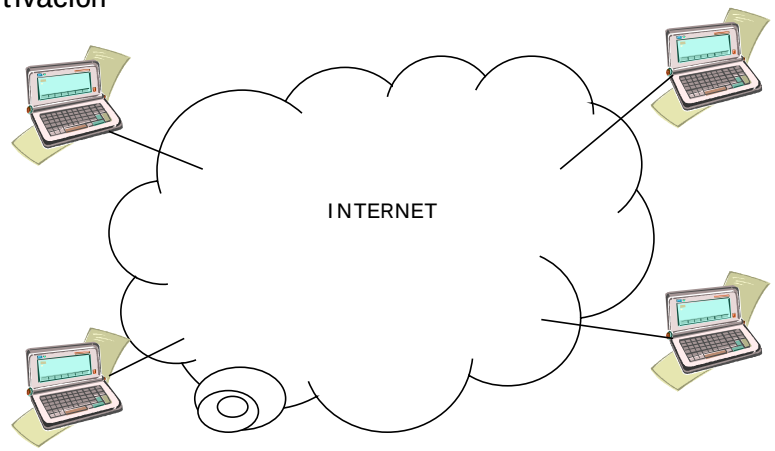
Capítulo XXII: InternetWorking[29]

Motivación



La red de redes de computadores más grande del mundo es la que conocemos en este momento como Internet. Tanto en las esferas universitarias, personales y en Hollywood siempre ha sido un mundo atrayente, ya que el mundo virtual y la magia de estar conectados desde la comodidad de su hogar con otros usuarios a través del mundo en solo segundos es realmente algo que a nuestra mente le agrada mucho.

¿Cómo es posible que podamos enviar un mensaje a esa nebulosa, que conocemos como Internet y que le llegue a nuestro destinatario sin ningún problema? ¿Cómo es que con la diversidad de lenguas, los computadores de la China nos entiendan? ¿Cómo es posible que un Hacker pueda pinchar la línea internet y capturar información y cómo las empresas se protegen contra ellos?

Pues esas preguntas no son tan complejas y podremos entenderlas un poco en este capítulo.

Conceptos

> *Una Red de Computadores es una agrupación de una serie de estaciones y servidores que se conectan a través de cables y que pueden comunicarse entre sí utilizando una forma estándar que entienden entre sí.*

Internet no es más que una gran red de computadores que abarca todo el mundo, un concepto bastante simple si pensamos que en el día de hoy lo estamos utilizando durante al menos 4 horas al día en promedio, y que para muchos es realmente su propia fuente de trabajo.

[29] Basado en el material entregado por el profesor Nelson Baloian

Entonces ¿de qué forma se comunican los computadores a través de la red?. La definición formal habla de una forma común. A eso le llaman Protocolo.

> *Un Protocolo es un formato con el cual se envía la información y que es compartida tanto por el generador como por el receptor.*

En efecto, todos los computadores utilizan protocolos de red para comunicarse. Es así como se usaban antes protocolos Novell, UDP, NetBIOS y TCP. Internet usa el protocolo TCP/IP para comunicarse y actualmente es el protocolo más usado por los distintos tipos de computadores.

¿Cómo funciona?

Veamos ahora algunas definiciones más técnicas del protocolo:

Sintaxis[30]

El protocolo TCP/IP se caracteriza por transportas los paquetes de información a través de una red hacia otro computador que está direccionado según un número IP con el siguiente formato:

x	x	x	.	x	x	x	.	x	x	x	.	x	x	x

Por ejemplo:

164.182.219.12
10.10.10.1
255.255.0.1

Fíjate que cada número entre los puntos está entre 0 y 255, por lo que es posible guardar un número IP en 4 bytes.

El número IP es un número único que identifica a cada terminal dentro de la red. No pueden haber dos número iguales, pues entran en conflicto. Físicamente podemos decir que es como el rut de cada computador. Es la forma en que la red se da cuenta dónde está cada terminal físicamente (no geográficamente, sino que en qué final del cable está).

La clase InetAddress en Java identifica un terminal dentro de la red. De esta forma, un objeto de tipo InetAddress perfectamente puede representar a tu computador:

```
...
InetAddress miTerminal = InetAddress.getLocalHost();
System.out.println(miTerminal.getHostAddress());
...
```

[30] Todas las clases de este capítulo están en el package java.net

Este pequeño código muestra el número IP de tu computador dentro de la red donde estés conectado (esto inclusive puede indicar en qué número IP estás dentro de Internet).

Otra forma de lograr llegar a un terminal dentro de la red es usando el *hostname*. Este nombre es el nombre que enmascara la IP y generalmente debe pasar a través de un DNS (Domain Name Server) para identificarse. Es así como para encontrar un terminal usando su hostname se puede hacer de la siguiente forma:

```
...
InetAddress terminal = InetAddress.getByname(hostname);
System.out.println(terminal.getHostName());
System.out.println(terminal.getHostAddress());
...
```

Veamos un ejemplo que funcione:

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

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

            try {
                  InetAddress yo = InetAddress.getLocalHost();
                  System.out.println("Mi nombre : " +
                        yo.getHostName());
                  System.out.println("Mi IP : " +
                        yo.getHostAddress());
                  System.out.println("Mi clase : " +
                        iPClass(yo.getAddress()));
            }
            catch (UnknownHostException e) {
                  System.out.println("No me encontre");
            }
            BufferedReader kbd = new BufferedReader(
                  new InputStreamReader(System.in));
            String nombre;
            while (true) {
                  try {
                        System.out.print("Ingrese un host :");
                        System.out.flush();
                        nombre = kbd.readLine();
                        if (nombre.equals("Fin"))
                              break;
                        System.out.println("Lookup DNS: " +
                              nombre);
                        InetAddress remoto =
                              InetAddress.getByName(nombre);
                        System.out.println("El IP : " +
                              remoto.getHostAddress());
                        System.out.println("El Nombre : " +
                              remoto.getHostName());
                        System.out.println("La clase : " +
                              iPClass(remoto.getAddress()));
                  }
                  catch (UnknownHostException e) {
                        System.out.println("No lo encontre");
                  }
                  catch (Exception e) {
```



```
                        System.out.println("Problemas "+e);
                  }
            }

      }

      public static char iPClass(byte[] ip) {
            int byteMayor = 0xff & ip[0];
            if (byteMayor < 128 ) return 'A';
            if (byteMayor < 192 ) return 'B';
            if (byteMayor < 224 ) return 'C';
            if (byteMayor < 240 ) return 'D';
            return 'E';
      }
}
```

Corriendo este ejemplo en un terminal local aislado de internet, la salida es:

```
Mi nombre : nb_portal01
Mi IP : 127.0.0.1
Mi clase : A
Ingrese un host :www.dcc.uchile.cl
Lookup DNS: www.dcc.uchile.cl
No lo encontre
Ingrese un host :Fin
```

En cambio que si conectamos ese mismo terminal a una red de área local con conexión a internet tenemos que:

```
Mi nombre : nb_portal01
Mi IP : 192.168.0.172
Mi clase : C
Ingrese un host : www.dcc.uchile.cl
Lookup DNS : www.dcc.uchile.cl
El IP : 192.80.24.4
El Nombre : www.dcc.uchile.cl
La clase : C
Ingrese un host : Fin
```

Vemos que hay una gran diferencia en los resultados, pues porque en un caso, la dirección IP del terminal no existe (está desconectado) y por defecto asigna una dirección "nula" que es 127.0.0.1. En el segundo caso, la red asigna una dirección en forma dinámica (si, igual como lo hacen con tu computador cuando te conectas a internet en tu casa) pero tiene salida y va a buscar la IP a través del DNS del sitio www.dcc.uchile.cl. Es muy simple.

Existen otras clases y formas de comunicarse a través de la red. Es por eso que vamos a ver otros conceptos que nos servirán para entender más las maneras que Java tiene para utilizar TCP/IP.

Otros métodos de la clase InetAddress[31]:

Método	Descripción
boolean equals(Object obj)	Compares this object against the specified object.

[31] Sacado desde la API del JDK 1.3.1 ubicada en http://java.sun.com/j2se/1.3/docs/api/

Método	Descripción
byte[] getAddress()	Returns the raw IP address of this InetAddress object.
static InetAddress[] getAllByName(String host)	Determines all the IP addresses of a host, given the host's name.
static InetAddress getByName(String host)	Determines the IP address of a host, given the host's name.
String getHostAddress()	Returns the IP address string "%d.%d.%d.%d".
String getHostName()	Gets the host name for this IP address.
static InetAddress getLocalHost()	Returns the local host.
int hashCode()	Returns a hashcode for this IP address.
boolean isMulticastAddress()	Utility routine to check if the InetAddress is an IP multicast address.
String toString()	Converts this IP address to a String.

Conceptos

Uniform Resource Locator (URL) consiste en la dirección de un recurso que un servidor en la internet publica para que pueda ser leído por distintos usuarios de la internet.

Este concepto sencillo, pero muchas veces usado, nos permite acceder a muchos lugares de la internet en forma sencilla. La URL se compone de varias partes:

```
http  ://  www.dcc.uchile.cl  :80  /home.html
Protocolo          Hostname     Puerto  Recurso
```

Protocolo: Estándar de más alto nivel (sobre TCP/IP) que indica con qué formato se está transfiriendo la información. Por ejemplo: Hyper Text Transfer Protocol (http), File Transfer Protocol (ftp), Gopher o News.

Hostname: Es el nombre del servidor al cuál nos estamos conectando.

Puerto: Es el número del puerto en donde el servidor recibe el requerimiento. Cada uno de los protocolos tiene un número de puerto estándar, así es para http es el 80 y para ftp el 21, pero el servidor puede recibir requerimientos en otros puertos definidos por él.

Recurso: Es el archivo, directorio o información que el servidor tiene publicada y que el cliente desea leer.

Con estas definiciones tenemos que también son URL la siguiente lista:

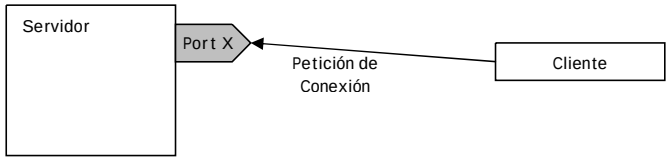
- http://www.google.cl
- ftp://jazzfortuna.sytes.net/Multimedia
- http://www.portal.cl:8080/mypage.jsp
- https://secure.network.com/var/spool/cgi?a=3

De esta misma formas, algo que no sabíamos antes es el tema del puerto:

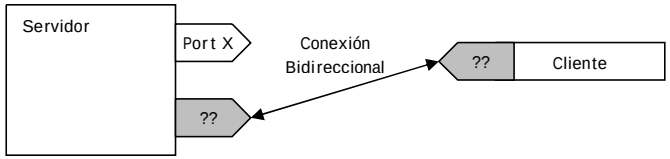
Un Socket es un link activo que permite comunicar dos puntos de una red. Un Puerto es un lugar donde se instaura el Socket para mantener la conexión.

Con esta definición podemos decir que un servidor puede tener muchos puertos abiertos en espera de una conexión directa y por supuesto el socket es el que permite la comunicación entre el servidor y el cliente.

El servidor está esperando con un socket abierto en el puerto x. Es así que el cliente realiza una petición de conexión al puerto, el cual si está activo, acepta la conexión:



Una vez que la conexión es aceptada, en el servidor se crea otro socket asociado a un port que normalmente ni siquiera lo conoce el cliente, liberando el "escuchador" para nuevos requerimientos.



Allí comienza la comunicación de lectura y/o escritura en el nuevo socket.

Ahora veamos como se hace esto en Java.

Sintaxis

Primero que todo, veamos como se usan las URL en Java. En este caso, existe una clase URL que nos permite trabajar con el destino de una URL:

