FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Electrical Engineering Department

Telecommunication Systems

2012/2013

Laboratory Work 0:
Demonstration of the Java environment
Learning application development

Class 3 – Aplication with connection oriented sockets

Integrated Master in Electrical Engineering and Computers

Luis Bernardo
Paulo da Fonseca Pinto

Index

1. OBJETIVE

Familiarization with the Java programming language and applications using connection oriented sockets, developed in the NetBeans environment. The work consists in the introduction of part of the code following the instructions set out, learning to use the environment and the set of library classes from the Java language. A project is provided with the start of the work, which is completed in a set of exercises.

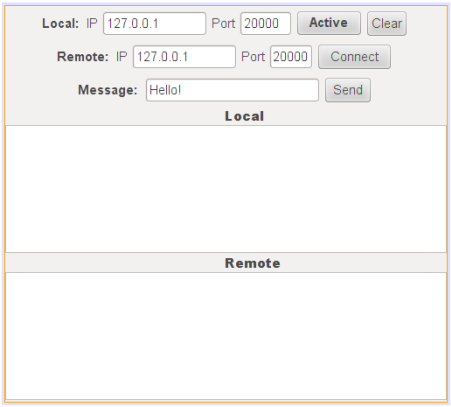
2. THIRD APPLICATION – NETWORK CHAT WITH TCP

This section illustrates the development of an application using the connection oriented sockets (TCP). The application supports the exchange of messages and files over the network, where each participant has a window which receives messages from other elements (*Remote*) and a window where he types its messages (*Local*). The user specifies the IP address and port number of the machine he wants to connect, connecting or accepting a connection from another user, and he may terminate the connection.

2.1. BASIC CHATTCP

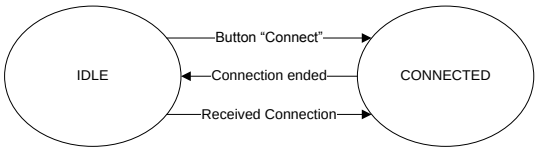
The first version only supports simple exchange of messages between users and a connection to one user each time. Students are provided with the complete NetBeans project of this first part.

This project uses practically the same GUI of the example of the previous class (*ChatUDP*) and associated graphics functions, except that it adds a toggle button ("*Connect*"), associated to `jToggleButtonConnect`, which was placed on the Remote panel, as is pictured in the right. The name of the main window was changed to "*Chat TCP*".



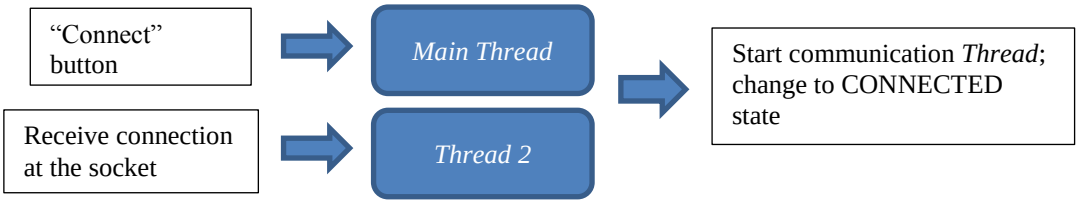
While the GUI is almost equal, the implementation is substantially different because communication with TCP sockets uses two classes: `ServerSocket` (for incoming calls) and `Socket` (to create connections and to communicate). In addition, the communication includes two phases: first sets up the connection; only after that it can communicate. This way, the application has two states:

- In the "IDLE" state it is waiting to connect, on its own initiative ("*Connect*" button) or receiving a connection from another user;
- In the "CONNECTED" state there is an active connecton and receives and sends data concurrently.

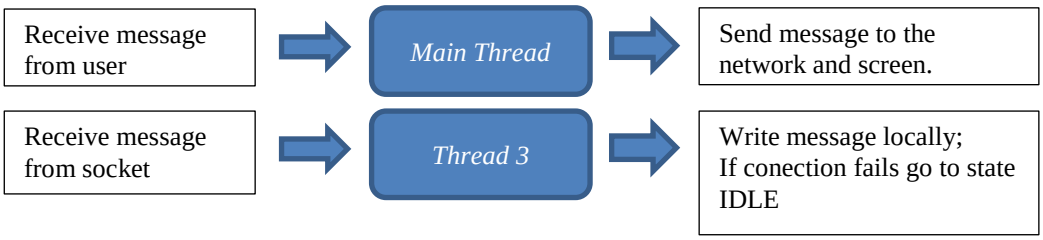


Thus, for each state there is always the need to have two activities in parallel.

