Indice de la Unidad IV

Autor . . . . Ing. Tymoschuk, Jorge
Colaboradores . (Introducción) Ing. Guzmán, Analía
(JSP) Al. Celia, Salvador

INTRODUCCION A LA PROGRAMACION DISTRIBUIDA

Los sistemas distribuidos surgen para dar solución a las siguientes necesidades:

- Repartir el volumen de información.
- Compartir recursos, ya sea en forma de software o hardware.

La construcción de sistemas distribuidos presenta una solución que aumenta nuestras capacidades, ya no estamos sujetos a las restricciones de la máquina, ahora somos capaces de utilizar los recursos de toda una red.
Los dos tipos principales de sistemas distribuidos son:

- **sistemas computacionales distribuidos:** un conjunto de ordenadores conectados por una red son usados colectivamente para realizar tareas distribuidas.
- **sistemas de procesamiento paralelo.** Por otro lado en los sistemas paralelos, la solución a un problema importante es dividida en pequeñas tareas que son repartidas y ejecutadas para conseguir un alto rendimiento.

Los sistemas distribuidos se pueden implementar usando dos modelos:

- **modelo de cliente-servidor:** contiene un conjunto de procesos clientes y un conjunto de procesos servidor, se precisan además de unos recursos (software), todos los estos recursos son manejados por los procesos servidor. Cliente y Servidor deben hablar el mismo lenguaje para conseguir una comunicación efectiva, el primero solicita al segundo unos recursos y este último los concede, le hace esperar o lo deniega según los permisos que tenga.
- **modelo basado en objetos:** Tenemos una serie de objetos que solicitan servicios (clientes) a los proveedores de los servicios (servidores) a través de una interfaz de encapsulación definida. Un cliente envía un mensaje a un objeto (servidor) y éste decide qué ejectutar, RMI y CORBA son algunos de esos sistemas basados en objetos. *(Este es el que veremos en detalle en esta unidad)*

La programación distribuida no es fácil:

- Hay múltiples modelos, dependiendo de los sistemas a los que accedemos.
- Hay muchos aspectos de seguridad.
- Se mezclan diversas tecnologías.
- Dificultar para probar y depurar.

Llegados a este punto ya podemos darnos cuenta de una de las principales dificultades de la programación distribuida ¿cómo se comunican cliente y servidor?, ¿a través de que lenguaje, entorno o herramienta?, es aquí cuando debemos hacer referencia a Java:

- Java es una herramienta ideal para programación distribuida
  - debido a su portabilidad,
  - seguridad y
  - amplio abanico de componentes.
- Desarrollando en Java podemos olvidarnos:
  - dónde vamos a ejecutar nuestro programa,
  - ¿será en una máquina UNIX o
  - en un servidor Windows 2000 Server?,
  - ¿lo podemos preparar para que se ejecute en varios entornos diferentes? Con Java todo esto es posible.

Finalicemos esta sección dando una definición muy básica:

Un sistema distribuido consiste en una colección de ordenadores autónomos unidos por una red y con un sistema que les permite compartir recursos de hardware, software y datos.

La cuestión básica en la programación de un sistema distribuido es cómo comunicarse con otras máquinas a través de la red. Resumiremos aquí diversas formas de atacar la programación de aplicaciones a través de la red.

SOCKETS

Los sockets son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos recibiendo y transmitiendo a través de sockets, el tipo de socket describe la forma en la que se transfiere la información.

Sockets proveen al usuario de un interfaz para comunicarse a través de la red con otro sistema. Cada dirección de identificación para sockets consiste en una dirección de Internet y en un puerto. Se suelen utilizar los conocidos protocolos TCP/IP o UDP/IP, preferentemente el primero.

Distinguimos los siguientes tipos de sockets:

- Sockets Stream: Son un servicio orientado a la conexión, donde los datos se transfieren sin encuadrarlos en registros o bloques.
- Sockets Datagram: Son un servicio de transporte sin conexión, son más eficientes que TCP pero su utilización no es del todo fiable. Los datos se envían y reciben en paquetes cuya entrega no está garantizada.
- Sockets Raw: son sockets que dan acceso a protocolos de más bajo nivel, no están soportados por Java.
- Socket Multicast es un DatagramSocket con una serie de capacidades adicionales para enlazar con grupos y hosts en Internet

Una de las ventajas principales de la programación con sockets es su sencillez.

A la hora de desarrollar una aplicación con sockets debemos tener en cuenta lo siguiente:

Es necesario tratar de manera especial los sockets para evitar problemas de seguridad, lo hacemos a través de la clase de Java SecurityManager y del paquete java security, podemos a través de estos recursos incluso generar mensajes codificados a través de algoritmos DSA.

- Generalmente conectamos con servidores PROXY.
- A la hora de conectar con BBDD, JDBC debe ser una opción a tener en cuenta, JDBC es un modelo de conexión a BBDD diseñado por JAVA, su utilidad radica en que es útil para acceso a bases de datos de diferentes tipos y vendedores, SQL Server, Oracle, etc.

Dentro de las diversas técnicas de programación destaca la serialización. La serialización de objetos es una técnica mediante la cual un programa puede salvar el estado de los objetos a un fichero y después recuperar los datos a la memoria y enviarlos a través de la red usando sockets de bajo nivel. *(Serialización lo hemos visto en Almacenamiento de Datos, AED)* Esta es una opción muy interesante que

haría posible evitar lo que hoy en día se hace continuamente en proyectos comerciales, el recurrir a una base de datos para guardar el estado de un objeto, lo que repercute en el rendimiento final de la aplicación y sobre todo en su arquitectura. Java lo implementa con la interfaz java.io.Serializable.

RMI (Remote Method Invocation)

RMI fue el primer framework con la idea de crear sistemas distribuidos que apareció para Java.

Además, viene integrado en cualquier máquina virtual Java posterior a la versión 1.0 y está pensado para hacer fácil la creación de sistemas distribuidos a partir de una aplicación cuyas clases ya están implementadas. RMI es una forma de RPC (Remote Procedure Call).

La invocación de métodos remotos permite que un objeto que se ejecuta en una máquina puede invocar métodos de un objeto que se encuentra en ejecución bajo el control de otra máquina (por supuesto no hay problemas para las relaciones entre los objetos cuando ambos son locales). En definitiva, RMI permite crear e instanciar objetos en máquinas locales y al mismo tiempo crearlos en otras máquinas (máquinas remotas), de forma que la comunicación se produce como si todo estuviese en local. RMI se convierte así en una alternativa muy viable a los sockets de bajo nivel con una serie de particularidades destacables:

- RMI permite abstraer las interfaces de comunicación a llamadas locales, no necesitamos fijarnos en el protocolo y las aplicaciones distribuidas son de fácil desarrollo.
- RMI te permite trabajar olvidándote del protocolo.
- RMI es flexible y extensible, destaca su recolector de basura.

Arquitectura de RMI:

- stub/skeleton: es responsable de transferir datos a Remote Reference Layer, permite a los objetos java ser transmitidos a diferentes espacios de direcciones.
- Remote Reference Layer es responsable de crear independencia de los protocolos.
- Transpor Layer es responsable de establecer las conexiones a los espacios remotos de direcciones.

En RMI toda la información sobre los servicios del servidor se dan en una definición de interface remota.

Si comparamos RMI con sockets percibiremos que RMI es más simple.

Además, esta tecnología permite despreocuparnos del protocolo, en sockets depende de la aplicación, si ésta es grande puede que nos tengamos que preocupar del protocolo.

En el material del apunte no está contemplado RMI. La idea es que debemos empezar por Sockets, futuramente se verá

CORBA

Hasta ahora hemos hablado de técnicas de programación que adquieren implementación a través de unas clases o interfaces definidas, con CORBA (desarrollado por OMG un consorcio de 700 compañías) el concepto cambia totalmente, CORBA (Common Object Request Broker Architecture) ES UNA ESPECIFICACIÓN PARA CREAR Y USAR OBJETOS DISTRIBUIDOS NO UN LENGUAJE DE

PROGRAMACIÓN. Desarrollar aplicaciones distribuidas con CORBA es similar a hacerlo con RMI porque una interfaz debe ser definida primero, pero las interfaces de RMI son definidas en JAVA mientras que las de CORBA emplean IDL.

- Los objetos CORBA son diferentes porque:
- Pueden ejecutarse en cualquier plataforma.
- Se pueden situar en cualquier punto de la red (ventaja)
- El uso de CORBA es complejo

Otra de las características y al mismo tiempo diferencias con RMI es que CORBA fue desarrollado con un lenguaje independiente donde los objetos de mueven en un sistema heterogéneo, RMI fue desarrollado para un lenguaje simple donde los objetos se desenvuelven en un entorno con un lenguaje homogéneo, además CORBA soporta parámetros de entrada y salida pero RMI no, lo cual son una ventajas claras de CORBA sobre RMI sin embargo debemos tener en cuenta que los objetos CORBA no tienen recolector de basura y es nuestra obligación preocuparnos de ello, al contrario que con RMI.

Ventajas de trabajar con CORBA:

- Disponibilidad y Versatilidad
- Eficiencia
- Adaptación a Lenguajes de programación

Desventajas de trabajar con CORBA:

- Complejidad
- Necesidad de un compilador traductor de tipos de datos estándares a los tipos del lenguaje en el que se vaya a programar (IDL)
- Hay que ser conscientes a la hora de diseñar qué objetos van a ser remotos y cuáles no (los remotos sufren restricciones en cuanto a sus capacidades con respecto a un objeto normal)
- Es necesario emplear tipos de datos que no son los que proporciona de manera habitual el lenguaje de programación (muchas veces hay que emplear tipos de datos adaptados de IDL).
- Incompatibilidad entre implementaciones

Antes de seguir describiendo Corba, transcribo aquí textualmente un comentario un tanto irónico de un par de autores de un importante libro de Java: JAVA 2 – Características Avanzadas (Cay Horstmann/Gary Cornell): " … tras pegarnos con algunos problemas de CORBA en la escritura de este libro … nuestro sentimiento hacia CORBA es similar al expresado por el presidente Charles de Gaulle sobre Brasil, " ***tiene un gran futuro*** *…* ***y siempre lo tendrá***"(El autor del apunte no está de acuerdo en lo que a Brasil se refiere)

AGENTES MOVILES

Los agentes móviles son entidades que circulan libremente por la red esperando llamadas del cliente. Representan un nuevo paradigma en la programación distribuida.
Estos agentes móviles presentan una serie de ventajas principales:

- Eficiencia: consumen menos recursos de red
- Utilizan menos ancho de banda
- Son robustos y tolerantes a fallos.
- Soportan entornos heterogéneos.
- Buenos para aplicaciones de comercio electrónico.
- Fácil desarrollo.

Las cuestiones de seguridad son especialmente importantes en los agentes móviles, se pretende cubrir dos objetivos:

- Proteger a los host de agentes destructivos.
- Proteger a los agentes de host maliciosos.

La solución estándar a estas cuestiones de seguridad consiste en utilizar la autentificación y la firma digital.
Voyager es una plataforma de computación 100% distribuida en Java que es útil para crear sistemas distribuidos de alto impacto rápidamente.
El uso de Voyager lleva consigo una serie de beneficios como pueden ser:

- Productividad
- Compatibilidad
- Eficiencia
- Flexibilidad

Además presenta una serie de innovaciones:

- Remote enabling a class
- Control de excepciones
- Basura distribuida
- Agregación dinámica
- CORBA
- Movilidad
- Agentes móviles autónomos
- Activación
- Seguridad

Voyager viene sobradamente preparado para integrarse con CORBA, generando las clases IDL de la aplicación.

SOAP

SOAP (*Simple Object Access Protocol*) es un protocolo basado en XML que permite comunicar componentes y aplicaciones mediante el conocido HTTP. Sería similar a hacer RPC (llamada a procedimientos remotos) mediante HTTP y XML. De hecho es la evolución del protocolo XML-RPC que permite hacer llamadas a procedimientos remotos usando XML como lenguaje común y HTTP como protocolo de transporte.

Internet fue diseñada para que múltiple sistemas se pudiesen comunicar entre si. Lo ideal es que una aplicación pueda usar componentes de otras aplicaciones, e incluso desarrollados en otros lenguajes. Eso se consigue actualmente mediante CORBA y DCOM. No obstante un desarrollo con estos estándares no es fácil de lograr y, sobre todo, la comunicación entre módulo situados remotamente da problemas. El más típico es que haya un "firewall" por el medio. Entonces la cosa se complica bastante. Como SOAP trabaja sobre HTTP, este problema se minimiza ya que HTTP es un protocolo que se maneja muy bien con los "cortafuegos".

Si queremos que la comunicación SOAP sea segura no tenemos más que usar el protocolo HTTPS en lugar de HTTP.

Propiedades de SOAP:

- Es un protocolo ligero
- Es simple y extensible
- Se usa para comunicación entre aplicaciones
- Está diseñado para comunicarse vía HTTP
- No está ligado a ninguna tecnología de componentes
- No está ligado a ningún lenguaje de programación
- Está basado en XML

- Está coordinado por el W3C
- Se convertirá en un estándar del W3C

SOAP es la pieza clave de la tecnología .NET de Microsoft pero, al ser un protocolo abierto, nos permite usar todo su potencial desde cualquier lenguaje y plataforma.

JINI

JINI construye redes que pueden adaptarse a las demandas de sistemas dinámicos, requiere una arquitectura innovadora que se acomode con eficiencia a un sistema complejo y a los cambios. El ideal de JINI es que se adapte de forma automática a las modificaciones que se hagan en la red. Sin embargo, a pesar de esta complejidad aparente JINI también ha sido pensado para ser fácil de usar y aprender. Además, Jini es una ventaja para el programador ya que el código necesario se simplifica de forma destacable.
Jini puede convivir perfectamente con todas las tecnologías y servicios típicos de una red.
Todas las representaciones en Jini se hacen con objetos java, a esos objetos se accede a través de interfaces en Java (como se puede comprobar el acceso a través de una interfaz es típico en la programación distribuida).

Gracias a esta tecnología podemos:
- Buscar servicios "multicast".
- Buscar servicios para autentificar código.
- Buscar servicios que mandan alertas.

Para encontrar un servicio, un cliente Jini lo localiza por tipo en "Lookup Service" (Interfaz de Java), después mueve el servicio al cliente y el código que el servicio necesita usar es dinámicamente cargado en demanda.
Debemos saber que a la hora de requerir un servicio, la comunicación se establece entre el servicio y su proxy, esto actúa de forma independiente al cortafuegos, el proceso es independiente del protocolo, es decir, en el caso de que el protocolo cambiase no afectaría al cliente.
Jini utiliza RMI, varias son las razones de esto:
- RMI puede pasar de forma completa los objetos (código incluido) programados en Java.
- Se trabaja mejor con la Java Virtual Machine.
- Obtenemos serialización en el proceso.
- Hay robustez y las configuraciones de seguridad son diversas.

En Jini tenemos presente el concepto de JavaSpaces. El JavaSpace es un servicio Jini que se almacena como un objeto persistente y compartido. Es necesaria su consideración a la hora de crear sistemas distribuidos, programación paralela y sistemas cooperativos.
Su uso tiene múltiples ventajas:
- Anominato entre aplicaciones.
- Comunicación desparejada.
- Los programas se pueden comunicar a través del tiempo o del espacio.
- Se mejora en diseño y tiempo de desarrollo.

Jini permite también la integración con CORBA.

Nota final: los contenidos de la unidad IV son el resultado del consenso de nuestras reuniones de Cátedra. Varios de los docentes son profesionales en diseño e implementación de sistemas, ha tiempo que trabajan en Java y estos contenidos son el mejor resultado que en este entorno tan velozmente cambiante hemos podido lograr.

Introducción al paradigma

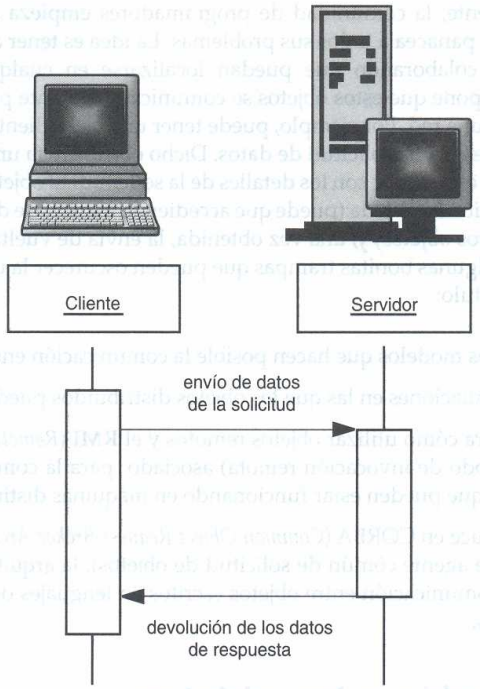
La idea de tener un equipo de objetos colaborando en la solución de problemas es ya conocida por nosotros (Apunte AED, Unidad II – Modularidad). Se supone que estos objetos están relacionados, se comunican y realizan la tarea en forma cooperativa y coordinada.

Este concepto es perfectamente extensible a un ambiente de red de ordenadores. Las comunicaciones deben realizarse usando protocolos estándar. Por ejemplo, podemos tener un objeto cliente que el usuario utilice para rellenar una solicitud de datos. Dicho objeto envía un mensaje al objeto que se encuentra en el servidor con los detalles de la solicitud. El objeto servidor recopila la información solicitada (puede que accediendo a una base de datos o comunicando con otros objetos) y, una vez obtenida, la envía de vuelta al cliente.

En esta unidad necesariamente comenzamos presentando el ambiente de la "red de redes", INTERNET. Definiciones, servicios, y poco a poco entramos en el detalle.

Introducción a los objetos remotos: los papeles de cliente y servidor

Suponemos que el usuario de la computadora local rellenará un formulario de solicitud de información. Dichos datos se envían después a un servidor, el cual será el encargado de procesar la solicitud y devolver, de forma ordenada, la información al usuario.



En un modelo tradicional de cliente/servidor, la solicitud es convertida a un formato intermedio (como parejas nombre/valor o datos XML). El servidor determina el formato de la solicitud, procesa la respuesta y la formatea para transmitida al cliente. El cliente entonces recupera esa respuesta y se la muestra al usuario.

Pero si sus datos están estructurados, existe un importante problema a la hora de codificar: será necesario convertir los datos entrantes, además de los salientes, al formato de transmisión adecuado. Y allí deberemos conocer del protocolo base de Internet, el TCP/IP y los servicios que sobre él se pueden implementar.

Un método muy usual para el envío de datos a un servidor es con formularios HTML. En este caso, el cliente es un navegador, por lo que el servidor deberá recopilar la información solicitada y formatearla en HTML. Incluso en este modelo, se separa el procesamiento de la recopilación de datos. El procesamiento ocurre en el servidor de aplicaciones, mientras que la captura de datos es en cliente (o servidor web)

El nivel de interacción entre cliente y servidor depende de cómo se aborde el problema. Suscintamente:

- Cliente y servidor están bien separados. Por ejemplo, el cliente es un formulario HTML y el servidor es un programa java.
- Cliente y servidor son programas Java. Sun desarrolló específicamente un sencillo mecanismo, llamado RMI (Remote Method Invocation, Método de invocación remota), para la comunicación entre aplicaciones Java.

- El cliente es un programa Java, el objeto servidor no lo es. Se requiere algún sistema para que los objetos puedan hablar entre sí sin importar el lenguaje de programación en el que estén escritos. El estándar CORBA (Common Object Request Broker Architecture, Arquitectura de agente común de solicitud de objetos) del OMG (Object Management Group, Grupo de administración de objetos; www.omg.org) define un mecanismo común para el intercambio de datos y el descubrimiento de servicios.

CORBA puede resultar interesante en un futuro, RMI es más sencillo de comprender y más cómodo de usar. Por supuesto, RMI sólo es útil para la comunicación entre objetos Java. En esta unidad trataremos conceptos y código relativos al primer nivel y haremos una introducción al segundo.

Introducción a INTERNET.

Internet es un conjunto de redes informáticas distribuidas por todo el mundo que intercambian información entre sí mediante la familia de protocolos TCP/IP. Puede imaginarse Internet como una gran nube con ordenadores conectados entre si, tomando/prestando servicios. Por ello se habla de clientes y servidores.

Internet surgió de un programa de investigación realizado por la Agencia de Proyectos de Investigación Avanzados de Defensa (DARPA) de los Estados Unidos sobre la conexión de redes informáticas. El resultado fue ARPANet (1969). Esta red crece y a principios de los ochenta se conecta con CSNet y MILNet, dos redes independientes, lo que se considera como el nacimiento de Internet (International Network of computers). También forman parte de esta red, NSI (NASA Science Internet) y NSFNet (National Science Foundation Net).

Durante muchos años Internet ha servido para que muchos departamentos de investigación de distintas universidades distribuidas por todo el mundo pudieran colaborar e intercambiar información. Muy recientemente ha comenzado a formar parte de los negocios y de nuestra vida cotidiana. Esto ha ocasionado el replanteo de los sistemas de comunicación internos y externos; en la mayoría de los casos se han resuelto vía Internet. Y la forma de organizar su conectividad ha generado intranets y extranets.

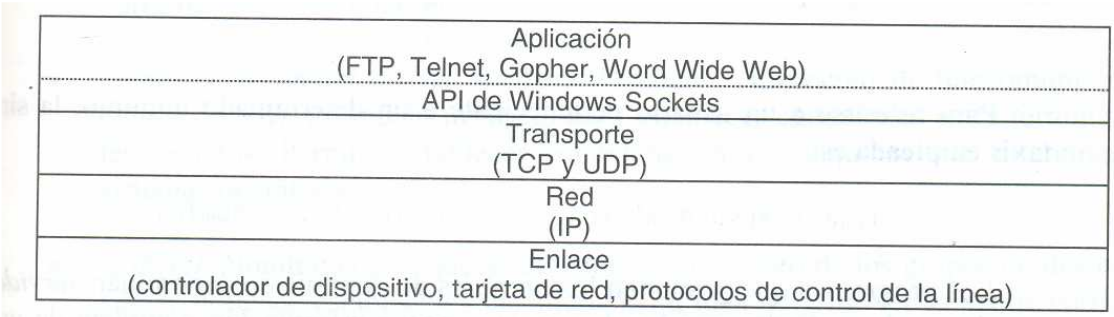
Una intranet no es más que una red local que utiliza los mismos protocolos que Internet, independientemente de que esté o no conectada a Internet. ¿Qué ventajas tiene una intranet? Fundamentalmente dos: independencia de los proveedores habituales de soluciones y una única forma de trabajar que evita tener que aprender sistemas nuevos, lo que redunda en un ahorro de formación. Por otra parte, una intranet suele estar dotada de una velocidad bastante mayor que la habitual en Internet, lo que posibilita una comunicación muy fluida, incluso, cuando se trata de flujos de información multimedia.