```
...
URL miURL = new URL("http://www.dcc.uchile.cl:80/index.html");
...
```

Con esta sentencia, lo que hacemos es crear una referencia activa a la url http://www.dcc.uchile.cl:80. Otras formas de hacer lo mismo es:

```
...
URL miURL2 = new URL("http", "www.dcc.uchile.cl:80", "index.html");
URL miURL3 = new URL("http", "www.dcc.uchile.cl", 80,
                     "index.html");
```

```
...
```

Estos objetos hacen referencia al mismo url, pero usan los otros constructores de la clase URL. Otro punto importante es que al momento de crear un URL es que ésta puede lanzar una excepción MalformedURLException y debe ser atrapada. De esta forma quedaría mejor:

```
...
try {
      URL miURL = new URL("http://www.dcc.uchile.cl");
}
catch (MalformedURLException) {
      System.out.println("La URL es inválida");
}
...
```

Veamos entonces un ejemplo más completo que se conecta a Yahoo y obtiene el contenido de esa dirección a la salida estándar de Java:

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

public class PURL {
      public static void main(String[] args) throws Exception {
            URL yahoo = new URL("http://www.yahoo.com");
            URLConnection yc = yahoo.openConnection();
            BufferedReader in = new BufferedReader(
                                    new InputStreamReader(
                                          yc.getInputStream()));
            String inputLine;

            while ((inputLine = in.readLine()) != null)
                  System.out.println(inputLine);
            in.close();
      }
}
```

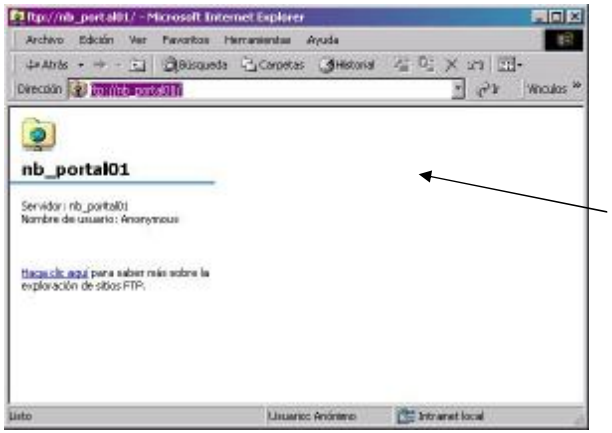
Al ejecutar este programa obtenemos (mostramos solo parte de):

```
<html><head>

<title>Yahoo!</title>
<meta http-equiv="PICS-Label" content='(PICS-1.1
"http://www.icra.org/ratingsv02.html" l r (cz 1 lz 1 nz 1 oz 1 vz 1)
gen true for "http://www.yahoo.com" r (cz 1 lz 1 nz 1 oz 1 vz 1)
"http://www.rsac.org/ratingsv01.html" l r (n 0 s 0 v 0 l 0) gen true
for "http://www.yahoo.com" r (n 0 s 0 v 0 l 0))'>
<base href=http://www.yahoo.com/ target=_top>
<style type="text/css"><!--
.yhmpabd{border-left:solid #4d99e5 1px;border-right:solid #4d99e5
1px;border-bottom:solid #4d99e5 1px;}
.yhmnwbd{border-left:solid #9b72cf 1px;border-right:solid #9b72cf
1px;}
.yhmnwbm{border-left:solid #9b72cf 1px;border-right:solid #9b72cf
1px;border-bottom:solid #9b72cf 1px;}
//--></style>
.
.
.
```


Como podemos ver obtenemos directamente el HTML que muestra la interfaz de inicio del sitio de Yahoo. ¿Para qué entonces sirve?. Piensa si quisieras hacer búsquedas o hacer tu propio browser. Probemos ahora cambiando la dirección de Yahoo por la de un FTP vacío:

```
<html>
<head>
<title>Directory: /@nb_portal01</title>
</head>
<body>
<h2>Directory: /@nb_portal01</h2>
<pre>
  <a href="ftp://nb_portal01/../"><img align="middle"
src="doc:/lib/images/ftp/directory.gif" border=0 width=24 height=26>
&lt;Parent Directory&gt;/</a>
</pre></body>
</html>
```

La salida es similar (también es HTML) pero ¿qué de especial tiene?, pues que si pones lo mismo en un browser, puedes ver el contenido del servidor FTP:



Otros métodos de la clase URL[32]:

Método	Descripción
URL(String spec)	Creates a URL object from the String representation.
URL(String protocol, String host, int port, String file)	Creates a URL object from the specified protocol, host, port number, and file.
URL(String protocol, String host, int port, String file, URLStreamHandler handler)	Creates a URL object from the specified protocol, host, port number, file, and handler.
URL(String protocol, String host, String file)	Creates a URL from the specified protocol name, host name, and file name.
URL(URL context, String spec)	Creates a URL by parsing the given spec within a specified context.
URL(URL context, String spec, URLStreamHandler handler)	Creates a URL by parsing the given spec with the specified handler within a specified context.

[32] Sacado desde la API del JDK 1.3.1 desde http://java.sun.com/j2se/1.3/docs/api/

Método	Descripción
boolean equals(Object obj)	Compares two URLs.
String getAuthority()	Returns the authority part of this URL.
Object getContent()	Returns the contents of this URL.
Object getContent(Class[] classes)	Returns the contents of this URL.
String getFile()	Returns the file name of this URL.
String getHost()	Returns the host name of this URL, if applicable.
String getPath()	Returns the path part of this URL.
int getPort()	Returns the port number of this URL.
String getProtocol()	Returns the protocol name of this URL.
String getQuery()	Returns the query part of this URL.
String getRef()	Returns the anchor (also known as the "reference") of this URL.
String getUserInfo()	Returns the userInfo part of this URL.
int hashCode()	Creates an integer suitable for hash table indexing.
URLConnection openConnection()	Returns a URLConnection object that represents a connection to the remote object referred to by the URL.
InputStream openStream()	Opens a connection to this URL and returns an InputStream for reading from that connection.
boolean sameFile(URL other)	Compares two URLs, excluding the "ref" fields.
protected void set(String protocol, String host, int port, String file, String ref)	Sets the fields of the URL.
protected void set(String protocol, String host, int port, String authority, String userInfo, String path, String query, String ref)	Sets the specified 8 fields of the URL.
static void setURLStreamHandlerFactory(URLStreamHandlerFactory fac)	Sets an application's URLStreamHandlerFactory.
String toExternalForm()	Constructs a string representation of this URL.
String toString()	Constructs a string representation of this URL.

Con URL entonces se tiene acceso de lectura para todos aquellos servicios que pueden ser alcanzados usando un URL. Sin embargo cuando se desea mayor control sobre la comunicación, es necesario usar sockets.

Para conectarse con un socket, solo basta conocer la dirección de éste y el puerto en el cual reside el socket, el resto del trabajo lo hace el mismo socket.

```
...
Socket llamada = new Socket(host, port);
...
```

De esta forma estamos conectándonos con el Socket ubicado en el host y port indicados. Después de la llamada, el cliente se queda en espera a que el servidor conteste y le permita conectarse a través del nuevo socket dedicado a la conexión.

En el caso del socket, éste puede retornar un problema y es recomendado atrapar la excepción UnknownHostException. De esta forma quedaría el programa:

```
...
try {
      Socket s = new Socket("anakena.dcc.uchile.cl", "25");
}
catch (UnknownHostException e) {
```



```
      System.out.println("Error: Host desconocido");
}
...
```

Así si el servidor no responde, o no existe, el programa enviaría un mensaje de Host desconocido.

Veamos un ejemplo práctico de Sockets

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

public class Sockets {
      public static void main(String args[]) {
            String host = "anakena.dcc.uchile.cl";
            int port = 25;
            try {
                  Socket s = new Socket(host, port);
                  System.out.println("The local host   : " +
                        s.getLocalAddress());
                  System.out.println("The local port   : " +
                        s.getLocalPort());
                  System.out.println("The remote host  : " +
                        s.getInetAddress());
                  System.out.println("The remote port  : " +
                        s.getPort());
                  System.out.println("The SoTimeout is : " +
                        s.getSoTimeout());
                  System.out.println("The SoLinger is  : " +
                        s.getSoLinger());
            }
            catch (Exception e) {
                  System.out.println("Error: "+e);
            }
      }
}
```

Este programa se tratará de conectarse al puerto 25 del servidor anakena.dcc.uchile.cl y la salida, después de un rato que el programa espera la respuesta, se ve como la que sigue:

```
The local host   : nb_portal01.stgo.codelco.cl/165.182.190.22
The local port   : 2420
The remote host  : anakena.dcc.uchile.cl/192.80.24.3
The remote port  : 25
The SoTimeout is : 0
The SoLinger is  : -1
```

Local Host y Port nos muestran el computador donde está corriendo el programa y el puerto en donde crea el socket local. En Remote Host y Port nos muestra el destino al cual queremos llegar, incluyendo IP. Por último el timeout del socket y de mantención de conexión (-1 es infinito).

Veamos ahora un ejemplo en donde escribimos en el Socket para que el servidor nos devuelva información: Un cliente de correos:

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

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

public class cliente_pop {

      public static void main(String[] args){
            Socket s;
            String cmd;
            InetAddress a;
            BufferedReader in;
            PrintWriter out;

            if (args.length!=3){
                  System.out.println("Uso: cliente_pop <host>
                                    <username> <password>\n\n");
                  System.exit(0);
            }

            try {
                  s=new Socket(args[0], 110);
                  System.out.println("Creado socket: "+s);

                  in=new BufferedReader(new
                        InputStreamReader(s.getInputStream()));
                  out=new PrintWriter(s.getOutputStream(), true);

                  System.out.println("Creado buffered in=" +
                        in.toString());
                  System.out.println(in.readLine());

                  //Envio login
                  out.println("user "+ args[1]);
                  System.out.println(in.readLine());

                  //Envio password
                  out.println("pass "+ args[2]);
                  System.out.println(in.readLine());

                  out.println("stat");

                  System.out.println(in.readLine());

                  out.println("list");

                  String line;
                  while ( !(line = in.readLine()).equals("."))
                        System.out.println(line);

            } catch (IOException e) {
                  System.out.println("Imposible Crear socket\n");
            }
      }
}
```

Si ejecutamos con una cuenta válida en el servidor de pop3.terra.cl, tenemos la salida:

```
Creado socket:
Socket[addr=pop3.terra.cl/200.28.216.13,port=110,localport=2447]
Creado buffered in=java.io.BufferedReader@113750
+OK POP3 server ready (6.5.034)
<77C193C192304628EA6DA4763A1C196922931EC3@genesis.terra.cl>
+OK Password required
```



```
+OK 27 messages
+OK 27 1081620
+OK
1 1241
2 35581
3 2705
4 361330
5 332819
6 2701
7 5211
8 21391
9 8989
10 15639
11 12506
12 21773
13 2997
14 14366
15 33907
16 21454
17 21716
18 21583
19 3631
20 21178
21 4066
22 20871
23 41523
24 29331
25 8563
26 12476
27 2072
```

¡¡Tengo 27 correos en mi inbox!! ¡¡Increíble!!

