FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Electrical Engineering Department

Telecommunication Systems

2013/2014

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

Class 2 – Aplication with datagram sockets

Integrated Master in Electrical Engineering and Computers

Luís Bernardo
Paulo da Fonseca Pinto

Index

1. OBJECTIVE

Familiarization with the Java programming language and applications using datagram 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. SECOND APPLICATION – NETWORK CHAT WITH DATAGRAMS

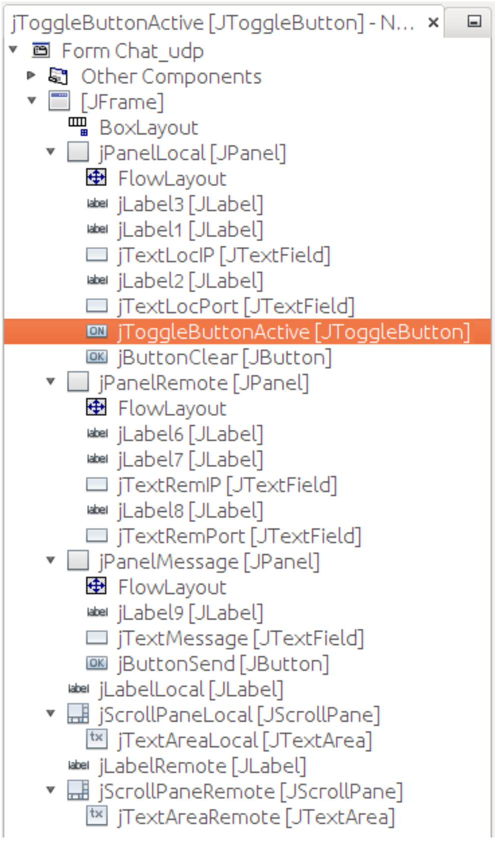
This section illustrates the development of an application using datagram sockets. The application supports the exchange of messages in a network, where each participant has a window which receives messages from other elements (*Remote*) and a window where you type your messages (*Local*). The user selects the IP address and port number of the machine where to send the messages and can shut down the application.

2.1. BASIC CHATUDP

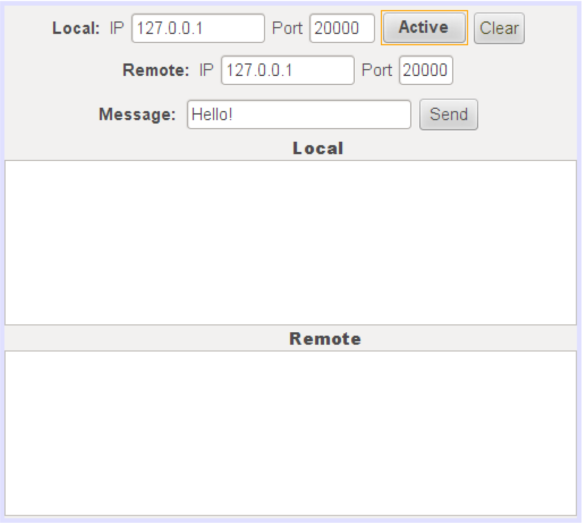
The first version only supports simple exchange of messages between users, uniquely identified on the network by the set *{endereço IP : número de porto}*. The NetBeans project with this first part is provided to the students.

As in the example of the previous class, the first phase starts with opening the project `Chat_udp` distributed with the assignment, with the design of the GUI. In this project we defined the window shown below, where we used the following components:

- 2 objects **Button** OK {buttons 'Clear' and 'Send'}
- 1 object **ToggleButton** {button 'Active'}
- 5 objects **Text Field** □ {message, IP and local and remote ports}
- 2 objects **Text Area** tx {local and remote texto}
- 2 objects **Scroll Pane** ▦ {Scroll bars for *TextArea*}
- 3 objects **Panel** □ {three lines with groups of buttons }