Terminología Internet

Desde el punto de vista fisico, Internet no es una simple red, sino miles de redes informáticas que trabajan conjuntamente bajo los protocolos TCP/IP (Transmision Control Protocol/lnternet Protocol - Protocolo de Control de Transmisiones/Protoco lo Internet), entendiendo por protocolo un conjunto de normas que regulan la comunicación entre los distintos dispositivos de una red. Desde el punto de vista del usuario, Internet es una red pública que interconecta universidades, centros de investigación, servicios gubernamentales y empresas.

El conjunto de protocolos de Internet está compuesto por muchos protocolos relacionados con la asociación formada por TCP e IP y relacionados con las diferentes capas de servicios de la red; esto es, las funciones de una red se pueden agrupar en capas de servicios de la red. Imagínese las capas como distintas estaciones por las que debe pasar un paquete de información cuando realiza la ruta de un ordenador a otro conectados a diferentes puntos dentro de

Internet. Por ejemplo, el protocolo TCP/IP visto desde este punto de vista puede imaginárselo de forma resumida así:



Entre dos capas puede haber una interfaz de programación (API) para interpretar los mensajes o paquetes a medida que van pasando.

Utilizar una interfaz de programación, como la API de Windows Sockets, libera al programador de tratar con detalles de cómo se pasan los paquetes de información entre las capas inferiores.

La capa de enlace y de red se encargan de empaquetar la información y de llevar los paquetes de un lugar a otro de la red. ¿Cómo se identifican estos lugares?

La respuesta es: con direcciones de Internet que permitan identificar tanto el ordenador como el usuario, ya que un mismo ordenador puede ser usado por diferentes usuarios. Estas direcciones son especificadas según un convenio de sistema de nombres de dominio (DNS).

Un DNS tiene el formato siguiente:

[subdominioJ.[subdominioJ.[...].dominio

ejemplo: uni.alcala.es // subdominio.subdominio.dominio

Algunos dominios de orden superior muy conocidos:

Dominio	*Cobertura*
com	organizaciones comerciales
edu	instituciones educativas
net	suministradores de servicios de red
us	Estados Unidos
de	Alemania
es	España

Cada nombre de dominio se corresponde con una única dirección de Internet o dirección IP. Una dirección IP es un valor de 32 bits dividida en cuatro campos de 8 bits. Por ejemplo:

200.16.20.12

Para referirse a un usuario dentro de un determinado subdominio, la sintaxis es:

usuario@[subdominio].[subdominio].[.. .].dominio

Los programas que gestionan los nombres de dominio se denominan servidores de nombres. Cada servidor de nombres posee información completa de una determinada zona (subconjunto de un dominio) y de otros servidores de nombres responsables de otras zonas. De esta forma, cuando llega una solicitud de información sobre la zona de la que es responsable un determinado servidor, éste sencillamente proporciona la información. Sin embargo, cuando llega una solicitud de información para una zona diferente, el servidor de nombres se pone

en contacto con el servidor de esa zona. Los servidores DNS constituyen la base de datos distribuida de nombres de Internet.

Veamos un poco de programación relacionando ambos conceptos

Direcciones Internet y dominios

Habitualmente, no tendrá que preocuparse por las direcciones Internet (la dirección numérica de un servidor consta de cuatro bytes, o seis en el caso de IPv6; por ejemplo, 132.163.4.102). Sin embargo, puede usar la clase InetAddress si necesita convertir nombres de servidor en direcciones Internet.

A partir del SDK 1.4, el paquete java.net soporta direcciones Internet IPv6, ofreciendo lo que hace el sistema operativo del servidor. Por ejemplo, podrá hacer uso de las direcciones IPv6 en Solaris, pero no en Windows (o tal vez ya si, el tiempo pasa …).

El método estático getByName devuelve un objeto InetAddress de un servidor. Por ejemplo:

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

devuelve un objeto InetAddress que encapsula la secuencia de cuatro bytes 132.163.4.102. Se puede acceder a los bytes a través del método getAddress.

```
byte[] addressBytes = address.getAddress();
```

Para facilitar el balanceo de la carga, los nombres de servidores con mucho tráfico suelen corresponderse con varias direcciones Internet. Interesante saber con cuantas cuenta actualmente el servidor java.sun.com. Cuando se accede al servidor, se selecciona una de ellas de forma aleatoria. Se puede llamar a todos los servidores mediante el método getAllByName.

```
InetAddress[] addresses = InetAddress.getAIIByName(host);
```

Por último, hay veces en las que se necesita la dirección del servidor local. Si sólo se pregunta por el valor de localhost, siempre se obtendrá la dirección 127.0.0.1, lo que no resulta especialmente útil. En su lugar, mejor usar el método estático getLocalHost.

```
InetAddress address = InetAddress.getLocalHost();
```

El ejemplo que sigue es un sencillo programa que muestra la dirección Internet de su servidor local, en el caso de que no especifique nada en la línea de entrada, o las correspondientes al servidor especificado, si informa.

```
import java.net.*;
import java.io.*;
public class InetAddressTest{
   public void direccion() throws IOException{
      BufferedReader in = new BufferedReader
            (new InputStreamReader(System.in));
      String dominio = "xxx" , auxText;
      InetAddress[] addresses;
      while(!dominio.startsWith("/")){
         try{
            System.out.println("de que dominio necesita su(s)
                              direccion(es) IP? ");
            dominio = in.readLine();
            if(dominio.length() == 0)
                auxText = "La mia, la de mi PC ";
            else{
                auxText = "Me interesan las de ";
            }
            System.out.println(auxText+dominio+", por favor");
            if (dominio.length() > 0){
               if(dominio.startsWith("/")) break;
```

```
                addresses = InetAddress.getAllByName(dominio);
                for (int i = 0; i < addresses.length; i++)
                    System.out.println("Va...("+i+1+") "+ addresses[i]);
             }
             else{
                InetAddress localHostAddress
                   = InetAddress.getLocalHost();
                System.out.println(localHostAddress);
             }
          }
          catch (Exception e){e.printStackTrace();}
       } //  while(...)
   }    //  direccion

   public static void main(String[] args){
       InetAddressTest iAT = new InetAddressTest();
       try{
            iAT.direccion();
            System.out.println("Demo finished!");
       }catch(IOException e){System.out.println("problema en lectura");}
   }
}
```

```
run:
de que dominio necesita su(s) direccion(es) IP?
Me interesan las de www.java.sun.com, por favor
Va...(0) www.java.sun.com/72.5.124.93
de que dominio necesita su(s) direccion(es) IP?
Me interesan las de www.LaNacion.com.ar, por favor
Va...(0) www.LaNacion.com.ar/200.59.146.34
de que dominio necesita su(s) direccion(es) IP?
Me interesan las de www.frc.utn.edu.ar, por favor
Va...(0) www.frc.utn.edu.ar/200.69.137.165
Va...(1) www.frc.utn.edu.ar/200.69.137.166
Va...(2) www.frc.utn.edu.ar/170.210.46.20
de que dominio necesita su(s) direccion(es) IP?
La mia, la de mi PC , por favor
tymoschuk/200.82.18.39
de que dominio necesita su(s) direccion(es) IP?
Me interesan las de /, por favor
Demo finished!
BUILD SUCCESSFUL (total time: 1 minute 13 seconds)
```

La capa de transporte es responsable de la entrega fiable de los datos. En esta capa se emplean dos protocolos diferentes: TCP y UDP. TCP toma mensajes de usuario de longitud variable y los pasa al nivel de red, solicitando acuse de recibo; UDP es similar, salvo en que no solicita acuse de recibo de los datos.
La capa de aplicación proporciona una interfaz a la aplicación que ejecuta un usuario. Dicho de otra forma, proporciona el conjunto de órdenes que el usuario utiliza para comunicarse con otros ordenadores de la red.

Existen muchos protocolos en TCP/IP. A continuación, se indican algunos bastante conocidos:

Protocolos en TCP/IP

. **FTP** (File Transfer Protocol - Protocolo de transferencia de ficheros). Copia ficheros de una máquina a otra.
. **Gopher**. Protocolo que permite buscar y recuperar documentos mediante un sistema de menús.
. **POP 3** (Post Office Protocol- Protocolo de oficina de correos). Protocolo para gestionar el correo electrónico, en base a su recepción y envío posterior, entre el usuario y su servidor de correo. Con este fin son empleados también los protocolos lMAP (Internet Message Access Protocol) y HTTP (correo Web).
. **SMTP** (Simple Mail Transfer Protocol - Protocolo de transferencia correo). Protocolo para controlar el intercambio de correo electrónico entre dos servidores de correo en Internet.

. **Telnet** (Telecomunications NetWork Protocol - Protocolo de telecomunicaciones de red). Protocolo utilizado para establecer conexiones entre terminales remotos. Permite establecer conexiones entre máquinas con diferentes sistemas operativos.
. **USENet.** Nombre con el que se denomina al conjunto de los grupos de discusión y noticias, establecidos en Internet. La idea de USENet es servir como tablón electrónico de anuncios.
. **HTTP** (HyperText Transfer Protocol)o Protocolo de transferencia de hipertexto utilizado por WWW (World Wide Web - La telaraña mundial). Se trata de un sistema avanzado para la búsqueda de información en Internet basado en hipertexto y multimedia. El software utilizado consiste en exploradores (browsers), también llamados navegadores, con una interfaz gráfica.

SERVICIOS EN INTERNET

Los servicios más comunes son el correo electrónico, la conexión remota, transferencia de ficheros, grupos de noticias y la WWW. Programas que facilitan estos servicios hay muchos, y su manejo, además de sencillo, es similar. Por eso, independientemente de los que se utilicen en los ejemplos, usted puede emplear los suyos de forma análoga.

En cualquier caso, tenga presente que sólo podrá acceder a Internet si además de un ordenador y un módem (un módem es un dispositivo que conecta un ordenador a una línea telefónica), ha contratado dicho servicio con un proveedor de Internet, o bien si su ordenador forma parte de una red que le ofrece acceso a Internet.

El correo electrónico (correo-e o e-mail) es uno de los servicios más utilizados en todo el mundo. Como su nombre indica, tiene como finalidad permitir enviar y recibir mensajes. Un ejemplo de aplicación que proporciona este servicio es Microsoft Outlook Express.

La orden telnet (de Windows, Unix, etc.) proporciona la capacidad de mantener sesiones como un terminal del ordenador remoto, lo que le permite ejecutar órdenes como si estuviera conectado localmente.

El protocolo de transferencia de ficheros (ftp) es un método sencillo y efectivo de transferir ficheros ASCII y binarios entre ordenadores conectados a una red TCP/IP. Las órdenes ftp utilizadas con mayor frecuencia son las siguientes:

Orden ftp	*Significado*
ascii	Establece el modo ASCII para la transferencia de ficheros.
binary	Establece el modo binario para la transferencia de ficheros.
bye	Finaliza la sesión *ftp* y sale.
cd	Cambia de directorio de trabajo en el ordenador remoto.
close	Finaliza la sesión *ftp*.
ftp	Inicia una sesión *ftp*.
get	Obtiene un fichero del ordenador remoto.
help	Proporciona las órdenes *ftp* disponibles o información relativa a la orden especificada.
lcd	Cambia al directorio local de trabajo. Suele utilizarse para seleccionar los directorios al que irán a parar los ficheros transferidos.
ls	Lista el contenido de un directorio remoto.
mget	Obtiene un grupo de ficheros que pueden haberse especificado utilizando algún comodín.
mput	Envía un grupo de ficheros que pueden haberse especificado utilizando algún comodín.
open	Inicia una conexión *ftp* con el ordenador remoto especificado.
put	Envía un fichero al ordenador remoto.

Además de utilizar línea de comandos, existen interfaces gráficas de usuario que facilitan mucho la tarea.

Los grupos de noticias, noticias USENET, netnews o simplemente news, son foros de discusión en línea. Los artículos de noticias de USENET se clasifican en grupos de noticias por temas (ordenadores, aplicaciones, alpinismo, etc.).

La world wide web, también conocida por www o simplemente Web, es uno de los logros más interesantes en Internet. La Web es un sistema hipermedia interactivo que permite conectarse a grandes cantidades de información en Internet.
Un sistema hipermedia está compuesto por páginas que contienen texto cuidadosamente formateado, imágenes llenas de color, sonido, vídeo y enlaces a otras páginas distribuidas por todo el mundo. Puede acceder a esas otras páginas simplemente seleccionando uno de los enlaces. La información se recupera automáticamente sin necesidad de saber dónde está.

El proyecto Web, que fue iniciado en el centro europeo de investigación nuclear (European Center for Nuclear Research), ha crecido a una velocidad impresionante. Esto se debe fundamentalmente a que soporta información de todo tipo: texto, sonido y gráficos.

PÁGINAS WEB

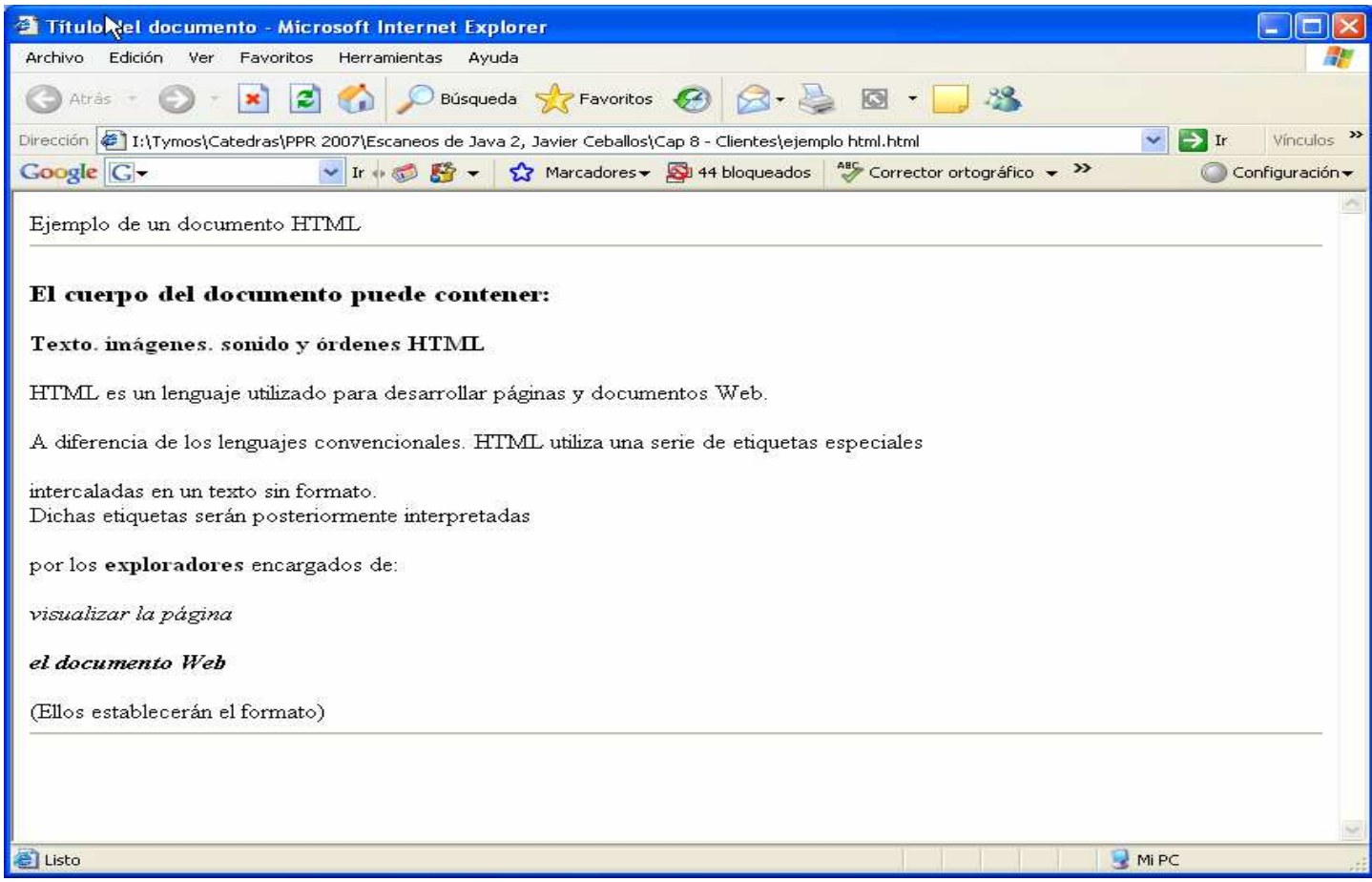
Sin duda UD ya ha accedido a muchas páginas de la Web; Pero, ¿cómo se puede crear una página Web de estas características a la que otros usuarios puedan acceder? Hay muchas herramientas que le permitirán realizardo. Para ello, antes debe conocer básicamente lo que es HTML (HyperText Markup Language), el lenguaje utilizado para construir páginas Web.

Qué es HTML

HTML es un lenguaje utilizado para desarrollar páginas y documentos Web. (Hemos comenzado a verlo en la Unidad II). A diferencia de los lenguajes convencionales, HTML utiliza una serie de etiquetas especiales intercaladas en un documento de texto sin formato. Dichas etiquetas serán posteriormente interpretadas por los exploradores encargados de visualizar la página o el documento Web con el fin de establecer el formato.

Para editar una página HTML y posteriormente visualizarla, todo lo que necesita es un editor de texto sin formato y un explorador Web. Para ver una página HTML no necesita una conexión a la red; cualquier explorador Web debe permitirle hacerlo trabajando en local. No obstante, existen otras herramientas como FrontPage que facilitan la generación de páginas HTML. Posteriormente, las páginas deben ser colocadas en un servidor Web para que otros usuarios puedan acceder a ellas. Veamos un ejemplo.

```
<html>
   <head>
    <title>Título del documento</title>
   </head>
   <body> <hl>Ejemplo de un documento HTML</hl>
    <hr> <h3>El cuerpo del documento puede contener:</h3>
    <h4>Texto. imágenes. sonido y órdenes HTML</h4>
    <p> HTML es un lenguaje utilizado para desarrollar
           páginas y documentos Web.
    <p> A diferencia de los lenguajes convencionales. HTML
           utiliza una serie de etiquetas especiales
    <p> intercaladas en un texto sin formato.
    <br> Dichas etiquetas serán posteriormente interpretadas
    <p>  por los <b>exploradores</b> encargados de:
    <p><i>    visualizar la página</i>
    <p><b><i>    el documento Web</i></b>
    <p>  (Ellos establecerán el formato)
    <hr>
   </body>
</html>
```



URL's

A los recursos de la Web se accede por medio de una dirección descriptiva conocida como URL (Universal Resource Locator - Localizador de recursos universal). Todo aquello a lo que se puede acceder en la Web tiene un URL.

Los URL son básicamente una extensión de la ruta que define a un fichero (path). Un URL añade, además, un prefijo que identifica el método de recuperación de la información que ha de emplearse (http, ftp, gopher, telnet, news, etc.), así como un nombre de dominio o dirección IP que indica el servidor que almacena la información. Finalmente aparece la ruta de acceso al fichero. Por ejemplo:

http://www.sun.com/software/download/app_dev.html

Enlaces entre páginas

Hemos dicho que la world wide web es un servicio de información basado en hipertexto. Se denomina hipertexto a la capacidad de asociar a una palabra o imagen de un documento un código oculto, de forma que cuando un usuario haga clic en esa palabra o imagen, el sistema lo transporta desde el lugar actual a otro lugar del mismo o de otro documento. Estos códigos especiales se denominan enlaces (links).

Para definir un enlace, se utiliza la etiqueta a (anchor o ancla). Esta etiqueta delimita el texto que se quiere utilizar para enlazar con otra página. Este texto aparecerá subrayado para el usuario.

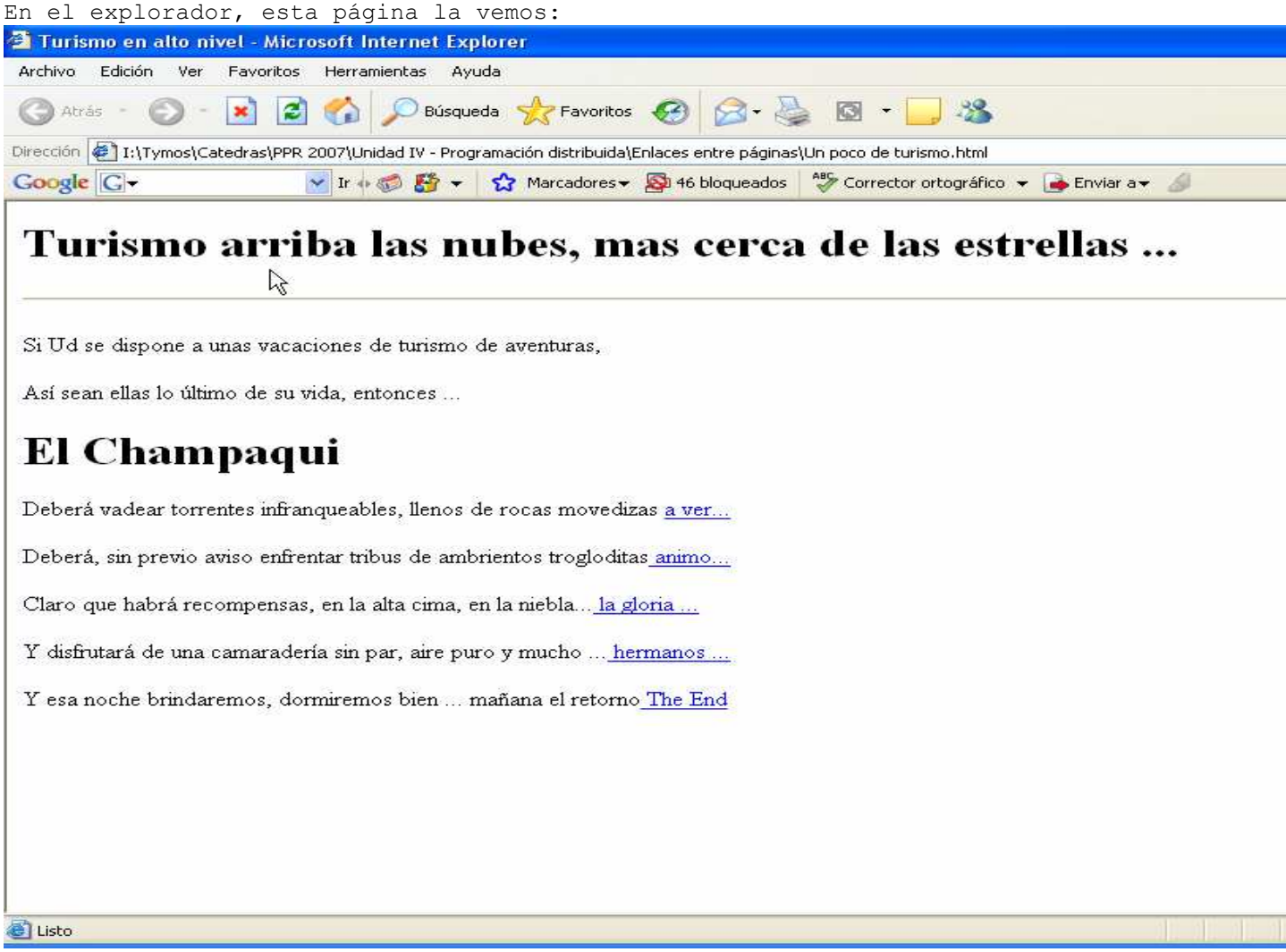
Para indicar al explorador la página que tiene que recuperar cuando el usuario haga clic en un enlace, incluya en la etiqueta de apertura a el atributo href y escriba a continuación el URL de la página que se quiere recuperar.