Veamos los métodos adicionales de la clase Socket[33]:

Método	Descripción
Socket(InetAddress address, int port)	Creates a stream socket and connects it to the specified port number at the specified IP address.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Creates a socket and connects it to the specified remote address on the specified remote port.
Socket(String host, int port)	Creates a stream socket and connects it to the specified port number on the named host.
Socket(String host, int port, InetAddress localAddr, int localPort)	Creates a socket and connects it to the specified remote host on the specified remote port.
void close()	Closes this socket.
InetAddress getInetAddress()	Returns the address to which the socket is connected.
InputStream getInputStream()	Returns an input stream for this socket.
boolean getKeepAlive()	Tests if SO_KEEPALIVE is enabled.
InetAddress getLocalAddress()	Gets the local address to which the socket is bound.
int getLocalPort()	Returns the local port to which this socket is bound.
OutputStream getOutputStream()	Returns an output stream for this socket.
int getPort()	Returns the remote port to which this socket is connected.
int getReceiveBufferSize()	Gets the value of the SO_RCVBUF option for this Socket, that is the buffer size used by the platform for input on this Socket.
int getSendBufferSize()	Get value of the SO_SNDBUF option for this Socket, that is the buffer size used by the platform for output on this Socket.

[33] Sacado desde la API del JDK 1.3.1 ubicada en http://java.sun.com/j2se/1.3/docs/api/

Método	Descripción
int getSoLinger()	Returns setting for SO_LINGER.
int getSoTimeout()	Returns setting for SO_TIMEOUT.
boolean getTcpNoDelay()	Tests if TCP_NODELAY is enabled.
void setKeepAlive(boolean on)	Enable/disable SO_KEEPALIVE.
void setReceiveBufferSize(int size)	Sets the SO_RCVBUF option to the specified value for this Socket.
void setSendBufferSize(int size)	Sets the SO_SNDBUF option to the specified value for this Socket.
static void setSocketImplFactory(SocketImplFactory fac)	Sets the client socket implementation factory for the application.
void setSoLinger(boolean on, int linger)	Enable/disable SO_LINGER with the specified linger time in seconds.
void setSoTimeout(int timeout)	Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.
void setTcpNoDelay(boolean on)	Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).
void shutdownInput()	Places the input stream for this socket at "end of stream".
void shutdownOutput()	Disables the output stream for this socket.
String toString()	Converts this socket to a String.

Ahora que vemos como el cliente se conecta con un servidor, veamos como esto se ve por parte del servidor.

Conceptos

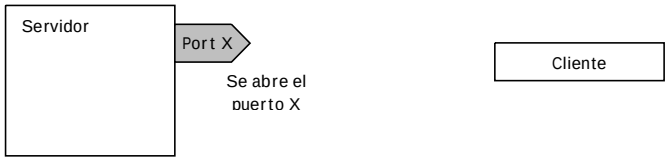
> *Un Servidor es un terminal en la red que permite compartir recursos con otros terminales, a los que se le llama Clientes.*

En efecto, hemos visto como un terminal llama a otro para obtener recursos usando las URLs y las IP's, pero no hemos visto aún como estos terminales "servidores" pueden responder a los requerimientos de los clientes. Veamos la sintaxis de esto.

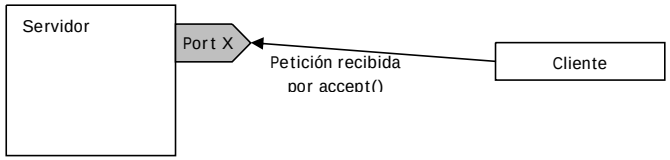
Sintaxis

Recordando como se comunican los clientes con los servidores y analizando paso a paso podemos ver como funciona.

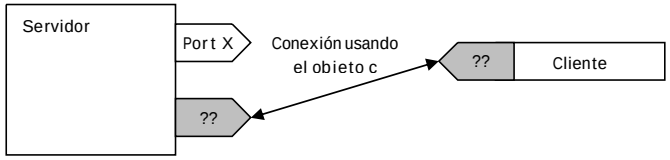
El servidor espera con un socket abierto en el puerto x.




```
...
ServerSocket s = new ServerSocket(X);
Socket c = s.accept();
...
```

El servidor se queda esperando el requerimiento del cliente en la línea del s.accept() y cuando éste hace la llamada, es capturada por la variable c que representa la conexión con el socket del cliente.



Desde este momento comienza la conexión. Cualquier requerimiento se puede obtener y enviar usando la variable c del cliente.

Veamos un ejemplo simple: Un servidor Echo. Este programa lo que hace es repetir lo que el cliente le envíe, y se lo envía a la salida del cliente. Ahora veremos el programa Servidor:

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

public class EchoServer {

      public static void main(String args[]) throws Exception {
            ServerSocket server = new ServerSocket(4445);

            System.out.println("Waiting for client...");
            Socket client = server.accept();
            System.out.println("Accepted from " +
                  client.getInetAddress());
            PrintWriter out =
                  new  PrintWriter(client.getOutputStream(),true);
            BufferedReader in = new  BufferedReader(new
                  InputStreamReader(client.getInputStream()));

            while (true) {
                  String line = in.readLine();
                  if (line.equals("**"))
                        break;
                  out.println(line);
            }

            client.close();
      }
}
```

El servidor abre un socket escuchador en el port 4445 y se queda con el mensaje "Waiting for client..." hasta que un cliente se conecta. Al momento de que se conecta un usuario solo muestra un mensaje del cliente que se conectó. Ahora veamos el programa cliente:

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

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

            Socket echoSocket = null;
            PrintWriter out = null;
            BufferedReader in = null;

            echoSocket = new Socket(args[0], 4445);
            out = new PrintWriter(echoSocket.getOutputStream(),
                  true);
            in = new BufferedReader(new
                  InputStreamReader(echoSocket.getInputStream()));

            BufferedReader stdIn = new BufferedReader(new
                  InputStreamReader(System.in));
            String userInput;
            System.out.println("echo starts...");

            while (true) {
                  userInput = stdIn.readLine();
                  out.println(userInput);
                  if (userInput.equals("**"))
                        break;
                  System.out.println("echo: " + in.readLine());
            }

            echoSocket.close();
      }
}
```

Como se puede ver, el cliente lo único que hace es recibir un texto y enviarlo al socket. Luego de que se ha enviado, es leído desde allá mismo para saber qué hay en el socket. Simple.

Otro ejemplo que también es muy simpático es el Talk, que envía mensajes de un lado a otro:

```
import java.net.*;
import java.io.*;
public class TalkServer {
      public static void main(String args[]) throws Exception {
            ServerSocket ss = null;
            PrintWriter out=null;
            BufferedReader in=null;
            Socket s;
            ss = new ServerSocket(4446);
            while(true) {
                  System.out.println("Waiting for a client ");
                  s = ss.accept();
                  System.out.println("Someone is calling: " +
                        s.getInetAddress());
                  in = new BufferedReader(new InputStreamReader(
                        s.getInputStream()));
                  while (true) {
```



```
                        String line = in.readLine();
                        if (line.equals("**"))
                              break;
                        else
                              System.out.println("Msg: "+line);
                  }
                  in.close();
                  s.close();
                  System.out.println("Call ended ");
            }
      }
}
```

```
import java.io.*;
import java.net.*;
class TalkClient {
      public static void main(String args[] ) throws Exception {
            PrintWriter outSocket = null;
            BufferedReader usin =  new BufferedReader(
                  new InputStreamReader(System.in));
            Socket s;
            while (true) {
                  System.out.print("Type hostname to call to: ");
                  String line = usin.readLine();
                  if (line.equals("end"))
                        break;
                  s = new Socket(line, 4446);
                  outSocket = new
                        PrintWriter(s.getOutputStream(),true);
                  System.out.println("Connected, start talking
                        (** for ending)");
                  while (true) {
                        System.out.print("? > ");
                        line = usin.readLine();
                        outSocket.println(line);
                        if (line.equals("**"))
                              break;
                  }
                  s.close();
                  outSocket.close();
                  System.out.println("call is over");
            }
      }
}
```

Échalo a correr y ve qué pasa cuando lo ejecutas.

Veamos los métodos que posee la clase ServerSocket[34]:

Método	Descripción
ServerSocket(int port)	Creates a server socket on a specified port.
ServerSocket(int port, int backlog)	Creates a server socket and binds it to the specified local port number, with the specified backlog.
ServerSocket(int port, int backlog, InetAddress bindAddr)	Create a server with the specified port, listen backlog, and local IP address to bind to.
Socket accept()	Listens for a connection to be made to this socket and accepts it.
void close()	Closes this socket.

[34] Sacado desde la API del JDK 1.3.1 ubicada en http://java.sun.com/j2se/1.3/docs/api/

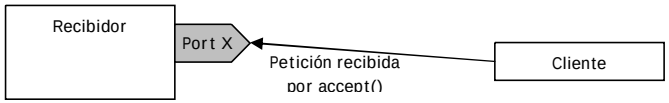
Método	Descripción
InetAddress getInetAddress()	Returns the local address of this server socket.
int getLocalPort()	Returns the port on which this socket is listening.
int getSoTimeout()	Retrive setting for SO_TIMEOUT.
protected void implAccept(Socket s)	Subclasses of ServerSocket use this method to override accept() to return their own subclass of socket.
static void setSocketFactory(SocketImplFactory fac)	Sets the server socket implementation factory for the application.
void setSoTimeout(int timeout)	Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.
String toString()	Returns the implementation address and implementation port of this socket as a String.

