

# Java Avanzado

---

## Comunicaciones TCP/IP



## Copyright

- Copyright (c) 2004  
José M. Ordax
- Este documento puede ser distribuido solo bajo los términos y condiciones de la Licencia de Documentación de javaHispano v1.0 o posterior.
- La última versión se encuentra en  
<http://www.javahispano.org/licencias/>

# Introducción



Un sistema de comunicaciones se compone de una pila de niveles encargados de distintas tareas.

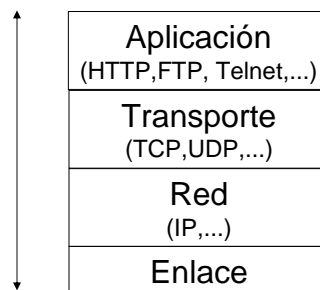


A este modelo teórico se le llama el Modelo de Referencia OSI.

## Introducción (cont.)



En una red TCP/IP (por ejemplo Internet) algunos niveles se fusionan quedando la siguiente pila:



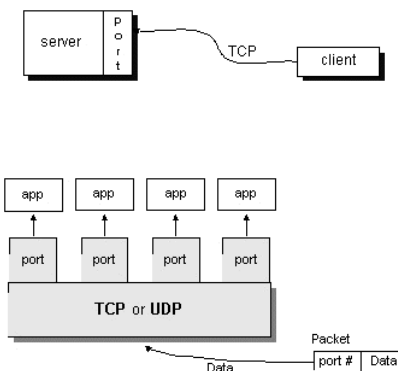
Cuando programamos en Java, solo nos preocupamos del nivel de Aplicación. Java y el Sistema Operativo ya se encargan del resto.

## Introducción (cont.)

- Las máquinas tienen habitualmente una conexión a la red por la que le van a llegar todos los datos.
- Pero en una misma máquina puede haber n aplicaciones esperando datos.
- ¿Cómo se sabe para que aplicación son los datos? Gracias a los puertos.
- Para enviar algo por una red TCP/IP se direcciona mediante una dirección y un puerto:
  - Dirección: especifica la máquina destino.
  - Puerto: especifica la aplicación destino.

## Introducción (cont.)

- Gráficamente:



## Introducción (cont.)



Algunas direcciones especiales:



127.0.0.1: Conocida con el nombre de LoopBack. Apunta a la propia máquina.



192.168.1.x: Utilizadas para las redes locales privadas conectadas a Internet (por ejemplo vía ADSL).



Algunos puertos conocidos:



21: Servidores FTP.



23: Servidores Telnet.



25: Servidores SMTP.



80: Servidores Web.

## Introducción (cont.)



Siglas:



HTTP: HyperText Transport Protocol.



FTP: File Transport Protocol.



TCP: Transmission Control Protocol.



UDP: User Datagram Protocol.



IP: Internet Protocol.



URL: Uniform Resource Locator.



Todas las clases referentes a la comunicación en redes TCP/IP se encuentran en `java.net`.\*

# URL

- Una URL es el identificador unívoco de un recurso en Internet.
- Una URL consta de las siguientes partes:
  - Protocolo.
  - Dirección de la máquina (host).
  - Puerto.
  - Path.
- Por ejemplo:  
`http://download.eclipse.org:80/downloads/index.php`

## Creando una URL

- En Java, la URL está implementada por la clase `java.net.URL`
- Lo mas sencillo es utilizar el constructor donde se le pasa la URL como un String:  
`URL aURL = new URL("http://www.ibm.com/index.html");`
- Esta es una URL absoluta.
- Pero también podemos crear URLs relativas mediante otro constructor:  
`URL base = new URL("http://www.ibm.com/");  
URL final = new URL(base,"index.html");`

## Creando una URL (cont.)

- Existen mas constructores donde se puede especificar por separado el puerto, el path, etc....
- Todos los constructores de la clase `java.net.URL` tienen la siguiente cláusula throws:  
`public URL(String spec) throws MalformedURLException`
- Por tanto, siempre tendremos que construir las URLs en un bloque try & catch donde recojamos la excepción `java.net.MalformedURLException`

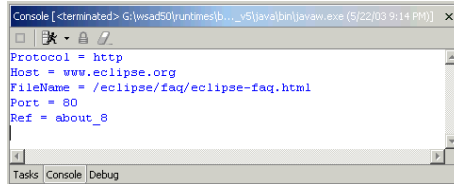
## Atributos de una URL

- Podemos acceder a los atributos de una URL mediante los siguientes métodos:
  - `getProtocol()`: Devuelve el protocolo de la URL.
  - `getHost()`: Devuelve el host (dirección) de la URL.
  - `getPort()`: Devuelve el puerto de la URL.
  - `getFile()`: Devuelve el path del recurso.
  - `getRef()`: Devuelve la referencia dentro del recurso.