Hagamos un ejemplo simple. En una carpeta ponemos **Un poco de turismo.html**, quien tiene referencias a otras cinco URL's, cada una de las cuales incorpora una imagen. Una forma de organizar nuestros álbums, no creen?

Un poco de turismo.html

```
<html>
  <head>
    <title>
      Turismo en alto nivel
    </title>
  </head>
  <body>
    <h1>Turismo arriba las nubes, mas cerca de las estrellas ...</h1>
    <hr>
    <p> Si Ud se dispone a unas vacaciones de turismo de aventuras,
    <p> Así sean ellas lo último de su vida, entonces ... <h1>El Champaqui</h1>
    <p> Deberá vadear torrentes infranqueables, llenos de rocas movedizas <a
          href="Huy, que dificil.html"> a ver...</a>
    <p> Deberá, sin previo aviso enfrentar tribus de ambrientos trogloditas<a
          href="trogloditas.html"> animo...</a>
    <p> Claro que habrá recompensas, en la alta cima, en la niebla...<a
          href="cimaJorge.html"> la gloria ...</a>
    <p> Y disfrutará de una camaradería sin par, aire puro y mucho ...<a
          href="cimaMiguel.html"> hermanos ...</a>
    <p> Y esa noche brindaremos, dormiremos bien ... mañana el retorno<a
          href="campamentoBase.html"> The End</a>
  </body>
</html>
```

En el explorador, esta página la vemos:



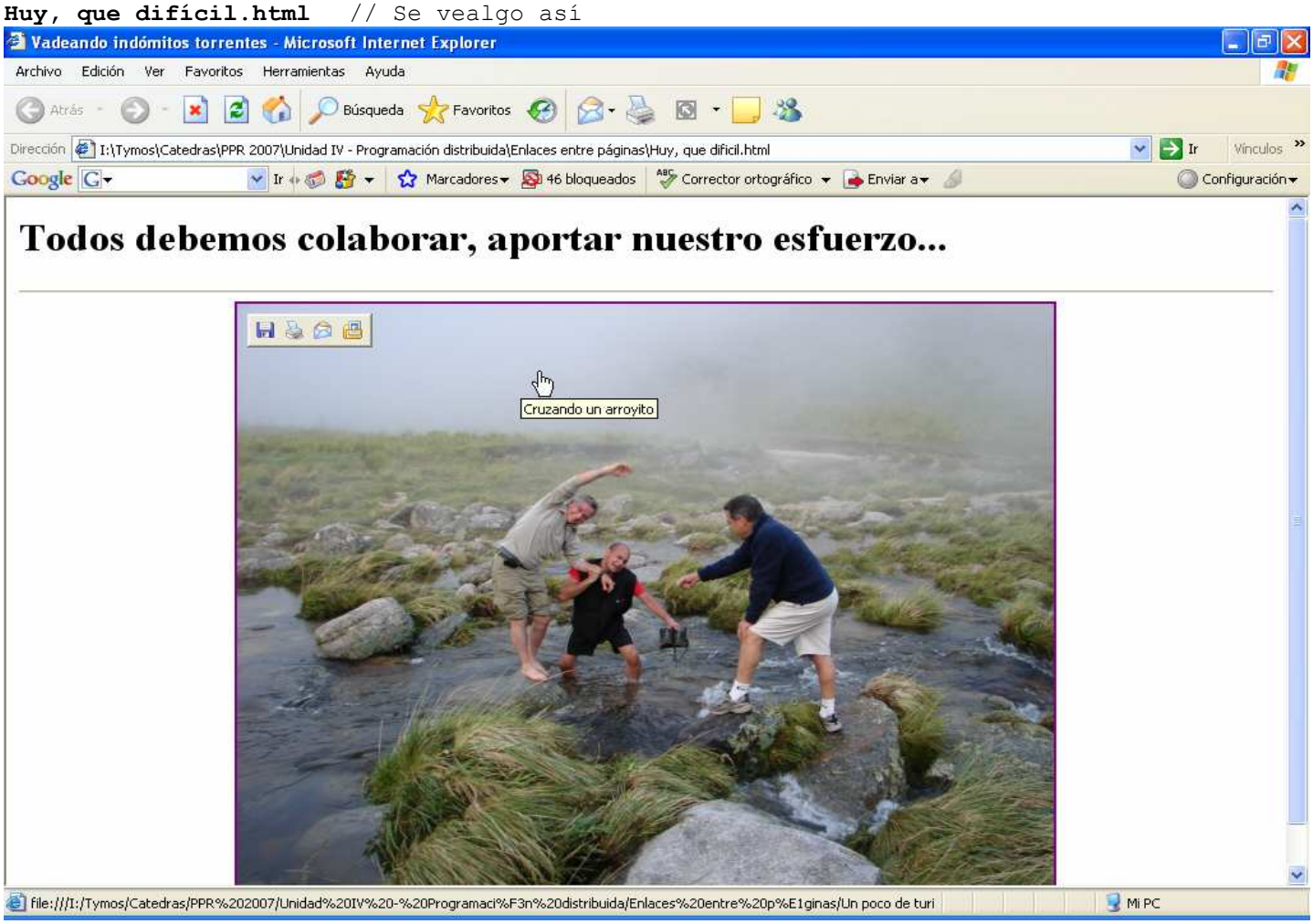
Huy, que difícil.html // primer página referenciada

```
<html>
  <head> <title>Vadeando indómitos torrentes </title> </head>
  <body>
    <h1>Todos debemos colaborar, aportar nuestro esfuerzo...</h1>
    <hr>
       <center>
       <a href="Un poco de turismo.html">
       <img src="Huy, que dificil.jpg" align=bottom
       alt="Cruzando un arroyito">
       </a>
       </center>
    <hr>
  </body>
</html>
```

Trogloditas.html // segunda página, resto similar (Verlo todo en la web)

```
<html>
  <head> <title> Enfrentando inauditos riesgos </title> </head>
  <body>
    <h1>Parecen amigables, no lo son... a correr!!!</h1>
    <hr>
       <center>
       <a href="Un poco de turismo.html">
       <img src="Trogloditas.jpg" align=bottom
       alt="Un momento, que momento ...">
       </a>
       </center>
    <hr>
  </body>
</html>
```

Huy, que difícil.html // Se vealgo así



Formularios

Los formularios permiten crear interfaces gráficas de usuario dentro de una página Web. Los componentes de estas interfaces, denominados también controles, serán cajas de texto, cajas para clave de acceso, botones de pulsación, botones de opción, casillas de verificación, menús, tablas, listas desplegables, etc. (Gran parte de esto lo vimos en la unidad II, en Java; En HTML también se puede)

Para crear un formulario, se utiliza la etiqueta form. Por ejemplo:

```
<form method={get|post} action="fichero para procesar los datos">
      Componentes del formulario
</form>
```

El atributo method es opcional y su valor por omisión es **get** indicando así al navegador que vienen los nombres de los campos del formulario y sus datos inmediatamente después del URL especificado por action (acción a tomar cuando se pulse el botón enviar). La cantidad de datos que se pueden concatenar al URL está limitada, truncándose la información en exceso. Esto no ocurre si el valor para este atributo es **post**; en este caso **adicionalmente** se puede enviar por flujo un fichero con los datos (o algunos de ellos) del formulario que serán recibidos en el servidor en la entrada estándar del componente de procesamiento. Ya veremos como tratamos esto del lado del servidor, en cada caso.

Los componentes que forman parte del formulario se definen dentro de la etiqueta **form**. A continuación vemos como se hace. En general, por cada componente se envía al servidor su nombre y su contenido o valor especificado (si no se especifica un valor, se envía uno por omisión).

Entrada básica de datos

Existe una amplia variedad de componentes de entrada de datos. Para crearlos, se utiliza la etiqueta **input** y los atributos **type y name**:

```
<input
      type {text|password|checkbox|radio|hidden|image|submit|reset}
      name "Variable que toma el valor"
>
```

El valor del atributo type especifica el tipo de componente que se creará, y el atributo name especifica el nombre de la variable que almacenará el dato. Como ya los vimos en el cap. II, resumimos:

Caja de texto (JTextField)

```
Nombre: <input type="text" name="nombre" size="35">
```

El valor del atributo size especifica el tamaño de la caja. Otros atributos que se pueden utilizar son value para especificar un valor inicial, readonly para indicar que la caja es de sólo lectura y maxlength para especificar el número máximo de caracteres que un usuario puede escribir.

Caja de clave de acceso

Una caja de clave de acceso es un control de entrada de tipo password. Se trata de una caja de texto en la que los caracteres escritos son reemplazados por asteriscos.

```
Clave de acceso: <input type="password" name="clave" size="25" maxlength="20">
```

Casilla de verificación (JCheckBox)

Puede presentar dos estados: seleccionado y no seleccionado. Se utilizan para mostrar y registrar opciones que un usuario puede elegir; **puede seleccionar**

varias de un grupo de ellas. Por ejemplo, las siguientes líneas de código muestran tres casillas de verificación. La primera se mostrará seleccionada:

```
<input type="checkbox" name="cvl" value="l" checked> Opción 1 <br>
<input type="checkbox" name="cv2" value="2"> Opción 2 <br>
<input type="checkbox" name="cv3" value="3"> Opción 3 <br>
```

Botón de opción (JRadioButtom)

Igual que ocurre con la casilla de verificación, puede presentar dos estados: seleccionado y no seleccionado. Se utilizan para mostrar y registrar una opción que un usuario puede elegir entre varias; **cuando selecciona una, la que estaba seleccionada dejará de estarlo**. Por ejemplo, las siguientes líneas de código muestran tres botones de opción. El segundo se mostrará seleccionado:

```
<input type="radio" name="opcion" value="l"> Opción 1 <br>
<input type="radio" name="opcion" value="2" checked> Opción 2 <br>
<input type="radio" name="opcion" value="3"> Opción 3 <br>
```

Para el comportamiento descrito, todos los botones de opción tendrán el mismo atributo name y con un valor distinto del atributo value. El valor enviado será el correspondiente al botón seleccionado. El atributo checked permitirá seleccionar por omisión uno de los botones de un grupo.

Parámetros ocultos

Un parámetro oculto es un componente de entrada de tipo hidden. En este caso no se muestra ningún campo de entrada de datos al usuario, pero el par variable/ valor especificado es enviado junto con el formulario.

```
<input type="hidden" name="variable" value="valor">
```

Se suelen utilizar para mantener datos durante una sesión.

Enviar datos

Un botón enviar es un componente de entrada de tipo submit. Se utiliza para enviar los datos del formulario, pasando el control al programa indicado por el atributo action del formulario. Todo formulario debe tener un botón submit, a menos que incluya una caja de texto.

```
<input type="submit" value="Enviar datos">
```

El atributo value especifica una etiqueta no editable que se mostrará como título del botón. Normalmente este componente no envía datos, pero si se incluye el atributo name con un nombre de variable, será enviada la variable con el valor de value. Esto puede ser útil para distinguir cuál fue el botón pulsado cuando se incluyan varios.

```
<input type="submit" name="enviar" value="Enviar">
<input type="submit" name="buscar" value="Buscar">
```

Borrar los datos de un formulario

Un botón borrar es un control de entrada de tipo reset. Se utiliza para restablecer los valores iniciales de los controles del formulario.

```
<input type="reset" value="Borrar datos">
```

Imágenes

Una imagen es un componente de entrada de tipo image. Su finalidad es análoga al botón submit, pero en este caso se presenta una imagen en lugar de un botón. Los datos del formulario se enviarán al hacer clic sobre la imagen jpg o .gif.

Caja de texto multilínea (JCheckBox)

Necesaria si la entrada son varias líneas de texto libre; un mensaje, una resolución.
Mensaje: <br)<textarea name="mensaje" rows="5" co1s="20" wrap) </textarea)

Listas desplegables

Una lista desplegable permite seleccionar una opción entre varias. Es la mejor alternativa para añadir menús a una interfaz gráfica. La etiqueta que permite crear un control de este tipo es **select**. Las opciones de la lista se especifican utilizando la etiqueta option.

```
<se1ect name="opcion")
      <option va1ue="1") Opción 1
      <option se1ected va1ue="2") Opción 2
      <option va1ue="3") Opción 3
</se1ect)
```

El atributo value de option indica el valor asociado con la opción especificada; si se omite este atributo, el valor que se toma es el texto especificado para la opción. La etiqueta option que contenga el atributo selected será considerada la opción por omisión; caso de no especificarse ninguna se considerará la primera de las opciones. Se puede también especificar el atributo size para indicar el número de opciones que la lista visualizará a la vez.

Para permitir realizar **selecciones múltiples** utilizando las teclas Ctrl o Alt, hay que añadir el atributo multiple; en este caso se mostrará una lista desplegada. Por ejemplo:

```
<select name="opcion" multiple)
      <option value="l") Opción 1
      <option selected value="2") Opción 2
      <option value="3") Opción 3
</select)
```

Utilizando la etiqueta **optgroup**, se pueden agrupar las opciones (como si de un submenú se tratara); cada grupo será identificado con una etiqueta especificada por el atributo label. Por ejemplo:

```
<select name="opcion")
      <optgroup label="grupo 1")
            <option value="l") Opción 1
            <option value="2") Opción 2
            <option value="3") Opción 3
      </optgroup)
      <optgroup label="grupo 2")
            <option value="4") Opción 4
            <option value="5") Opción 5
      </optgroup)
</select)
```

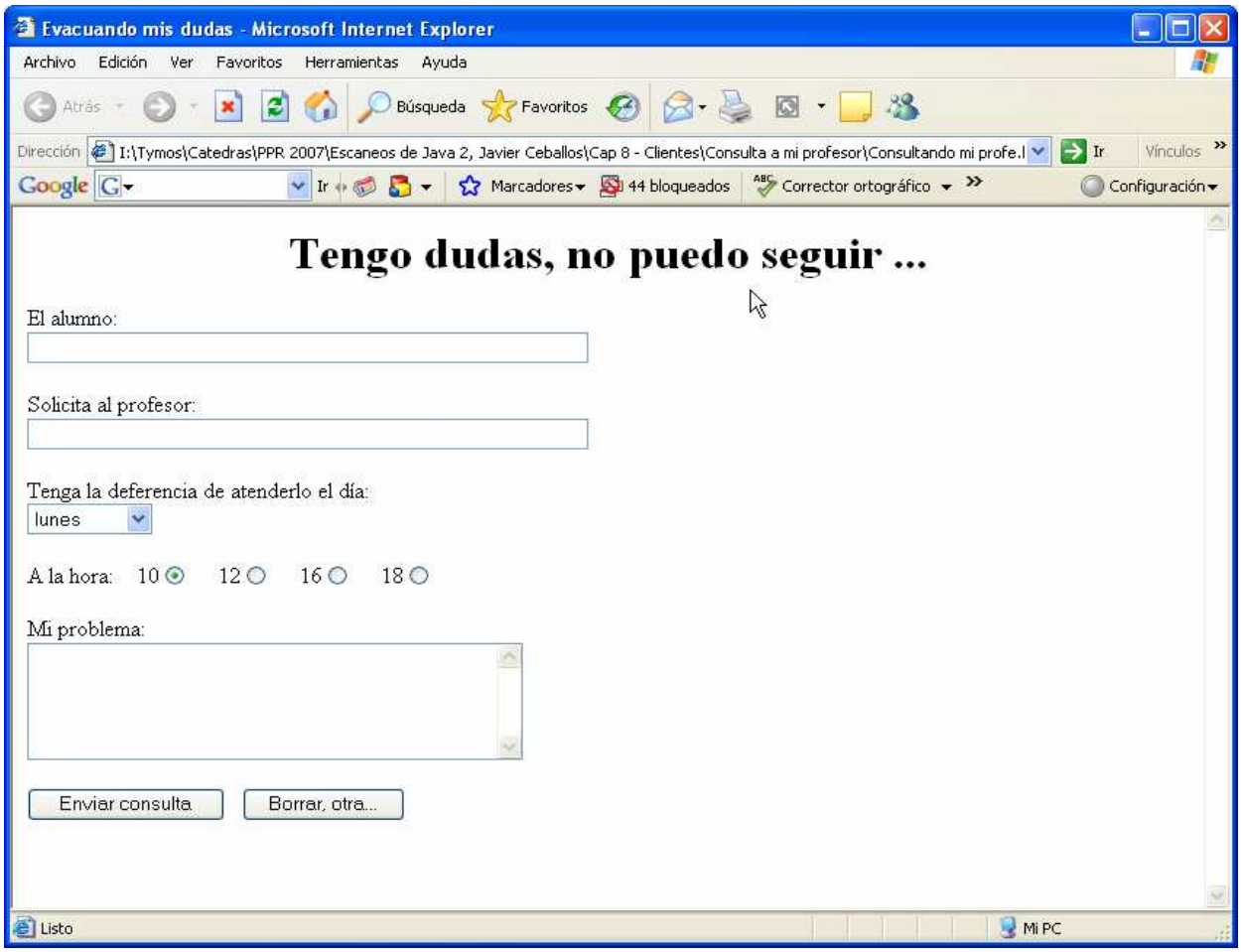
Aplicando la teoría expuesta hasta ahora, vamos a diseñar un formulario, como el que muestra la figura siguiente, que permita a un alumno solicitar una consulta a su profesor.

Una consulta a mi profesor

```
<html>
  <head>
    <title>Evacuando mis dudas</title>
  </head>
  <body>
```

```
    <h1 align="center">Tengo dudas, no puedo seguir ...</h1>
    <form action="http:\\localhost:8080/servlet/ConsultaProfe"
          method=post>
      El alumno:<br><input type="text" name="alumno" size="60">
      <br><br>Solicita al profesor:
      <br><input type="text" name="profesor" size="60">
      <br><br>Tenga la deferencia de atenderlo el día:<br>
      <select name="día">
        <option>lunes <option>miércoles <option>jueves
      </select>
      <br><br>A la hora:   
      10<input type="radio" name="hora" value="10" checked>
         
      12<input type="radio" name="hora" value="12">
         
      16<input type="radio" name="hora" value="16">
         
      18<input type="radio" name="hora" value="18">
      <br><br>Mi problema:<br><textarea name="miDuda"
                          rows="5" cols="40" wrap></textarea>
      <br><br><input type="submit" value="Enviar consulta">
        <input type="reset" value="Borrar, otra...">
    </form>
  </body>
</html>
```

Y esto lo vemos:



HTML no es la unica forma de diseñar formularios. Además existen:

XML(Extensible Markup Language - lenguaje extensible para análisis de documentos) está convirtiéndose rápidamente en el estándar para el intercambio de datos en la Web. Como en el caso de HTML, los datos se identifican utilizando etiquetas, y a diferencia de HTML las etiquetas indican lo que los datos significan en lugar de cómo visualizar los datos. Por ejemplo, la fecha de nacimiento de una persona se puede colocar en un párrafo HTML así:

```
<p>Ol/Ol/Z004</p>
```

Sin embargo, la etiqueta de párrafo no describe que el dato es la fecha de nacimiento. Simplemente indica al navegador que el texto que aparece entre las etiquetas se debe mostrar en un párrafo. En cambio, el dato sí queda descrito en una línea como la siguiente:

```
<FechaNacimiento>Ol/Ol/Z004</FechaNacimiento>
```

El lenguaje XML es un lenguaje que se utiliza para crear otros lenguajes que definen los componentes de un documento. Por ejemplo, podríamos utilizar XML para describir una persona: fecha de nacimiento, sexo, color, altura, peso, etc.

La creación de un documento XML tiene dos partes: crear un esquema XML y crear el documento utilizando los elementos definidos en el esquema.

El esquema se puede considerar como un diccionario que define las etiquetas que se utilizan para describir los elementos de un documento. Estas etiquetas son similares a las etiquetas HTML, pero con la diferencia de que quien crea el esquema crea los nombres y los atributos de las etiquetas. Por ejemplo:

```
<foto origen="Girafales.jpg">Profesor Girafales</foto>
```

La realidad es que un navegador no puede leer y mostrar un documento XML.
Pero sí puede convertir un documento XML en un documento HTML mediante una hoja de estilos XSTL *(Extensible Stylesheet Language Transformation)*. Una hoja de estilos se compone de una o más definiciones de plantillas que procesa un procesador XSLT cuando la etiqueta de la plantilla corresponde a un componente del documento XML.

XHTML (Extensible HyperText Markup Language - lenguaje mejorado para la escritura de páginas Web) es la siguiente generación de HTML que incorpora muchas de las características de XML. Utiliza prácticamente todos los elementos de HTML, pero impone nuevas reglas; por ejemplo, es sensible a mayúsculas y minúsculas, toda etiqueta de apertura tiene que tener una etiqueta de cierre, etc.

PÁGINAS WEB DINÁMICAS

El lenguaje HTML que mínimamente vimos es suficiente para visualizar documentos, imágenes, sonidos y otros elementos multimedia; pero el resultado es siempre una página estática.

Entonces, ¿qué podemos hacer para construir una página dinámica?, entendiendo por página dinámica una página que actualiza su contenido mientras se visualiza.

Haciendo un poco de historia, una de las primeras formas que se encontraron para dar dinamismo a las páginas HTML fue CGI (Common Gateway Interface). Esta interfaz permite escribir pequeños programas que se ejecutan en el servidor para aportar un contenido dinámico. El resultado es un código HTML, que se incluye en la página Web justo antes de ser enviada al cliente. Pese a que la CGI es fácil de utilizar, en general, no es un buen sistema porque cada vez que un cliente solicita una página con algún programa basado en esa interfaz, el programa tiene que ser cargado en memoria para ser ejecutado, lo que ocasiona un tiempo de espera excesivo. Además, si el número de usuarios es grande, los requerimientos de memoria también serán elevados, ya que cada proceso requiere de una copia propia en memoria antes de ejecutar.