¿Qué pasa si abres 2 clientes al mismo servidor? Los requerimientos del segundo cliente quedan "encolados" hasta que el primero termine y luego puede continuar. Esto no es en la realidad de los servidores, porque ellos pueden recibir muchos clientes en forma concurrente. Es por eso que podemos ver este nuevo concepto.

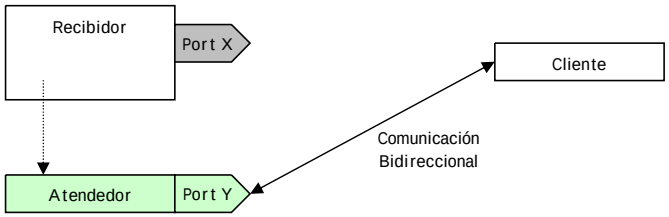
Los servidores concurrentes, entonces pueden atender a más de un cliente a la vez. Esto se puede hacer de varias formas, pero veamos una forma en que usaremos Threads.

> *Los Servidores en la red permiten muchas conexiones en forma simultánea, por lo que el programa que permite recibir las conexiones de los clientes debe ser "concurrente".*

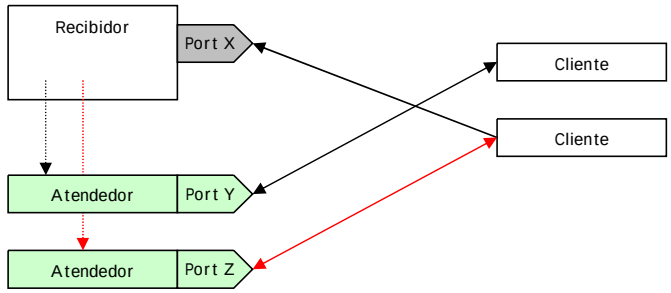
Esta es la típica problemática que hay en el día a día sobre internet. Un servidor web, un servidor de archivos, un servidor de chat, etc, que permiten que múltiples usuarios accedan a sus recursos compartidos.



El cliente solicita conexión y se conecta con el proceso recibidor del servidor.



El recibidor genera un nuevo proceso (Thread) con el programa que solo comunica ambos puntos (cliente y servidor) de esta forma haciendo que el recibidor quede libre y a la espera de un nuevo cliente.



De esta forma, cuando llega otro cliente, el recibidor lo puede atender y derivarlo a otro atendedor creado especialmente para la comunicación, sin interferir en el trabajo que tiene el primer atendedor y el primer cliente.

¿Cómo hacemos esto en formato Java?

Pues no es tan difícil como parece. Cómo la lógica de recibimiento y atención está por parte del servidor y no del cliente, los programas que hasta ahora vimos como "clientes" no se ven afectado en lo absoluto. En cambio el servidor sufre una pequeña alteración:

El programa principal es el Recibidor: Con esta declaración, y generalizando la forma de trabajo de los servidores, podemos llegar a que la parte recibidora del servidor quedaría con la siguiente sintaxis:

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

public class Recibidor {
      public static void main(String[] args) throws Exception {
            // Creación del puerto de escucha y recibimiento
            ServerSocket servidor = new ServerSocket(<port>);

            // Ciclo de espera por clientes
            while(true) {
                  // Obtenemos el socket del cliente
                  Socket cliente = servidor.accept();

                  // Creamos nuevo "atendedor" entregándole Socket
                  Thread atiende = new Atendedor(cliente);

                  // Atendemos al cliente
                  atiende.start();
            }
      }
}
```

Esta versión por supuesto no es la única que puedes usar y solo debe ser considerada como un ejemplo genérico dentro del contexto del problema.

Si analizamos un poquito el programa, podemos ver que el servidor siempre está escuchando por un cliente. Si no le llega un cliente, el programa suelta el procesador y sigue esperando hasta que alguien se conecte.

La Clase de Thread es el Atendedor: Tal como dice el nombre, la clase que extiende de Thread entonces sería el atendedor y es atendido usando el método run que Thread obliga a implementar:

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

public class Atendedor extends Thread {
	private Socket cliente;

	public Atendedor(Socket c) {
		this.cliente = c;
	}

	public void run() {
		// AQUI va la lógica de proceso entre el servidor y
		// el cliente.
	}
}
```

Tal como se ve, y si comparamos un poco con los servidores no concurrentes, lo único que hacemos es pasar la parte de proceso de la comunicación a un Thread. No es dificil, pero al principio cuesta comprender bien.

Ejemplos Prácticos

Existe en linux un programa que se llama Fortune. La idea de este programa es enviar mensajes de índole de fortuna al usuario que lo solicita. Veamos como se ve implementado con Java:

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

public class FortuneServer extends Thread {
	private ServerSocket myServer;

	public FortuneServer(int port) {
		System.out.print("Trying to opening port: " + port + "... ");
		try {
			this.myServer = new ServerSocket(port);
		}
		catch (IOException e) {
			System.out.println("ERROR: " + e.getMessage());
			System.exit(0);
		}
		System.out.println("Ok");
	}

	public void run() {
		System.out.println("Waiting for connections...");
		while (true) {
			Socket client = null;
			try {
				client = this.myServer.accept();
			}
			catch (IOException e) {
				System.out.println("ERROR: " + e.getMessage());
```



```
				System.exit(0);
			}
			System.out.println("-> Incoming from " +
				client.getInetAddress() + " on port " +
				client.getPort());
			FortuneConnection fc = new FortuneConnection(client);
			fc.start();
		}
	}

	static public void main(String[] args) {
		FortuneServer fs = new FortuneServer(4446);
		fs.run();
	}
}

class FortuneConnection extends Thread {
	private Socket client;

	public FortuneConnection(Socket client) {
		this.client = client;
	}

	public void run() {
		try {
			PrintWriter send = new PrintWriter(
				this.client.getOutputStream(), true);
			BufferedReader receive = new BufferedReader(
				new InputStreamReader(
				this.client.getInputStream()));
			send.println("% Hello!... What do you want?");
			while(true) {
				try {
					super.sleep(1);
				}
				catch(InterruptedException e) {
					System.out.println("ERROR: " +
						e.getMessage());
					System.exit(0);
				}
				String line = receive.readLine();
				if (".".equals(line))
					break;
				if ("cookie".equals(line)) {
					BufferedReader file =
						new BufferedReader(
						new FileReader("fortune.txt"));
					int x_max = Integer.parseInt(
						file.readLine());
					int x = (int) Math.round(Math.random() *
						x_max) + 1;
					System.out.println("(" + x + "/" +
						x_max + ")");
					String txt = "";
					for (int i=0; i<x; i++) {
						String l = file.readLine();
						if (l == null) break;
						txt = l;
					}
					file.close();
					send.println("% " + txt);
				}
				else {
					send.println("% I don't understand " +
						line);
				}
			}
			send.close();
			receive.close();
			System.out.println("<- Closing connection!");
			this.client.close();
```

```
                }
                catch (Exception e) {
                        System.out.println("ERROR: " + e.getMessage());
                }
        }
}
```

El concepto es bien simple. El Servidor recibe la llamada, espera el comando cookie, y responde con el mensaje que desea. Fíjate que en esta versión podemos interactuar con él directamente desde el cliente (consola, entrada estándar). Veamos como quedaría un programa con eso:

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

public class FortuneClient {
        public static void main(String[] args) throws Exception {
                Console c = new Console();
                InetAddress local = InetAddress.getLocalHost();
                c.print("Creating connection to " +
                        local.getHostAddress());
                Socket server = new Socket(local.getHostAddress(), 4446);
                c.print("port " + server.getPort() + "... ");
                PrintWriter send = new PrintWriter(server.getOutputStream(),
                        true);
                BufferedReader receive = new BufferedReader(
                        newInputStreamReader(server.getInputStream()));
                String line = receive.readLine();
                c.println("Ok!");
                c.println(line);
                while(true) {
                        c.print("> ");
                        String cmd = c.readLine();
                        send.println(cmd);
                        if (".".equals(cmd))
                                break;
                        line = receive.readLine();
                        c.println(line);
                }
                send.close();
                receive.close();
                c.print("Closing connection... ");
                server.close();
                c.println("Ok!");
        }
}
```

En este caso es una consola con la cual interactuamos al Servidor. Pero ¿y si hacemos una versión simple de línea de comando automática?:

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

public class Fortune {
        public static void main(String[] args) throws Exception {
                InetAddress local = InetAddress.getByName("nb_portal01");
                Socket server = new Socket(local.getHostAddress(), 4446);
                PrintWriter send = new PrintWriter(server.getOutputStream(),
                        true);
                BufferedReader receive = new BufferedReader(
                        newInputStreamReader(server.getInputStream()));
                String line = receive.readLine();
                send.println("cookie");
                line = receive.readLine();
                System.out.println(line);
                send.println(".");
                send.close();
```



```
                receive.close();
                server.close();
        }
}
```

Muy similar pero se ejecuta solo una vez.

Ahora, todos conocemos lo que son los Chats. Pues bien, imaginemos que queremos construir uno con esta tecnología. ¿Cómo lo hacemos?. Pues prefiero que lo veas con tus propios ojos:

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

public class ChatServer {
        public ServerSocket server;
        public List clients;

        public ChatServer(int port) {
                try {
                        this.server = new ServerSocket(port);
                        this.clients = new ArrayList();
                }
                catch (Exception e) {
                        System.err.println("ERROR: " + e.getMessage());
                        System.exit(0);
                }
        }

        public void startServer() {
                System.err.println("Esperandoconexiones...");
                while (true) {
                        try {
                                Socket client = this.server.accept();
                                clients.add(new PrintWriter(
                                        client.getOutputStream(), true));
                                System.err.println("[" +
                                        client.getInetAddress().getHostAddress()
                                        + "] Cliente conectado");
                                ChatConnection cc = new ChatConnection(client);
                                cc.start();
                        }
                        catch (Exception e) {
                                System.err.println("ERROR: " + e.getMessage());
                                System.exit(0);
                        }
                }
        }

        class ChatConnection extends Thread {
                private Socket client;
                private int n;

                public ChatConnection(Socket client) {
                        this.client = client;
                        this.n = clients.indexOf(client);
                }

                public void run() {
                        try {
                                BufferedReader in = new BufferedReader(
                                        new InputStreamReader(
                                        this.client.getInputStream()));
                                while (true) {
                                        String line = in.readLine();
                                        if (line == null) {
                                                System.err.println("[" +
                                                this.client.getInetAddress().
```

```
                                        getHostAddress()
                                        + "] Client disconnected");
                                        break;
                                }
                                for (int i=0; i<clients.size(); i++) {
                                        PrintWriter out = (PrintWriter)
                                                clients.get(i);
                                        out.println(line);
                                }
                        }
                        in.close();
                        client.close();
                        clients.remove(n);
                }
                catch (Exception e) {
                        System.err.println("ERROR: " + e.getMessage());
                }
        }
    }

    public static void main(String[] args) {
        if (args.length <= 0) {
                System.err.println("Uso: ChatServer <port>");
                System.exit(0);
        }
        ChatServer cs = new ChatServer(Integer.parseInt(args[0]));
        cs.startServer();
    }
}
```

En este caso la clase ChatServer es el "Recibidor" y es creada a partir de un programa principal que reside en la misma clase ChatServer. De esta forma es creado un objeto en el servidor de tipo ChatServer que lo único que se preocupa es de escuchar.