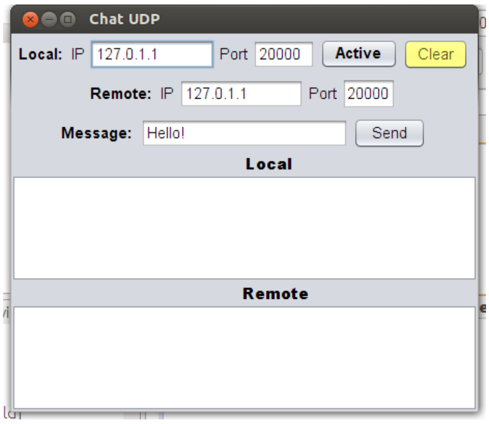
```
jToggleButtonActive [JToggleButton] - N... ×
▾ Form Chat_udp
  ▸ Other Components
  ▾ [JFrame]
      BoxLayout
    ▾ jPanelLocal [JPanel]
        FlowLayout
        jLabel3 [JLabel]
        jLabel1 [JLabel]
        jTextLocIP [JTextField]
        jLabel2 [JLabel]
        jTextLocPort [JTextField]
        jToggleButtonActive [JToggleButton]
        jButtonClear [JButton]
    ▾ jPanelRemote [JPanel]
        FlowLayout
        jLabel6 [JLabel]
        jLabel7 [JLabel]
        jTextRemIP [JTextField]
        jLabel8 [JLabel]
        jTextRemPort [JTextField]
    ▾ jPanelMessage [JPanel]
        FlowLayout
        jLabel9 [JLabel]
        jTextMessage [JTextField]
        jButtonSend [JButton]
      jLabelLocal [JLabel]
    ▾ jScrollPaneLocal [JScrollPane]
        jTextAreaLocal [JTextArea]
      jLabelRemote [JLabel]
    ▾ jScrollPaneRemote [JScrollPane]
        jTextAreaRemote [JTextArea]
```



Local: IP 127.0.0.1 Port 20000

Remote: IP 127.0.0.1 Port 20000

Message: Hello!



Chat UDP

Local: IP 127.0.1.1 Port 20000

Remote: IP 127.0.1.1 Port 20000

Message: Hello!

The name of each graphical object is shown in the figure of the previous page, and the text of each button is represented above. In order to have the graphical aspect shown on the right, the following object properties were modified:

- object '`JFrame`'
  - ✓ `title`= “*Chat UDP*”;
  - ✓ `Layout=BoxLayout`, com `Axis`= “*Y Axis*”.
- object '`jPanelLocal`'
  - ✓ `Layout=FlowLayout`;
  - ✓ `preferredSize`= [450,38] e `maximumSize`= [450,40], limiting the vertical growth.
- object '`jTextLocIP`'
  - ✓ `preferredSize`= [120,28], fixing the width of the box after “Local: IP”.
- object '`jTextLocPort`'
  - ✓ `preferredSize`= [60,28], fixing the width of the box after “Local: Port”.
- object '`jButtonClear`'
  - ✓ `background`= [220,220,100], changing the color to yellow.
- object '`jPanelRemote`'
  - ✓ `Layout=FlowLayout`;
  - ✓ `preferredSize`= [450,38] and `maximumSize`= [450,40], limiting vertical growth.
- object '`jTextRemIP`'
  - ✓ `preferredSize`= [120,28], fixing the width of the box after “Remote: IP”.
- object '`jTextRemPort`'
  - ✓ `preferredSize`= [50,28], fixing the width of the box after “Remote: Port”.
- object '`jPanelMessage`'
  - ✓ `Layout=FlowLayout`;
  - ✓ `preferredSize`= [450,38] and `maximumSize`= [450,40], limiting vertical growth.
- object '`jTextMessage`'
  - ✓ `preferredSize`= [200,28], fixing the width of the box after “Message:”.
- object '`jLabelLocal`'
  - ✓ `preferredSize`= [50,22] and `maximumSize`= [50,22], limiting vertical growth.
- object '`jScrollPanelLocal`'
  - ✓ `preferredSize`= [222,87], defining an unlimited maximum to let it grow.
- object '`jTextAreaLocal`'
  - ✓ `preferredSize`= [220,85], defining an unlimited maximum to let it grow.
- object '`jLabelRemote`'
  - ✓ `preferredSize`= [64,22] and `maximumSize`= [64,22], limiting vertical growth.