Una alternativa a la CGI fue ISAPI (Internet Server Application Programming Interface). Esta API proporciona la funcionalidad necesaria para construir una aplicación servidora de Internet. A diferencia de CGI que trabaja sobre ejecutables, ISAPI trabaja sobre bibliotecas DLL. Esta diferencia hace que ISAPI sea un sistema más rápido, ya que por tratarse de una biblioteca dinámica sólo será cargada una vez y podrá ser compartida por múltiples procesos, lo que supone pocos requerimientos de memoria.

Posteriormente, las técnicas anteriores fueron sustituidas por la incorporación de secuencias de órdenes (scripts) ejecutadas directamente en el interior de la página HTML. Esto es, en lugar de consultar al servidor acerca de un ejecutable, el explorador puede ahora procesar las secuencias de órdenes a medida que carga la página HTML. El tratamiento de estas secuencias de órdenes puede hacerse tanto en el servidor Web como en el cliente. Los lenguajes más comunes para la escritura de secuencias de órdenes son JavaScript y VBScript.

Apoyándose en la técnica anterior y en un intento de potenciar la inclusión de contenido dinámico en páginas Web, Microsoft lanza una nueva tecnología, las páginas ASP (Active Server Page - Página activa del servidor o activada en el servidor). Una página ASP, sencillamente es un fichero .asp que puede contener: texto, código HTML, secuencias de órdenes y componentes *ActiveX.*
Con tal combinación se pueden conseguir de una forma muy sencilla páginas dinámicas y aplicaciones para la Web muy potentes. Un inconveniente de esta tecnología es que es propietaria de Microsoft y solamente está disponible para el servidor IIS (Internet Information Server) que se ejecuta sobre plataformas Windows. También es cierto que hay herramientas que permiten utilizar ASP sobre un servidor Apache en plataformas Unix pero aun hoy los componentes ActiveX no están disponibles para plataformas que no sean Microsoft Windows.

Cuando un cliente solicita una ASP, el servidor la intenta localizar dentro del directorio solicitado, igual que sucede con las páginas HTML. Si la encuentra, ejecutará las rutinas VBScript o JScript que contenga. Cuando el servidor ejecuta estas rutinas, genera un resultado consistente en código HTML estándar que sustituirá a las rutinas VBScript o JScript correspondientes de la página ASP. Una vez procesada la página, el servidor envía al cliente el contenido de la misma en HTML estándar, siendo así accesible desde cualquier navegador.

Actualmente, la tecnología ASP ha sido sustituida por ASP.NET, que es parte integrante de Microsoft .NET Framework. ASP.NET es algo más que una nueva versión de ASP; es una plataforma de programación Web unificada que proporciona todo lo necesario para que podamos crear aplicaciones Web. Al crear una aplicación Web, podemos elegir entre formularios Web y servicios Web XML, o bien combinarlas. Los formularios Web permiten crear páginas Web basadas en formularios. Por lo tanto este tipo de aplicaciones son una alternativa más para crear páginas Web dinámicas. Los servicios Web XML proporcionan acceso al servidor de manera remota, permitiendo el intercambio de datos en escenarios cliente-servidor utilizando estándares como HTTP y XML.

También, cuando Sun Microsystems presentó Java, una de las cosas que más impactaron fueron los *applets*: pequeños programas interactivos que son ejecutados por un navegador. Esto supuso una alternativa más para crear páginas Web dinámicas, ya que esta tecnología permite vincular código estándar Java con las páginas HTML (HyperText Markup Language - Lenguaje para hipertexto) que luego utilizaremos como interfaz de usuario, dotándolas de contenido dinámico e interactivo.

Posteriormente, *Sun Microsystems* introdujo los ***servlets***. Mientras que los applets incorporaron funcionalidad interactiva a los navegadores, los servlets la incorporaron a los servidores. Los servlets es la alternativa de Sun Microsystems para sustituir a la programación CGI. Ambas tecnologías proporcionan la misma funcionalidad básica, es decir, un cliente realiza una petición a un servidor; se ejecuta en el servidor el programa que responde a tal petición y devuelve el resultado al cliente para que lo muestre al usuario. La

diferencia es que utilizando servlets no es necesario lanzar un nuevo proceso por cada petición de un cliente, es decir, sólo se carga una copia de un servlet en la máquina virtual Java, independientemente del número de peticiones que tenga que atender. Con cada petición se inicia un hilo en vez de un proceso, lo cual reduce el uso de memoria del servidor y el tiempo de respuesta. Los servlets serán estudiados mas adelante en este capítulo.

Un servlet se escribe en Java y las respuestas, que deben tener formato HTML, se codifican como objetos String para ser enviadas a través del método println. Esto supone que cualquier cambio en la presentación requiere modificar el código Java, recompilarlo y volver a descargar los ficheros en el servidor.
Además, obliga a los desarrolladores a conocer Java y HTML. Evidentemente, separar la programación de la presentación, no sólo permitiría trabajar independientemente a desarrolladores en HTML y a desarrolladores en Java, sino que facilitaría el mantenimiento y abaratatía costes. Por eso Sun Microsystems, dándose cuenta de esta limitación, lanzó finalmente la tecnología **JSP (Java Server Pages)** con cuyo estudio finalizaremos este paradigma.

Bueno, consideremos que lo visto hasta aquí es una introducción a Internet. Vamos an entrar en detalles. Comencemos por servicios dentro del protocolo TCP/IP, iniciando con el cliente.

Conexión con un servidor

Antes de escribir nuestro primer programa en red, vamos a aprender algo acerca de una gran herramienta de depuración para la programación en red: telnet. Muchos sistemas operativos (UNIX, Linux, Windows) incluyen telnet.

En la linea de comandos escriba:

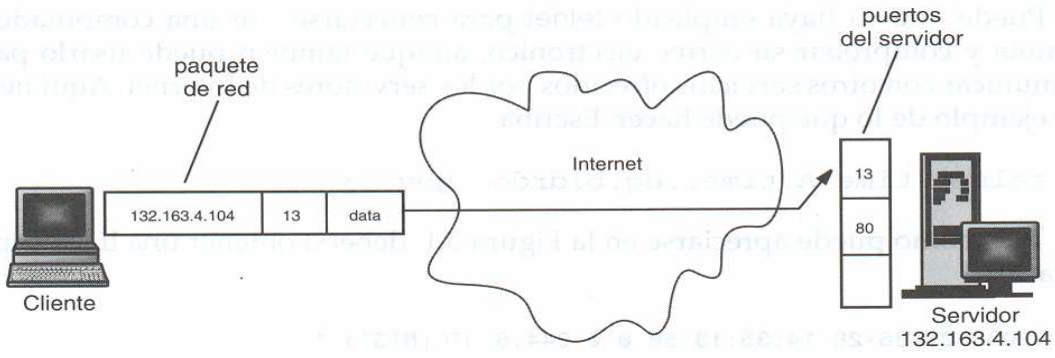
```
telnet tirne-A.tirnefreq.bldrdoc.gov 13
```

Y obtendrá una (críptica) respuesta:

```
53923 06-07-07 10:25:24   50 0  0  0.0 UTC (NIST) *

Se ha perdido la conexión con el host
```

¿Qué ha sucedido? Pues que ha conectado con el servidor del National Institute of Standards and Technology en Boulder, Colorado, y lo que ha obtenido es la medición de un reloj atómico de Cesio (desde luego, la hora no está totalmente actualizada debido a los retrasos de la red y además, si estamos en Argentina, hay un defasaje de 3 hs respecto a Colorado, EEUU, donde hay tres husos horarios, el reloj de la PC está indicando 7:25:xx). Por convenio, el servicio "hora del día" siempre está unido al puerto 13.

En lenguaje de red, **un puerto** no es un dispositivo físico, sino una abstracción para facilitar la comunicación entre un servidor y un cliente



Lo que ocurre es que el software del servidor está ejecutándose continuamente en la máquina remota, esperando cualquier tráfico de red que dialogue con el puerto 13. Cuando el sistema operativo de este servidor recupera un paquete de red que contiene una petición para conectar con el puerto 13, activa el servicio de escucha del servidor y establece la conexión, que permanece activa hasta que es finalizada por alguna de las dos partes. (El servidor, en este ejemplo)

Cuando se inicia un sesión telnet con time-A.timefreq.bldrdoc.gov en el puerto 13, una parte del software de red transforma la cadena "time-A.timefreq.bldrdoc.gov" a su dirección IP (Protocolo Internet) correcta, 132.163.4.102, se envía una petición de conexión a dicho servidor, solicitándole la entrada al puerto 13. Una vez establecida, el programa remoto devuelve una línea de datos y cierra la conexión.

Y como puede hacerse esto en Java?. Una primera y básica forma, usamos recursos de la plataforma J2SE, la clase Socket, para comunicarnos con el puerto 13 de la URL donde está la computadora conectada al reloj de Cesio… Y entramos en el tema:

Implementación de clientes

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

/**
   Este programa realiza una conexión Socket a un reloj atómico en
   Boulder, Colorado, y muestra la hora que el servidor envía
*/
public class SocketTest{
   public static void main(String[] args){
      try{
         Socket s = new Socket("time-A.timefreq.bldrdoc.gov",13);
         // Instanciamos un objeto Socket s;
         // parametros: direccion remota y puerto

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

            //  Concretamente, hemos establecido un flujo de datos
            //  de entrada desde el puerto 13 del servidor del
            //  National Institute of Standards and Technology …

         boolean more = true;
         while (more){    //  Ciclo de lectura del flujo in
            String line = in.readLine();
            if (line == null)
               more = false;
            else
               System.out.println(line);
         }
      }catch (IOException e){e.printStackTrace();}
   }
}
```

```
53923 06-07-07 11:03:31 50 0 0 431.6 UTC(NIST) *
```

Este programa es muy simple, observe que hemos importado el paquete java.net y que hemos capturado cualquier error de entrada/salida que se produzca (el código está encerrado en un bloque try/catch), ya que pueden ocurrir muchas cosas en una conexión de red, la mayor parte de los métodos relacionados con ella " amenazan" con lanzar errores de este tipo, por lo que es necesario capturarlos para que el código pueda compilarse.

El proceso continúa hasta que finaliza el flujo y el servidor desconecta. Esto se sabe cuando el método readLine devuelva una cadena null.

Este programa sólo funciona con servidores muy simples, como un servicio de "hora del día". En programas de red más complejos, el cliente envía peticiones de datos al servidor, y puede que éste no realice la desconexión inmediatamente después del final de una de estas peticiones. Veremos cómo implementar este comportamiento mas adelante.

En esencia, la clase Socket es agradable y fácil de usar porque la tecnología Java oculta las complejidades derivadas del establecimiento de la conexión de red y del envío de datos a través de ella. En esencia, el paquete java.net le proporciona la misma interfaz de programación que podría utilizar en el trabajo con un archivo.

Implementación de servidores

Ahora que ya tenemos implementado un cliente de red básico que recibe datos a través de la Red, vamos a desarrollar un sencillo servidor que pueda devolver información. Una vez iniciado el programa de servidor, espera hasta que algún cliente conecta con su puerto. Hemos elegido el 8189 porque no se utiliza por ninguno de los servicios estándar. La clase ServerSocket se utiliza para establecer un Socket; la sentencias:

ServerSocket s = new ServerSocket(8189); //establece un servidor que monitoriza el puerto 8189.

Socket incoming = s.accept(); // indica al programa que debe esperar indefinidamente hasta que algún cliente conecte con ese puerto. Una vez que alguien establezca una conexión válida a través de la red, este método devuelve un objeto Socket que representa la conexión que se ha establecido. Puede usar este objeto para obtener un lector (input reader) y un escritor (output writer) para ese socket, tal y como se verá en el siguiente fragmento de código:

```
BufferedReader in = new BufferedReader (new          // Entrada al socket prove
      InputStreamReader(incoming.getInputStream())); // niente del cliente


PrintWriter out = new PrintWriter         // Salida por el Socket en dirección
      (incoming.getOutputStream(), true /* autoFlush */); // al cliente
```

El flujo de salida del cliente es el flujo de entrada del servidor. Y a su vez **El flujo de salida del servidor se convierte en el flujo de entrada del cliente.**

En varios ejemplos que siguen, transmitiremos texto a través de sockets. Y entonces podemos alternar los streams entre lectores y escritores. A continuación, usamos el método readLine (definido en BufferedReader, no en InputStream) y print (definido en PrintWriter, no en OutputStream). Si quisiéramos transmitir datos binarios, deberíamos cambiar los streams por DatalnputStream y DataOutputStreams. Para trabajar con objetos serializados, usaremos ObjectlnputStream y ObjectOutputStreams.

Ya tenemos un cliente de red básico (SocketTest.java) que solicita/recibe la hora desde un servidor en el sitio **timefreq.bldrdoc.gov**. El programa que está en ese sitio no lo conocemos, no sabemos en que lenguaje está codificado, no lo podemos modificar.

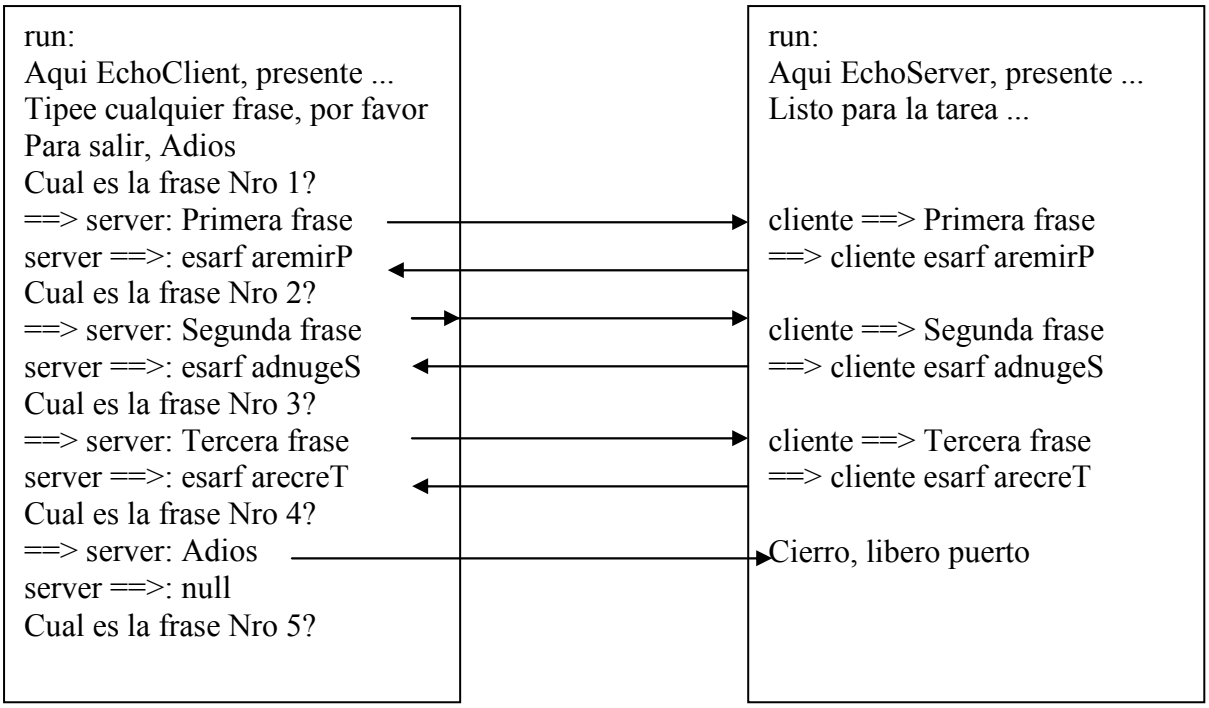
Queremos algo con más control de parte nuestra. Por ejemplo, un par cliente/servidor, ambos en Java, en el cliente tipeamos una frase, la enviamos al servidor (localhost), puerto 8189, y este nos la devuelve revertida. Finaliza cuando el cliente tipea Adiós

Aquí va el codigo.

```
package javaapplication10;
import java.io.*; import java.net.*;
public class EchoClient{

    //  Comunicacion con el humano
    String entrada;  // lo que el operador digita en el teclado
    String salida;   // lo que me devuelve el servidor, lo mostrare en pantalla
    BufferedReader teclado = new BufferedReader(
            new InputStreamReader(System.in));//recibo del teclado

    //  Comunicacion con el servidor
    BufferedReader in;      // el flujo de entrada desde EchoServer
    PrintWriter   out;      // el flujo de salida  hacia EchoServer
    Socket cliente;         // Referencia al puerto de   EchoServer

    public void tarea(){
       try{
          cliente = new Socket("Localhost",8189); // Instanciamos un cliente
                               // informando su direccion remota y puerto

          in = new BufferedReader  // Instanciamos in para establecer el flujo
              (new InputStreamReader(cliente.getInputStream()));//desde servidor

          out = new PrintWriter // Instanciamos out para establecer el flujo
               cliente.getOutputStream(), true); // hacia el servidor

          int contFrases = 0;
          System.out.println("Aqui EchoClient, presente ...");
          System.out.println("Tipee cualquier frase, por favor");
          System.out.println("Para salir, Adios");

          while (true){ //  Ciclo de lectura y proceso en el servidor
             contFrases++;
             System.out.println("Cual es la frase Nro "+contFrases+"?");
             entrada = teclado.readLine();
             if (entrada == "Adios")  // Terminamos
                 break;
             else{                    // Tengo una frase
                 System.out.println("==> server: "+entrada);// Recibo de teclado
                 out.println(entrada);  // La envio al servidor EchoServer
                 salida = in.readLine(); // leo lo que EchoServer me devuelve
                 System.out.println("server ==>:"+salida);//muestro por pantalla
                 if (entrada == "Adios")  // Terminamos
                     break;
             }   //  else
          }  //    while
          cliente.close();
       }catch (IOException e){e.printStackTrace();}
        finally{System.out.println("Conexion cerrada, nos vamos ...");}
    } //    void tarea

    public static void main(String[] args){
        EchoClient client = new EchoClient();
        client.tarea();
    }
}     // EchoClient

package javaapplication9;
import java.io.*; import java.net.*;
public class EchoServer{
    ServerSocket server;
    Socket incoming;
    String revLine;
    public String reverse(String linea){
```

```
        revLine = "";
        for(int i = linea.length()-1;i>=0;i--)
            revLine+=linea.charAt(i);
        return revLine;
    }
    public void tarea(){
        try{
            server = new ServerSocket(8189);// servidor escucha pto 8189;

            incoming = server.accept();// espera una conexión indefinidamente
            // El método accept() bloqueará el thread actual hasta que la
            // conexión se haya establecido. El método devuelve un objeto Socket
            // a través del cual el programa puede comunicar con el cliente.

            // Flujo de entrada (cliente ➔ servidor)
            BufferedReader in = new BufferedReader
                (new InputStreamReader(incoming.getInputStream()));

            // Flujo de salida (servidor ➔ cliente)
            PrintWriter out = new PrintWriter
                (incoming.getOutputStream(), true);
            int contFrases = 0;
            System.out.println("Aqui EchoServer, presente ...");
            System.out.println("Listo para la tarea ...");

            // Ciclo de entrada / salida
            boolean done = false;
            while (true){
                String entrada = in.readLine();
                if (entrada.trim().equals("Adios")){
                    break;
                }
                else{
                  System.out.println("cliente ==> "+entrada);// del cliente
                  System.out.println("==> cliente "+reverse(entrada));
                  out.println(revLine);   // al cliente
                }
            }     //    while
            System.out.println("Cierro, libero puerto");
            incoming.close();//  Liberar el puerto !!!
        }
        catch (Exception e){
             e.printStackTrace();}
    }   // tarea

    public static void main(String[] args )throws IOException{
        EchoServer server = new EchoServer();
        server.tarea();
   } //  public static void main
}   //  public class EchoServer
```

A continuación mostramos el resultado registrado con el entorno NetBeans Ide para el cliente y servidor, por separado. NetBeans IDE genera un archivo tipo texto para cada main() que esté ejecutando. En este ejemplo tenemos dos, uno de cada lado de la red, aunque ella esté contenida dentro de una única PC.



La dirección IP 127.0.0.1, llamada dirección local del bucle de vuelta (loopback), hace referencia a la máquina local. Ud, si ejecuta en local, esta es la forma de conectar.

Y cualquiera podrá acceder a su EchoServer, (conociendo su IP y puerto)
Para hacer funcionar este par en su propia PC:

- Desde su IDE Java ejecute el proyecto **EchoServer**
- Desde su IDE Java ejecute el proyecto **EchoClient**
- Desde el lado cliente tipee una frase cualquiera <Enter>

Observación: Si utiliza una conexión por marcación (Modem, ADSL) necesita tenerla en funcionamiento para llevar a cabo esta corrida. Incluso aunque esté trabajando localmente en su máquina, el software de red debe estar cargado. (Pero no tiene por que estar pagando pulsos; desconecte el modem y acepte el mensaje de error de conexión).

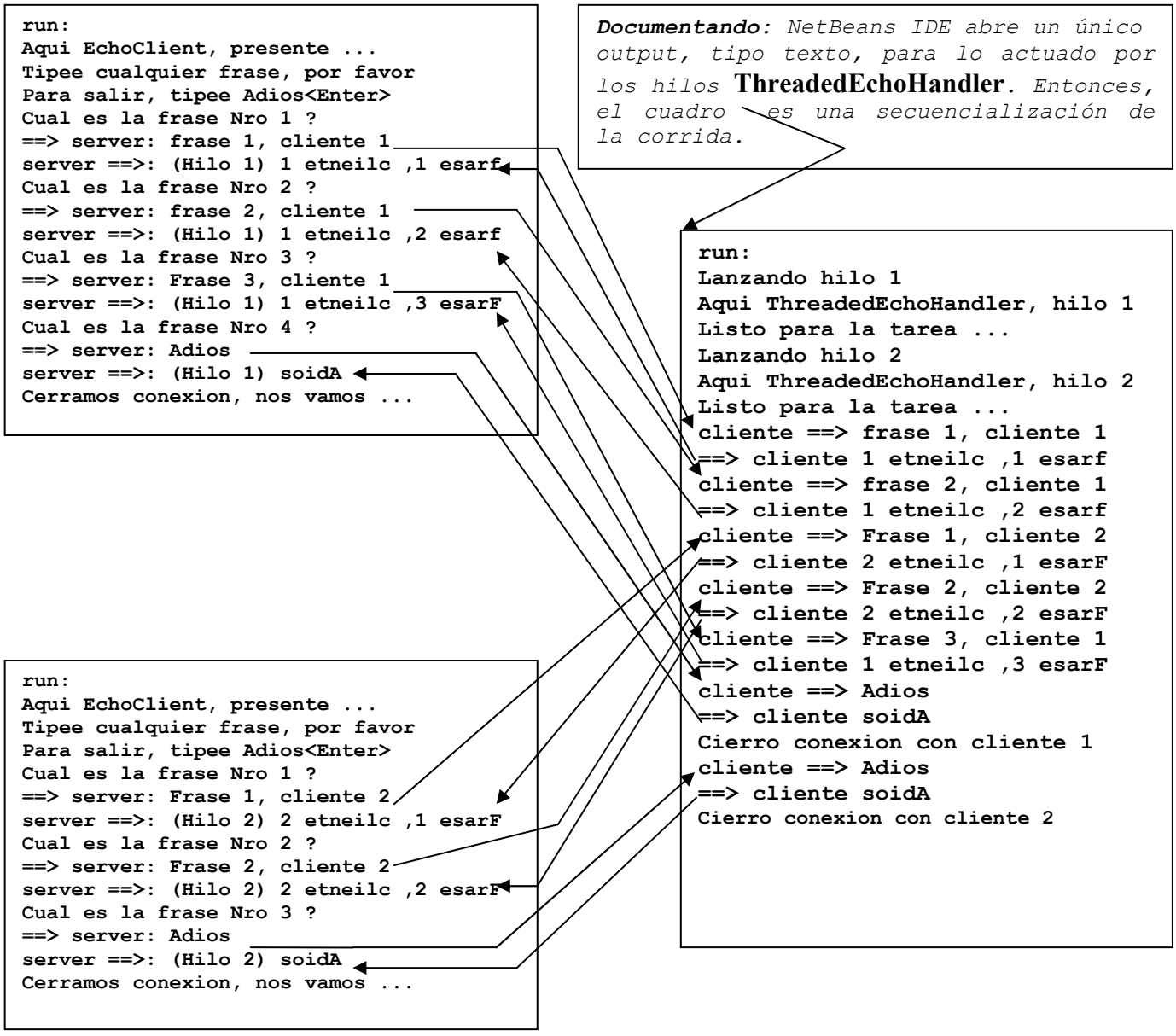
Servicio a varios clientes

Existe una seria limitación en el ejemplo del servidor que acabamos de ver. El servidor atiende un único cliente. **Son un par indisoluble.** Si probamos de arrancar un segundo cliente, la corrida será efectivamente arrancada, pero el servidor lo ignorará. Solo ve el primer cliente, nada más. Muy malo. En el mundo real lo normal es que un servidor esté ejecutándose constantemente en una computadora, y que muchos usuarios de Internet de cualquier parte del mundo puedan simultáneamente conectarse al mismo. Necesitamos un servidor con capacidad de atender a todos, a cada uno de ellos individualmente, en forma simultánea. Todo esto podemos hacerlo muy bien a través de (claro!!)los threads.

La estrategia de diseño es la siguiente: cada vez que una petición de conexión de un cliente tenga éxito, el servidor lanzará un nuevo thread que será el encargado de cuidar la conexión entre el servidor y ese cliente. Concretamente, el programa servidor (**ThreadedEchoServer**) sólo se encargará de seguir esperando nuevas conexiones y cada vez que detecte una arrancará un hilo **ThreadedEchoHandler.**
En el ejemplo que sigue seguiremos usando **EchoClient**, sin modificaciones.
EchoServer distribuirá su lógica en las dos clases dichas; numeraremos a los hilos para que quede claro cual atiende a cada cliente. Vamos con el código.