Por otro lado, existe una clase interna que se llama ChatConnection que es quien se preocupa de conectar el cliente con el servidor para recibir los mensajes de él.

Pero ¿cuál es la diferencia con el problema del Fortune?.

Simple. En este caso cuando el servidor recibe el mensaje de un cliente, debe enviar a todos los demás clientes conectados un mensaje. A esto se le conoce como Broadcasting.

Adjuntamos un posible cliente de este chat para que puedan ejecutarlo.

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

public class ChatClient {
    private Socket server;
    private String nick;

    private Frame ventana;
    private TextArea texto;
    private TextField mensaje;
    private Button salir;

    private PrintWriter out = null;

    public ChatClient(String server, int port, String nick) {
        ventana = new Frame("Java Chat v2.0");
        ventana.setLayout(new BorderLayout());
```



```
        mensaje = new TextField(100);
        mensaje.addActionListener(newMensajeListener());
        ventana.add(mensaje, "North");

        texto = new TextArea(25, 100);
        texto.setEnabled(true);
        texto.setBackground(Color.white);
        texto.setForeground(Color.black);
        ventana.add(texto, "Center");

        Panel opciones = new Panel();
        opciones.setLayout(newFlowLayout());
        salir = new Button("Salir");
        salir.addActionListener(newSalirListener());
        opciones.add(salir);
        ventana.add(opciones, "South");

        ventana.pack();
        ventana.show();

        try {
                this.nick = nick;
                this.server = new Socket(server, port);
                ChatClientReader ccr = new ChatClientReader();
                ccr.start();
                out = new PrintWriter(
                        this.server.getOutputStream(), true);
                out.println("*** " + nick + " se ha conectado ***");
        }
        catch(Exception e) {
                System.err.println("ERROR: " + e.getMessage());
                System.exit(0);
        }
    }

    class ChatClientReader extends Thread {

        public void run() {
                try {
                        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                server.getInputStream()));
                        while(true) {
                                String line = in.readLine();
                                if (line == null)
                                        break;
                                texto.append(line + "\n");
                        }
                        in.close();
                        server.close();
                        System.exit(0);
                }
                catch(Exception e) {
                        System.err.println("ERROR: " + e.getMessage());
                        System.exit(0);
                }
        }

    }

    class MensajeListener implements ActionListener {

        public void actionPerformed(ActionEvent x) {
                try {
                        out.println("[" + nick + "] " +
                                mensaje.getText());
                        mensaje.setText("");
                }
                catch(Exception e) {
                        System.err.println("ERROR: " + e.getMessage());
```

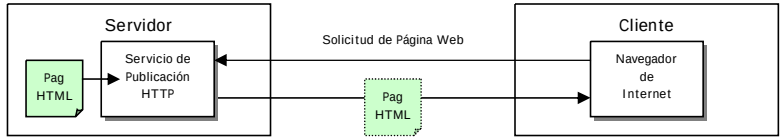
```
                                System.exit(0);
                        }
                }

        }

        class SalirListener implements ActionListener {

                public void actionPerformed(ActionEvent x) {
                        System.exit(0);
                }

        }

        static public void main(String[] args) throws Exception {
                BufferedReader stdin = new BufferedReader(
                        new InputStreamReader(System.in));
                System.out.print("   Nick > ");
                String nick = stdin.readLine();
                System.out.print(" Server > ");
                String server = stdin.readLine();
                System.out.print("   Port > ");
                String port = stdin.readLine();
                ChatClient cc = new ChatClient(server,
                                        Integer.parseInt(port),
                                        nick);
        }
}
```

Esta versión está hecha con interfaz gráfica para que puedan probarla.

Servlets

Motivación

Sobre la internet, existe el protocolo HTTP (HyperText Transfer Protocol) que permite enviar paquetes de textos a través del TCP/IP escritos en lenguaje HTML (HyperText Markup Language).



La figura muestra el proceso de cómo funciona la pertición o llamada de una URL desde tu browser (Microsoft Internet Explorer, Netscape Navigator, Opera, Mosaic, etc). Esto se puede explicar de la siguiente forma:

- El Cliente realiza una llamada al Servicio de Publicación Web o HTTP que se encuentra residente en el Servidor escuchando en el puerto 80 (normalmente).

- El Servicio de Publicación (también llamado Web Server y que no es más que un programa escuchador que genera un nuevo socket para responder al cliente) detecta la dirección y va a buscar el archivo que contiene el programa HTML que está pidiendo el usuario. Normalmente este programa es un archivo de texto plano con códigos HTML.

- El Web Server envía a través de la red entonces el contenido del archivo HTML al cliente directamente.

- El socket del Navegador recibe el archivo plano y lo interpreta para mostrar gráficamente lo que viene escrito en HTML.

¡¡Claro!!. El programa Navegador que está en el cliente es el programa "cliente" de nuestro servidor de publicación web. Entonces, él se preocupa de abrir el Socket con el web server.

¿Cuál es la gracia del HTML?

Algunas características es que se puede usar estilos, colores, imágenes, videos y muchos medios que hacen las páginas atractivas. De hecho si navegas en internet encontrarás que todo lo que ves es HTML (o casi todo).

La desventaja es que lo que el servicio de publicación envía es exactamente lo que contiene el archivo, el cuál es "estático", es decir, no cambia su contenido por nada del mundo. Pero ¿cómo entonces hacer cosas más "dinámicas"?.

Como respuesta a esa pregunta, existe algo que se llama DHTML (Dynamic HTML) que básicamente mezcla lo que es HTML estático y algo de Javascript (¡¡hey, no es Java!!). Si ves en una página botones que cambian de colores, algunas sorpresas, ventanas y cosas más llamativas aún, éstas se pueden hacer con DHTML.

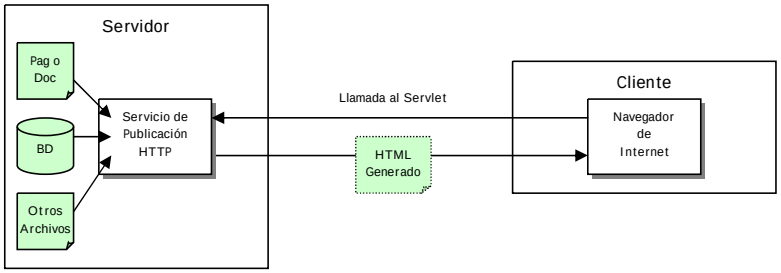
Sin embargo, el DHTML, al igual que el HTML, son interpretados en el lado del cliente. ¿Qué pasa si en el servidor tuviera una Base de Datos y quiero leer su información?. Pues, debemos recordar que el web server es un flojo y solo envía texto, por lo que nunca vamos a saber cómo leer la base de datos o qué información posea. A menos que...

Concepto

> *Un Servlet es un programa que reside en el servidor y que responde a llamadas usando URL (igual que las páginas web) en lenguaje HTML, pero cuya respuesta está dada por la lógica del programa.*

Los Servlets son programas básicamente hechos en Java. La diferencia con los programas estándar que hasta ahora conocemos es que se imprime directamente en el Socket para que el navegador capture el HTML que nosotros enviamos.

De esta forma el comportamiento se ve algo distinto ahora:



El Servlet también reside en un servicio de publicación, pero la ejecución genera HTML dependiendo del llamado que se haga del Servlet y la comunicación que mantenga el cliente con él.

Veamos realmente cómo funciona esto:

Sintaxis

La anatomía de un Servlet no se aleja mucho de los programas tradicionales. Solo es necesario conocer algunos elementos que se usan ella:

La Clase HttpServlet es la superclase de los Servlets de tipo HTTP (que responden usando este protocolo) y su definición es la siguiente[35]:

[35] Sacado de la API de Servlets en http://java.sun.com/products/servlet/2.2/javadoc/

Método	Descripción
HttpServlet()	Does nothing, because this is an abstract class.
protected void doDelete(HttpServletRequest req, HttpServletResponse resp)	Called by the server (via the service method) to allow a servlet to handle a DELETE request.
protected void doGet(HttpServletRequest req, HttpServletResponse resp)	Called by the server (via the service method) to allow a servlet to handle a GET request.
protected void doOptions(HttpServletRequest req, HttpServletResponse resp)	Called by the server (via the service method) to allow a servlet to handle a OPTIONS request.
protected void doPost(HttpServletRequest req, HttpServletResponse resp)	Called by the server (via the service method) to allow a servlet to handle a POST request.
protected void doPut(HttpServletRequest req, HttpServletResponse resp)	Called by the server (via the service method) to allow a servlet to handle a PUT request.
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)	Called by the server (via the service method) to allow a servlet to handle a TRACE request.
protected long getLastModified(HttpServletRequest req)	Returns the time the HttpServletRequest object was last modified, in milliseconds since midnight January 1, 1970 GMT.
protected void service(HttpServletRequest req, HttpServletResponse resp)	Receives standard HTTP requests from the public service method and dispatches them to the doXXX methods defined in this class.
void service(ServletRequest req, ServletResponse res)	Dispatches client requests to the protected service method.

Esta clase no sirve de mucho tal cuál está, ya que es una clase abstracta y debe ser extendida por el servlet que uno quiere construir. Según esta definición, podemos mirar un servlet sencillo para analizarlo:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class MiPrimerServlet extends HttpServlet {
      public void doGet(HttpServletRequest req,
                        HttpServletResponse res)
                  throws ServletException, IOException {
            res.setContentType("text/html");
            PrintWriter out = res.getWriter();
            out.println("Hola Mundo");
            out.close();
      }
}
```

Este servlet pone en la ventana del browser la siguiente salida:

Pero ¿cómo lo hace?. Bueno, veamos cómo este código queda "publicado", ya que podemos ver la parte de programación pero no la parte más "administrativa".

Primero que todo, para publicar un servlet es necesario tener un servidor de publicación también conocido como Servidor de Aplicaciones que soporte Servlets por supuesto. Uno simple y que pueden descargar en forma gratuíta es el Tomcat (http://jakarta.tomcat.org) y es bastante bueno para hacer pruebas.

El Tomcat posee una estructura de directorios bastante amplia, sin embargo lo importante se encuentra en la carpeta webapps. Dentro de esta carpeta se encuentran los "proyectos" en los cuales uno puede trabajar. Entonces, una vez instalado, podemos poner nuestro archivo .class en el directorio webapps/examples/WEB-INF/classes.

