**Network Programming – Sockets**

7. Sending objects through sockets (TCP)

The serialization mechanism provides the programmer with a method that saves an object on disk and restored when needed. Also by the same mechanism, an object can be transmitted remotely to another machine by using the sockets.

In order to serialize an object, it will need to implement the **Serializable** interface.

For writing and reading serialized objects, input/ output streams are used: **ObjectInputStream** and **ObjectOutputStream**.

The following listing shows how to serialize/ deserialize an object.

*import java.io.\*;*

*import java.net.\*;*

*public class SerialTest extends Thread{*

*public void run(){*

*try{*

*ServerSocket ss = new ServerSocket(1977);*

*Socket s = ss.accept();*

*ObjectInputStream ois = new ObjectInputStream(s.getInputStream());*

*Pers p = (Pers)ois.readObject();*

*System.out.println(p);*

*s.close();*

*ss.close();*

*}catch(Exception e){e.printStackTrace();}*

*}*

*public static void main(String[] args) throws Exception{*

*//trimite obiect prin socket*

*(new SerialTest()).start();*

*Socket s = new Socket(InetAddress.getByName("localhost"),1977);*

*ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());*

*Pers p = new Pers("Alin",14);*

*oos.writeObject(p);*

*s.close();*

*}*

*}*

*class Pers implements Serializable{*

*String nume;*

*int varsta;*

*Pers(String n, int v){*

*nume = n; varsta = v;*

*}*

*public String toString(){*

*return "Persoana: "+nume+" varsta: "+varsta;*

*}*

*}*

There are situations when the state of an object is saved by serialization, but it is not desired to save all the states of the object, respectively, it is not desired to save or transmit certain parameters of the object. To block the serialization of an attribute of a serializable object, the keyword **transient** is used.

8. HTTP Server

In this section, An HTTP server that can respond to GET requests is created. The **main()** function of the **HttpServer** class starts a thread of execution. This thread instantiates a **ServerSocket** object and begins listening to port 80, which is the standard HTTP protocol port.

When a request appears (a client connects to port 80), the **accept()** method will return a Socket object. Next, a **ProcesRequest** object (which is of type thread of execution) is created. Then it will receive the Socket object that is returned by the **accept()** method as a parameter. After the **ProcesRequest** object is created, the server resumes waiting and will be able to serve other clients.

The **ProcesRequest** class implements a simplified version of the HTTP protocol. After the thread is started, the input / output stream is created in the constructor of the **ProcesRequest** class. The request received from the client is analyzed in the thread of execution, and if this is a valid GET request, then the requested resource will be sent to the client.