- object 'jScrollPanelRemote'
  - ✓ preferredSize= [222,87], defining an unlimited maximum to let it grow.
- objeto 'jTextAreaRemote'
  - ✓ preferredSize= [220,85], defining an unlimited maximum to let it grow.

Thus, if you enlarge the window, only the local and remote text boxes get larger. In addition to the changes above, the font of the labels was changed to "*Arial Bold 15*" and the other fonts for "*Arial 15 Plain*" in order to adopt a font that exists on Windows, MacOS and Linux, making the code more portable across platforms.

Then, empty methods for the buttons were created using double click.

To facilitate writing in text boxes "*Local*" and "*Remote*" two methods were set up in the class associated with the window (Chat_udp): Log_loc and Log_rem. The two are synchronized to ensure that write operations are not interrupted.

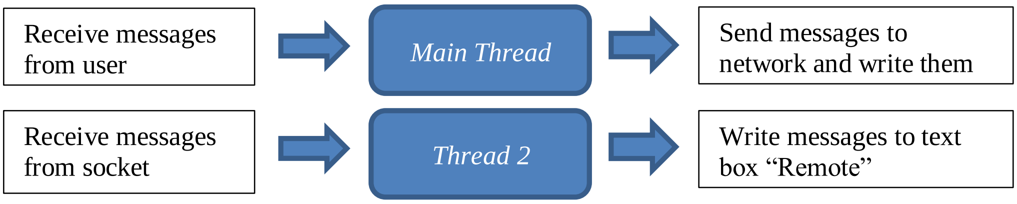
```
public synchronized void Log_loc(String s) {
  try {
    jTextAreaLocal.append(s);           // Write to the Local text area
    System.out.print("Local: " + s);    // Write to the terminal
  } catch (Exception e) {
    System.err.println("Error in Log_loc: " + e + "\n");
  }
}

public synchronized void Log_rem(String s) {
  try {
    jTextAreaRemote.append(s);         // Write to the Remote text area
    System.out.print("Remote: " + s); // Write to the terminal
  } catch (Exception e) {
    System.err.println("Error in Log_rem: " + e + "\n");
  }
}
```

These text boxes are cleaned in the method that handles the event associated with the button "*Clear*":

```
private void jButtonClearActionPerformed(java.awt.event.ActionEvent evt) {
  jTextAreaLocal.setText("");
  jTextAreaRemote.setText("");
}
```

2.1.1. Daemon_udp class – Reception of data from UDP socket

The datagram sockets are permanently connected. As message are sent and received concurrently, in Java there is the need for two concurrent activities:


Receive messages from user
Main Thread
Send messages to network and write them
Receive messages from socket
Thread 2
Write messages to text box “Remote”

The main program is already waiting for graphical events to send messages (clicks on button "*Send*"). Therefore, you must create an object of class Daemon_udp, of type ***thread***, which is run concurrently with the main program to receive messages from the socket.

The class code is shown below, containing the constructor (where the values of the class variables root and ds are initialized), the run method (which contains the code run in this thread), and stopRunning method, which allows stopping the thread, controlling the Boolean variable keepRunning. The socket is created in the main class, that creates the graphics window (Chat_udp), and is passed in the constructor for this class (argument ds) being stored in the variable with the same name (this identifies the object location) along with the reference to the object main (root). The run method is cyclically waiting for messages, invoking the receive_packet method of class Chat_upd per message received and passing as parameter the object for reading the message fields (dis). It also deals with the exceptions resulting from errors in communication.

```
public class Daemon_udp extends Thread {   // inherits from Thread class
  volatile boolean keepRunning = true;
  Chat_udp root;                        // Main window object
  DatagramSocket ds;                    // datagram socket

  public Daemon_udp(Chat_udp root, DatagramSocket ds) {  // Constructor
    this.root = root;
    this.ds = ds;
  }

  public void run() {               // Function run by the thread
    byte[] buf = new byte[Chat_udp.MAX_PLENGTH];  // buffer with maximum message size
    DatagramPacket dp = new DatagramPacket(buf, buf.length);
    try {
      while (keepRunning) {
        try {
          ds.receive(dp);    // Wait for packets
          ByteArrayInputStream BAis = new ByteArrayInputStream(buf, 0, dp.getLength());
          DataInputStream dis = new DataInputStream(BAis);
          root.receive_packet(dp, dis);  // process packet in Chat_udp object
        } catch (SocketException se) {
          if (keepRunning) {
            root.Log_rem("recv UDP SocketException : " + se + "\n");
          }
        }
      }
    } catch (IOException e) {
      if (keepRunning) {
        root.Log_rem("IO exception receiving data from socket : " + e);
      }
    }
  }

  public void stopRunning() {    // Stops loop by turning off keepRunning
    keepRunning = false;
  }
}
```