Con esto, ya estaría publicado, por lo que solo debemos llamar a la dirección:

http://localhost:8080/examples/servlet/MiPrimerServlet

¡Y listo! ¡Funciona!

Ahora analicemos nuestro servlet para que entendamos más cómo está construído.

- Es necesario usar clases de los packages javax.servlet y javax.servlet.http: Las clases que se utilizan para usar servlets están en uno paquetes de clases especiales, que son descargables de http://java.sun.com/products/servlet/. En esta dirección se encuentra tanto las clases como la documentación de la misma.

- Para construir un servlet es necesario extender de la clase HttpServlet : Esta clase permite el funcionamiento de los servlet en un servidor de aplicaciones.

- El Servlet recibe desde el Cliente información: Para ello utiliza las interface HttpServletRequest. Esta clase posee los siguientes métodos:

Método	Descripción
String getAuthType()	Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the servlet was not protected.
String getContextPath()	Returns the portion of the request URI that indicates the context of the request.
Cookie[] getCookies()	Returns an array containing all of the Cookie objects the client sent with this request.
long getDateHeader(String name)	Returns the value of the specified request header as a long value that represents a Date object.
String getHeader(String name)	Returns the value of the specified request header as a String.
Enumeration getHeaderNames()	Returns an enumeration of all the header names this request contains.
Enumeration getHeaders(String name)	Returns all the values of the specified request header as an Enumeration of String objects.
int getIntHeader(String name)	Returns the value of the specified request header as an int.
String getMethod()	Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.

Método	Descripción
String getPathInfo()	Returns any extra path information associated with the URL the client sent when it made this request.
String getPathTranslated()	Returns any extra path information after the servlet name but before the query string, and translates it to a real path.
String getQueryString()	Returns the query string that is contained in the request URL after the path.
String getRemoteUser()	Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated.
String getRequestedSessionId()	Returns the session ID specified by the client.
String getRequestURI()	Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
String getServletPath()	Returns the part of this request's URL that calls the servlet.
HttpSession getSession()	Returns the current session associated with this request, or if the request does not have a session, creates one.
HttpSession getSession(boolean create)	Returns the current HttpSession associated with this request or, if if there is no current session and create is true, returns a new session.
java.security.Principal getUserPrincipal()	Returns a java.security.Principal object containing the name of the current authenticated user.
boolean isRequestedSessionIdFromCookie()	Checks whether the requested session ID came in as a cookie.
boolean isRequestedSessionIdFromURL()	Checks whether the requested session ID came in as part of the request URL.
boolean isRequestedSessionIdValid()	Checks whether the requested session ID is still valid.
boolean isUserInRole(String role)	Returns a boolean indicating whether the authenticated user is included in the specified logical "role".

Y los métodos que hereda desde la interface ServletRequest:

Método	Descripción
Object getAttribute(String name)	Returns the value of the named attribute as an Object, or null if no attribute of the given name exists.
Enumeration getAttributeNames()	Returns an Enumeration containing the names of the attributes available to this request.
String getCharacterEncoding()	Returns the name of the character encoding used in the body of this request.
int getContentLength()	Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.
String getContentType()	Returns the MIME type of the body of the request, or null if the type is not known.
ServletInputStream getInputStream()	Retrieves the body of the request as binary data using a ServletInputStream.
Locale getLocale()	Returns the preferred Locale that the client will accept content in, based on the Accept-Language header.
Enumeration getLocales()	Returns an Enumeration of Locale objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client based on the Accept-Language header.
String getParameter(String name)	Returns the value of a request parameter as a String, or null if the parameter does not exist.
Enumeration getParameterNames()	Returns an Enumeration of String objects containing the names of the parameters contained in this request.
String[] getParameterValues(String name)	Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
String getProtocol()	Returns the name and version of the protocol the request uses in the form protocol/majorVersion.minorVersion, for example, HTTP/1.1.
BufferedReader getReader()	Retrieves the body of the request as character data using a BufferedReader.

Método	Descripción
String getRemoteAddr()	Returns the Internet Protocol (IP) address of the client that sent the request.
String getRemoteHost()	Returns the fully qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.
RequestDispatcher getRequestDispatcher(String path)	Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path.
String getScheme()	Returns the name of the scheme used to make this request, for example, http, https, or ftp.
String getServerName()	Returns the host name of the server that received the request.
int getServerPort()	Returns the port number on which this request was received.
boolean isSecure()	Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
void removeAttribute(String name)	Removes an attribute from this request.
void setAttribute(String name, Object o)	Stores an attribute in this request.

De esta forma, se pueden obtener muchas cosas del cliente y que puedan servir al Servlet. Además, si se desea obtener una parte del Header, se tienen algunas variables:

Accept:
Accept-Charset:
Accept-Encoding:
Accept-Language:
Authorization:
Host:
Referer:
Cookie:
Connection:

- El Servlet envía al Cliente información: Para ello utiliza las interface HttpServletResponse como canal de comunicación. Esta clase posee los siguientes métodos:

Método	Descripción
void addCookie(Cookie cookie)	Adds the specified cookie to the response.
void addDateHeader(String name, long date)	Adds a response header with the given name and date-value.
void addHeader(String name, String value)	Adds a response header with the given name and value.
void addIntHeader(String name, int value)	Adds a response header with the given name and integer value.
boolean containsHeader(String name)	Returns a boolean indicating whether the named response header has already been set.
String encodeRedirectURL(String url)	Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
String encodeURL(String url)	Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
void sendError(int sc)	Sends an error response to the client using the specified status.
void sendError(int sc, String msg)	Sends an error response to the client using the specified status code and descriptive message.
void sendRedirect(String location)	Sends a temporary redirect response to the client using the specified redirect location URL.
void setDateHeader(String name, long date)	Sets a response header with the given name and date-value.
void setHeader(String name, String value)	Sets a response header with the given name and value.

Método	Descripción
void setIntHeader(String name, int value)	Sets a response header with the given name and integer value.
void setStatus(int sc)	Sets the status code for this response.

Y los métodos que hereda desde la interface ServletResponse:

Método	Descripción
void flushBuffer()	Forces any content in the buffer to be written to the client.
int getBufferSize()	Returns the actual buffer size used for the response.
String getCharacterEncoding()	Returns the name of the charset used for the MIME body sent in this response.
Locale getLocale()	Returns the locale assigned to the response.
ServletOutputStream getOutputStream()	Returns a ServletOutputStream suitable for writing binary data in the response.
PrintWriter getWriter()	Returns a PrintWriter object that can send character text to the client.
boolean isCommitted()	Returns a boolean indicating if the response has been committed.
void reset()	Clears any data that exists in the buffer as well as the status code and headers.
void setBufferSize(int size)	Sets the preferred buffer size for the body of the response.
void setContentLength(int len)	Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
void setContentType(jString type)	Sets the content type of the response being sent to the client.
void setLocale(Locale loc)	Sets the locale of the response, setting the headers (including the Content-Type's charset) as appropriate.

De esta forma, se pueden enviar muchas cosas del cliente y que puedan servir. Además, si se desea enviar algo en el Header, se tienen algunas variables:

Content-Type:
Content-Length:
Content-Encoding:
Content-Language:
Connection:
Cache:
Refresh:
www-Authenticate:

- El Servlet Responde a los Métodos GET y POST: Dependiendo del método de llamada (request), el servlet lo discrimina con dos métodos doGet y doPost, los cuales reciben el canal de comunicación de lectura (Request) del cliente y de escritura (Response) al mismo.

- Es necesario decirle el tipo de salida: Al momento de preparar la salida, es necesario indicarle qué tipo de salida es. En este caso usaremos "text/html" en el resp.setContentType().

- La salida debe ser escrita en un Stream: Para poder enviar información al cliente se usa un Stream de salida, que se obtiene con el response y el método getWriter de ese objeto. La escritura es como en cualquier otro stream.

De esta forma, podemos comentar nuestro servlet "hola mundo" explicando qué hace cada línea e instrucción:

```
// Importación de los paquetes de clases de Servlets
import javax.servlet.*;
import javax.servlet.http.*;

// Importación del paquete de entrada y salida
import java.io.*;

// Clase extendiendo de HttpServlet
public class MiPrimerServlet extends HttpServlet {

      // Método de retorno de información a un método GET
      // Recibe ambos parámetros de entrada y salida
      public void doGet(HttpServletRequest req,
                        HttpServletResponse res)
                  throws ServletException, IOException {

            // Se setea el tipo de contenido de salida a HTML
            res.setContentType("text/html");

            // Se obtiene el dispositivo de salida
            PrintWriter out = res.getWriter();

            // Se escribe en el dispositivo de salida (cliente)
            out.println("Hola Mundo");

            // Se cierra el dispositivo de salida
            out.close();
      }
}
```

Concepto

Una Cookie es un objeto que se almacena en el terminal del cliente y que nos permite guardar cierta información particular.

La información de una cookie normalmente es solo texto y se almacena en un directorio en el computador cliente. Las cookies son manejadas normalmente por el browser del cliente, por lo que si el cliente decide eliminar las cookies puede hacerlo, y el servlet no tiene control directo sobre ellas.

Aún cuando las cookies entonces son un medio tan volátil, es una muy buena forma de hacerle seguimiento al cliente, de hecho cada vez que un sitio es contactado por el cliente, éste le envía automáticamente todas sus cookies.

Las cookies son simples pares de strings que se guardan. Por ejemplo, la cookie "fecha" puede almacenar "15 de Noviembre de 2003".

Sintaxis

Las cookies son simples pares de Strings. De todas formas, en Java tienen una representación clara utilizando la clase Cookie:

Método	Descripción
Cookie(String name, String value)	Constructs a cookie with a specified name and value.
Object clone()	Overrides the standard java.lang.Object.clone method to return a copy of this cookie.
String getComment()	Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
String getDomain()	Returns the domain name set for this cookie.
int getMaxAge()	Returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
String getName()	Returns the name of the cookie.
String getPath()	Returns the path on the server to which the browser returns this cookie.
boolean getSecure()	Returns true if the browser is sending cookies only over a secure protocol, or false if the browser can send cookies using any protocol.
String getValue()	Returns the value of the cookie.
int getVersion()	Returns the version of the protocol this cookie complies with.
void setComment(String purpose)	Specifies a comment that describes a cookie's purpose.
void setDomain(String pattern)	Specifies the domain within which this cookie should be presented.
void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
void setPath(String uri)	Specifies a path for the cookie to which the client should return the cookie.
void setSecure(boolean flag)	Indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL.
void setValue(String newValue)	Assigns a new value to a cookie after the cookie is created.
void setVersion(int v)	Sets the version of the cookie protocol this cookie complies with.