In IDLE state, the main thread linked to the graphical interface is used to handle the "*Connect*" button; another thread is used to wait for connections from other users.



In the state CONNECTED the main thread is used to send user messages after pressing button "Send"; another thread is used to receive message from the TCP socket.



In this initial version of the program it was decided to have only one extra thread running; "Thread 2" and "Thread 3" will never be running at the same time. The main class `Chat_tcp` is used to coordinate the starting and stopping of threads.

2.1.1. `Daemon_tcp` class – Accepting connections

Class `Daemon_tcp` performs the functionality of "*Thread 2*": it blocks waiting to receive connections on `ServerSocket ss`. As it should only accept one connection at a time, the thread (as defined in `run` method) only runs once the instruction `ss.accept()`, which returns an object `s` of class `Socket`. `s` is then used to exchange messages. It should be noted also the use of Java instructions `try-catch-finally`, to ensure that even with errors, the variable `isRunning` always gets the value `false` at the end of the thread. A novelty was introduced, compared to *Chat UDP*: method `stopRunning` relies on the instruction `this.interrupt()` in order to ensure that the operation `accept` is interrupted, avoiding accepting more active connections.

```
public class Daemon_tcp extends Thread {
  volatile boolean isRunning = false;
  Chat_tcp root;                    // Main window object
  ServerSocket ss;                  // server socket

  public Daemon_tcp(Chat_tcp root, ServerSocket _ss) {  // Constructor
    this.root = root;
    this.ss = _ss;
  }

  public boolean isRunning() {
    return isRunning;
  }

  public void run() {  // Thread "2" code
    isRunning = true;
    try {
      Socket s = ss.accept();     // Waits for a new connection; s is a new Socket
      root.start_connection_thread(s);  // Asks Chat_tcp to start the connection thread
    } catch (Exception e) { // Exception
      if (isRunning) {
        root.Log_rem("Exception in Daemon_tcp : " + e);
      }
    } finally { // Always runs
      isRunning = false;
    }
  }

  public void stopRunning() {   // Stops main thread, interrupting the accept operation
```

```
      if (isRunning) {
        isRunning = false;
        this.interrupt();
      }
    }
  } // end of class Daemon_ tcp
```

2.1.2. `Connection_tcp` class – message reception in a connection

Class `Connection_tcp` implements the functionality of "*Thread 3*": it cycles waiting to receive messages from the socket associated with a connection, invoking the `receive_message` method of class `Chat_tcp` for each message received (i.e. for each new line of text). In addition, this class centralizes all communication through the socket: `send_message` provides the method for sending messages, and `toString` returns a string with a connection unique identifier consisting of "*IP address : port number*". Due to define the `toString` method, an object of this class can be automatically converted to a `String` object by the Java compiler.

The protocol used to send the messages is to add a line feed (`"\n"`) to the end of the message, allowing it to be read using the instruction `in.readline()`, which reads to the end of the line, extracting the end of line. Again, the instructions `try-catch-finally` are used to ensure that the connection is closed when the thread ends. The code inside `finally` is always executed regardless of whether an error occurred, return were called to exit the function or reached the end of the code inside the `try`.

```
  public class Connection_tcp extends Thread {
    volatile boolean keepRunning = true;
    Chat_tcp root;          // Main window object
    Socket s;               // socket
    PrintStream pout;       // Device used to write strings to the socket
    BufferedReader in;      // Device used to read from socket

    Connection_tcp(Chat_tcp _root, Socket _s) {
      this.root = _root;
      this.s = _s;
    }

    public String toString() {  // Returns the remote's ID
      if ((s==null) || !s.isConnected()) {
        return "null";
      }
      return s.getInetAddress().getHostAddress()+":"+s.getPort();
    }

    public boolean send_message(String msg) {  // Sends a message through the connection
      try {
        pout.print(msg + "\n");
        return true;
      } catch (Exception e) {
        return false;
      }
    }

    public void run() {  // Threads code
      try {
        String message;
        in = new BufferedReader(new InputStreamReader(s.getInputStream(), "8859_1"));
        pout = new PrintStream(s.getOutputStream(), false, "8859_1");
        while (keepRunning) {   // Loop waiting for messages
          message = in.readLine();    // Blocks waiting for new messages (line of text)
          if (message == null) {   // End of connection
            return;
          }
          root.receive_message(this, message);  // Calls Chat_tcp object
        }
      } catch (Exception e) { // Catches all errors
        if (keepRunning) {
          System.out.println("Error " + e);
```

```
        }
      } finally {   // Always runs this code, even when return is called!
        try {
          s.close();   // Close the socket and all devices associated
        } catch (Exception e) { /* Ignore everything */ }
        root.connection_thread_ended(this); // Inform Chat_tcp object that thread ended
      }
    }

    public void stopRunning() {  // Stops the thread
      keepRunning = false;
      try {
        s.close(); // Closes the socket and all devices associated; triggers an exception
      } catch (Exception e) {
        System.err.println("Error closing socket: "+e);
      }
    }
  } // end of class Connection_tcp
```