# Ejemplo

```
import java.net.MalformedURLException;
import java.net.URL;

public class URLTest
{
    public static void main(String[] args)
    {
        try
        {
            URL aURL = new URL("http://www.eclipse.org:80/eclipse/faq/eclipse-faq.html#about_8");
            System.out.println("Protocol = " + aURL.getProtocol());
            System.out.println("Host = " + aURL.getHost());
            System.out.println("FileName = " + aURL.getFile());
            System.out.println("Port = " + aURL.getPort());
            System.out.println("Ref = " + aURL.getRef());
        }
        catch (MalformedURLException ex)
        {
            ex.printStackTrace();
        }
    }
}
```



```
Console [ <terminated> G:\wsad50\runtimes\jre\bin\java.exe (5/22/03 9:14 PM) ]
Protocol = http
Host = www.eclipse.org
FileName = /eclipse/faq/eclipse-faq.html
Port = 80
Ref = about_8
```

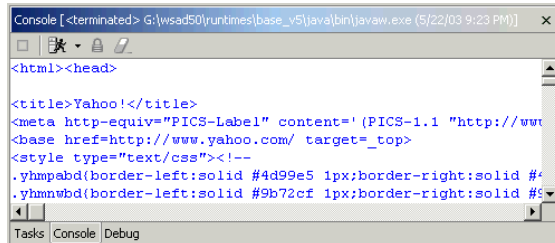
## Leyendo de una URL

- La clase `java.net.URL` tiene un método para poder acceder a su stream de entrada.
- Este stream es una instancia de la clase `java.io.InputStream`
- El método es:  
`public final java.io.InputStream openStream() throws java.io.IOException`
- A partir del stream ya podremos leer directamente o usar cualquier tipo de filtro sobre este stream.

# Ejemplo

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

public class LeerURLTest
{
    public static void main(String[] args)
    {
        try
        {
            URL yahoo = new URL("http://www.yahoo.com/");
            BufferedReader in = new BufferedReader(new InputStreamReader(yahoo.openStream()));
            String aux;
            while ((aux = in.readLine()) != null)
                System.out.println(aux);
            in.close();
        }
        catch (MalformedURLException ex)
        {
        }
        catch (IOException ex)
        {
        }
    }
}
```



## Conectando con una URL

- Una vez que hemos creado una URL, podemos establecer una conexión con ella a través del método:

```
public java.net.URLConnection openConnection() throws java.io.IOException
```

- Mediante una conexión podremos establecer un diálogo con la URL: escribir y leer.

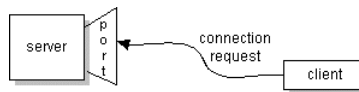
Nota: Una URL no tiene porque ser solo un recurso estático como una página HTML o una imagen. También puede ser un programa CGI, un script PHP, un Servlet Java, una página JSP, etc.....

- Lo lograremos mediante los métodos `getOutputStream` y `getInputStream`.



# Socket

- Habitualmente, tenemos una aplicación servidora esperando conexiones de un cliente en un puerto.
- Y tenemos una aplicación cliente que quiere conectar con ese servidor en ese puerto.
- Un socket es el link entre una aplicación servidora y un puerto.



## Socket (cont.)

- Cuando un cliente conecta con el servidor se crea un nuevo socket a un nuevo puerto.
- De esta forma, el servidor puede seguir esperando conexiones en el socket principal y comunicarse con el cliente conectado.
- De igual forma se establece un socket en el cliente en un puerto local.
- Por tanto, la comunicación se establece entre dos sockets.



## Socket vs URL

- Las URLs son un caso específico del trabajo con sockets. Las clases Java relacionadas con las URLs utilizan por debajo sockets.
- Para el desarrollador es más fácil usar las clases URL, URLConnection, etc.... que los sockets directamente.
- Pero hay muchos casos donde se requieren clases de comunicación de bajo nivel. Por ejemplo, la implementación de un servidor, el uso de un protocolo propietario, etc....

## Socket (cont.)

- Las clases Java que implementan los sockets son:
  - java.net.Socket
  - java.net.SocketServer
- java.net.Socket sirve para establecer un socket en un cliente. Establecer la conexión con un socket servidor.
- java.net.SocketServer sirve para establecer un socket en un servidor. Poder escuchar posibles conexiones desde sockets cliente.

# java.net.Socket



Trabajar con un socket cliente comprende los siguientes pasos:



Abrir el socket: mediante uno de sus constructores, habitualmente dándole la dirección y el puerto destino.



Abrir el stream de entrada y/o de salida: mediante los métodos `getInputStream()` y `getOutputStream()`.



Leer y/o escribir al socket: mediante los métodos de los streams o los filtros que hayamos creado.



Cerrar los streams: mediante el método `close()`.



Cerrar el socket: mediante el método `close()`.

## Atributos de un socket



Podemos acceder a los atributos de un socket mediante los siguientes métodos:



`getInetAddress()`: Devuelve la dirección destino (remota).



`getPort()`: Devuelve el puerto destino (remoto).



`getLocalAddress()`: Devuelve la dirección origen (local).



`getLocalPort()`: Devuelve el puerto origen (local).

## Ejemplo

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

public class EchoClient
{
    public static void main(String[] args)
    {
        try
        {
            Socket clientSocket = new Socket("192.168.1.2",1234);
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),true);
            BufferedReader in = new BufferedReader(new InputStreamReader
                (clientSocket.getInputStream()));

            System.out.println("Enviando: ¡¡¡¡Ecoooo!!!!");
            out.println("¡¡¡¡Ecoooo!!!!");

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

            out.close();
            in.close();
            clientSocket.close();
        }
        catch (UnknownHostException e) { }
        catch (IOException e) { }
    }
}
```

## java.net.ServerSocket



Trabajar con un socket servidor comprende los siguientes pasos:



Abrir el socket: mediante uno de sus constructores, habitualmente dándole el puerto donde escuchar.



Esperar una petición cliente: mediante el método accept() que devuelve el nuevo socket.



Abrir el stream de entrada y/o de salida: mediante los métodos getInputStream() y getOutputStream().



Leer y/o escribir al socket: mediante los métodos de los streams o los filtros que hayamos creado.



Cerrar los streams: mediante el método close().



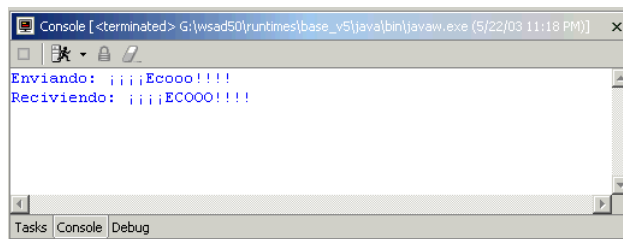
Cerrar el socket: mediante el método close().

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

## Ejemplo

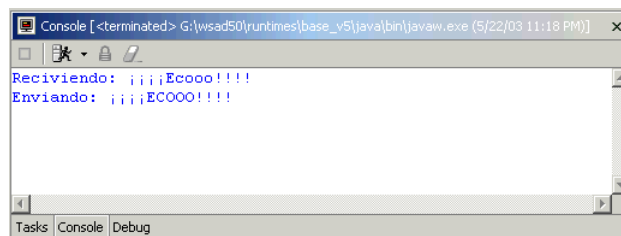
```
public class EchoServer
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket serverSocket = new ServerSocket(1234);
            Socket clientSocket = serverSocket.accept();
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader
                (clientSocket.getInputStream()));
            String inputLine = in.readLine();
            System.out.println("Recibiendo: " + inputLine);
            String outputLine = inputLine.toUpperCase();
            System.out.println("Enviando: " + outputLine);
            out.println(outputLine);
            out.close();
            in.close();
            clientSocket.close();
            serverSocket.close();
        }
        catch(IOException ex) { }
    }
}
```

## Ejemplo



Cliente

Servidor



# Aceptar múltiples clientes



Para desarrollar un servidor que pueda gestionar mas de un cliente a la vez debemos:



Introducir la ejecución del método `accept()` en un bucle. Este bucle podrá ser infinito o controlado, dependiendo de como queramos terminar el servidor.