De esta forma, crear una Cookie es muy sencillo. La creación de ella se realiza creando un objeto cookie:

```
Cookie c = new Cookie("miPrimeraCookie", "Esta es una cookie");
```

Y luego es necesario enviarla al cliente para que se registre:

```
res.addCookie(c);  // Recuerda: res es de tipo HttpServletResponse
```

Y ahora el cliente posee una cookie llamada miPrimeraCookie.

Al obtenerla se nos hace solo un poquito más dificil, ya que las cookies vienen todas juntas. De esta forma, si queremos obtener la misma cookie desde el servidor, deberemos hacer lo siguiente:

```
Cookie[] cs = req.getCookies();  // req es un HttpServletRequest
for (int i=0; i<cs.length; i++) {
      if (cs[i].getName().equals("miPrimeraCookie")) {
            System.out.println(cs[i].getValue());
      }
}
```

Este código, entonces, retorna el contenido de la cookie "miPrimeraCookie" en la salida estándar.

Veamos ahora un par de ejemplos de cookies.

Primero que nada, veamos un servlet que solo se preocupe de mostrar todas las cookies que posee el usuario en su computador, para "espiar" lo que posee:

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class MisCookies extends HttpServlet {
      public void doGet(HttpServletRequest req,
                        HttpServletResponse res)
                  throws ServletException, IOException {
            res.setContentType("text/html");
            PrintWriter out = res.getWriter();
            Cookie[] cs = req.getCookies();
            for (int i=0; i<cs.length; i++) {
                  out.print("<LI><B>");
                  out.print(cs[i].getName());
                  out.print("</B>: ");
                  out.print(cs[i].getValue());
                  out.print("</LI>");
                  out.println();
            }
            out.close();
      }
}
```

Este servlet lo que hace es imprimir todas las cookies que envía el cliente en el reques req y las pones con el formato:

- Nombre: Valor

Por lo que logramos ver, es simple utilizarlas, pero nuevamente debemos recordar que las cookies son un espacio volátil de información que el usuario puede borrar o impedir en su configuración personalizada.

Concepto

> *La Session es un objeto que se almacena en el servidor y que guarda información que existe durante el intercambio que hay entre un cliente y un servidor.*

A diferencia de una cookie, la sesión solo dura desde el momento en que el cliente se contacta una vez con el servidor, hasta que ha pasado un tiempo de inactividad. Generalmente el tiempo de inactividad se extiende durante los siguientes 30 minutos desde que el cliente no se ha vuelto a comunicar con el servidor de ninguna forma. Una vez que ha pasado este tiempo de timeout, la sesión deja de existir en el servidor.

La sesión no es manejada ni por el cliente ni tampoco por el servlet, si no que existe SIEMPRE en el servidor web, y el servlet puede utilizarla para almacenar y obtener información de ella.

Sintaxis

Al igual que las cookies, la sesión se obtiene del request del usuario (o en realidad, se obtiene a partir de la información que trae el request). De esta forma, y usando la interfaz HttpSession podemos obtener la sesión actual del usuario con:

```
HttpSession sesion = request.getSession(true);
```

Con esta línea, creamos un objeto sesion que representa la sesión del cliente en el servidor web. Veamos más métodos que podemos utilizar con HttpSession:

Método	Descripción
Object getAttribute(String name)	Returns the object bound with the specified name in this session, or null if no object is bound under the name.
Enumeration getAttributeNames()	Returns an Enumeration of String objects containing the names of all the objects bound to this session.
long getCreationTime()	Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
String getId()	Returns a string containing the unique identifier assigned to this session.
long getLastAccessedTime()	Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
int getMaxInactiveInterval()	Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
void invalidate()	Invalidates this session and unbinds any objects bound to it.
boolean isNew()	Returns true if the client does not yet know about the session or if the client chooses not to join the session.
void removeAttribute(String name)	Removes the object bound with the specified name from this session.
void setAttribute(String name, Object value)	Binds an object to this session, using the name specified.
void setMaxInactiveInterval(int interval)	Specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

La ventaja que tiene una sesión con respecto a una cookie salta a la vista al ver los métodos de la interfaz HttpSession, pues se pueden almacenar objetos en vez de solo strings. De esta forma podemos pasar información durante la conexión que nos puede servir de muchas formas.

Veamos un ejemplo simple que obtiene información directamente de la Session que el usuario crea y que se mantiene durante la conexión:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionExample extends HttpServlet {
```

```
      public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
                  throws IOException, ServletException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            HttpSession session = request.getSession(true);

            // print session info

            Date created = new Date(session.getCreationTime());
            Date accessed =
                  new Date(session.getLastAccessedTime());
            out.println("ID " + session.getId());
            out.println("Created: " + created);
            out.println("Last Accessed: " + accessed);

            // set session info if needed

            String dataName = request.getParameter("dataName");
            if (dataName != null && dataName.length() > 0) {
                  String dataValue =
                        request.getParameter("dataValue");
                  session.setAttribute(dataName, dataValue);
            }

            // print session contents

            Enumeration e = session.getAttributeNames();
            while (e.hasMoreElements()) {
                  String name = (String)e.nextElement();
                  String value =
                        session.getAttribute(name).toString();
                  out.println(name + " = " + value);
            }
      }
}
```

Una salida válida de este servlet sería:

```
ID 7D09C750A21350C64A6D388D44048A90
Created: Sun Nov 16 05:22:16 GMT 2003
Last Accessed: Sun Nov 16 05:22:16 GMT 2003
```

¿Cuándo usar Cookies o Sessions?

La respuesta es compleja, pero simple a la vez, ya que se pueden utilizar ambas a gusto, pero siempre teniendo en cuenta las siguientes consideraciones generales:

- Una Session se crea en el servidor. Una cookie se crea en el caché del usuario.
- Una Session existe mientras se mantena una conexión del usuario. Una Cookie existe mientras el usuario no la borre de su caché.
- Una Session se mantiene local al servlet en ejecución. Una cookie viaja a través de la red cada vez que es contactado el cliente.
- Una Session puede guardar un Object. Una cookie solo guarda String.
- Una Session siempre se crea. Una Cookie puede ser bloqueada para impedir su creación.

JavaServer Pages

Motivación

Hasta ahora, vimos que sobre la internet podemos generar distintas formas de comunicación. Los canales más comunes son los punto a punto (servidores y clientes escritos en Java) y utilizando la web (programas que responden a través de http).

En el segundo caso, conocimos que los Servlets son programas compilados y listos para ejecutar que responden a llamadas HTTP directamente, pero está basado en lenguaje Java con la salida en HTML.

De ahí es donde se contrapone con lenguajes más fáciles de utilizar en la web, como ASP y PHP, en donde el cuerpo de los programas se basa en el lenguaje HTML pero con capacidades de procesamiento con lenguajes de más alto nivel (ASP -> Visual Basic).

Sin embargo, SUN Microsystems no se quedó atrás entregando una alternativa a su solución de Servlets, con capacidades similares a lo conocido como ASP de Microsoft.

Conceptos

> *JavaServer Pages (JSP) son páginas dinámicas escritas en código Java para ser interpretado en el servidor y código HTML para ser interpretado en el cliente.*

En efecto, bajo este concepto se pueden utilizar estas páginas de una manera sencilla. Normalmente los servidores de aplicación (Tomcat) puede interpretar las páginas JSP sin problemas, por lo que no se necesita nada adicional.

Pero ¿qué diferencia un Servlet de una JSP?

A nivel de funcionalidad podrían hacer exactamente lo mismo, ya que ambos son llamados a través de una URL al servidor de aplicación, ejecutados en el servidor y la salida de ellos va a dar directamente al browser del cliente en formato HTML (que es lo que entiende). Para el cliente es absolutamente transparente. Pero en el proceso son bastante distintos:

- Los Servlets son construídos como clases escritas en Java, las JSP son construídas como páginas HTML: La diferencia sutil es el punto de vista de programación. Para construir el Servlet, se debe pensar en algún editor Java para ello (punto de vista del programador), en cambio la JSP se piensa directamente en la salida del cliente (punto de vista del diseño gráfico).

- Los Servlets se ejecutan después de compilados, las JSP se compilan en runtime: Esto significa que la JSP en realidad solo se construye (el código fuente) y se instala directamente en el servidor de aplicación "sin compilar". Al momento de ser llamada la

JSP, el servidor la compila momentáneamente y se ejecuta. Por otro lado el Servlet funciona igual que cualquier programa Java.

- Los Servlets usan URL's especiales, las JSP usan URL's más estándares: Por usar llamadas a un programa, normalmente las URL que usa un Servlet terminan en un nombre de clase sin extensión ni nada, algo que tiende a confundir a algunos usuarios al verlas en su browser. Por otro lado, las JSP poseen direcciones más "normales" ya que llaman físicamente a un archivo de extensión .jsp de la misma manera que se llaman los .htm (archivos HTML estándares).

- Los Servlets poseen una estructura fija, las JSP no poseen métodos, variables de instancia ni imports varios: En pocas palabras, la estructura rígida de un Servlet se convierte en una estructura más lineas cuando se trabaja en JSP.

Pero, ¿cómo se ejecuta en realidad la JSP siendo que no es un "programa compilado". Simplemente, cuando el servidor de aplicación es llamado para ejecutar la JSP, lo que hace es compilarla en un servlet (generado) y luego ejecutar, lo que es Java, dentro del Servlet y luego unirlo con lo que tiene HTML para enviarlo al browser cliente.

Con estas características poco técnicas podemos darnos cuenta que los Servlets y las JSP son distintas. Veamos ahora un poco de anatomía de JSP para que nos quede más claro aún.

Sintaxis

Básicamente, la JSP no posee una estructura flexible y que depende mucho de donde uno quiere insertar código. De esta forma, la sintaxis de una JSP se reduce a:

```
...
Código HTML
<% Código Java %>
Código HTML
...
```

En donde lo importante está que, el código que está dentro de "<% ... %>" es un código JSP. En particular:

Sintaxis	Significado
<%= expresión %>	JSP Expressions: Expresión evaluada lista para mostrar en la página
<% código %>	JSP Scriptles: Líneas de código que se ejecutan en la JSP
<%! código %>	JSP Declarations: Código que va inserto fuera de la declaración del método SERVICE
<%@ page att = "val" %>	JSP Page Directives: Directivas (en negritas el valor por defecto): ü import="package.class" ü content-Type="MIME-Type"

	ü isThreadSafe="true|false" ü session="true|false" ü buffer="sizekb|none" ü autoflush="true|false" ü extends="package.class" ü info="message" ü errorPage="url" ü isErrorPage="true|false" ü language="java"
<%@ include file = "url" %>	JSP Include Directive: Un archivo local al servidor que será utilizado cuando el servlets sea generado a partir de la JSP.
<!-- comentario -->	Comentarios dentro de la JSP.