```
package javaapplication11;
import java.io.*;
import java.net.*;
/**
   Este programa hace lo mismo que EchoServer, solo que con tantos clientes
   como se conecten. Para cada EchoClient se abre un hilo ThreadedEchoHandler
*/
public class ThreadedEchoServer{
    ServerSocket server;
    Socket incoming;
    String revLine;
    public String reverse(String linea){
        revLine = "";
        for(int i = linea.length()-1;i>=0;i--)
            revLine+=linea.charAt(i);
        return revLine;
    }
    public void tarea(){
        int hilo = 1;   // Control de hilos
        try{
            server = new ServerSocket(8189);// servidor escucha pto 8189;
            for(;;){
                incoming = server.accept( );// espera indefinidamente
                System.out.println("Lanzando hilo "+hilo);
                Thread t = new ThreadedEchoHandler(incoming, hilo,this);
                t.start();
                hilo++;
            }
        }
        catch (Exception e){e.printStackTrace();}
    }   // tarea

    public static void main(String[] args )throws IOException{
        ThreadedEchoServer server = new ThreadedEchoServer();
        server.tarea();
   } //  public static void main
}   //  public class ThreadedEchoServer

class ThreadedEchoHandler extends Thread{
   private Socket incoming;
   private int hilo;
   private ThreadedEchoServer server;

   public ThreadedEchoHandler(Socket i, int hilo, ThreadedEchoServer server){
      incoming = i; this.hilo = hilo; this.server = server;
   }

   public void run(){
      try{
          // Flujo de entrada del servidor)
          BufferedReader in = new BufferedReader
             (new InputStreamReader(incoming.getInputStream()));

          // Flujo de salida del servidor
          PrintWriter out = new PrintWriter
              (incoming.getOutputStream(), true /* autoFlush */);
          int contFrases = 0;
          System.out.println("Aqui ThreadedEchoHandler, hilo "+hilo);
          System.out.println("Listo para la tarea ...");

            // Ciclo de entrada / salida
          boolean done = false;
          while (!done){
              String entrada = in.readLine(), revLine;
```

```
                System.out.println("cliente ==> "+entrada);// recibo del ciente
                if (entrada.trim().equals("Adios")) done = true;
                System.out.println("==> cliente "+(revLine =
                                                server.reverse(entrada)));
                out.println("(Hilo "+hilo+") "+revLine);
                if (done){
                    System.out.println("Cierro conexion con cliente "+hilo);
                    incoming.close();//  Liberar el puerto !!!
                    break;
                }
            }   //  while
        }
        catch (Exception e){e.printStackTrace();}
    }    //  void run
}       //  class ThreadedEchoHandler
```



Un programa Chat.

A continuación presentamos otra aplicación, un programa de Chat. Usamos las clases Socket, ServerSocket, agregamos una interface gráfica con eventos, en fin, mejoramos bastante el nivel de lo que estuvimos haciendo hasta ahora. Toda esta aplicación es un desarrollo del alumno Rodrigo Nogues, quien la presenta

como trabajo de investigación en febrero de 2004. Comenzamos por su propia descripción de su trabajo.

Introducción

RS Chat es un pequeño programa de chateo tipo IRC que básicamente funciona mediante el uso de sockets. Este programa hace uso de la facilidad que da Java a los desarrolladores de software para la construcción de programas que requieran trabajar en red, y a su vez muestra la flexibilidad y el poder del paquete Swing para crear sofisticadas interfaces gráficas con las que el usuarios pueda interactuar intuitivamente. También nos da la pauta de la capacidad de Java para manejar la concurrencia.
RS Chat consta de tres clases fundamentales en archivos separados:

- **Servidor.class**

  Esta clase es la responsable de recibir los pedidos de conexión de los distintos clientes y derivarlos a una clase concurrente ServidorHilo que luego se encargará de manejarlos.

- **ServidorHilo.class**

  Esta clase se encarga de los clientes para dejar libre a la clase Servidor y que esta pueda seguir recibiendo peticiones. Esta clase es heredera de Thread por lo cual se comporta como un hilo, el cual es creado y arrancado desde Servidor.
  ServidorHilo se encarga de difundir los mensajes enviados entre los clientes, avisar a los clientes cuándo se une uno nuevo y cuándo uno abandona la conversación. También debe enviar la lista de clientes a los usuarios que se conectan y dirigir correctamente los mensajes privados.

- **Cliente.class**

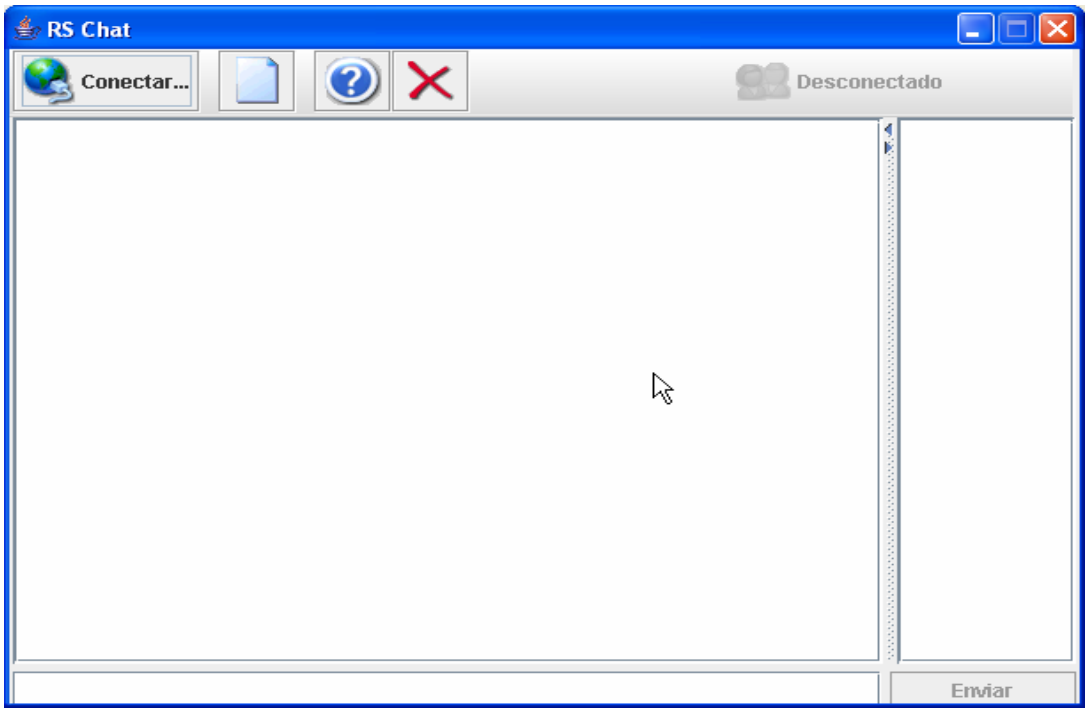
  Como su nombre lo indica esta clase es el cliente de chat, es decir, el programa con el que los usuarios interactuarán. Por este motivo esta clase desciende de JFrame transformándola así en una aplicación gráfica facilitando la tarea de los usuarios.
  La parte visible de la aplicación se mueve en una ventana gráfica creada con componentes Swing susceptibles a eventos.
  Por otro lado, internamente esta clase se encarga de conectarse y desconectarse del servidor, interpretar y reaccionar ante los comandos de éste, y recibir y enviar los mensajes del usuario y trasladar éstos al entorno gráfico.

Además cada una de las clases se encarga de su propio manejo de errores utilizando captura de excepciones lanzadas por Java. **Fin presentación Nogues** */

Comenzamos la descripción de esta aplicación capturando la ventana cliente, tal como aparece cuando la arrancamos desde el IDE. Para ello, cumplimentamos los siguientes pasos:

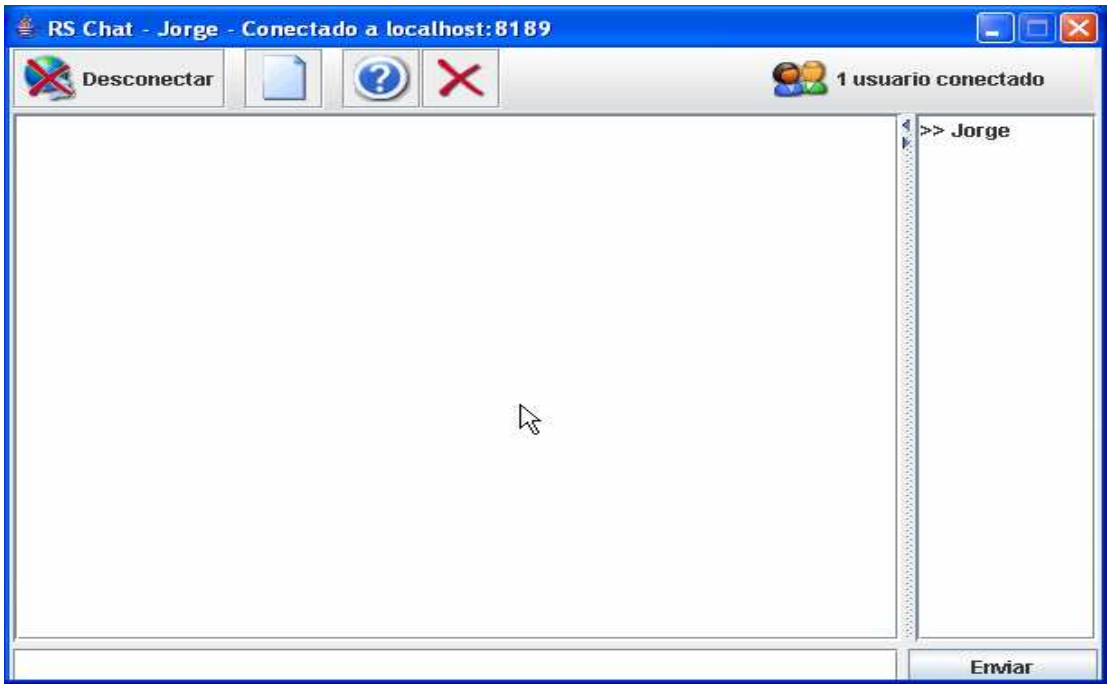
- Nos conectamos a Internet. Sea por Modem, ADSL, etc. Si lo que tenemos es una conexión domiciliar, por pulsos, podemos desconectar la ficha, (economizamos…). Lo que importa es que el soft de red quede cargado.
- Arrancamos servidor.java: desde el IDE, run, run File, run "Servidor.java"
- Arrancamos cliente.java: desde el IDE, run, run File, run "cliente.java".

Estaremos viendo la siguiente ventana:



Clickeamos el botón conectar, aparece el dialogo abajo, llenamos los datos pedidos, clickeamos en Conectar

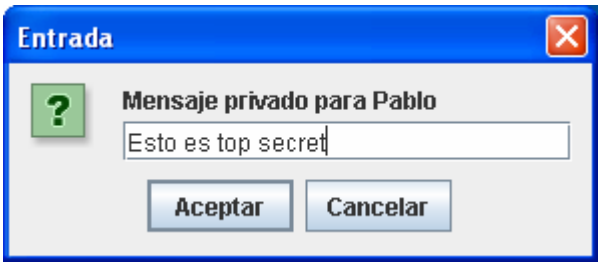
Y nuestra ventana cliente se ve ahora



La clase cliente.java, a continuación.

```
import java.net.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import java.awt.event.*;
import java.awt.Dimension;

public class Cliente extends JFrame implements ActionListener, Runnable{

// Componentes interfase usuario

    private JTextArea txtCharla;    // Area de texto con todo el chat
    private JScrollPane scrCharla;  // Panel para area texto con scrollBar
    private DefaultListModel lm;    // Clase que implementa lista sobre vector
                //  dinámico y notifica a los escuchas cambios en la lista
    private JList lstConectados;    // Lista de usuarios conectados
    private JScrollPane scrConect;  // Lista desplegable
    private JTextField txtEnvio;    // Texto a enviar
    private JButton btnEnviar;      // Boton para evento enviar
    private JSplitPane splitPane;   // Panel div. p/2 componentes, dimensionable
    private JToolBar tbBarra;       // barra de herramientas
    private JButton btnConectar;    // Boton pedir ventana conexion con servidor
    private JButton btnCerrar;      // Boton para cerrar conexión
    private JLabel lblUsuarios;     // Etiquetas de usuarios

    // Objetos
    private String mensaje; // Texto a enviar a todos los clientes
    private String nick;    // Apodo, alias del usuario cliente
    private Socket servidor = null; // Puerto de escucha en el servidor
    private DataInputStream entrada = null; // Entrada (servidor al cliente)
    private DataOutputStream salida = null; // Salida (cliente al servidor)
    private Thread receptor;    // Objeto receptor
    private boolean conectado = false;


    public Cliente(){   // Constructor
      btnConectar = new JButton("Conectar...",new ImageIcon("conectar.gif"));
      btnConectar.addActionListener(this);
      lblUsuarios = new JLabel("Desconectado", new
      ImageIcon("usuarios.gif"),JLabel.LEFT);
      lblUsuarios.setEnabled(false);

      // Barra de tareas

      tbBarra = new JToolBar();   // Instanciamos
      tbBarra.setBounds(0,0,600,45);  //  Dimensionamos
      tbBarra.setFloatable(false);    //  No podrá ser arrastrada, movida
      tbBarra.add(btnConectar);     // Agregamos boton conectar a la barra
      tbBarra.addSeparator();       // Separamos
      JButton botonTB = new JButton(new ImageIcon("limpiar.gif"));
      botonTB.setToolTipText("Limpiar área de conversación");// Microtexto
      botonTB.addActionListener(new ActionListener(){ // Respuesta al evento
        public void actionPerformed(ActionEvent e){txtCharla.setText("");}});
      tbBarra.add(botonTB);         // Lo incorporamos a la barra
      tbBarra.addSeparator();       //  Separacion
      botonTB = new JButton(new ImageIcon("acercaDe.gif"));
      botonTB.setToolTipText("Acerca de");
      botonTB.addActionListener(new ActionListener(){ //  Respuesta al evento
        public void actionPerformed(ActionEvent e){acercaDe();}});
      tbBarra.add(botonTB);         // Incorporamos a la barra
      btnCerrar = new JButton(new ImageIcon("cerrar.gif"));
      btnCerrar.setToolTipText("Salir");
```

```
btnCerrar.addActionListener(new ActionListener(){ //  Respuesta al evento
  public void actionPerformed(ActionEvent e){
    if(conectado){conectado=!desconectar();}
      dispose();        //  Metodo heredado de java.awt.Windows. Libera
      System.exit(0);   //  todos los recursos de este componente
  }
});
tbBarra.add(btnCerrar);
tbBarra.addSeparator(new Dimension(150,10));
tbBarra.add(lblUsuarios);   //  Etiqueta para mensajes a usuarios

//  Area de conversación

txtCharla = new JTextArea();     //  Instanciamos
txtCharla.setEditable(false);    //  No editable (Solo salida)
txtCharla.setLineWrap(true);     //  Desplazamiento vertical
txtCharla.setWrapStyleWord(true); // Envoltura usado p/lineas desplazables
txtCharla.setToolTipText("Conversación en curso");
scrCharla = new JScrollPane(txtCharla); // Incorporamos en contenedor
scrCharla.setMinimumSize(new Dimension(100,50)); // definimos tamaño mín.

//  Lista de usuarios (Conectados)

lm = new DefaultListModel();     // Definimos modelo de lista, con ella
lstConectados = new JList(lm);     // instanciamos lista de usuarios
lstConectados.setSelectionMode(0);  // Seleccion simple
lstConectados.setToolTipText("Usuarios conectados");

//  Mecanica de envio de mensajes privados entre los usuarios conectados
```




```
lstConectados.addListSelectionListener(new ListSelectionListener(){
  /*  Incorporamos y desarrollamos el escucha. El metodo valueChanged
   *  responde al evento cambio de selección en JList              */
  public void valueChanged(ListSelectionEvent e){
    String nickdest=lstConectados.getSelectedValue().toString();
    // A quien de los usuarios seleccionamos (para mensage privado)
    if(nickdest!=null && !nickdest.startsWith(">> ")){
      //  Si no me seleccione (a mi mismo), ventana para dialogo
      String msg = JOptionPane.showInputDialog("Mensaje privado para
                  "+nickdest);
      if(msg!=null && !msg.matches("")){ //   si algo escribo
        //  preparo el JTextField y lo envio a todos sus escuchas
        txtEnvio.setText("/p"+lstConectados.getSelectedValue()+"\t"+msg);
        txtEnvio.postActionEvent();
      }
    }
  }   //    valueChanged
});   //    addListSelectionListener

scrConect = new JScrollPane(lstConectados); // incorporamos al contenedor
scrConect.setMinimumSize(new Dimension(50,50));

/* Usaremos un objeto JSplitPane, (Panel dividido) que permite situar
 * dos componentes, el izquierdo el area de conversación y el derecho
```

```
     * la lista de usuarios conectados. Dentro de el, podemos redimensionar
     * el tamaño de los componentes dinamicamente                          */

    splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT); // Dist. horiz.
    splitPane.setLeftComponent(scrCharla);  //  Situamos comp. izquierdo
    splitPane.setRightComponent(scrConect); //  Idem derecho
    splitPane.setOneTouchExpandable(true);  //  Usuario puede redimensionar
    splitPane.setDividerLocation(490);      //  Posicion de la divisoria
    splitPane.setDividerSize(10);           //  Ancho de la divisoria
    splitPane.setBounds(2,45,600,360);      //  Dimensiones

    //  El campo de texto p/mensaje y boton enviar

    txtEnvio = new JTextField();
    txtEnvio.setBounds(2,410,490,25);
    txtEnvio.addActionListener(this);
    txtEnvio.setToolTipText("Escriba aquí su mensaje");
    btnEnviar = new JButton("Enviar");
    btnEnviar.setEnabled(false);
    btnEnviar.setBounds(497,410,105,25);
    btnEnviar.setToolTipText("Presione aquí para enviar su mensaje");
    btnEnviar.addActionListener(this);

    // Incorporamos todos los componentes a la ventana principal

    getContentPane().setLayout(null);
    getContentPane().add(splitPane);
    getContentPane().add(txtEnvio);
    getContentPane().add(btnEnviar);
    getContentPane().add(tbBarra);
    this.addWindowListener(new CierroVentana()); //  Escucha de la ventana
    txtEnvio.requestFocus();
  }   //  constructor (public Cliente)

  public void acercaDe(){
    JOptionPane.showMessageDialog(this, "RS Chat v1.0\n\nDesarrollado por:\n<<
       Rodrigo Nogués >>\npara el final de PPR.","Acerca de RS
       Chat",JOptionPane.INFORMATION_MESSAGE,new ImageIcon("usuarios.gif"));
  }

  /* A continuación definimos la respuesta a eventos a nivel ventana cliente
   * (JFrame), o sea todos aquellos que no tienen su escucha definida junto
   * al componente botonTB, btnCerrar, por ejemplo)                       */

  public void actionPerformed(ActionEvent evt){
    if(evt.getActionCommand().equals("Desconectar")){
       conectado=!desconectar();
       return;
    }

    if(conectado){
       String texto = txtEnvio.getText();
       texto.trim();
       if (texto.length()>0){
          try{
             salida.writeUTF(texto); //  Enviamos al servidor un flujo
                      // tipo string UTF-8 (Independiente de la maquina)
             salida.flush(); //  Forzamos el envio de los datos por el
                                //  flujo de salida
          }
          catch(IOException e){;}
          txtEnvio.setText("");   //  Limpiamos textField
          }
    }   //  if(conectado)
```

