**Subir:** [HLP 05-06](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/doc0506.html) **Anterior:** [7. Las clases Java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node8.html)   **[Índice General](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node1.html)**   
  
**Subsecciones**

* [8.1 Introducción](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00910000000000000000)

* [8.2](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00920000000000000000) *[Sockets](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00920000000000000000)*

* + [8.2.1 La clase Socket](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00921000000000000000)

* + [8.2.2 La clase ServerSocket](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00922000000000000000)

* [8.3 La clase](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00930000000000000000) *[Thread](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00930000000000000000)* [y la implementación de servidores](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00930000000000000000)

* [8.4 Ejercicios](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00940000000000000000)

* [8.5 Códigos Fuente del Servidor de chistes iterativo](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00950000000000000000)

* [8.6 Códigos Fuente del Servidor de chistes concurrente](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html" \l "SECTION00960000000000000000)

**8. Las clases Java Socket y ServerSocket**

El objetivo de esta sesión práctica es mostrar el modo de funcionamiento de las clases Java para definir *sockets* de flujo (*stream*).

**8.1 Introducción**

El paradigma *Cliente/Servidor* es quizás el más conocido de los paradigmas para aplicaciones de red. Se usa para describir un modelo de interacción entre dos procesos, que se ejecutan de forma simultánea. Este modelo es una comunicación basada en una serie de preguntas y respuestas, que asegura que si dos aplicaciones intentan comunicarse, una comienza la ejecución y espera indefinidamente que la otra le responda y luego continua con el proceso.

|  |
| --- |
| \includegraphics[width=0.5\textwidth]{img/clienserv.eps} |
| **Figura 8.1:** Paradigma Cliente/Servidor |

Los dos componentes del paradigma son:

**Cliente**: aplicación que inicia la comunicación, es dirigida por el usuario.

**Servidor**: es quien responde a los requerimientos de los clientes, son procesos que se están ejecutando indefinidamente.

Los procesos clientes son más sencillos que los procesos de los servidores, los primeros no requieren de privilegios de sistemas para funcionar, en cambio los procesos servidores sí.

Los usuarios cuando quieren acceder a un servicio de red, ejecutan un software cliente. El diseño de los servidores debe ser muy cuidadoso, debe incluir código para la manipulación de:

* *autenticación*: verificar la identidad del cliente.
* *seguridad de datos*: para que estos no puedan ser accedidos inapropiadamente.
* *privacidad*: garantizar que la información privada de un usuario, no sea accedida por alguien no autorizado.
* *protección*: asegurar que las aplicaciones no monopolicen los recursos del sistema.
* *autorización*: verificar si el cliente tiene acceso al servicio proporcionado por el servidor.

La mayoría de las comunicaciones punto-a-punto en las redes (incluida Internet), están basadas en el modelo Cliente/Servidor. Desde el punto de vista Internet/Intranet, se tendría:

* Un servidor es un ordenador remoto - en algún lugar de la red - que proporciona información según petición.
* Un cliente funciona en su ordenador local, se comunica con el servidor remoto, y pide a éste información.
* El servidor envía la información solicitada.

Un único servidor típicamente sirve a una multitud de clientes, ahorrando a cada uno de ellos el problema de tener la información instalada y almacenada localmente.

**8.2 *Sockets***

Normalmente, un servidor se ejecuta en una máquina específica y tiene un *socket* asociado a un número de puerto específico. El servidor simplemente espera a la escucha en el *socket* a que un cliente se conecte con una petición. El cliente conoce el *nombre de la máquina* sobre la que está ejecutándose el servidor y el *número de puerto* al que está conectado. Solicitar una conexión consiste en intentar establecer una cita con el servidor en el puerto de la máquina servidora.

|  |
| --- |
| \includegraphics[width=0.5\textwidth]{img/socketnuevo.eps} |
| **Figura 8.2:** *socket* Servidor |