2.1.2. Class Chat_udp – initialization

The class Chat_udp is associated with the GUI, and is responsible for conducting all program logic and for defining of settings: MAX_PLENGTH is a constant (final variable) at the class level (static variable) with the maximum size for the messages exchanged.

```
public static final int MAX_PLENGTH = 8096;   // Constant - Maximum packet length
```

Two main variables are used: sock and serv. The variable sock holds the object of type DatagramSocket used both to send and to receive data; variable serv contains a thread object of class Daemon_udp that receives messages concurrently. It also defines the auxiliary variable formatter, to format the writing of dates to "*hour:minutes*", which is initialized in the declaration because it does not depend on any external value.

```
// Variables declaration
private DatagramSocket sock;                           // datagram socket
private Daemon_udp serv;                               // thread for message 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 (method Chat_udp). In addition, the constructor fills the text boxes with IP addresses with the local address, and sets the value of the local port to 20000 by default.

```
public Chat_udp() {
  initComponents();  // defined by NetBeans, creates the graphical window
  sock = null;       // Set null value - meaning "not initialized"
  serv = 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");
}
```

The application startup is controlled in the method that handles the toggle button "*Active*". When the button is activated, the function reads the local port number, creates the socket, and creates and starts a thread that receives data from the socket, 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 disabled, the function stops the thread and closes the socket.

```
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 {
      sock = new DatagramSocket(port);         // Create UDP socket
      jTextLocPort.setText("" + sock.getLocalPort());
      jTextRemPort.setText("" + sock.getLocalPort());
      serv = new Daemon_udp(this, sock);       // Create the receiver thread
      serv.start();                            // Start the receiver thread
      Log_loc("Chat_udp 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 thread is running
      serv.stopRunning();    // Stop the thread
      serv = null;           // Thread will be garbage collected after it stops
    }
    if (sock != null) {      // If socket is active
      sock.close();          // Close the socket
      sock = null;           // Forces garbadge collecting
    }
    Log_loc("Chat_udp stopped\n");
  }
}
```

2.1.3. Class `Chat_udp` – message handling

The message format defined in the application is:

Length (short)	Message contents (byte[])

As seen above, the UDP messages are received in `Daemon_udp` thread that calls the method `receive_packet`, of the object of class `Chat_udp` to handle message content. In this method the received data is written in the text area *Remote*:

```
public synchronized void receive_packet(DatagramPacket dp, DataInputStream dis) {
  try {
    Date date = new Date();  // Get reception hour
    String from = dp.getAddress().getHostAddress() // IP address of the sending host
                  + ":" + dp.getPort();            // + port of sending host = User ID
    // Read the packet fields using 'dis'
    int len_msg = dis.readShort();  // Read message length
    if (len_msg > MAX_PLENGTH) {
      Log_rem(formatter.format(date) + " - received message too long (" + len_msg +
                 ") from " + from + "\n");
      return;    // Leaves the function
    }
    byte[] sbuf2 = new byte[len_msg];    // Create na array to store the message
    int n = dis.read(sbuf2, 0, len_msg); // returns number of byte read
    if (n != len_msg) {
      Log_rem(formatter.format(date) + " - received message too short from " +
                  from + "\n");
      return;
    }
    String msg = new String(sbuf2, 0, n);  // Creates a String from the buffer
    if (dis.available() > 0) {  // More bytes after the message
      Log_rem("Packet too long\n");
      return;
    }
    // Write message contents
    Log_rem(formatter.format(date)+ " - received from " + from + " - '" + msg + "’\n");
  } catch (IOException e) {
    Log_rem("Packet too short: " + e + "\n");
  }
}
```

2.1.4. Class `Chat_udp` – sending messages

Sending messages is performed using two functions. The first function, `send_one_packet`, receives a packet (`DatagramPacket` class) with a message, sets the IP address and port number, and sends it to the destination:

```
private void send_one_packet(DatagramPacket dp, InetAddress netip /* destination IP */,
                             int port /* destination port */, String message) {
  try {
    dp.setAddress(netip);   // Set destination ip
    dp.setPort(port);       // Set destination port
    sock.send(dp);          // Send packet
    // Write message to jTextAreaLocal
    String to = netip.getHostAddress() + ":" + port;   // ‘name’ of remote host
    String log = formatter.format(new Date()) + " - sent to " + to
                  + " - '" + message + "'\n";
    Log_loc(log);           // Writes to Local text area
  } catch (IOException e) {  // Communications exception
    Log_loc("Error sending packet: " + e + "\n");
  } catch (Exception e) {    // Other exception (e.g. null pointer, etc.)
    Log_loc("Error sending packet: " + e + "\n");
  }
}
```

The previous function is invoked by `send_packet` function that reads the IP address and port of the respective text boxes and prepares the message to send. Thus, it does not require parameters:

```
public synchronized void send_packet() {
  if (sock == null) {
    Log_loc("Socket isn't active!\n");
    return;
  }
  InetAddress netip;
  try { // Get IP address
    netip = InetAddress.getByName(jTextRemIP.getText());
  } catch (UnknownHostException e) {  // O endereço IP não é válido
    Log_loc("Invalid remote host address: " + e + "\n");
    return;
  }
  int port;
  try { // Get port
    port = Integer.parseInt(jTextRemPort.getText());
  } catch (NumberFormatException e) {
    Log_loc("Invalid remote port number: " + e + "\n");
    return;
  }
  String message= jTextMessage.getText();
  if (message.length() == 0) {
    Log_loc("Empty message: not sent\n");
    return;
  }
  // Create and send packet
  ByteArrayOutputStream os = new ByteArrayOutputStream();  // Prepares a message
  DataOutputStream dos = new DataOutputStream(os);         //   writting object
  try {
    dos.writeShort(message.length());     // Write the message's length to buffer
    dos.writeBytes(message);              // Write the message contents to buffer
    byte[] buffer = os.toByteArray();     // Convert to byte array
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length);  // Create packet
    send_one_packet(dp, netip, port, message);   // Send packet and log the event
  } catch (Exception e) {  // Catches all exceptions
    Log_loc("Error sending packet: " + e + "\n");
  }
}
```

Messages are sent in when the button "*Send*" is pressed:

```
private void jButtonSendActionPerformed(java.awt.event.ActionEvent evt) {
  send_packet();
}
```

2.2. ADVANCED CHATUDP – EXERCISES

In this second phase of the work it is intended that the students perform a set of exercises on the project with the first phase of the work. The first is to perform the sending of five messages (one per second) for 5 seconds controlled by a timer, pressing a "*Send 5*" key. In the second, the last message received from each of the other users is memorized. The third is to send a message to a port on all machines on a network.

2.2.1. Sending messages using a timer

To accomplish this you must create a variable (object) `timer`, which will support the sending of messages. To control the number of times that lack sending, it is also used a variable `cnt` serving as counter – it counts the number of times that the timer should still fire.

Exercise 4.2.1A: add the following fields (variables) to the class `Chat_udp`:

```
private javax.swing.Timer timer;    // Timer object
public volatile int cnt;            // Counter - messages left to send
```

To use a timer, you must create the `set_timer_function` function that creates the timer object, and defines the function that will be run when the time passes. Note that the timer is created but not active - only when the `start()` method is invoked.

Exercício 4.2.1B: add the following method to the class `Chat_udp`:

```
private void set_timer_function(int period /*ms*/) {
  java.awt.event.ActionListener act;      // Callback object
  act = new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
      // Define the timer callback function
      if (cnt > 0) {     // While there are more packets to send …
        send_packet();   // Send the packet
        cnt--;           // Decrement the counter
```