2.1.3. `Chat_tcp` class – initialization

The `Chat_tcp` class was created with the GUI, and is responsible for conducting the entire program logic. Three main variables used are: `ss`, `conn` and `serv`. The variable `ss` is of type `ServerSocket`, used to receive connections, the `conn` variable contains a thread object of type `Connection_tcp` that will receive and send messages (or is `null` if the thread is inactive); `serv` contains the thread object of class `Daemon_udp` (or is `null` if the thread is inactive), which will receive new connections. It also defines the auxiliary variable `formatter`, to format dates for writing as "*hour: minutes*", which is initialized in the declaration as it does not depend on any external value.

```
  // Variables declaration
  private ServerSocket ss;                              // server socket
  private Connection_tcp conn                           // Connection object;
  private Daemon_tcp serv;                              // thread for connection reception
  private java.text.SimpleDateFormat formatter =     // Formatter for dates
                      new java.text.SimpleDateFormat("hh:mm:ss");
```

The initial value of the variables is defined in the class constructor (`Chat_tcp` method). The constructor also populates the text boxes with the IP addresses of local address, and sets the value of the local port to 20000 by default.

```
  public Chat_tcp() {
    initComponents();  // defined by NetBeans, creates the graphical window
    ss = null;         // Set null value – meaning "not initialized"
    serv = null;       // Set null value – meaning "not initialized"
    conn = null;       // Set null value – meaning "not initialized"
    try {
      // Get local IP and set port to 0
      InetAddress addr = InetAddress.getLocalHost();  // Get the local IP address
      jTextLocIP.setText(addr.getHostAddress());      // Set the IP text fields to
      jTextRemIP.setText(addr.getHostAddress());      //    the local address
    } catch (UnknownHostException e) {
      System.err.println("Unable to determine local IP address: " + e);
      System.exit(-1);    // Closes the application
    }
    jTextLocPort.setText("20000");
  }
```

As in *ChatUDP*, the application startup is controlled in the function that handles the toggle button "*Active*". When the button is selected, the function reads the number of the local port, creating `ServerSocket ss`, and creates and starts a thread that receives connections, pre-filling the number of local and remote port. On error, it turns off the button back to its initial state. When the button is deselected, it closes the threads and `ss`.

```
private void jToggleButtonActiveActionPerformed(java.awt.event.ActionEvent evt)
  if (jToggleButtonActive.isSelected()) {  // The button is ON
    int port;
    try { // Read the port number in Local Port text field
      port = Integer.parseInt(jTextLocPort.getText());
    } catch (NumberFormatException e) {
      Log_loc("Invalid local port number: " + e + "\n");
      jToggleButtonActive.setSelected(false);  // Set the button off
      return;
    }
    try {
      ss = new ServerSocket(port);       // Create TCP Server socket
      jTextLocPort.setText("" + sock.getLocalPort());
      jTextRemPort.setText("" + sock.getLocalPort()+1);
      serv = new Daemon_tcp(this, ss);   // Create the connection receiver thread
      serv.start();                      // Start the thread
      Log_loc("Chat_tcp active\n");
    } catch (SocketException e) {
      Log_loc("Socket creation failure: " + e + "\n");
      jToggleButtonActive.setSelected(false);  // Set the button off
    }
  } else { // The button is OFF
    if (serv != null) {      // If connection receiver thread is running
      serv.stopRunning();    // Stop the thread
      serv = null;           // Thread will be garbadge collected after it stops
    }
    if (conn != null) {      // If connection thread is running
      conn.stopRunning();    // Stop the thread
      conn = null;           // Thread will be garbadge collected after it stops
    }
    if (ss != null) {        // If server socket is active
      try {
        ss.close();          // Close the socket
      } catch (IOException ex) {  /* Ignore */  }
      ss = null;             // Forces garbadge collecting
    }
    Log_loc("Chat_tcp stopped\n");
  }
}
```