Si todo va bien, el servidor acepta la conexión. Pero antes, el servidor crea un nuevo *socket* en un puerto diferente. Es necesario crear un nuevo *socket* (y consecuentemente un número de puerto diferente) de forma que en el *socket* original se continue a la escucha de las peticiones de nuevos clientes mientras se atiende a las necesidades del cliente conectado. En el cliente, si se acepta la conexión, el *socket* se crea satisfactoriamente y se puede utilizar para comunicarse con el servidor.

|  |
| --- |
| \includegraphics[width=0.5\textwidth]{img/socketcliente.eps} |
| **Figura 8.3:** *socket* Cliente |

Un *socket* es el extremo final de un enlace punto-a-punto que comunica a dos programas ejecutándose en una red.

Los *sockets* siempre están asociados a un número de puerto que es utilizado por TCP para identificar la aplicación a la que está destinada la solicitud y poder redirigirsela.

**8.2.1 La clase Socket**

La clase Socket del paquete java.net es fácil de usar comparada con la que proporcinan otros lenguajes. Java oculta las complejidades derivadas del establecimiento de la conexión de red y del envío de datos a través de ella. En esencia, el paquete java.net proporciona la misma interfaz de programación que se utiliza cuando se trabaja con archivos.

**Ejemplo 1**

El siguiente ejemplo, [ClienteFecha.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/ClienteFecha.java), muestra la implementación de un cliente que accede al servicio UNIX ``fecha y hora". El servidor concreto al que se conecta es al ``localhost". El servicio ``fecha y hora", por convenio, siempre está en el puerto 13. Lo que ocurre es que el software del servidor está ejecutándose continuamente en la máquina remota, esperando cualquier tráfico de red que ``*hable con él*" en el puerto 13. Cuando el Sistema Operativo de este servidor recupera un paquete de red que contiene una petición para conectar con el puerto 13, activa el servicio de escucha del servidor y establece la conexión, que permanece activa hasta que es finalizada por alguna de las dos partes.

import java.net.\*;

import java.io.\*;

import java.util.\*;

class ClienteFecha {

public static void main( String[] args ) {

String servidor = "localhost";

int puerto = 13; // puerto de daytime

try {

// Se abre un socket conectado al servidor y al

// puerto estándar de echo

Socket socket = new Socket( servidor,puerto );

System.out.println( "Socket Abierto." );

// Se consigue el canal de entrada

BufferedReader entrada = new BufferedReader(

new InputStreamReader( socket.getInputStream() ) );

System.out.println( "Hora actual en localhost:" );

System.out.println( "\t"+entrada.readLine() );

System.out.println( "Hora actual con la clase date:" );

System.out.println( "\t" + new Date() );

// Se cierra el canal de entrada

entrada.close();

// Se cierra el socket

socket.close();

} catch( UnknownHostException e ) {

System.out.println( e );

System.out.println(

"Debes estar conectado para que esto funcione bien." );

} catch( IOException e ) {

System.out.println( e );

}

}

}

**Ejemplo 2**

El ejemplo [EchoClient.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/EchoClient.java) muestra la implementación de un cliente que accede al servicio UNIX ``eco". El servidor concreto al que se conecta es a ``manis'' en la Escuela. El servicio ``eco", por convenio, siempre está en el puerto 7. Aunque con frecuencia por razones de seguridad está cerrado.

import java.io.\*;

import java.net.\*;

public class EchoClient {

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

String serverName = "exthost.csi.ull.es";

int portNumber = 7;

Socket echoSocket = null;

PrintWriter out = null;

BufferedReader in = null;

try {

echoSocket = new Socket(serverName, portNumber);

out = new PrintWriter(echoSocket.getOutputStream(), true);

in = new BufferedReader(new InputStreamReader( echoSocket.getInputStream()));

} catch (UnknownHostException e) {

System.err.println("Don't know about host: " + serverName);

System.exit(1);

} catch (IOException e) {

System.err.println("Couldn't get I/O for " + "the connection to: " + serverName);

System.exit(1);

}

BufferedReader stdIn = new BufferedReader( new InputStreamReader(System.in));

String userInput;

while ((userInput = stdIn.readLine()) != null) {

out.println(userInput);

System.out.println("echo: " + in.readLine());

}

out.close();

in.close();

stdIn.close();

echoSocket.close();

}

}

**8.2.2 La clase ServerSocket**

La clase ServerSocket es la que se utiliza a la hora de crear servidores, al igual que como se ha visto, la clase Socket se utilizaba para crear clientes.