Abrir un thread por cada conexión recibida, pasándole como parámetro el nuevo socket generado.

## Ejemplo

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

public class EchoMultiServer
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket serverSocket = new ServerSocket(1234);
            while(true)
            {
                new echoThread(serverSocket.accept()).start();
            }
            serverSocket.close();
        }
        catch(IOException ex)
        {
        }
    }
}
```

# Ejemplo

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

public class echoThread extends Thread
{
    private Socket clientSocket = null;

    public echoThread(Socket param)
    {
        this.clientSocket = param;
    }

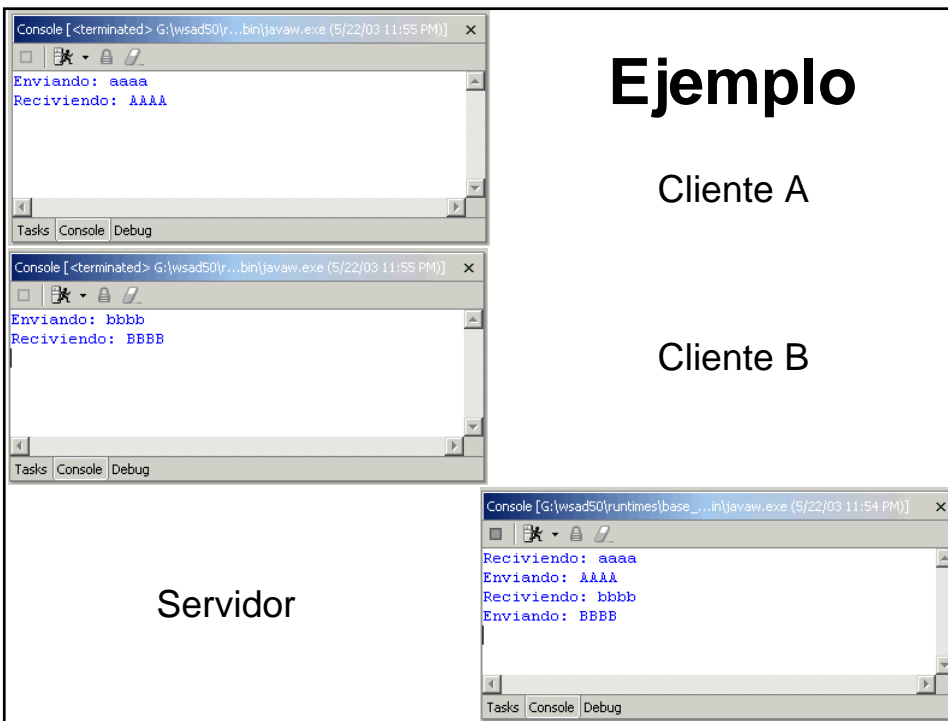
    public void run()
    {
        try
        {
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            String inputLine = in.readLine();
            System.out.println("Recibiendo: " + inputLine);
            String outputLine = inputLine.toUpperCase();
            System.out.println("Enviando: " + outputLine);
            out.println(outputLine);
            out.close();
            in.close();
            clientSocket.close();
        }
        catch(IOException ex) { }
    }
}
```

# Ejemplo

Cliente A

Cliente B

Servidor

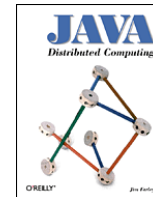


# Bibliografía

Java Network Programming (2nd edition)  
Elliote Rusty Harold.  
O'Reilly.



Java Distributed Computing  
Jim Farley.  
O'Reilly.



Java Network Programming  
M. Hughes, M. Shoffner, D. Hamner y C. Hughes.  
Manning Publications Company .



The Java tutorial (on-line)  
<http://java.sun.com/docs/books/tutorial/networking/index.html>

# Bibliografía

TCP/IP Network Administration (3rd edition)  
Craig Hunt.  
O'Reilly.



Windows TCP/IP Network Administration  
Craig Hunt y Robert Bruce Thompson.  
O'Reilly.



Networking Personal Computers (with TCP/IP)  
Craig Hunt.  
O'Reilly.