2.1.4. `Chat_tcp` class – connection control

There are two ways to start a connection: either receiving a call through the `Daemon_tcp` object, or starting a new connection using the "*Connect*" button. The first case was described in section 2.1.1, where we showed that after receiving the connection the method `start_connection_thread` of class `Chat_tcp` is invoked. In the method's code, shown below, it is created and activated an object of class `Connection_tcp`. In addition, it selects the "*Connect*" button, allowing the user to stop the connection.

```
public void start_connection_thread(Socket s) {
  conn = new Connection_tcp(this, s);     // Create the connection thread object
  Log_rem("Connected to " + conn.toString() + "\n");
  jToggleButtonConnect.setSelected(true); // Set ON the "Connect" button
  conn.start();                           // Starts the connection thread
}
```

The second method of creating a connection is implemented in the function that handles the toggle button "*Connect*". If it is selected, it reads the values of text fields *Remote IP* and *Remote Port* and creates the object `cs` of class `Socket` connected to the remote user. If the connection is successful, the thread `serv` is stopped not to receive more connections and starts the thread `conn` associated with the connection using function `start_connection_thread`. On error, the "*Connect*" button is deselected.

The "*Connect*" button is used to stop a call, when the button is not selected.

```
private void jToggleButtonConnectActionPerformed(java.awt.event.ActionEvent evt) {
  if (jToggleButtonConnect.isSelected()) { // The button is ON - Start the connection
    if (con != null) {  // A Connection is active; ignore request
      return;
    }
    InetAddress netip;
    try { // Test IP address in Remote IP text box
      netip = InetAddress.getByName(jTextRemIP.getText());
    } catch (UnknownHostException e) {
      Log_loc("Invalid remote host address: " + e + "\n");
      jToggleButtonConnect.setSelected(false);
      return;
    }
    int port;
    try { // Test port
      port = Integer.parseInt(jTextRemPort.getText());
    } catch (NumberFormatException e) {
      Log_loc("Invalid remote port number: " + e + "\n");
      jToggleButtonConnect.setSelected(false);
      return;
    }
    try {
      Socket cs = new Socket(netip, port); // Create and connect a socket to the remote
      if (cs != null) {   // Is connected
        start_connection_thread(cs);       // Start the connection thread
        serv.stopRunning();                // Stop connection receive thread
        serv= null;
      }
    } catch (Exception ex) {
      Log_loc("Connection to " + jTextRemIP.getText() + ":" + jTextRemPort.getText() +
                  " failed\n");
      jToggleButtonConnect.setSelected(false);
    }
  } else {  // The button is OFF - stop the connection
    if (con != null) {
      con.stopRunning();  // Stop the connection
      con= null;
    }
  }
}
```

When `Connection_tcp` thread ends, the method `connection_thread_ended` is invoked at the main object of the class `Chat_tcp`. This method reactivates the thread `Daemon_tcp` and turns off the "*Connect*" button, making it ready to create new connections.

```
public void connection_thread_ended(Connection_tcp th) {
  Log_rem("Connection to "+ th.toString() + " ended\n");
  conn = null;
  serv = new Daemon_tcp(this, ss);   // Create the connection receiver thread
  serv.start();                      // Start the thread
  jToggleButtonConnect.setSelected(false); // Set OFF the "Connect" button
}
```

2.1.5. `Chat_tcp` class – sending messages

Messages are sent via the "*Send*" button, which invokes the method `send_message`. The implementation is simple in this case because the method uses `send_message` of class `Connection_tcp`, after getting the text that is in the *Message* text box.

```
public synchronized void send_message() {
  if (conn == null) {
    Log_loc("Connection isn't active!\n");
    return;
  }
  String message = jTextMessage.getText();   // Get the text from the Message box
  if (message.length() == 0) {
    Log_loc("Empty message: not sent\n");
    return;
  }
  if (conn.send_message(message)) {          // Send the message using the conn object
```

```
      // Write message to jTextAreaLocal
      Log_loc(formatter.format(new Date()) + " - sent '" + message + "'\n");
    }
  }
```