**Ejemplo**

Este ejemplo muestra cómo escribir un servidor y su cliente. Está sacado del Tutorial de Java de Sun. El servidor sirve chistes. Funciona de la siguiente forma:

miranda:~/clases/psd/> java KnockKnockClient

Server: Knock! Knock!

Who's there?

Client: Who's there?

Server: Turnip

Turnip who?

Client: Turnip who?

Server: Turnip the heat, it's cold in here! Want another? (y/n)

n

Client: n

Server: Bye.

miranda:~/clases/psd/>

El ejemplo consta de dos programas Java ejecutándose de forma independiente KnockKnockClient y KnockKnockServer. Sin embargo, está constituido por tres ficheros:

* [KnockKnockServer.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/KnockKnockServer.java) (implementación del Servidor)

* [KnockKnockProtocol.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/KnockKnockProtocol.java) (implementación del protocolo)

* [KnockKnockClient.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/KnockKnockClient.java) (implementación del cliente)

**8.3 La clase *Thread* y la implementación de servidores**

Existe un problema con el ejemplo del servidor de chistes de la sección anterior. Suponga que queremos permitir que varios usuarios se conecten a la vez. Lo normal es que un servidor esté ejecutándose constantemente en un ordenador, y que los usuarios se conecten simultáneamente al mismo. En el ejemplo que hemos visto, sólo se admite la conexión de un usuario. Esto podemos arreglarlo usando *threads*.

Cada vez que sepamos que el programa ha establecido una nueva conexión, esto es, siempre que una petición de servicio tenga éxito, lanzaremos un nuevo *thread* que será el encargado de monitorizar la conección entre el servidor y ese cliente. El programa principal sólo se encargará de seguir esperando nuevas conexiones. Para implementar esto, el bucle principal del servidor debería ser algo como:

while (true) {

Socket incoming = s.accept();

Thread t = new ThreadServerHandler(incoming);

t.start();

}

La clase ThreadServerHandler extiende a la clase Thread y contiene el bucle de comunicación entre el servidor y el cliente en su método run().

class ThreadServerHandler extends Thread {

...

public void run() {

try {

// Establecer los flujos de entrada/salida para el socket

// Procesar las entradas y salidas según el protocolo

// cerrar el socket

}

catch (Excepction e) {

// manipular las excepciones

}

}

}

**Ejemplo**

Este ejemplo amplía al de la sección anterior mostrando cómo escribir un servidor que atiende a múltiples clientes. El modo de funcionamiento es exactamente el mismo.

El ejemplo consta de dos programas Java ejecutándose de forma independiente KnockKnockClient y KKMultiServer. Para probarlo en una terminal lance al servidor, y en dos o más nuevas terminales lance a varios clientes. La aplicación está constituida por los siguientes ficheros:

* [KnockKnockProtocol.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/KnockKnockProtocol.java) (implementación del protocolo, no cambia respecto al ejemplo anterior)

* [KnockKnockClient.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/KnockKnockClient.java) (implementación del cliente, no cambia respecto al ejemplo anterior)

* [KKMultiServerThread.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/KKMultiServerThread.java) (implementación de un *thread* Servidor)

* [KKMultiServer.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/KKMultiServer.java) (implementación del Servidor)

**8.4 Ejercicios**

1. Modificar el ejemplo *Cliente de Eco* del enunciado de manera que:
   * Se especifique en la línea de comandos el nombre de la máquina servidora y el número de puerto. Si no se especifica nada, el servidor por defecto será ``*localhost*'' y el puerto el número 7.
   * Para indicar el final de una sesión cliente el usuario ha de introducir por teclado un punto ``**.**''. Cuando se introduzca un punto se ha de salir del bucle de entrada y se ha de cerrar el *socket* de datos.

