
  public void onUserValidated();
  public void onUserRejected(String reason);
  public void onUserJoin(String username);
  public void onUserLeave(String username);
  public void onMessageReceived(String from, String message);
  public void onMessageReceivedFromAll(String from, String message);
  public void onSendMessageError(int id, String description);
  public void onLostConnection();
}
```

7.18 Contents of CListenThread.java

```
import java.io.*;

class CListenThread extends Thread
{
  private BufferedReader in;
  private CListenThreadListener listener;

  public CListenThread(CListenThreadListener listener, InputStreamReader in)
  {
    this.listener = listener;
    this.in = new BufferedReader(in);
    this.start();
  }

  public void run()
  {
    String data;

    // Keep receiving data and posting it back
    try
    {
      for(;;)
      {
        data = in.readLine();
        if(data == null)
        {
          listener.onDataError(2, "Connection lost");
          return;
        }
        else
        {
          listener.onDataReceived(data);
        }
      }
    }
    catch (IOException e)
    {
      listener.onDataError(2, e.getMessage());
    }
  }
}
```

7.19 Contents of CListenThreadListener.java

```
public interface CListenThreadListener
{
  public void onDataReceived(String data);
  public void onDataError(int id, String description);
}
```