2.1.6. `Chat_tcp` class – receiving messages

The reception of messages is performed in `Connection_tcp` thread, but the content of the message is passed to `Chat_tcp` object through `receive_message` method, which simply write the content in respective window.

```
  public synchronized void receive_message(Connection_tcp con, String msg) {
    try {
      Date date = new Date();  // Get reception date
      // Write message contents
      Log_rem(formatter.format(date) + " - received '" + msg + "'\n");
    } catch (Exception e) {
      Log_rem("Error in receive_message: " + e + "\n");
    }
  }
```

2.2. ADVANCED CHATTCP – EXERCISES

In this second phase of the work, it is intended that students make two exercises using the project with the basic ChatTCP. The first is to add a button to send the contents of a file. The second is to modify the program to run multiple connections in parallel.

2.2.1. Sending the contents of a file

To accomplish this task it is necessary to add a new button "*Send File*" and a file choosing object (*File Chooser* on *Swing Windows*). The function that handles the new button should open a file selection window (see the example of the Calculator) and invoke a new method to send a file (e.g. `send_file`) at the class `Connection_tcp`. This new method `send_file` receives the file (eg `File f`) and should, in this order, open the file in read mode, create a `FileInputStream` object, create a buffer (`byte []`), read the file contents into the buffer and send the content to the socket, not forgeting to close the `FileInputStream` object:

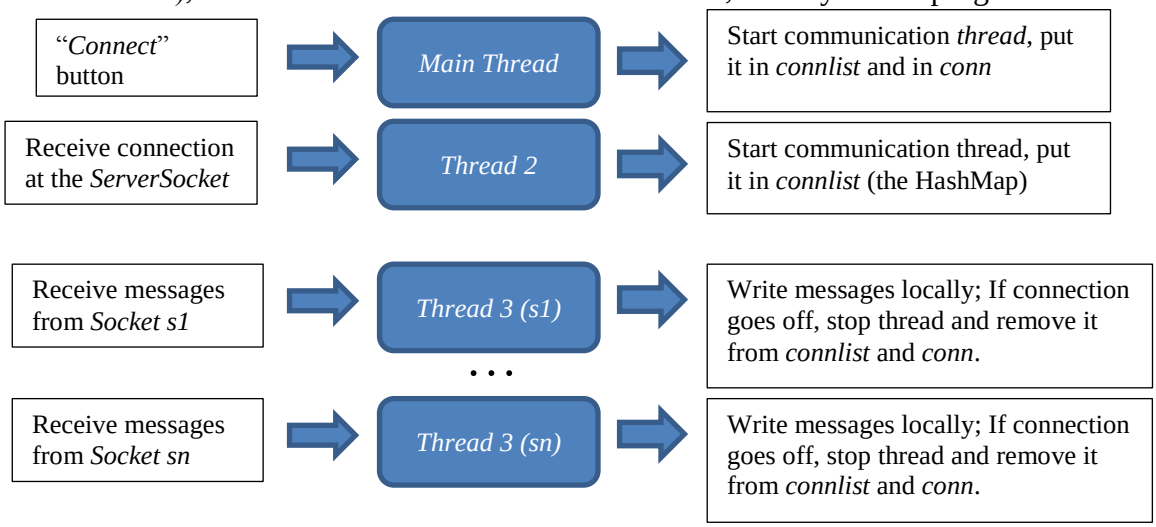
```
  public boolean send_file(File f) {   // Should be public because is called by Chat_tcp
    FileInputStream fis = null;
    try {
      fis = new FileInputStream (f); // Open file input stream
      byte[] buffer= new byte[fis.available()];// allocate a buffer with the
                                               //     length of the file
      int n= fis.read(buffer); // read the entire file to buffer;
                               //n counts the number of bytes actually read
      if (n != buffer.length) {
        root.Log_loc("Did not read the entire file\n");
        return false;
      }
      pout.write(buffer, 0, n); // write to socket
      return true;
    } catch (IOException ex) {
      root.Log_loc("Error sending file "+f+"\n");
      return false;
    } finally {
      try {
        fis.close();   // Always close the file
      } catch (IOException ex) { /* Ignore error */ }
    }
  }
```

Exercise 5.2.1: Add the method `send_file` to the class `Connection_tcp`. Implement the file sending functionality presented above at class `Chat_tcp`. See the support documentation and the Calculator example.