Solución:

[EchoClient.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/ej1/EchoClient.java)

1. Implementar un servidor de ``eco'' como el que proporcionan los servidores Unix en el puerto 7. Diseñe un servidor iterativo, para ello:
   * Se ha de especificar en la línea de comandos el número de puerto en el que el servidor acepta conexiones (por ejemplo, 8180). Si no se especifica nada, el número de puerto por defecto será el 7.
   * El servidor se ha de quedar esperando las solicitudes de conexión de los clientes. Utilizando la clase BufferedReader junto con la clase InputStreamReader se ha de abrir un flujo de entrada desde el *socket* servidor. Con la clase PrintWriter junto con la clase OutputStreamWriter abrir un flujo de salida al *socket*. El programa ha de actuar como repetidor, recogiendo las líneas que llegan por el canal de entrada y escribiéndolas en el canal de salida, hasta que el usuario le indique que ha terminado, escribiendo un punto ``**.**''. Cuando ya no haya más líneas que leer, se recibirá un punto, lo cual hará que el servidor salga del bucle de entrada y cierre el *socket* servidor.
   * Para probar que funciona, ejecute el programa servidor en un consola, y en otra terminal escriba: telnet 127.0.0.1 8180.

Solución:

[EchoServer.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/ej2/EchoServer.java)

[MyStreamSocket.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/ej2/MyStreamSocket.java)

1. Con los dos programas anteriores realice las siguientes operaciones:
   * Ejecute los programas empezando por el servidor y a continuación el cliente.
   * En una terminal diferente arranque a otro cliente.

Dibuje el diagrama de secuencia. ¿Se pueden realizar las dos sesiones en paralelo? Explique su respuesta.

1. Modificar el servidor de ``eco'' del ejercicio [2](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html#ej:ser), para que sea un servidor concurrente.

Solución:

[EchoServer.java](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/src/ej4/EchoServer.java)

1. Con el programa anterior y el *cliente de eco* del ejercicio [1](http://nereida.deioc.ull.es/%7Ecleon/doctorado/doc06/doc06/html/node9.html#ej:cli) realice las siguientes operaciones:
   * Ejecute los programas empezando por el servidor y a continuación el cliente.
   * En una terminal diferente arranque a otro cliente.

Dibuje el diagrama de secuencia. ¿Se pueden realizar las dos sesiones del cliente en paralelo? Explique su respuesta.

1. Describa las diferencias, desde el punto de vista del cliente, entre un servidor iterativo y un servidor concurrente para un servicio que involucre múltiples rondas de intercambios de mensajes.

**8.5 Códigos Fuente del Servidor de chistes iterativo**

import java.net.\*;

import java.io.\*;

public class KnockKnockProtocol {

private static final int WAITING = 0;

private static final int SENTKNOCKKNOCK = 1;

private static final int SENTCLUE = 2;

private static final int ANOTHER = 3;

private static final int NUMJOKES = 5;

private int state = WAITING;

private int currentJoke = 0;

private String[] clues = { "Turnip", "Little Old Lady", "Atch", "Who", "Who" };

private String[] answers = { "Turnip the heat, it's cold in here!",

"I didn't know you could yodel!",

"Bless you!",

"Is there an owl in here?",

"Is there an echo in here?" };

public String processInput(String theInput) {

String theOutput = null;

if (state == WAITING) {

theOutput = "Knock! Knock!";

state = SENTKNOCKKNOCK;

} else if (state == SENTKNOCKKNOCK) {

if (theInput.equalsIgnoreCase("Who's there?")) {

theOutput = clues[currentJoke];

state = SENTCLUE;

