Programación en Red en Java

El paquete java.net contiene muchas herramientas que se usan para establecer conectividades de red. Dentro del paquete esta la clase URL (Uniform Resource Locator), la cual le permite leer un recurso URL y escribirlo en un objeto. El URL es la dirección de un recurso en internet.

Para emplear un recurso en internet se crea un objeto URL que contenga la dirección del recurso deseado después se lee la información delobjeto URL usando el método openStream(), el cual hace que la lectura de información a través de una red sea tan fácil como leer de un flujo de entrada.

Ejemplo programa que muestra el código fuente de una pagina HTML usando un URL

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

public class VerFuente1{
      public static void main(String args[]){
            String linea;
            URL u;
            if(args.length>0){
                  try{
                        u=new URL(args[0]);
                        try{
                              BufferedReader is=new BufferedReader(
                              new InputStreamReader(u.openStream()));
                              try {
                                    while((linea=is.readLine())!=null){
                                          System.out.println(linea);
                                    }
                              }
                              catch(Exception e){ System.err.println(e);}
                        }
                        catch(Exception e){ System.err.println(e);}
                  }
                  catch(MalformedURLException e){ System.err.println(e);}
            }
      }
}
```

Sockets

Un socket es un punto de comunicación por el cual un proceso puede emitir o recibir información. Los sockets son una innovación del UNIX de Berkeley. Estos permiten tratar una conexión a la red como otro flujo de bytes del que se puede leer o escribir. Históricamente los sockets son la extensión de una de las ideas más importantes de UNIX: que toda la entrada/salida debe parecerle entrada/salida de archivos al programador. Los sockets esconden del programador los detalles de bajo nivel de la red.

Un socket puede realizar siete operaciones básicas.

- Conectarse a una maquina remota.
- Enviar datos.
- Recibir datos.
- Cerrar una conexión.
- Ligarse a un puerto.
- Escuchar datos entrantes.
- Aceptar conexiones de maquinas remotas en el puerto al que se ligo.

Sockets clientes
Los sockets clientes se usan de la forma siguiente:

1.- El nuevo socket es creado usando un constructor Socket().
2.- El socket intenta conectarse a un host remoto.
3.- Una vez que la conexión es establecida los hosts local y remoto obtienen flujos de entrada y salida para el socket y usan esos flujos para enviarse datos uno a otro. Esta conexión es full-duplex, ambos hosts pueden enviar y recibir datos simultáneamente. Lo que significan los datos depende del protocolo. Normalmente se pondrían de acuerdo sobre algún handshaking antes de transmitir datos de uno a otro.
4.- Cuando la transmision de datos esta completa, uno o ambos cierra la conexión.

Como ejemplo usaremos uno de los protocolos más simples el llamado “dayTime”. En este protocolo el cliente abre un socket en el puerto 13 en el servidor dayTime; en respuesta el servidor envia el tiempo en un formato legible por las personas y cierra la conexión.

//stdin- teclado
//stdout-pantalla
//stderr-siempre pantalla

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

public class daytimeClient {
  public static void main(String[] args) {
    Socket theSocket;
    String hostname;
    DataInputStream theTimeStream;

    if (args.length > 0) { hostname = args[0];}
    else { hostname = "localhost";}
    try {
      theSocket = new Socket(hostname, 13);//1 y 2
      //                                        InputStream byte a la vez
      theTimeStream = new DataInputStream(theSocket.getInputStream());//3
      String theTime = theTimeStream.readLine();
      System.out.println("It is " + theTime + " at " + hostname);
    }  // end try
    catch (UnknownHostException e) { System.err.println(e);}
    catch (IOException e) {System.err.println(e);}
  }  // end main
} // end daytimeClient
```

Como un ejemplo mas de cliente otro programa que muestra el código fuente de una pagina HTML, esta vez usando un socket.

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

public class VerFuente {
      public static void main(String [] args){
      Socket cliente;
      PrintStream salida;
      BufferedReader entrada;
      String host ="localhost";
      String userAgent="MiCliente";
      String accept="text/html,text/plain,*/*";
      String request="index.html";
      String linea="";
      int puerto=23;

      if(args.length>1){
            host=args[0];
            request=args[1];
      }
      try {
            cliente=new Socket(host, puerto) ;
            salida=new PrintStream(cliente.getOutputStream());
            entrada=new BufferedReader(new
            InputStreamReader(cliente.getInputStream()));
            salida.print("GET "+request+"HTTP/1.0\r\n"+
                        "User-Agent: "+userAgent+"\r\n"+
                        "Accept: "+accept+"\r\n"+
                       "\r\n");
            while((linea=entrada.readLine())!=null){
                  System.out.println(linea);
            }