2.2.2. Several concurrent connections

This exercise is complex, and will require the advanced use of lists, threads and classes. It is recommended that before you perform this exercise you remember how *HashMaps* were at *Chat_udp* and watch carefully as the two threads are managed in the project, as it is now.

When it comes to allowing the use of multiple concurrent connections, the need arises to be "waiting" for messages on many sockets. In Java, we need a thread for each connection, so "many" threads will be used in parallel. The previous threads (`Daemon_tcp` and `Connection_tcp`) will be slightly modified: the object `Daemon_tcp` should be always active and there may be several `Connection_tcp` objects active. To keep the information about all `Connection_tcp` objects a data structure of type `HashMap<String, Connection_tcp>` is used, i.e. a list of connection objects, indexed by the string "*IP:port*" (which can be obtained in the class `Connection_tcp` with the method `toString`). Each user starts a single connection (using the "*Connect*" button), which will be stored in the variable `conn`, already on the program.



Exercise 5.2.2A: Declare e initialize the list `connlist` at class `Chat_tcp`:

```
private HashMap<String, Connection_tcp> connlist =
                                        new HashMap<String, Connection_tcp>();
```

The class `Connection_tcp` is already prepared to have multiple objects running in parallel. It is only necessary to modify what happens each time an object of this class is started or stopped. When you create an object, it is necessary to add it to the list (you can not modify the `conn` object, because it is now used to identify the connection initiated locally).

Exercise 5.2.2B: Modify the method `start_connection_thread` of class `Chat_tcp` in order to add the new object to the list `connlist`, returning the object created.

```
public Connection_tcp start_connection_thread(Socket s) {
  Connection_tcp c = new Connection_tcp(this, s);// Create the connection thread object
  Log_rem("Connected to " + c.toString() + "\n");
  c.start();                              // Start the connection thread
  connlist.put(c.toString(), c);  // Add object c to the list
  return c;
}
```

When a `Connection_tcp` object ends, it should be removed from the list, and if it is the `conn` object, the variable must be set to `null`. The thread *Daemon_tcp* should not be modified.

Exercise 5.2.2C: Modify the method `connection_thread_ended` of class `Chat_tcp` such that it adds the new object to `connlist`, returning the object created.

```
public void connection_thread_ended(Connection_tcp th) {
  Log_rem("Connection to " + th.toString() + " ended\n");
  if (th == conn) {      // if it is the thread initiated locally
    conn = null;
    jToggleButtonConnect.setSelected(false);  // Set the Connect button OFF
  }
  connlist.remove(th.toString());  // Removes the thread from the list using the key
}
```

To support multiple connections in parallel, thread `Daemon_tcp` should be in cycle indefinitely waiting for new connections, similarly to what happened in class `Daemon_udp`.

Exercise 5.2.2D: Modify the thread `Daemon_tcp`, which should be in cycle waiting for new connections, and only stop due to an error or when the user terminates the thread.

The "*Connect*" button also changes its operation. When you connect, you should save the `Connection_tcp` thread created in the variable `conn`, not changing the thread `Daemon_tcp`. The value of `conn` can be used to know if the object started (`conn != null`) or not (`conn == null`) a connection. When it is unselected, it should stop the thread started by the user, and remove it from the list.

Exercise 5.2.2E: Modify the method `jToggleButtonConnectActionPerformed` of class `Daemon_tcp` to turn on/off the `Connection_tcp` thread started by the user, according to the description above.

Messages (and files) should be sent through all connections.

Exercise 5.2.2F: Modify the method `send_message` of class `Chat_tcp` in order to send messages through all connections. The method `send_message` should be invoked for all objects in the list `connlist`. For all objects, one can use an iterator over the values, obtained with:

```
Iterator<Connection_tcp> it= connlist.values().iterator();
```

The message should be sent as long as the list is not empty (i.e. `!connlist.isEmpty()`).

Exercise 5.2.2G: Modify the method associated with sending file in class `Daemon_tcp` in order to send the file to all connections.

To complete the application, it is missing only the modification on the handling of "*Active*" button, which should terminate ALL active connections when the application shuts down.

Exercise 5.2.2H: Modify the method `jToggleButtonActiveActionPerformed` of class `Daemon_tcp` in order to stop all active connections and clear the list at the end (`connlist.clear()`). As in previous exercises, you should stop each thread individually.