```
      if(evt.getActionCommand().equals("Conectar..."))
            new ConectarDlg(this);  //  Presenta ventana de dialogo para esta-
                                    //  blecer conexion con nuevo usuario
    }   //  public void actionPerformed

    public void run(){  // Arranca el cliente, objeto ya construido
      try{
         while(true){
            if(mensaje.startsWith("/")){    //  Inicia con '/', (Es de control)
               char tipo = mensaje.charAt(1); // obtengo tipo
               mensaje=mensaje.substring(2);   //  recorto 2 primeras pos.
               if(mensaje.matches(nick)){ //  si mensaje contiene apodo
                  mensaje=">> "+mensaje;  // antepongo ">> "
               }
               if(tipo=='c'){             // 'c' por conectar
                  lm.addElement(mensaje);//   Incorporo usuario a la lista
               }
               if(tipo=='d'){      // 'd' por desconectar
                  lm.remove(lm.lastIndexOf(mensaje));//   lo excluyo
               }
               if(lm.getSize()==1)
                  lblUsuarios.setText("1 usuario conectado");
               else
                  lblUsuarios.setText(lm.getSize() + " usuarios conectados");
               }
            else {   //  mensaje no de control (charla)
               txtCharla.append(mensaje + "\n");
            }
            mensaje = new String(entrada.readUTF());// leemos siguiente
         }   //  while(true)
      }
      catch (SocketException e){
            conectado=!desconectar();}
      catch (IOException e) {;}
    }   //  public void run()

    public boolean conectar(String ipserv, int puerto, String nick){
      try{
         lblUsuarios.setText("Conectando...");
         servidor = new Socket(ipserv, puerto);
         entrada = new DataInputStream(new
            BufferedInputStream(servidor.getInputStream()));
         salida = new DataOutputStream(new
            BufferedOutputStream(servidor.getOutputStream()));
         lblUsuarios.setText("Registrandose...");
         do{
            nick.trim();
            salida.writeUTF(nick);
            salida.flush();
            mensaje = new String(entrada.readUTF());
            if(mensaje.matches("/n")){
               nick=JOptionPane.showInputDialog(this,
               "El apodo (nickname) elegido esta siendo usado por otro
                   participante.\nElija otro por favor:","Apodo usado",
                  JOptionPane.INFORMATION_MESSAGE);
            }
         }while(mensaje.matches("/n"));
      }   //  try
      catch(UnknownHostException e){
            JOptionPane.showMessageDialog(this, "No se pudo encontrar el
servidor " + ipserv + ":" + puerto, "Error de conexión",
JOptionPane.ERROR_MESSAGE);
            desconectar();
```

```
            return false;
      }
      catch(IOException e){
            JOptionPane.showMessageDialog(this, "No se pudo obtener E/S de la
                  conexión", "Error de E/S", JOptionPane.ERROR_MESSAGE);
            desconectar();
            return false;
      }
      this.nick=nick;
      lblUsuarios.setText("Conectado.");
      receptor = new Thread(this);    //  Le abrimos un ServidorHilo
      receptor.start();               //  Lo arrancamos
      this.setTitle("RS Chat - " +nick+" - Conectado a "+ipserv+":"+puerto);
      btnConectar.setIcon(new ImageIcon("desconectar.gif"));
      btnConectar.setLabel("Desconectar");
      lblUsuarios.setEnabled(true);
      btnEnviar.setEnabled(true);
      return true;
    }   //  public boolean conectar


/* La clase interna ConectarDlg (Dialogo de conexion) que vemos a continuacion
 * se ocupa de pedir los datos(localhost, puerto, apodo) de nuevo usuario que
 * va a usar la ventana cliente que acabamos de arrancar. Esa ventana la
 * podemos arrancar desde el IDE. Por ejemplo, si estamos usando NetBeans IDE,
 * clickear Run, Run file, Run "Cliente.java" y ya en la ventana, <"Conectar">.
 * Esta clase tiene tambien comportamiento para desconectar y cerrar ventana */

class ConectarDlg extends JDialog implements ActionListener{
    private JLabel lblServ;
    private JLabel lblPto;
    private JLabel lblNick;
    private JTextField txtServ;
    private JTextField txtPto;
    private JTextField txtNick;
    private JButton btnConec;
    private JButton btnCancelar;
    private JDialog dlgConectar;

    public ConectarDlg(JFrame origen){
      super(origen,"Conectar", true);
      lblServ = new JLabel("Servidor:");
      lblPto = new JLabel("Puerto:");
      lblNick = new JLabel("Apodo (nickname):");
      txtServ = new JTextField();
      txtPto = new JTextField();
      txtNick = new JTextField();
      btnConec = new JButton("Conectar");
      btnCancelar = new JButton("Cancelar");
      lblServ.setBounds(5,5,120,25);
      lblPto.setBounds(5,35,120,25);
      lblNick.setBounds(5,65,120,25);
      txtServ.setBounds(130,5,120,25);
      txtPto.setBounds(130,35,120,25);
      txtNick.setBounds(130,65,120,25);
      btnConec.setBounds(15,95,100,25);
      btnCancelar.setBounds(15,95,100,25);
      btnConec.addActionListener(this);
      btnCancelar.addActionListener(this);
      txtServ.addActionListener(this);
      txtPto.addActionListener(this);
      txtNick.addActionListener(this);
      setSize(260,150);
      setResizable(false);
```

```
    setLocationRelativeTo(origen);
    getContentPane().setLayout(null);
    getContentPane().add(lblServ);
    getContentPane().add(lblPto);
    getContentPane().add(lblNick);
    getContentPane().add(txtServ);
    getContentPane().add(txtPto);
    getContentPane().add(txtNick);
    getContentPane().add(btnConec);
    getContentPane().add(btnCancelar);
    show();
  }   //  public ConectarDlg

  public void actionPerformed(ActionEvent evt){
    if(evt.getActionCommand().equals("Conectar")){
      if(txtServ.getText().length()==0 || txtPto.getText().length()==0 ||
          txtNick.getText().length()==0){
             JOptionPane.showMessageDialog(this,"Debe ingresar todos los
                      datos.");
      }
      else{
         int puerto;
         try{
            puerto = Integer.parseInt(txtPto.getText());
         }
         catch(NumberFormatException e){
            JOptionPane.showMessageDialog(this, "El puerto debe ser un número
                                                  entero.");
            return;
         }
         conectado=conectar(txtServ.getText(),puerto,txtNick.getText());
         dispose();
      }   //  if(txtServ.getText
    }
    else if(evt.getActionCommand().equals("Cancelar"))dispose();
         else{
            if(txtServ.isFocusOwner())        txtPto.requestFocus();
            else if(txtPto.isFocusOwner())    txtNick.requestFocus();
                 else                         btnConec.requestFocus();
         }
    }   //  if(evt.getActionCommand()
  }   //  public void actionPerformed

  public boolean desconectar(){
    try{
       servidor.close();
       entrada=null;
       salida=null;
       servidor=null;
       receptor=null;
       lm.clear();
       this.setTitle("RS Chat");
       btnConectar.setIcon(new ImageIcon("conectar.gif"));
       btnConectar.setLabel("Conectar...");
       lblUsuarios.setEnabled(false);
       lblUsuarios.setText("Desconectado");
       btnEnviar.setEnabled(false);
       return true;
    }
    catch (Exception ex){return false;}
  }   //  public boolean desconectar()

  class CierroVentana extends WindowAdapter{
    public void windowClosing(WindowEvent e){
```

```
            btnCerrar.doClick();
        }
    }   //   class CierroVentana

    public static void main(String []args){
      Cliente ventana = new Cliente();
      ventana.setSize(610,463);
      ventana.setResizable(false);
      ventana.setTitle("RS Chat");
      ventana.setVisible(true);
    }
}   // public class Cliente
```

Ahora, la clase servidor. Es muy simple, por suerte …

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

public class Servidor{
    public static void main (String[] args) throws IOException{
      ServerSocket servidor=null;
      boolean escuchando = true;
      try{
            servidor = new ServerSocket(8189);
      }
      catch (IOException e){
            System.out.println("No se puede escuchar en el puerto " + 8189);
            System.exit(1);
      }
      while (escuchando){
            Socket cliente=null;
            try{
                cliente = servidor.accept();
            }
            catch (IOException e){
            System.out.println("Fall¢ la conexi¢n: " + args[0] + ", " +
                  e.getMessage());
            continue;
            }
            System.out.println("Se conect¢ " + cliente.getInetAddress());
            new ServidorHilo(cliente).start();
      }
      try{
            servidor.close();
            }
            catch (IOException e){
            System.err.println("No se pudo cerrar el servidor. " +
                  e.getMessage());
      }
    }   //  public static void main
}   //  public class Servidor
```

La clase servidor starta ServidorHilo, tambien bastante sencilla.

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

public class ServidorHilo extends Thread{
```

```
private Socket cliente = null;
private DataInputStream entrada = null;
private DataOutputStream salida = null;
private static Vector conectados = new Vector();
private String nick;

//  El constructor establece los flujos de entrada/salida con el cliente

public ServidorHilo(Socket cliente) throws IOException{
  super(new String(cliente.getInetAddress().toString()));
  this.cliente=cliente;
  entrada = new DataInputStream(new
                   BufferedInputStream(cliente.getInputStream()));
  salida = new DataOutputStream(new
                   BufferedOutputStream(cliente.getOutputStream()));
}

public void run(){
  try{
     boolean n=true;
     do{
        nick = new String(entrada.readUTF()); // Quien se quiere conectar?
        ServidorHilo ni = buscarnick(nick); // Veamos si el apodo existe

        /* buscarnick retorna el objeto ServidorHilo en el cual ya tenemos
         * al apodo (nick) ya conectado o null caso contrario           */

        if(ni!=null) transmitir("/n",this); // ya tenemos alguien así...

        /* Ya tenemos alguien así apodado, entonces los parámetros a
         * transmitir son el mensaje "/n" y un objeto ServidorHilo null.
         * Con ello transmitir activa comportamiento del cliente que
         * hace la correspondiente advertencia...                   */

        else n=false;
        }while(n);  //  Salimos de este bucle si el apodo (nick) es nuevo
        transmitir(" >> " + nick + " se ha unido al chat.",null);
        transmitir("/c" + nick, null);  // Indicamos la nueva conexion
        conectados.addElement(this);    // Incorporamos en la lista
        transmitir("", this);
        while(true){    //  Ciclo de atencion al cliente
        String ent = new String(entrada.readUTF()); // Que nos envia ?
        if(ent.startsWith("/p")){   //  Control, (msge privado)

          /* Buscamos el destinatario del mensaje, está inmediatamente
          * despues de /p, o sea la posición 2, termina con \t   */

           String nickdest=ent.substring(2,ent.indexOf("\t"));

           /* El mensaje está a continuación del destinatario, o sea
           * posicion de "\t" + 1, hasta el fin de la cadena      */

           String msg=ent.substring(ent.indexOf("\t")+1);
           // Busquemos al destinatario de ese mensaje privado
           ServidorHilo dest=buscarnick(nickdest);
           if(dest==null)  // O sorpresa, no existe (Esto no ocurre)
              transmitir(">> No se pudo entregar el mensaje privado a
                           "+nickdest,this);
           else{           //  Lo tenemos, comuniquemonos con el
              transmitir("- Privado de "+ nick + ": " + msg, dest);
              transmitir("- Privado para "+ nickdest + ": " + msg, this);
           }
           ent="";
        }     //    Fin del tratamiento mensaje privado
```

```
            else    // No es un mensaje privado, nos comunicaremos con todos
                    transmitir(nick + ": "+ ent,null);
            }   // while(true)
      }       // try
      catch (IOException e) {;}
      finally{
            conectados.removeElement(this);
            transmitir("/d" + nick,null);
            transmitir(" >> " + nick + " ha dejado el chat.",null);
            System.out.println("Se desconect¢ "+ cliente.getInetAddress());
            try{
            entrada.close();
            salida.close();
            cliente.close();
            }
            catch (IOException e) {;}
      }
    } // public void run

    public ServidorHilo buscarnick(String n){

     /* A continuación vemos un ejemplo de uso de comportamiento de la clase
      * Enumeration. Ya la hemos usado en la U I, la conocemos. Aquí la esta-
      * mos usando para recorrer la lista de usuarios ya conectados al chat */

      Enumeration con = conectados.elements();
      while(con.hasMoreElements()){
          ServidorHilo cl = (ServidorHilo) con.nextElement();
          if(n.matches(cl.nick)){return cl;}  //  Ya tenemos ese alias (nick)
      }
      return null;    //  El apodo, alias (nick) es nuevo
    }
    private static synchronized void transmitir(String mensaje,
      ServidorHilo sh){
      Enumeration conect = conectados.elements();
        while(conect.hasMoreElements()){
            ServidorHilo ctes = (ServidorHilo) conect.nextElement();
            if(sh==null){   // este alias ya existía
                try{
                    ctes.salida.writeUTF(mensaje);
                    ctes.salida.flush();
                }
                catch (IOException e){ctes.stop();}
            }
            else{
                try{
                    if(mensaje==""){
                        sh.salida.writeUTF("/c"+ ctes.nick);
                        sh.salida.flush();
                    }
                    else{
                        sh.salida.writeUTF(mensaje);
                        sh.salida.flush();
                        break;
                    }
                }   // try
                catch(IOException e){;}
            }   // if((sh==null)
      }   // while(conect.hasMoreElements())
    }   // private static synchronized void transmitir
}   // public class ServidorHilo
```

A continuación, cerrando este ejemplo, vamos a mantener un mini chat con tres usuarios. La secuencia de pasos que ejecutamos, con el proyecto ya abierto en el

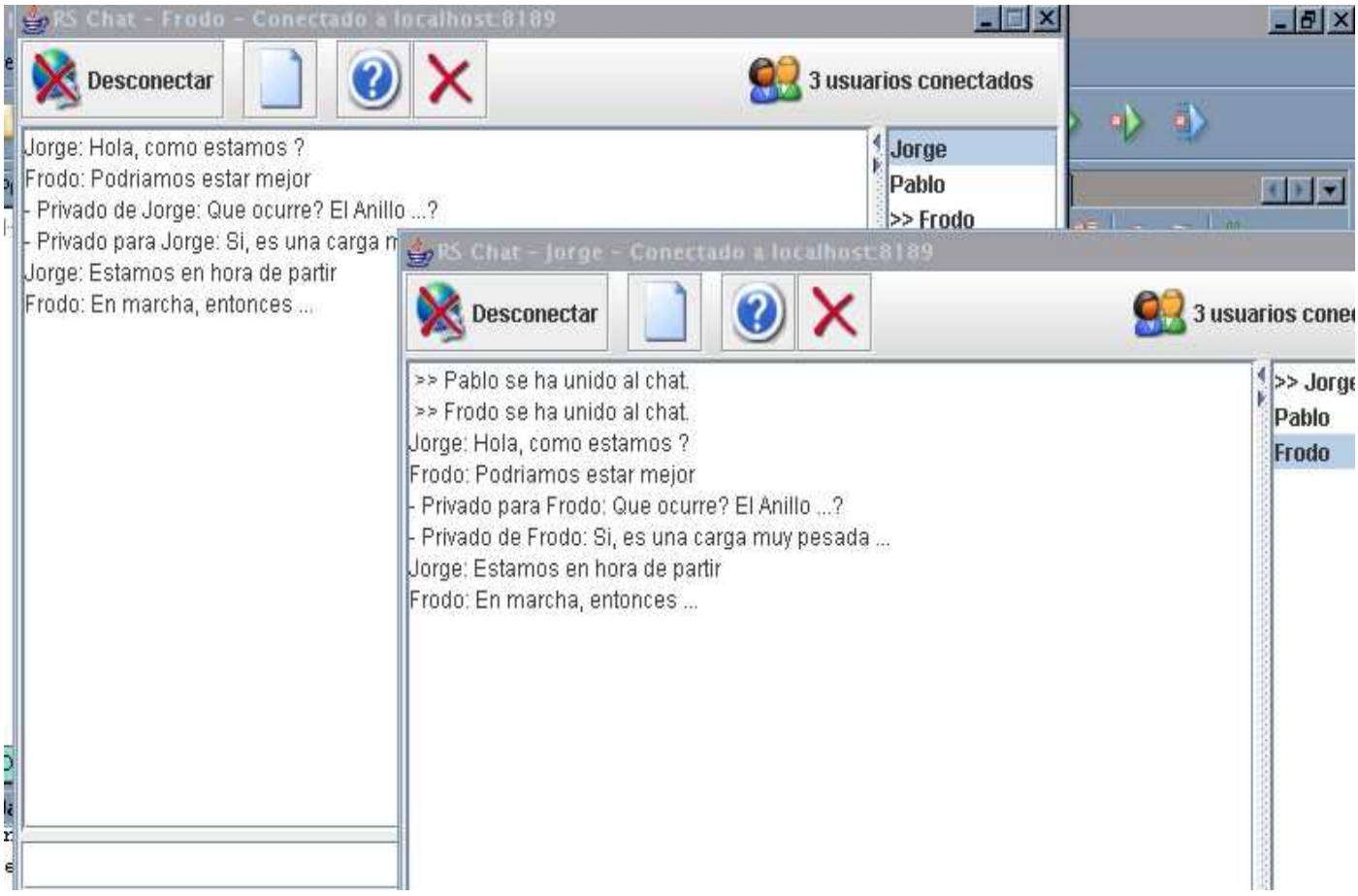
NetBeans IDE y con el soft de red previamente cargado (Fullzero, Tutopia,etc), es la siguiente.
1. – Abrimos los tres fuentes (cliente.java, Servidor.java, ServidorHilo.java
2. – Editamos el fuente de Servidor.java y <run>, <run file>, run Servidor.java"

3. – Editamos el fuente de Cliente.java y <run>, <run file>, run Cliente.java"
4. – <Conectar>,localhost,8189,Jorge, <Conectar>

5. – <run>, <run file>, run Cliente.java"
6. – <Conectar>,localhost,8189,Pablo, <Conectar>

7. – <run>, <run file>, run Cliente.java"
8. – <Conectar>,localhost,8189,Frodo, <Conectar>

9. – nos vamos posicionando en las distintas ventanas cliente y enviamos mensajes. Sigue un capture de la pantalla cliente del nick Jorge.



Envío de correo electrónico

En esta sección vamos a ver un ejemplo práctico de programación socket: un programa que envía correos electrónicos a un sitio remoto.
Para ello, establecemos una conexión socket al puerto 25, que es el puerto SMTP. SMTP es la abreviatura de Protocolo sencillo de transporte de correo, el cual describe el formato de los mensajes de correo electrónico. Puede conectar con cualquier servidor que ejecute un servicio SMTP. En máquinas UNIX, este servicio suele estar implementado por sendmail. Sin embargo, el servidor debe estar dispuesto para aceptar su petición. Solía ser frecuente que los servidores sendmail enrutaran gustosamente los correos electrónicos de cualquiera, pero en estos tiempos en los que los spams fluyen a diario por la Red, muchos servidores

están configurados para aceptar sólo las peticiones de los usuarios, dominios o rangos de direcciones IP en los que confían.

Una vez establecida la conexión con el servidor, envíe una cabecera de correo electrónico (en formato SMTP, que es fácil de generar), seguida del cuerpo del mensaje. Siguen los detalles:

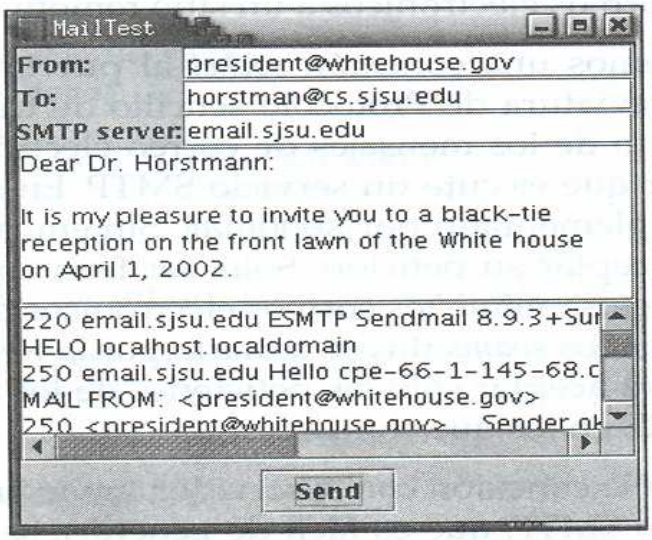
1. Abra un socket en su servidor.