*import java.io.\*;*

*import java.net.\*;*

*class HttpServer extends Thread*

*{*

*//standard port*

*private final static int PORT = 80;*

*private final String iniContext="c:/temp/ServerHTTP/webdocs";*

*private boolean alive;*

*private ServerSocket ss;*

*//constructor*

*HttpServer()throws Exception{*

*System.out.println("Start server http.");*

*ss = new ServerSocket(PORT);*

*alive=true;*

*start();*

*}*

*public void run(){*

*while(alive){*

*//waiting for connections*

*try{*

*System.out.println("Server asteapta...");*

*new ProcesRequest(ss.accept(),iniContext);*

*}*

*catch(IOException e)*

*{System.err.println("EROARE CONECTARE:"+e.getMessage());}*

*//..resumes the waiting block when we create a thread for the client*

*}*

*System.out.println("STOP SERVER");*

*}*

*public static void main(String[] args) throws Exception*

*{*

*try{*

*new HttpServer();*

*}catch(Exception e){e.printStackTrace();}*

*}*

*}*

*import java.net.\*;*

*import java.io.\*;*

*import java.util.\*;*

*class ProcesRequest extends Thread*

*{*

*private PrintWriter outStr;*

*private BufferedReader inStr;*

*private Socket s;*

*private DataOutputStream dout;*

*private String iniContext;*

*ProcesRequest(Socket s, String iContext){*

*try{*

*outStr = new PrintWriter(new OutputStreamWriter(s.getOutputStream()),true);*

*inStr = new BufferedReader(new InputStreamReader(s.getInputStream()));*

*dout = new DataOutputStream(s.getOutputStream());*

*iniContext = iContext;*

*this.s = s;*

*start();*

*}catch(IOException e)*

*{System.err.println("EROARE CONECTARE: "+e.getMessage());}*

*}*

*public void run(){*

*try{*

*String fileName=null;*

*String request = inStr.readLine();*

*System.out.print(request);*

*if(request.lastIndexOf("GET")==0) fileName = interpretGET(request);*

*else throw new Exception("BAU");*

*byte[] data = readFile(fileName);*

*dout.write(data);*

*dout.flush();*

*}*

*catch(IOException e)*

*{outStr.println("<HTML><BODY><P>403 Forbidden<P></BODY></HTML>");}*

*catch(Exception e2)*

*{outStr.println("<HTML><BODY><P>"+e2.getMessage()+"<P></BODY></HTML>");}*

*finally{*

*try{s.close();}catch(Exception e){}*

*}*

*}*

*private String interpretGET(String rqst) throws Exception{*

*StringTokenizer strT = new StringTokenizer(rqst);*

*String tmp="";*

*String fileN=iniContext;*

*tmp=strT.nextToken();*

*if(!tmp.equals("GET")) throw new Exception("Comanda GET invalida .");*

*tmp=strT.nextToken();*

*if((tmp.equals("/")) || (tmp.endsWith("/"))) {*

*fileN = fileN+tmp+"index.htm";*

*System.err.println("CERERE:"+fileN);*

*return fileN;*

*}*

*fileN = fileN+ tmp;*

*System.err.println("CERERE:"+fileN);*

*return fileN;*

*}*

*private byte[] readFile(String fileN) throws Exception{*

*fileN.replace('/','\\');*

*File f = new File(fileN);*

*if(!f.canRead()) throw new Exception("Fisierul "+fileN+" nu poate fi citit");*

*FileInputStream fstr = new FileInputStream(f);*

*byte[] data = new byte[fstr.available()];*

*fstr.read(data);*

*return data;*

*}*

*}*

To verify the previous program, the **iniContextm** variable in the **HTTPServer** class will be modified so that it indicates the correct path to the original context (the directory where all the resources that the client can access).

Network programming - Datagrams and URLs

1. Objectives

The purpose of this work is to master the following mechanisms;

- UDP communication

             - working with URLs

             - communicating with a CGI

2. Preliminary notions

Customers communicating via TCP using sockets have a dedicated channel and secure data transmission. Data is received in the order in which it was sent.

In contrast, when data is transmitted by UDP, reaching them at destination is not guaranteed, the datagrams order of arrival at the destination may also differ from the order in which they were transmitted.

The advantage of working with datagrams is to increase the speed of packets (datagrams) reaching the destination. There are cases where the data transmission speed is more important than 100% guarantee of reaching the destination. For example, when transmitting a real-time audio signal, its transmission speed is more important than the guarantee of reaching the destination.

To implement UDP protocol in java, these classes are used: **DatagramPacket** and **DatagramSocket**. Unlike TCP programming, in the UDP case there is no ServerSocket concept. Both the **server** and the **client** use **DatagramSocket** to make the connection. **DatagramPacket** class is used to transmit and receive data.

Programming takes place at a higher level than socket programming. Using the URL class, you can open a connection to a resource on the web. Once the connection is opened to the resource, the client will be able to read data or send data to that resource.

The **URL** is the acronym of the Uniform Resource Locator (also the name of a java class). A URL is a pointer to a resource in the Internet.

  The general syntax of a URL is:

protocol://hostname[:port]/path/filename#ref

3. UDP Server

A time server is built in this section that will send the current date to customers requesting it. The client accessing the time server services is also built.

At the server level, a **DatagramSocket** object is created, this object will receive the port as a parameter on which the server will start listening.

   DatagramSocket socket = new DatagramSocket(1977);

Next, a DatagramPacket object is being built, which will be used by the server to receive the request from the client. Once the **DatagramPacket** object is built, the server will start listening to the 1977 port by invoking the receive() method.

           byte[ ] buf = new byte[256];

             DatagramPacket packet = new DatagramPacket(buf,buf.length);

             socket.receive(packet);

When a client wishes to use the server services, it will send a packet to the server. **The** **server** reads the client's port and address from the package, and will send it a package containing the current date.

InetAddress address = packet.getAddress();

             int port = packet.getPort();

             buf = ((new Date()).toString()).getBytes();

 packet = new DatagramPacket(buf,buf.length,address,port);

  socket.send(packet);

In order for a client to connect to the server, it must create a **DatagramSocket** object, and sends a packet to the server. Unlike the server, the client is not obligated to specify any port when the **DatagramSocket** object is created, since it automatically assigns a free port to that object.

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class TimeServer extends Thread{

   boolean running=true;

   public TimeServer() { start();}

   public void run(){

   try{

     DatagramSocket socket = new DatagramSocket(1977);

     while(running)

//waitnig for the client

       byte[] buf = new byte[256];

       DatagramPacket packet = new DatagramPacket(buf,buf.length);

       socket.receive(packet);

      //reads the client address and port no.

       InetAddress address = packet.getAddress();

       int port = packet.getPort();

      //sends a reply to the client

       buf = ((new Date()).toString()).getBytes();

       packet = new DatagramPacket(buf,buf.length,address,port);

       socket.send(packet);

    }

    }catch(Exception ex){ex.printStackTrace();}

  }

  public static void main(String[] args) {

    TimeServer timeServer1 = new TimeServer();

  }

}

 import java.io.\*;

 import java.net.\*;

 import java.util.\*;

public class Client {

  public static void main(String[] args) {

  try{

     DatagramSocket socket = new DatagramSocket();

     byte[] buf = new byte[256];

     DatagramPacket packet = new DatagramPacket(buf,buf.length,

      InetAddress.getByName("localhost"),1977);

     socket.send(packet);

     packet = new DatagramPacket(buf,buf.length);

     socket.receive(packet);

     System.out.println(new String(packet.getData()));

  }catch(Exception ex){ex.printStackTrace();}

  }

}

4. Working with URLs

This section discusses how to work in java with URLs.