} else {

theOutput = "You're supposed to say \"Who's there?\"! " +

"Try again. Knock! Knock!";

}

} else if (state == SENTCLUE) {

if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {

theOutput = answers[currentJoke] + " Want another? (y/n)";

state = ANOTHER;

} else {

theOutput = "You're supposed to say \"" +

clues[currentJoke] +

" who?\"" +

"! Try again. Knock! Knock!";

state = SENTKNOCKKNOCK;

}

} else if (state == ANOTHER) {

if (theInput.equalsIgnoreCase("y")) {

theOutput = "Knock! Knock!";

if (currentJoke == (NUMJOKES - 1))

currentJoke = 0;

else

currentJoke++;

state = SENTKNOCKKNOCK;

} else {

theOutput = "Bye.";

state = WAITING;

}

}

return theOutput;

}

}

import java.io.\*;

import java.net.\*;

public class KnockKnockClient {

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

Socket kkSocket = null;

PrintWriter out = null;

BufferedReader in = null;

try {

kkSocket = new Socket("localhost", 4444);

out = new PrintWriter(kkSocket.getOutputStream(), true);

in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));

} catch (UnknownHostException e) {

System.err.println("Don't know about host: manis.csi.ull.es.");

System.exit(1);

} catch (IOException e) {

System.err.println("Couldn't get I/O for the connection to: manis.csi.ull.es.");

System.exit(1);

}

BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));

String fromServer;

String fromUser;

while ((fromServer = in.readLine()) != null) {

System.out.println("Server: " + fromServer);

if (fromServer.equals("Bye.")) break;

fromUser = stdIn.readLine();

if (fromUser != null) {

System.out.println("Client: " + fromUser);

out.println(fromUser);

}

}

out.close();

in.close();

stdIn.close();

kkSocket.close();

}

}

import java.net.\*;

import java.io.\*;

public class KnockKnockServer {

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

ServerSocket serverSocket = null;

try {

serverSocket = new ServerSocket(4444);

System.out.println("estoy después de crear el socket");

} catch (IOException e) {

System.err.println("Could not listen on port: 4444.");

System.exit(1);

}

Socket clientSocket = null;

try {

clientSocket = serverSocket.accept();

System.out.println("estoy después de aceptar un cliente");

} catch (IOException e) {

System.err.println("Accept failed.");

System.exit(1);

}

PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

BufferedReader in = new BufferedReader(

new InputStreamReader(

clientSocket.getInputStream()));

String inputLine, outputLine;

KnockKnockProtocol kkp = new KnockKnockProtocol();

outputLine = kkp.processInput(null);

out.println(outputLine);

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

outputLine = kkp.processInput(inputLine);

out.println(outputLine);

if (outputLine.equals("Bye."))

break;

}

out.close();

in.close();

clientSocket.close();

serverSocket.close();

}

}

**8.6 Códigos Fuente del Servidor de chistes concurrente**

import java.net.\*;

import java.io.\*;

public class KKMultiServer {

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

ServerSocket serverSocket = null;

boolean listening = true;

try {

serverSocket = new ServerSocket(4444);

} catch (IOException e) {

System.err.println("Could not listen on port: 4444.");

System.exit(-1);

}

while (listening)

new KKMultiServerThread(serverSocket.accept()).start();

serverSocket.close();

}

}

import java.net.\*;

import java.io.\*;

public class KKMultiServerThread extends Thread {

private Socket socket = null;

public KKMultiServerThread(Socket socket) {

super("KKMultiServerThread");

this.socket = socket;

}

public void run() {

try {

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader in = new BufferedReader( new InputStreamReader(

socket.getInputStream()));

String inputLine, outputLine;

KnockKnockProtocol kkp = new KnockKnockProtocol();

outputLine = kkp.processInput(null);

out.println(outputLine);

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

outputLine = kkp.processInput(inputLine);

out.println(outputLine);

if (outputLine.equals("Bye"))

break;

}

out.close();

in.close();

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}