      }
      catch(IOException e){ System.err.println(e);}
      }
}
```

Sockets para Servidores
La clase ServerSocket contiene todo lo que se necesita para escribir sockets en Java.

El ciclo básico de vida de un servidor es:

1.- Un ServerSocket nuevo es creado en un puerto particular usando el constructor ServerSocket().
2.- El ServerSocket() escucha los intentos de conexión entrantes en ese puerto usando su metodo accept(). Accept() se bloquea hasta que un cliente intenta realizar una conexión, en dicho punto accept() regresa un Socket que conecta el cliente al servidor.

3.- Dependiendo del tipo de servidor, el metodo getInputStream() o el metodo getOutputStream del socket será llamado. Para obtener flujos de entrada y salida que comunican con el cliente.
4.- El servidor y el cliente interactuan de acuerdo a un protocolo establecido hasta que es tiempo de cerrar la conexión.
5.- El servidor el cliente, o ambos, cierran la conexión.
6.- El servidor regresa al paso 2, y espera la siguiente conexión.

El codigo siguiente es una implementación del servidor dayTime.

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

public class daytimeServer {

  public final static int daytimePort = 13;

  public static void main(String[] args) {

   ServerSocket theServer;
   Socket theConnection;
   PrintStream p;

   try {
     theServer = new ServerSocket(daytimePort);
     try {
       while (true) {
         theConnection = theServer.accept();
         p = new PrintStream(theConnection.getOutputStream());
         p.println(new Date());
         theConnection.close();
       }
     }
     catch (IOException e) {
       theServer.close();
       System.err.println(e);
     }
   }  // end try
   catch (IOException e) { System.err.println(e);}
  }
}
```

A continuación otro ejemplo de cliente-servidor: un tablero de gato distribuido. Primero se muestra el código del servidor.

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

public class ServiByte implements Runnable
{
      static Vector flujos= new Vector();
      Socket cliente;
```

```
public ServiByte(Socket cliente){
	this.cliente=cliente;
}
public void run(){
	try {
		OutputStream salida=cliente.getOutputStream();
		InputStream entrada=cliente.getInputStream();
		handle(entrada,salida);
	}
	catch (Exception ex){ex.printStackTrace();}
	finally {
		try {
			socket.close();
		}
		catch(IOException ignorada){}
	}
}
void handle(InputStream entrada, OutputStream salida) throws
IOException {
	try {
		synchronized(flujos){
			flujos.addElement(salida);
		}
		procesaMensajes(entrada);
	}
	finally {
		flujos.removeElement(salida);
	}
}
void procesaMensajes(InputStream entrada) throws IOException {
	int msg=-1;
	boolean error=false;
	while(true){
		error=false;
		try {
			msg=entrada.read();
		} catch ( IOException e) {
			error=true;
		}
		if(!error){
			broadcastMsg(msg);
		}
	}
}
static void broadcastMsg(int msg)
{
	for(int i=0; i<flujos.size();++i){
		OutputStream salida=(OutputStream)flujos.elementAt(i);
		try {
			salida.write(msg);
			salida.flush();
		}
		catch(IOException ex){ ex.printStackTrace();}
	}
}
public static void main(String [] args) throws IOException {
```

```
            ServerSocket serv=new ServerSocket(23);
            while(true){
                  try {
                        ServiByte gestor=new ServiByte(serv.accept());
                        new Thread(gestor).start();
                  }
                  catch (IOException ex){ ex.printStackTrace();}
            }
      }
}
```

Ahora se muestra el código del cliente

```
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.net.*;

public class Gato extends Applet implements Runnable, ActionListener{
      Thread hilo;
      Socket cliente;
      OutputStream salida;
      InputStream entrada;
      Button botones[]=new Button[9];
      boolean turno = true;
      String host="localhost";
      int puerto=23;
      Label etiq;

      public void init(){
            setLayout(new GridLayout(3,3));
            for(int i=0; i<9; i++){
                  add(botones[i]=new Button(""+i));
                  botones[i].addActionListener(this);
            }
            bolean error=true;
            while(error)
            {
                  error=false;
                  System.out.println("Esperando por el servidor . . .");
                  try {

                        cliente=new Socket(host, puerto);
                  } catch ( IOException e) {
                              System.out.println(e);
                              error=true;
                  }
            }
            System.out.println("Connectado al servidor.");
            try {
                  salida=cliente.getOutputStream();
            } catch ( IOException e) {System.out.println(e);}
            try {
                  entrada=cliente.getInputStream();
            } catch ( IOException e) { System.out.println(e);}
```

```
      hilo = new Thread (this);
      hilo.start ();
}
public void run()
{
      boolean error=false;
      int posicion=0;
      while(true)
      {
            error=false;
            try {
                  posicion=entrada.read();
            } catch ( IOException e) {
                  error=true;
            }
            if (!error) {
                  if(turno){
                        botones[posicion].setLabel("X");
                        etiq.setText("Es el turno de: el cero");
                  }
                  else {
                        botones[posicion].setLabel("O");
                        etiq.setText("Es el turno de: la equis");
                  }
                  botones.[posicion]setEnabled(false);
                  turno=!turno;
            }
      }
}
public void actionPerformed(ActionEvent e)
{
      Button btn=(Button)e.getSource();

      if(turno){
            btn.setLabel("X");
            etiq.setText("Es el turno de: el cero");
      }
      else {
            btn.setLabel("O");
            etiq.setText("Es el turno de: la equis");
      }
      btn.setEnabled(false);
      turno=!turno;
      for(int posicion=0;posicion<9;posicion++)
            if(botones[posicion]==btn)
                  try {
                        salida.write(posicion);
                  } catch (IOException ee){System.err.println(e);}
}
public static void main(String args[])
{
      Gato g=new Gato();
      g.init();
      g.start();
      Frame f=new Frame("Gato");
      f.add(g, BorderLayout.CENTER);
      f.add("South", g.etiq=new Label("Es el turno de: la equis"));
```

```
                f.setVisible(true);
        }
    }
```