```
Socket s = new Socket("mail.yourserver.com", 25)¡ // 25 es el "puerto SMTP
PrintWriter out = new PrintWriter(s.getOutputStream())¡
```

2. Envíe la siguiente información al flujo de impresión:

```
HELO host de envío
MAIL FROM: <dirección email del emisor>
RCPT TO: <dirección email del receptor>
DATA mensaje del correo
      (cualquier número de líneas)
.
QUIT
```

La especificación SMTP (RFC 821) detalla que las líneas deben estar finalizadas con un carácter \r seguido de un \n.
Ojo! (Aunque esto es obvio …)
Muchos servidores SMTP no comprueban la veracidad de la información: debe estar dispuesto a responder a cualquiera que lo solicite (la próxima vez que reciba un mensaje de correo electrónico procedente de president@whitehouse.gov invitándole a una fiesta de etiqueta en el jardín delantero, desconfie…
Si hacemos las cosas bien, al ejecutar deberiamos ver algo así:



El programa sólo envía la secuencia de comandos que ya hemos comentado antes. Muestra esos comandos que ha enviado al servidor SMTP y las respuestas que recibe. Observe que la comunicación con el servidor de correo se produce en un thread aparte para que el thread de la interfaz de usuario no se bloquee mientras intenta conectar con dicho servidor.

Y el programa es:

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import javax.swing.*;
```

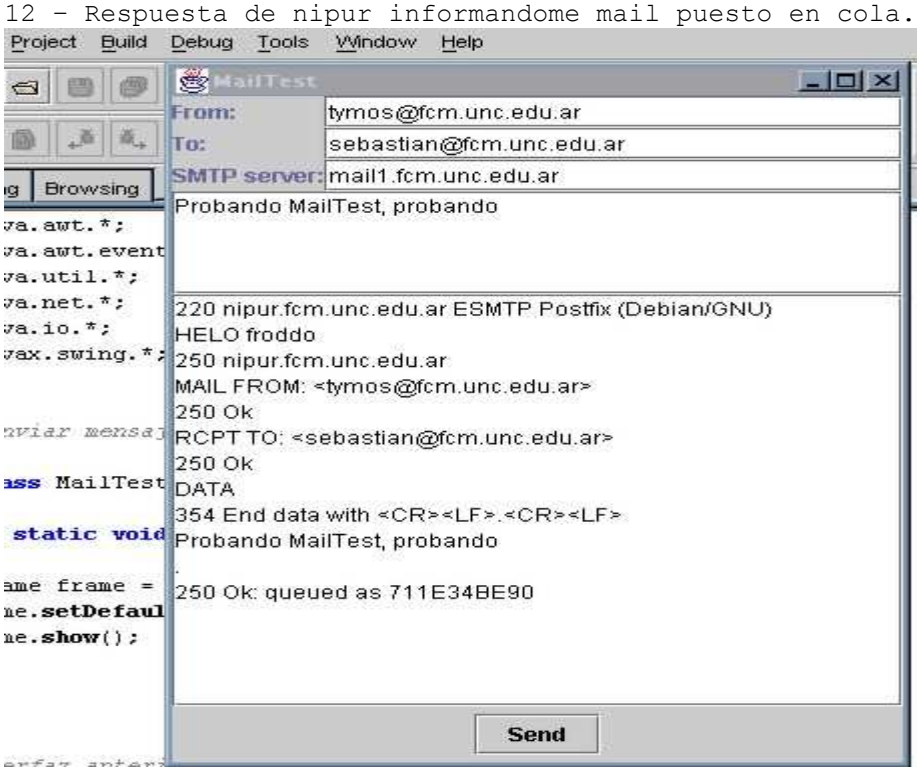
```
public class MailTest{  //   enviar mensajes de correo usando sockets
   public static void main(String[] args){
      JFrame frame = new MailTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

class MailTestFrame extends JFrame{  // interfaz ya vista
   public MailTestFrame(){   //    constructor
      setSize(WIDTH, HEIGHT);
      setTitle("MailTest");

      getContentPane().setLayout(new GridBagLayout());

      GridBagConstraints gbc = new GridBagConstraints();
      gbc.fill = GridBagConstraints.HORIZONTAL;
      gbc.weightx = 0;
      gbc.weighty = 0;

      gbc.weightx = 0;
      add(new JLabel("From:"), gbc, 0, 0, 1, 1);
      gbc.weightx = 100;
      from = new JTextField(20);
      add(from, gbc, 1, 0, 1, 1);

      gbc.weightx = 0;
      add(new JLabel("To:"), gbc, 0, 1, 1, 1);
      gbc.weightx = 100;
      to = new JTextField(20);
      add(to, gbc, 1, 1, 1, 1);

      gbc.weightx = 0;
      add(new JLabel("SMTP server:"), gbc, 0, 2, 1, 1);
      gbc.weightx = 100;
      smtpServer = new JTextField(20);
      add(smtpServer, gbc, 1, 2, 1, 1);

      gbc.fill = GridBagConstraints.BOTH;
      gbc.weighty = 100;
      message = new JTextArea();
      add(new JScrollPane(message), gbc, 0, 3, 2, 1);

      communication = new JTextArea();
      add(new JScrollPane(communication), gbc, 0, 4, 2, 1);

      gbc.weighty = 0;
      JButton sendButton = new JButton("Send");
      sendButton.addActionListener(new
            ActionListener(){
                  public void actionPerformed(ActionEvent evt){
                        new Thread(){
                              public void run(){sendMail();}}.start();
                  }     //    actionPerformed
            }     //    ActionListener()
      );    //    new ActionListener(){

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(sendButton);
      add(buttonPanel, gbc, 0, 5, 2, 1);
   }  //    constructor public MailTestFrame

   /**
      void add añade componentes a este marco.
```

```
      @parametro c es el componente a añadir
      @parametro gbc la GridBagConstraints
      @parametro x es la columna en la rejilla (frame)
      @parametro y es la fila idem
      @parametro w es la cantidad de columnas de la rejilla
      @parametro h idem, filas
    */
   private void add(Component c, GridBagConstraints gbc,
      int x, int y, int w, int h)
   {
      gbc.gridx = x;
      gbc.gridy = y;
      gbc.gridwidth = w;
      gbc.gridheight = h;
      getContentPane().add(c, gbc);
   }

   public void sendMail(){// Envia texto introducido en la interfaz
      try{
         Socket s = new Socket(smtpServer.getText(), 25);

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


         String hostName
            = InetAddress.getLocalHost().getHostName();

         receive();
         send("HELO " + hostName);
         receive();
         send("MAIL FROM: <" + from.getText() + ">");
         receive();
         send("RCPT TO: <" + to.getText() + ">");
         receive();
         send("DATA");
         receive();
         StringTokenizer tokenizer = new StringTokenizer(
            message.getText(), "\n");
         while (tokenizer.hasMoreTokens())
            send(tokenizer.nextToken());
         send(".");
         receive();
         s.close();
      }
      catch (IOException exception)
      {
         communication.append("Error: " + exception);
      }
   }

   /**
      Envia una cadena al socket y la repite
      en el area de texto de la comunicacion.
      @parametro s es la cadena a enviar.
   */

   public void send(String s) throws IOException{
      communication.append(s);
      communication.append("\n");
      out.print(s);
      out.print("\r\n");
      out.flush();
```

```
    }

    // recibe una cadena desde el socket y la muestra en la JTextArea
    public void receive() throws IOException{
       String line = in.readLine();
       if (line != null){
          communication.append(line);
          communication.append("\n");
       }
    }

    private BufferedReader in;
    private PrintWriter out;
    private JTextField from;
    private JTextField to;
    private JTextField smtpServer;
    private JTextArea message;
    private JTextArea communication;

    public static final int WIDTH = 300;
    public static final int HEIGHT = 300;
}
```

Un capture de una ejecución de MailTest.java en el "mundo real", en la página siguiente. En el area de texto inferior podemos seguir que ocurrió al clickear el botton Send. Detallamos esto por línea.

1 – Mi PC se conectó al puerto 25 de nipur (Nuestro servidor de correo) del cual mail1 es un alias.
2 – Mi PC saluda "Hello", ella es FRODO, (el heroe de El Señor de los Anillos)
3 – El acknowledgement (reconocimiento, acuse de recibo) del Servidor
4 – Mi PC informa envio de correo
5 – Servidor dice que lo recibe OK
6 – Se indica al destinatario del mail
7 – Destinatario dice ok
8 – Siguen datos
9 – Especificaciones técnicas (HTML)
10 – El texto de mi mail
11 – . (punto) generado por mi PC, informando fin del texto del mail.
12 – Respuesta de nipur informandome mail puesto en cola.



En esta Unidad sólo cubrimos el protocolo de red TCP (Transmission Control Protocol, Protocolo para el control de la transmisión), el cual establece una conexión fiable entre dos computadoras. Java también soporta el protocolo UDP (User Datagram Protocol, Protocolo de datagrama de usuario), que puede emplearse para enviar paquetes de datos (también llamados datagramas) con más carga de la necesaria para TCP. El inconveniente es que dichos paquetes pueden ser entregados de forma aleatoria, o, incluso, completamente descartados. Es obligación del receptor ponerlos en el orden correcto y solicitar la retransmisión de aquellos que no hayan llegado. UDP está mejor adaptado para las aplicaciones en las que se pueden tolerar la desaparición de paquetes, como, por ejemplo, en flujos de audio o vídeo, o en mediciones continuas.

Conexiones URL

Acabamos de ver cómo usar la programación a nivel de socket para conectar con un servidor SMTP y enviar un mensaje de correo electrónico. Sin embargo, si está planificando una aplicación que incorpore este servicio, lo más seguro es que prefiera trabajar a un nivel superior y usar una librería que encapsule los detalles del protocolo. Por ejemplo, Sun Microsystems ha desarrollado la API JavaMail como una extensión estándar de la plataforma Java. En dicha API, basta con efectuar una llamada como la siguiente: *Transport.send(mensaje);* para enviar un mensaje. La librería es la que se encarga de preocuparse por dicho mensaje, por la recepción múltiple, la manipulación de los adjuntos (attachments), etcétera.

Las clases URL y URLConnection encapsulan una gran parte de la complejidad asociada a la recuperación de datos desde un servidor remoto. Aquí tiene la forma de especificar un URL. Esta clase posee comportamiento para detectar sus componentes. Veamos algunos ejemplos.

```
import java.net.*;
import java.io.*;
public class ParseURL{
    public static void main(String[] args){
        String urlAux;
        try{
            System.out.println("Que URL desea Ud investigar?");
            urlAux = In.readLine();
            System.out.println("Investigamos "+urlAux);
            URL aURL = new URL(urlAux);
            System.out.println("protocol = " + aURL.getProtocol());
            System.out.println("host = " + aURL.getHost());
            System.out.println("filename = " + aURL.getFile());
            System.out.println("port = " + aURL.getPort());
            System.out.println("ref = " + aURL.getRef());
            System.out.println("--------------");
        }
        catch (IOException exception){exception.printStackTrace();}
    }
}

/* Un ejemplo a investigar
"http://java.sun.com:80/docs/books/"
             + "tutorial/index.html#DOWNLOADING"      */

Que URL desea Ud investigar?
Investigamos http://java.sun.com:80/docs/books/tutorial/index.html#DOWNLOADING·
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING·
--------------
```

```
// otro ejemplo
http://docs.mycompany.com/api/java/lang/String.html

Que URL desea Ud investigar?
Investigamos http://docs.mycompany.com/api/java/lang/String.html
protocol = http
host = docs.mycompany.com
filename = /api/java/lang/String.html
port = -1
ref = null
--------------
```

Uso de URLConnection para recuperar información

Si desea información adicional acerca de un recurso web, puede utilizar la clase URLConnection, siguiendo los siguientes pasos:

```
URL url = new URL(urlName);   //    Obtengo objeto URL

URLConnection connection = url.openConnection{}; // objeto URLConnection

// Establezca cualquier propiedad de la petición mediante los métodos
setDolnput,setDoOutput,setIfModifiedSince,setUseCaches, setAllowUserInteraction,
setRequestProperty. (UD. puede consultar su utilidad en el Help de su IDE)

connection.connect();   //    Conecte con el recurso remoto
```

Además de facilitar una conexión socket con el servidor, este método también le pregunta a éste la información de cabecera.

Hay dos métodos, **getHeaderFieldKey y getHeaderField**, que permiten obtener los nombres y valores de todos los campos de la cabecera.

Por último, puede acceder a los datos del recurso. Usaremos el método **getInputStream** para obtener un flujo de entrada para la lectura de la información (este flujo es el mismo que devuelve el método openStream de la clase URL). Vamos con un programa ejemplo

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

/**
 *   Este programa conecta con el URL solicitado por el operador, o con
 *  java.sun.com si no se informa nada.
 *  Utiliza los metodos vistos para exhibir datos de la cabecera,
 *  y las diez primeras lineas de los datos requeridos.
*/
public class URLConnectionTest{
   public static void main(String[] args){
      String urlName  = null;
      try{
         System.out.println("Que URL desea Ud investigar?");
         urlName = In.readLine();
         if (urlName == null)urlName  = "http://www.java.sun.com";
         System.out.println("Investigaremos ==> "+urlName);
         URL url = new URL(urlName);
         URLConnection connection = url.openConnection();

         int n = 1;
         String key;

         System.out.println("Van campos de cabecera");
         while ((key = connection.getHeaderFieldKey(n)) != null){
```

```
            String value = connection.getHeaderField(n);
            System.out.println(key + ": " + value);
            n++;
         }
         System.out.println("Fin campos de cabecera");
         System.out.println("");

         System.out.println("Van 10 lineas del archivo de definicion del URL");
         BufferedReader in = new BufferedReader(new
            InputStreamReader(connection.getInputStream()));
         String line;
         n = 1;
         while ((line = in.readLine()) != null && n <= 10){
            System.out.println(line);
            n++;
         }
         if (line != null)
             System.out.println("Fin 10 lineas del archivo de definicion");
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }
}
```

Una primera ejecución

```
run:
Que URL desea Ud investigar?
Investigaremos ==> http://www.fcm.unc.edu.ar
Van campos de cabecera
Date: Sat, 10 Mar 2007 01:49:40 GMT
Server: Apache/2.0.55 (Ubuntu) PHP/4.4.2-1build1 mod_ssl/2.0.55 OpenSSL/0.9.8a
Last-Modified: Thu, 28 Dec 2006 15:12:46 GMT
ETag: "aab8-1a9d-425ab94631780"
Accept-Ranges: bytes
Content-Length: 6813
Keep-Alive: timeout=15, max=500
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1
Fin campos de cabecera

Van 10 lineas del archivo de definicion del URL
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Facultad de Ciencias Médicas</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#9FBDB1" link="#000000" vlink="#000000" alink="#0000CC">
<table width="817" height="1032" border="0">
  <tr>
Fin 10 lineas del archivo de definicion
```

Una segunda

```
run:
Que URL desea Ud investigar?
Investigaremos ==> http://www.frc.utn.edu.ar
Van campos de cabecera
Date: Sat, 10 Mar 2007 01:52:21 GMT
```

```
Server: Microsoft-IIS/6.0
pragma: no-cache
cache-control: private
Content-Length: 46042
Content-Type: text/html
Expires: Fri, 09 Mar 2007 01:52:20 GMT
Set-Cookie: ASPSESSIONIDCSBDTBAS=GONBIDABKAJJJFKLPNMOKPIM; path=/
Cache-control: no-cache
Fin campos de cabecera