```
     } else {  // Counter reached 0
      timer.stop();    // Stop the timer
     }
    }
   };
   timer = new javax.swing.Timer(period, act);  // Create the timer's object
  }
```

To initialize the timer object, the method set_timer_function must be called when starting the application (button "*Active*"), in the method jToggleButtonActiveActionPerformed at the class Chat_udp. When it shuts down the application, it is also necessary to turn off the timer.

Exercise 4.2.1C: add the invocation set_timer_function in the method of the class Chat_udp that starts the application, passing as argument 1 second (1000 ms). It should be placed after the successful creation of the socket and the thread.

Exercise 4.2.1D: add the code to turn off the timer object in Chat_udp's class method that stops the application (associated with the button "*Active*").

To finish, it is only missing a button "*Send 5*", which starts the timer.

Exercise 4.2.1E: add a button "*Send 5*" next to "*Send*" and program the event handling function associated to start the timer (with timer.start();) if not already active (can know that using timer.isRunning()), and properly initialize the value of cnt in order to send 5 packets. Remember that timer can be equal to null (because the application can be not active).

2.2.2. Memorize the last message from users

The intention of this exercise that the application remember the last message it received from all users with whom it is communicating. To control this feature, it is proposed to use a toggle button "*Record*" with the name jToggleButtonRecord. When the button is selected, all incoming messages are saved. When the button is turned off all the latest messages from all users should be written one by one.

Exercise 4.2.2A: Add one *Toggle Button* to the GUI, with the name jToggleButtonRecord and with the text "*Record*". Create the button handling function.

Messages will be recorded in an indexed list, where the index is defined by the concatenation of *IP address*+":"+*port number* (HashMap<String,String>).

Exercise 4.2.2B: add the following declaration of the list record to the class Chat_udp:

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

Each time a message is received, you need to create a string with the user identifier to use as index. In the following exercises, you should save the message to the list associated with the received message identifier.

Exercise 4.2.2C: add to method receive_packet (presented in section 2.1.3) the message registration operation, if the button is selected:

```
  if (jToggleButtonRecord.isSelected()) {  // If button is selected
   String str = formatter.format(date) + " - received from " + from + " - '" + msg +
          "'\n";
   record.put(from, str);   // Put string str into the list associated to key from
  }
```

It is intended to write all records and sources stored in the list. An iterator will be used for going through the list of keys. For each key, method get returns the last message associated with the user. For instance, one can know the last message sent by "127.0.0.1:20000" doing the invocation String str= record.get("127.0.0.1:20000").

Exercise 4.2.2D: add to the class Char_udp the method write_record, which writes to the terminal the content of the list and clears the list.

```
public void write_record() {
  Iterator<String> it = record.keySet().iterator(); // interator over keys
  while (it.hasNext()) {  // While there are more keys to get
    String remote = it.next();    // remote has next key
    System.out.println("\nCommunication with " + remote);
    System.out.println(record.get(remote));// Write the last message from remore
  }
  record.clear();  // Clear the list
}
```

To finish, it is only missing that the function that handles the event associated to pressing the "*Record*" button (created in 4.2.2A) calls the method `write_record` when you turn off the "*Record*" button.

Exercise 4.2.2D: program the method associated to "*Record*" button created in exercise 4.2.2A so as to invoke the method `write_record` only when the button is set off; note that the method will also run when the button is turned on.

2.2.3. Sending messages to all computers in the network

The latter exercise should only be done if more than 20 minutes are left to the end of class.

It is intended to send a message to all computers that are connected to the network, instead of sending only to one IP address. Laboratory 3.4 has 11 machines on the network 172.16.54.0, occupying the IP addresses 172.16.54.101 - 172.16.54.111. If you are on another network, you can know the network address information from the machine's IP address and network mask. To control this "broadcast" sending, you should create a toggle button "*All*". If it is selected, messages must be send to all machines and to the port selected. Therefore, all changes have to be made only in `send_packet` method, which sould call the method `send_packet` for all machines on the network.

Exercise 4.2.3:

1) Add a *Toggle Button* "*All*";
2) Modify the method `send_packet` to generate all IP addresses and call the function `send_one_packet` when the button "*All*" is selected. Tip: You can create a string with the IP address, and then convert it to the type `InetAddress`.