The **URL** is the acronym for Uniform Resource Locator and represents a reference (address) to an Internet resource. This is generally a file representing a webpage or image, but a URL can also refer to database queries, the results of some commands (programs), etc.

To access an Internet resource identified by a URL in java, the first step is to create a URL object.

            URL utcn = new URL(“[www.utcluj.ro](http://www.utcluj.ro/)”);

The URL class contains methods that all the components of a URL can be found: getProtocol(), getPort(), getHost(), getFile(), getRef().

Once the URL object has been created, the openStream() method is used to open an input stream through which the content of that URL is read.

BufferedReader in = new BufferedReader(

                    new InputStreamReader(

                    utcn.openStream()));

               String inputLine;

               while ((inputLine = in.readLine()) != null)

                   System.out.println(inputLine);

               in.close();

5. Read and Write to a URL

If you wish to do more than just reading the contents of a URL, then you can call the **openConnection()** method within the URL class. This method will return a **URLConnection** object. This object can be used for writing, reading, and queries to a URL.

*URLConnection connection = utcn.openConnection();*

Here is a short program that reads the contents of a URL using the **URLConnection** class.

import java.net.\*;

import java.io.\*;

public class URLConnectionReader {

    public static void main(String[] args) throws Exception {

        URL utcluj = new URL([http://www.utcluj.ro](http://www.utcluj.ro/));

        URLConnection con = utcluj.openConnection();

        BufferedReader in = new BufferedReader(

                                new InputStreamReader(

                                con.getInputStream()));

        String inputLine;

        while ((inputLine = in.readLine()) != null)

            System.out.println(inputLine);

        in.close();

    }

}

Many HTML pages contain the formats (text fields, buttons, etc.) that allow data entry and transmission to the server. After filling in the fields, press a button and the web browser will send the data to the appropriate URL. The URL that receives the data is a **cgi-bin script**. It processes the data and sends a response to the client, which is usually another HTML page.

Java programs can interact with cgi-bin scripts. They must be able to write data to a URL. This is done using the **URLConnection** class. The following listing exemplifies how a Java program can interact with **a cgi-bin script**.

try{

       String data=””;

        data += URLEncoder.encode("nume") + "=" + URLEncoder.encode("Adi”);

        data += "&" + URLEncoder.encode("nota”) + "=" + URLEncoder.encode("8.50”);

        URL url = new URL(“http://193.226.6.117:80/test/test.php");

        URLConnection conn = url.openConnection();

        conn.setDoOutput(true);

        OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream());

        wr.write(data);

        wr.flush();

        // Get the response

        BufferedReader rd = new BufferedReader(new InputStreamReader(conn.getInputStream()));

        String line;

        while ((line = rd.readLine()) != null) {

            System.out.println(line);

         }

        rd.close();

        wr.close();

}catch(Exception ex){ex.printStackTrace();}

In the previous listing, it is assumed that there is a cgi-bin test.php script to which the java program sends two variables: nume = adi and nota = 8.50. The cgi script will read those variables and send a response to the java program. To test the previous listing, return to the laboratory that represents the servlets.

The following listing shows a complete example of an applet that reads the contents of a file that is identified by a URL. In this case, the applet even reads its own .class file.

*import java.applet.\*;*

*import java.awt.\*;*

*import java.net.\*;*

*import java.io.\*;*

*public class FileURL extends Applet*

*{*

*byte[] appletCode; //read storage file*

*public void init()*

*{*

*try {*

*// Create a URL object*

*URL url = new URL(getCodeBase(),*

*getClass().getName()+".class");*

*// Opens the connection towards a URL*

*URLConnection urlConn = url.openConnection();*

*// The use of ByteArrayOutputStream as a temporary container.*

*// After reading, it will be converted to an array*

*ByteArrayOutputStream tempBuffer;*

*tempBuffer = new ByteArrayOutputStream();*

*// Creation of a reading straeam towards a URL*

*InputStream instream = urlConn.getInputStream();*

*// Reading the contend of the URL in a temporary buffer*

*int ch;*

*while ((ch = instream.read()) >= 0) {*

*tempBuffer.write(ch);*

*}*

*// Convert the temporary buffer to an array*

*appletCode = tempBuffer.toByteArray();*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*public void paint(Graphics g)*

*{*

*g.setColor(Color.black);*

*if (appletCode == null) {*

*g.drawString("Nu s-a citit fisierul .class",*

*10, 30);*

*} else {*

*g.drawString("Lungimea fisierului class. "+*

*appletCode.length+" bytes .", 10, 30);*

*}*

*}*

*}*