Veamos un ejemplo de JSP. Considerando exactamente el mismo ejemplo usado para los Servlets del "Hola Mundo", el código era:

```
// Importación de los paquetes de clases de Servlets
import javax.servlet.*;
import javax.servlet.http.*;

// Importación del paquete de entrada y salida
import java.io.*;

// Clase extendiendo de HttpServlet
public class MiPrimerServlet extends HttpServlet {

      // Método de retorno de información a un método GET
      // Recibe ambos parámetros de entrada y salida
      public void doGet(HttpServletRequest req,
                        HttpServletResponse res)
                  throws ServletException, IOException {

            // Se setea el tipo de contenido de salida a HTML
            res.setContentType("text/html");

            // Se obtiene el dispositivo de salida
            PrintWriter out = res.getWriter();

            // Se escribe en el dispositivo de salida (cliente)
            out.println("<HTML>");
            out.println("<HEAD>");
            out.println("       <TITLE>Mi Primer Servlet</TITLE>");
            out.println("</HEAD>");
            out.println("<BODY>");
            out.println("       Hola Mundo");
            out.println("</BODY>");
            out.println("</HTML>");

            // Se cierra el dispositivo de salida
            out.close();
      }
}
```

Ahora, si queremos verlo del punto de vista de una JSP quedaría:

```
<HTML>
	<HEAD>
		<TITLE>Mi Primer JSP</TITLE>
	</HEAD>
	<BODY>
		<%= "Hola Mundo" %>
	</BODY>
</HTML>
```

¡Bastante fome!. De hecho es lo mismo que hacer una página en HTML:

```
<HTML>
	<HEAD>
		<TITLE>Esto no es JSP</TITLE>
	</HEAD>
	<BODY>
		Hola Mundo
	</BODY>
</HTML>
```

Ambos códigos muestran lo mismo como resultado "Hola Mundo". A pesar de todo, lo único posible utilizar fue una expresión de un String. Compliquemos el JSP para meterles conceptos de scriptlet:

```
<!-- Aquí comienza el código HTML -->
<HTML>
	<HEAD>
		<TITLE>Mi Primer JSP</TITLE>
<!-- Aquí comienza el JSP Scriptlet -->
<%
	String mensaje = "Hola Mundo";
%>
<!-- Aquí termina el JSP Scriptlet -->
	</HEAD>
	<BODY>
<!-- Aquí comienza la JSP Expression -->
		<%= mensaje %>
<!-- Aquí termina la JSP Expression -->
	</BODY>
</HTML>
<!-- Aquí termina el código HTML -->
```

Y el resultado de esta JSP es exactamente EL MISMO. Por último, mutamos lo mismo por:

```
<!-- Aquí comienza el código HTML -->
<HTML>
	<HEAD>
		<TITLE>Mi Primer JSP</TITLE>
	</HEAD>
	<BODY>
<!-- Aquí comienza el JSP Scriptlet -->
<%
	String mensaje = "Hola Mundo";
	out.println(mensaje);
%>
<!-- Aquí termina el JSP Scriptlet -->
	</BODY>
```



```
</HTML>
<!-- Aquí termina el código HTML -->
```

Otra vez lo mismo. Los distintos sabores de implementación de los scriptlets le dan una alta versatilidad a las JSP con respecto a otros componentes como los Servlets, ya que el código es más simple que en su pariente. Si miramos solo el código JAVA escrito en el Servlet y lo comparamos con el JSP nos queda:

Servlet:

```
 1  import javax.servlet.*;
 2  import javax.servlet.http.*;
 3  import java.io.*;
 4  public class MiPrimerServlet extends HttpServlet {
 5          public void doGet(HttpServletRequest req,
 6                            HttpServletResponse res)
 7                        throws ServletException, IOException {
 8                  res.setContentType("text/html");
 9                  PrintWriter out = res.getWriter();
                    out.println("<HTML>");
                    out.println("  <HEAD>");
                    out.println("          <TITLE>Mi Primer Servlet</TITLE>");
                    out.println("  </HEAD>");
                    out.println("  <BODY>");
10                  out.println("          Hola Mundo");
                    out.println("  </BODY>");
                    out.println("</HTML>");
11                  out.close();
            }
    }
```

JSP:

```
    <HTML>
            <HEAD>
                    <TITLE>Mi Primer JSP</TITLE>
            </HEAD>
            <BODY>
    <%
 1          String mensaje = "Hola Mundo";
 2          out.println(mensaje);
    %>
            </BODY>
    </HTML>
```

O sea, de 11 líneas Java puras que se utilizaron en el Servlet, se utilizan solo 2 en la JSP. ¡Interesante!.

Pero ¿qué gracia tiene la JSP, aparte de escribir menos, que la hace una buena solución para escribir "páginas dinámicas"?

En realidad tiene una característica que permite al programador acceder a los mismos servicios que poseen los Servelts:

- Utiliza la variable request para obtener los parámetros que vienen en la llamada del cliente. Al igual que el servlet utilizada un variable de tipo HttpServletRequest, aquí acceden directamente a la variable "request".

- Utiliza la variable response para obtener los parámetros que permiten comunicación con el cliente. Al igual que el servlet utilizada un variable de tipo HttpServletResponse, aquí acceden directamente a la variable "response"

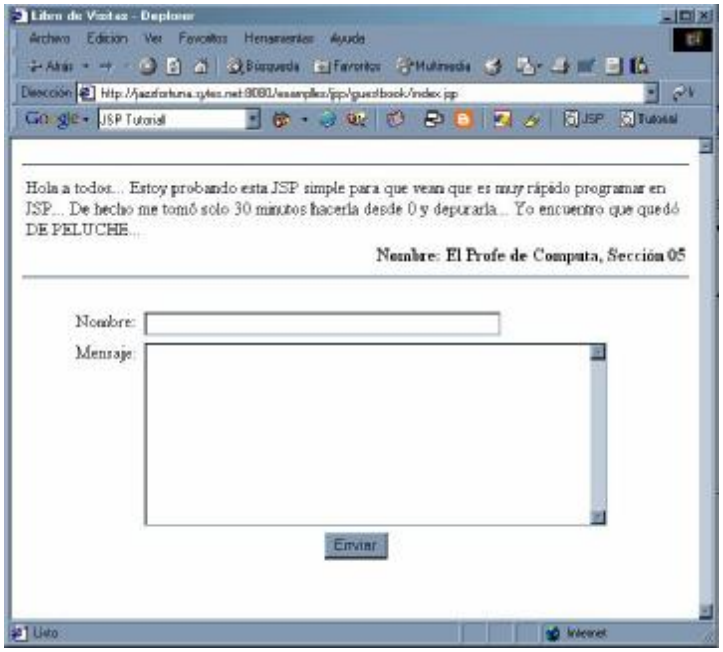
- Utiliza la variable out como salida estándar de mensajes hacia el cliente. En vez de tener que obtener el PrintWriter desde el HttpServletResponse, acá se usa la variable "out" directamente.

- Utiliza la variable session como si fuese un objeto HTTPSession. Al igual que el Servlet utiliza HTTPSession para acceder a la sesión del usuario, directamente trabajar con "session" es más natural.

Esto le da características similares a los servlets, por lo que tendrían tanta ventaja como ellos.

Veamos como se implementa en JSP un libro de visitas simple.

¿Qué es un libro de visitas?. No es nada más que una página que permite publicar mensajes para todos en forma pública y abierta. Básicamente lo que debería mostrar es esto:



Las características de este libro de visitas son:

- El libro se va guardando en un RAF en donde se guardan los 2 Strings: Nombre y Mensaje.

- Al leerlo, como al guardarlo, no necesita saber el largo del UTF, ya que no vamos a realizar saltos de registro (se lee en forma lineal).
- En este caso, suponga que el archivo queda en el directorio por defecto del TOMCAT (solo debe indicar el nombre del archivo).

La JSP que debe escribirse quedaría de la siguiente forma:

```
<HTML>
	<HEAD>
		<TITLE>Libro de Visitas</TITLE>
	</HEAD>
	<BODY>
		<!-- Importación de Packages -->
		<%@ page import="java.io.*" %>

		<!-- Ingreso de un mensaje, si es que aplica -->
		<%
			if
("NEW".equals(request.getParameter("action"))) {
				try {
					RandomAccessFile book = new
RandomAccessFile("book.raf", "rw");
					book.seek(book.length());

	book.writeUTF(request.getParameter("message"));

	book.writeUTF(request.getParameter("name"));
					book.close();
					out.println("<script
language=Javascript>");
					out.println("document.location =
'index.jsp';");
					out.println("</script>");
				}
				catch (Exception e) {
					out.println("ERROR: " +
e.getMessage());
				}
			}
		%>

		<!-- Lectura de los mensajes anteriores -->
		<HR>
		<%
			try {
				RandomAccessFile book = new
RandomAccessFile("book.raf", "rw");
				while(book.getFilePointer() <
book.length()) {
					out.println("<TABLE WIDTH=100%
BORDER=0>");
					out.println("<TR>");
					out.println("<TD>");
					out.println(book.readUTF());
					out.println("</TD>");
					out.println("</TR>");
					out.println("<TR>");
					out.println("<TD ALIGN=RIGHT>");
					out.println("<B>Nombre: " +
book.readUTF() + "</B>");
					out.println("</TD>");
```

```
                                    out.println("</TR>");
                                    out.println("</TABLE>");
                                    out.println("<HR>");
                              }
                              book.close();
                        }
                        catch (Exception e) {
                        }
            %>

            <!-- Formulario de Ingreso -->
            <FORM METHOD="GET">
                  <INPUT TYPE="HIDDEN" NAME="action" VALUE="NEW">
                  <TABLE WIDTH="85%" ALIGN="CENTER">
                        <TR VALIGN="TOP">
                              <TD>Nombre:</TD>
                              <TD><INPUT TYPE="TEXT" NAME="name"
SIZE="50"></TD>
                        </TR>
                        <TR VALIGN="TOP">
                              <TD>Mensaje:</TD>
                              <TD><TEXTAREA NAME="message"
COLS="50" ROWS="10"></TEXTAREA></TD>
                        </TR>
                        <TR VALIGN="TOP">
                              <TD COLSPAN="2"
ALIGN="CENTER"><INPUT TYPE="SUBMIT" VALUE="Enviar"></TD>
                        </TR>
                  </TABLE>
            </FORM>

      </BODY>
</HTML>
```

En este caso, no es necesario saber HTML para entender el código, solo lo básico. Para ejercicio suyo, haga la prueba ahora abriendo este archivo con un programa como Dreamweaver y cambie la gráfica con ese programa tratando de no afectar el código JSP (está bien delimitado). ¿Puede mejorarlo sin afectar el programa?. Algo que no se puede hacer con el Servlet.

Sin embargo eso no es todo. Existen más comandos de JSP que pueden servir cuando se trata de desarrollos profesionales. Allí encontraremos cosas como Java Beans, inclusión de contenidos, redireccionamienteo y plugins. Pero eso da para mucho más que un contenido de un curso básico de Java.