Van 10 lineas del archivo de definicion del URL

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html><!-- InstanceBegin template="/Templates/Principal.dwt.asp"
codeOutsideHTMLIsLocked="false" -->
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<meta name="keywords" content="UTN-FRC, UTN, ARGENTINA, FRC, Universidad
Tecnológica Nacional, Universidad Tecnológica Nacional - Facultad Regional
Córdoba, Facultad Regional Córdoba, U.T.N., F.R.C, UTN - Facultad Regional
Córdoba, U.T.N. - Facultad Regional Córdoba, ARGENTINE, CIUDAD DE CORDOBA,
CORDOBA, CORDOBA, CIUDAD DE CORDOBA" />
<meta name="description" content="UTN FRC, casa de altos estudios de la ciudad
de Córdoba. P&aacute;gina Web de la Universidad Tecnológica Nacional - Facultad
Regional Córdoba" />
<meta name="verify-v1" content="+iYX1Z0WEddeqeNwGIgtu68PXg0qXJcPRedckxgD6Xk=" />
<!-- InstanceBeginEditable name="doctitle" -->
<title>Universidad Tecnol&oacute;gica Nacional - Facultad Regional
C&oacute;rdoba</title>
Fin 10 lineas del archivo de definición
```

J2EE

La plataforma Java J2SE (Java 2 Standard Edition) permite construir aplicaciones Java robustas, pero no resuelve **eficientemente** las necesidades requeridas por el mundo empresarial en cuanto a las aplicaciones distribuidas.

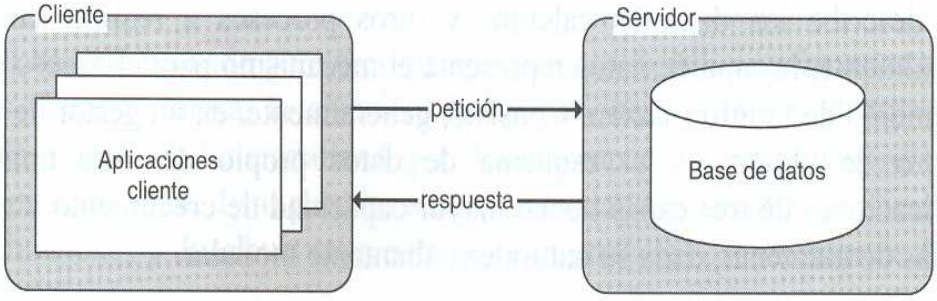
Sun Microsystems sintió la necesidad de revisar la tecnología Java.

El resultado fue **J2EE (Java 2 Enterprise Edition**), versión ampliada de la J2SE que incluye las API necesarias para construir aplicaciones para arquitecturas distribuidas multicapa, permitiendo la construcción de sistemas transaccionales basados en tecnologías Web, entendiendo por capa un grupo de tecnologías que proporcionan uno o más servicios.

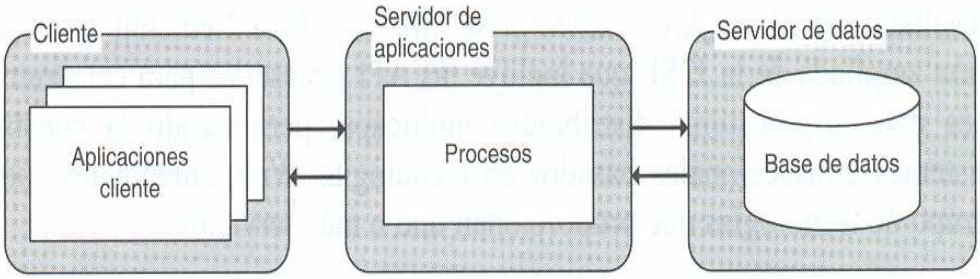
INTRODUCCIÓN

Tradicionalmente los sistemas corporativos se han venido diseñando utilizando el **modelo cliente servidor**, entendiendo por cliente una aplicación que inicia el diálogo con otra denominada servidor para solicitarle servicios que éste puede atender. (Es lo que hemos estado viendo en esta Unidad, hasta este momento)

Una aplicación cliente servidor tiene tres componentes fundamentales: presentación (interfaz de usuario), lógica de negocio y gestión de datos. Cada una de estas componentes puede estar en el cliente o en el servidor. Por ejemplo, se puede implementar la presentación en el cliente, la gestión de datos en el servidor, y distribuir la lógica de negocio entre el cliente y el servidor, o bien la presentación y la lógica del negocio en el cliente, y la gestión de datos en el servidor, etc. El resultado es una **arquitectura de dos capas de software.**



Actualmente, la lógica de negocio se ha convertido también en un servicio y puede residir en cualquier otro servidor, conocido como servidor de aplicaciones, dando lugar a una **arquitectura de tres capas**:



- **Cliente** Esta **capa de presentación** contiene todos los elementos que constituyen la interfaz con el usuario. Una interfaz gráfica basada en ventanas, un explorador, también llamado navegador, etc.
- **Servidor de aplicaciones** En la **capa de lógica de negocio** se modela el comportamiento del sistema, basándose en los datos provistos por la capa de datos, y actualizándolos según sea necesario. Esta capa describe los distintos cálculos y otros procesos a realizar con los datos.
- **Servidor de datos** Finalmente, la **capa de datos** representa el mecanismo para el acceso y el almacenamiento de la información. Consiste, generalmente, en un gestor de bases de datos o de objetos, y el esquema de datos propio de cada aplicación.

Las aplicaciones de tres capas tienen mayor capacidad de crecimiento y son más sencillas de mantener, dada su naturaleza altamente modular.

Normalmente, en una arquitectura cliente servidor multicapa, una petición de un cliente a un servidor genera peticiones a otros servidores conectados a través de una red. Un ejemplo:

- Una persona **(cliente)** que solicita a un
- agente de viajes que le organice **(aplicación)** un viaje de vacaciones.
- El agente, normalmente a través de mayoristas, contacta hoteles, líneas aéreas, restaurantes, alquiler de coches, **(datos)** …

La plataforma **J2EE** es el nuevo modelo diseñado para proporcionar un entorno de ejecución distribuido y multicapa (interfaz de usuario, servicios de aplicación y lógica de negocio, y servicios de gestión de datos).

¿Cómo se llega a J2EE?

Como se ha dicho anteriormente, Java es un lenguaje que se comienza a utilizar en el desarrollo de aplicaciones Web cuando Sun Microsystems liberó el JDK (Java Development Kit - paquete de desarrollo de Java). Esta utilización masiva, influenciada al principio por los applets (pequeños programas interactivos que son ejecutados por un navegador), hizo que los desarrolladores demandaran más

API, las cuales se iban añadiendo al JDK como extensiones al mismo. Tras este período de evolución de Java, Sun Microsystems decidió lanzar un nuevo paquete que incluyera JDK y sus extensiones, al que denominó **J2SDK**. (**Plataforma J2SE**)

Se produce la explosiva evolución de Internet, las empresas ven en las aplicaciones Web una forma económica y eficiente de ofrecer sus servicios, esto se traduce en mas programación del lado del servidor, nuevas API, independientes del proveedor, para conectar con sistemas servidores. Sun Microsystems crea el grupo de trabajo JCP (Java Community Process(SM), participan usuarios corporativos, proveedores, etc., desarrollando un estándar para las API empresariales demandadas. El resultado fue **J2EE**.

J2EE ha sido pensado para escribir **aplicaciones distribuidas** basadas en componentes. De esta forma, la creación de aplicaciones complejas distribuidas se simplifica, porque la funcionalidad que deben proporcionar se encapsula, modulariza en los componentes de J2EE. Esto permite a los desarrolladores dividir la aplicación según sus funciones y distribuir éstas entre los componentes del servidor de J2EE.

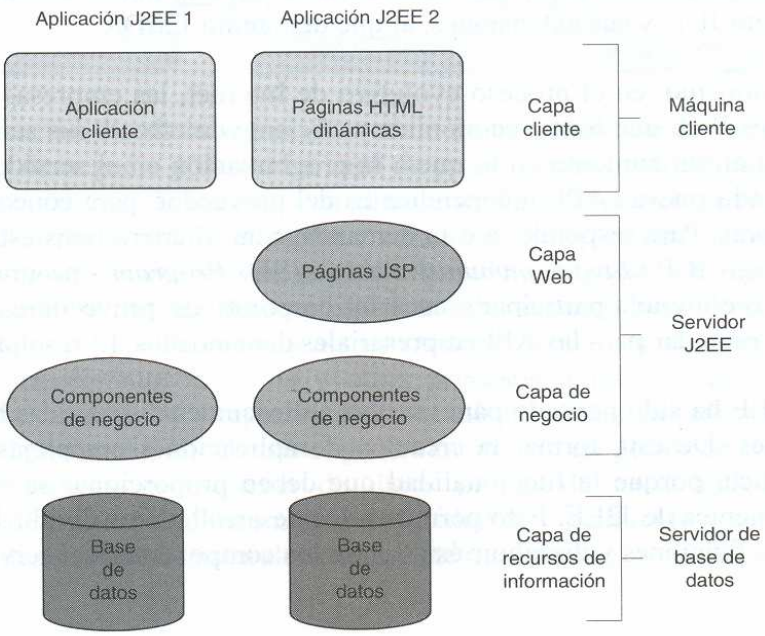
Qué es una aplicación empresarial? Brevemente:

- Utilizada concurrentemente por muchos usuarios.
- Utiliza recursos distribuidos (distintas bases de datos).
- Las tareas las realizan distintos objetos distribuidos.
- Utiliza tecnología J2EE para enlazar componentes distribuidos.

El software que maneja y soporta los diferentes componentes de un sistema distribuido se denomina **middleware**. Es el software del sistema que permite las interacciones a nivel de aplicación entre programas en un ambiente distribuido. Está localizado entre una aplicación y un sistema de menor nivel (sistema operativo, sistema gestor de bases de datos, servicios de red, etc.). En esencia, está localizado en el medio (middle) del sistema, entre otros componentes.

ARQUITECTURA J2EE MULTICAPA

La plataforma J2EE utiliza un modelo de aplicación distribuida multicapa para las aplicaciones empresariales. La lógica de negocio está dividida en componentes según su función, y los diversos componentes de aplicación que constituyen una aplicación J2EE normalmente son instalados en diferentes máquinas en función de la capa del entorno J2EE multicapa a la que pertenece el componente.



. **Capa cliente**. Ya la vimos en detalle. Podemos agregar, que en este modelo la mayoría de las peticiones van dirigidas a componentes de la capa Web.

· **Capa Web**. Los componentes de esta capa se ejecutan en el servidor J2EE y utilizan el protocolo HTTP para recibir las peticiones y enviar las respuestas a otros componentes. Esta capa proporciona a la aplicación J2EE funcionalidad de Internet, ya que permite proporcionar servicios a la capa cliente.

· **Capa de negocio**. Los componentes de esta capa se ejecutan en el servidor J2EE. Contienen la lógica de negocio (tareas específicas basadas en reglas que resuelven o cumplen las necesidades de un negocio particular) de las aplicaciones J2EE, con acceso concurrente, por múltiples clientes de esta capa. Por lo general, un componente de esta capa interactúa con la capa de recursos de información.

. **Capa de recursos de información** (middelware). Los componentes de esta capa se ejecutan en el servidor que administra estos recursos. Conecta la aplicación J2EE con los sistemas y recursos que se encuentren en la red corporativa (sistemas heredados, bases de datos, sistemas de terceros); la idea es que los desarrolladores de aplicaciones J2EE pueden aprovechar los recursos existentes con los que cuenta la empresa y no están obligados a replicarlos dentro de J2EE.

Concepto de componente en J2EE

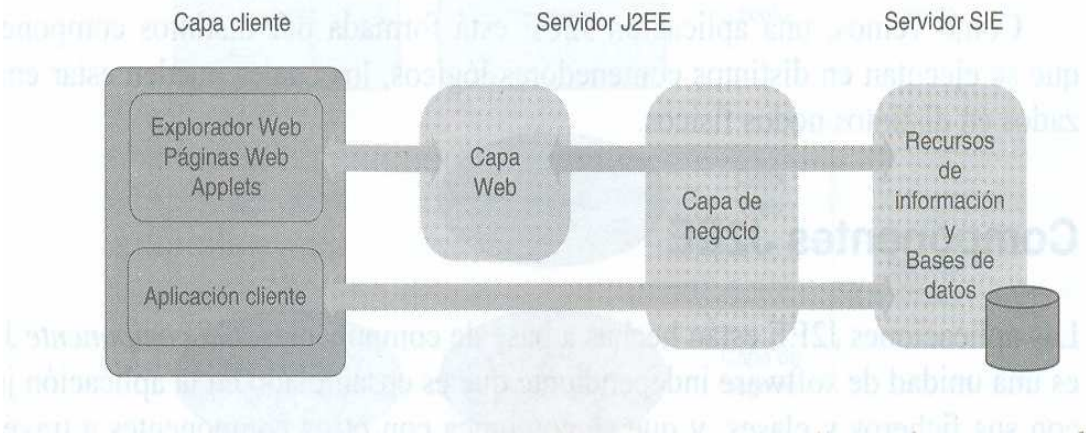
Las aplicaciones J2EE están armadas con componentes. Un componente J2EE es una unidad de software independiente que es ensamblado en la aplicación junto con sus ficheros y clases, y que se comunica con otros componentes a través de canales definidos. Estos componentes son, según La especificación J2EE:

- Aplicaciones cliente y applets. (ya los vimos).
- Servlets y JavaServer Pages (JSP). Son componentes Web que se ejecutan en un servidor J2EE, concretamente en el contenedor Web del mismo.
- Enterprise JavaBeans (EJB) o simplemente enterprise beans. Son componentes que se ejecutan en un servidor J2EE, concretamente en el contenedor EJB del mismo, y que contienen las reglas de negocio.

Un cliente J2EE normalmente suele ser un cliente Web o una aplicación cliente (aplicación de escritorio). Un cliente Web se compone de:

- páginas Web dinámicas generadas por componentes Web que se ejecutan en la capa Web,
- Un explorador Web que presenta las páginas recibidas desde el servidor. Una página Web recibida desde la capa Web puede incluir un applet; sin embargo, por seguridad, facilidad de desarrollo y mantenimiento, antes que los applets, son preferidos los componentes Web que se ejecutan en el servidor y que exponemos a continuación; estos proporcionan una forma clara de separar la programación de las aplicaciones, del diseño de la página Web. Una aplicación cliente normalmente se utiliza para mostrar una interfaz gráfica al usuario.

¿Cómo se comunica la capa cliente con la de negocio y con la capa de información empresarial? La capa de cliente se comunica con la de negocio directamente, o bien, en el caso de un cliente que se ejecute en un explorador, a través de la capa Web. La figura siguiente muestra cómo los componentes de la capa de negocio reciben datos de los componentes de la capa cliente, los procesan, si es necesario, y los envían a la base de datos. Los componentes de la capa de negocio también recuperan datos de la base de datos, los procesan, si es necesario, y los envían a los componentes de la capa cliente.



Los componentes de la capa Web (componentes Web) son los servlets y páginas JSP.

- Los servlets son clases escritas en Java que procesan peticiones y construyen respuestas dinámicamente; por lo general, un servlet obtiene datos como resultado de procesar la petición, que envía a una página JSP para que se los transmita al cliente.
- Las páginas JSP son documentos de texto que se ejecutan como los servlets, pero proporcionan una forma más natural de crear contenido dinámico.

La lógica de negocio se ejecuta en la capa de negocio. Esta capa es un elemento vital de la arquitectura J2EE, ya que proporciona concurrencia, escalabilidad, gestión de ciclo de vida y tolerancia a fallos. Los componentes de esta capa son enterprise JavaBeans (EJB). Hay tres clases de enterprise beans:

- **beans de sesión** (session beans), representa una conversación transitoria con un cliente; cuando termina la ejecución del cliente, el bean de sesión finaliza sin guardar sus datos.

- **beans de entidad** (entity beans); datos almacenados en una base de datos; si el cliente termina o se cierra el servidor, los servicios subyacentes aseguran que los datos del bean se guardan.

- **beans orientados a mensajes** (message-driven beans); combina características de un bean de sesión y un oyente de mensajes JMS, permitiendo a un componente de negocio recibir mensajes JMS de forma asíncrona.

La capa de recursos de información representa la conexión con aquellos recursos que no están dentro de J2EE; por eso, se denomina **middleware**. Esto es, esta capa proporciona la conectividad entre una aplicación J2EE y el software que no forma parte de J2EE, incluyendo sistemas administradores de bases de datos y otros servidores ya existentes. Esta conectividad es posible gracias al uso de diversas tecnologías, protocolos, y conectores Java.

Contenedores J2EE

Un contenedor es una interfaz entre un componente y el sistema de inferior nivel que da soporte al componente. Mientras que un contenedor se encarga de la persistencia de los datos, la gestión de recursos, la seguridad, los hilos y otros servicios a nivel de sistema para los componentes asociados con él, estos son responsables de implementar la lógica de negocio. Esto significa que durante el desarrollo de una aplicación el programador puede concentrar todos sus esfuerzos en codificar las reglas de negocio sin tener que preocuparse de los servicios del sistema. Comparativamente, la relación entre un componente y un contenedor es muy similar a la que existe entre un programa y el sistema operativo; en este caso, el sistema operativo proporciona al programa servicios,

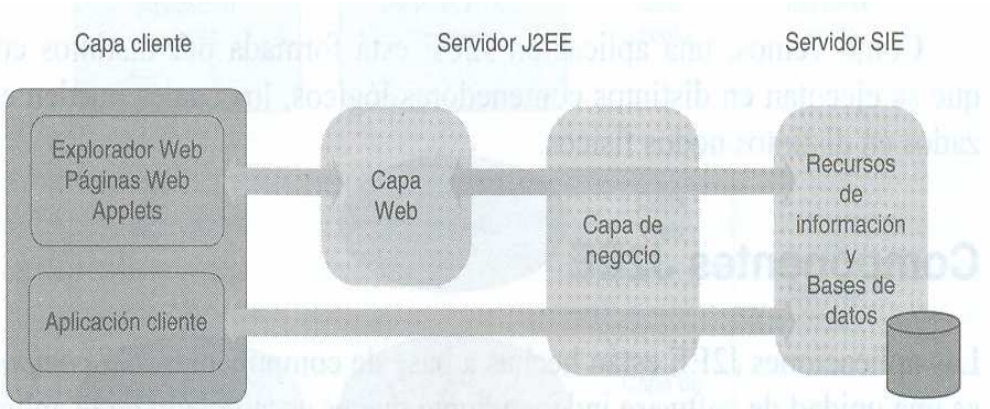
por ejemplo, de E/S, y si se instala un nuevo dispositivo de E/S, no es necesario modificar el programa, basta con re configurar el sistema operativo.

Antes de que un componente Web, EJB, o aplicación cliente pueda ser ejecutado, debe ser ensamblado en una aplicación J2EE y desplegado en su contenedor.

Resumiendo y según lo estudiado hasta ahora, un cliente se refiere a un programa que solicita un servicio de un componente. Un componente es, a su vez, otro programa que realiza una función para proporcionar el servicio; para ello, en algunos casos, puede necesitar obtener algún recurso. Y un contenedor es el software que gestiona al componente y le proporciona servicios de sistema.

Tipos de Contenedores

El proceso de instalación de una aplicación J2EE conlleva la instalación de los componentes ya vistos en sus contenedores. Gráficamente:



Mientras que una aplicación java se distribuye en un archivo .jar (Java Archive), una aplicación J2EE lo hace en un fichero .war (Web ARchive), .ear (Entreprise ARchive) o jar (Java ARchive). Se trata de un fichero ZIP que contiene módulos J2EE. Un módulo J2EE consta de uno o más componentes J2EE del mismo contenedor y un documento XML (**Descriptor de despliegue**, por ejemplo, web.xml) que describe detalles necesarios para su localización y ejecución.

Responsabilidad de la Capa Cliente

Servir de interfaz gráfica al usuario para interactuar con la aplicación. Su responsabilidad tiene que ver con la respuesta a eventos y la validación de datos. Y puede haber más de una solución.

- Definida en un formulario HTML o XML. Los eventos son tratados en el explorador, lo que ya incorpora una capa de soft adicional.
- Definida en un programa Java, ya sea usando las clases JApplet o JFrame; mas laboriosa su programación, pero con mejor tiempo de respuesta.
- Definida en el lado del servidor. Mas unificada, pero incomnveniente desde el punto de vista de los tiempos de respuesta.

Por ejemplo, es adecuado minimizar todo lo posible errores del usuario mediante el uso de componentes graficos adecuados, (listas, casillas de verificación, etc) y hacer toda la validación formal en el propio cliente. En el lado del servidor se debe validar unicamente lo que tenga que ver con la situación existente en la Base de Datos (alta duplicada, legajo inexistente, etc). Debe considerarse que el tiempo de respuesta de una red depende de muchos factores, y es conveniente validar todo lo que se pueda en forma inmediata.

La capa Web

La capa Web contiene componentes (servlets y JSP) que se comunican de forma directa con los clientes. Por lo general, las peticiones que requieren servlets o JSP se envían a un servidor de aplicaciones; por ejemplo, como el que implementa Tomcat. Esta técnica proporciona una seguridad adicional, ya que la ejecución de todos los procesos se localiza detrás del servidor de aplicaciones.

Un componente de la capa Web se identifica a través de un URL que se asocia a un enlace (link) que se encuentra en alguna página Web que se muestra al cliente. Cuando el usuario selecciona este enlace, el explorador llama al componente Web y éste se ejecuta. Esta ejecución finaliza generando una página Web que se envía de vuelta al cliente. Esto indica que bajo un componente Web subyace una lógica de presentación y otra de procesamiento. La lógica de presentación define el contenido que mostrará el cliente con los datos que generará la lógica de procesamiento.

Cuando utilizamos servlets y JSP, una buena programación supone separar el código de presentación y el de procesamiento, colocando el primero en una JSP y el segundo en un servlet. De esta forma, un especialista en HTML puede centrarse en crear la JSP y otro en Java en construir el servlet. De otra forma, (como se hacía antes de la existencia de las JSP) esto es, si colocáramos ambas cosas en un servlet, provocaríamos un inconveniente serio, ya que cualquier modificación respecto a la forma en la que deseamos se realice la presentación, por simple que sea, requeriría recompilar el servlet.

Asimismo, la capa Web debe controlar el acceso del cliente a los recursos. Para cumplir con este requisito, se aconseja utilizar las medidas de seguridad que existen en los sistemas administradores de bases de datos o en la red, siempre que sea posible. Si esta no es la mejor alternativa, también se puede utilizar un componente controlador (una JSP, un servlet, o un EJB), conocido como guardia de recursos; éste, cuando recibe una petición de un cliente, lo identifica y le otorga o deniega el acceso al recurso dependiendo de los derechos que indique su configuración. Es usual que los derechos definidos en el perfil del usuario:

- definan de que menú de opciones dispondrá
- arranquen un proceso determinado.

También, la capa Web debe evitar peticiones duplicadas enviadas por los clientes. Una interfaz de usuario basada en un navegador contiene elementos que pueden conducir al envío de una petición duplicada. Por ejemplo, pensemos en un usuario que después de pulsar el botón enviar para que el cliente envíe un formulario de pedido a la capa Web, pulsa el botón atrás del navegador (se vuelve a mostrar el formulario) y vuelve a pulsar el botón enviar. En realidad, ha enviado dos pedidos. La solución a este tipo de problemas es mantener un objeto de sesión para registrar el estado de la sesión. De esta forma, sería posible verificar si el formulario ha sido enviado con anterioridad. El estado de la sesión puede incluir prácticamente cualquier tipo de información, incluyendo el identificador del cliente o las selecciones realizadas en un formulario, y se puede gestionar en el cliente o en el servidor. No obstante, mantener el estado en la máquina del usuario presenta el inconveniente de que se perderá si ésta falla. Por eso, una mejor práctica es mantener el estado de sesión en el servidor.

La capa EJB (Enterprice Java Beans)

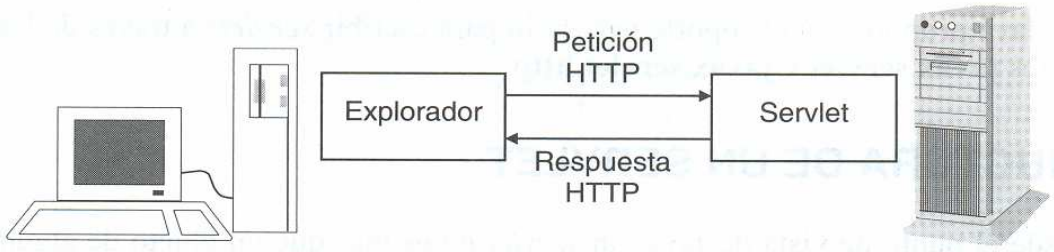
Permite construir componentes, que contengan lógica de negocio o que representen datos, reutilizables por distintas aplicaciones cliente y que normalmente estén distribuidos entre distintas máquinas.

SERVLETS

Vimos que una aplicación J2EE es un conjunto de componentes, cada uno de los cuales proporciona a la aplicación un servicio determinado; por ejemplo, utilizan un cliente ligero, por lo general un explorador (también llamado navegador), para interactuar con el usuario. Dicho cliente contiene poca o ninguna lógica de procesamiento, tarea que se deja para programas que se ejecuten en el servidor, como por ejemplo, un servlet.

¿QUÉ ES UN SERVLET?

Un servlet es un programa que se ejecuta en el contenedor Web de un servidor de aplicaciones. Los clientes pueden invocarlo utilizando el protocolo HTTP. Comparativamente, lo mismo que un applet es cargado y ejecutado por un explorador, un servlet es cargado y ejecutado por un contenedor Web.



La figura anterior indica que un servlet acepta peticiones de un cliente, procesa la información relativa a la petición realizada por el cliente y le devuelve a éste los resultados que podrán ser mostrados mediante applets, páginas HTML, etc.

No necesariamente un servlet responde directamente a una petición cliente. Antes puede comunicarse con otros servlets para completar su trabajo, o bien facilitar el acceso a bases de datos.

Características de un servlet

Los servlets fueron la alternativa de Sun Microsystems para sustituir a la programación CGI. Ambas tecnologías proporcionan la misma funcionalidad básica, con la diferencia de que utilizando servlets, con cada petición se inicia un hilo en vez de un proceso, lo cual reduce el uso de memoria del servidor y el tiempo de respuesta.

En definitiva, si comparamos los servlets con la tecnología CGI o con otras, llegaremos a la conclusión de que es bastante más sencilla y más potente. Sus principales caracteristicas son:

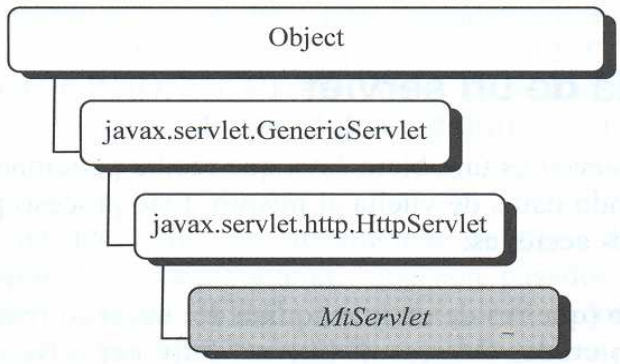
- Al estar escritos en lava, son independientes de la plataforma.
- Consumen menos recursos porque sólo son cargados la primera vez que se solicitan sus servicios. Las siguientes peticiones crearán hilos de ejecución.
- Son más rápidos que los programas CGI y que los scripts porque, por un lado se precompilan y, por otro, no se tiene que generar un nuevo proceso cada vez que se invocan.
- Son seguros y portables debido:
  - que se ejecutan bajo la máquina virtual de lava,
  - al mecanismo de excepciones de lava, y
  - al uso del administrador de seguridad de lava (Java Security Manager).
- No requieren soporte para lava en el explorador del cliente, ya que operan en el dominio del servidor y envían los resultados en HTML. No obstante, se pueden utilizar otras interfaces de cliente como aplicaciones lava o applets.

Java proporciona el soporte necesario para escribir servlets a través de los paquetes javax.servlet y javax.servlet.http.

ESTRUCTURA DE UN SERVLET

Desde el punto de vista de lava, un servlet no es más que un objeto de alguna de las clases de la API Java Servlet que implemente la interfaz Servlet, como son GenericServlet y HttpServlet. Cuando se implementa un servicio genérico normalmente se utiliza la clase GenericServlet. En cambio, la clase HttpServlet es la idónea para servicios específicos HTTP.

Los más comunes son los servlets que responden a peticiones realizadas desde un cliente HTML, entonces nos concentraremos a ellos como objetos de una clase derivada de HttpServlet:



El tipo de servlet http más básico puede ser de la forma siguiente: un objeto de una clase derivada de HttpServlet que encapsula los métodos init para iniciar el servlet, si procede, doPost y/o doGet para responder a las peticiones de los clientes y destroy para realizar operaciones de limpieza cuando se descarga el servlet.

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

public class MiServlet extends HttpServlet{
      public void init(ServletConfig config) throws ServletException{
            super.initCconfig);
            // ...
      }

      // Método doPost para responder a una petición POST
      public void doPost(HttpServletRequest request.
            HttpServletResponse response) throws ServletException.IOException{
            // …
      }

      // Método doGet para responder a una petición GET
      public void Donet(HttpServletRequest request.
            HttpServletResponse response) throws ServletException. IOException{
            // …
      }

      public void destroy(){
            // Liberar recursos
      }
}     //    public class MiServlet
```

Ciclo de vida de un servlet

Básicamente un servlet es un objeto Java que recibe peticiones de un cliente Web y responde enviando datos de vuelta al mismo. Este proceso puede descomponerse en las siguientes acciones:

- El cliente Web (que reside en la máquina del usuario) realiza una petición utilizando un explorador (Microsoft Internet Explorer o Netscape Navigator, por ejemplo). Si suponemos que la página Web del cliente contiene un formulario para interactuar con el usuario, los datos recogidos en el formulario son enviados también como parte de la petición.
- Utilizando Internet, la petición es trasladada al servidor (máquina que tiene instalado un servidor Web, o bien un servidor de aplicaciones, del que forma parte un contenedor Web) especificado por la dirección URL que escribimos en el explorador.
- El servidor recoge la petición, la analiza, se da cuenta de que hay que ejecutar un servlet y delega esta acción en el contenedor Web.
- El contenedor Web ejecuta el servlet.
- Los datos producidos que haya que enviar al cliente como resultado de la petición se trasladan en forma de página Web al servidor.
- El servidor, utilizando Internet, envía la respuesta al cliente que realizó la petición.
- El cliente recoge la página Web enviada por el servidor y la muestra al usuano.

¿Cómo se ejecuta el servlet? Todo servlet debe implementar la interfaz Servlet ya que es ésta quien declara los métodos que definen el ciclo de vida del mismo: init, service y destroy.

- **init** es invocado por el contenedor Web para iniciar la ejecución del servlet. Este método se ejecuta una sola vez y tiene como misión iniciar las variables y recursos (por ejemplo, conectar con una base de datos) necesarios para la ejecución del servlet. Observe el esqueleto de este método en la clase MiServlet expuesta anteriormente. Existen dos formas de este método, una sin parámetros y otra con un parámetro de tipo ServletConfig (la que muestra el ejemplo) que contiene la configuración del servlet y los parámetros de iniciación. Si la ejecución de este método no se realiza con éxito, lanzará la excepción ServletException.

- **service** se llama cada vez que el servidor recibe una petición para el servlet. Este método puede recibir varias llamadas simultáneas. Por cada una de ellas, crea un nuevo hilo y examina el tipo de petición HTTP (GET, POST, PUT, DELETE, TRACE, OPTIONS o HEAD) con el fin de llamar al método adecuado para atenderla; por ejemplo, doGet, doPost, doPut, etc. El tipo de petición por omisión es GET.

  El método service tiene dos parámetros de tipos HttpServletRequest y HttpServletResponse, respectivamente, que son pasados al método invocado. Por ejemplo, si el formulario HTML ejecutado por el cliente realiza una petición de tipo POST, se ejecutará el método doPost y si la realiza de tipo GET, se ejecutará el método doGet. Si la ejecución no se desarrolla con éxito, se lanzará una excepción de la clase ServletException o IOException.

  El objeto de la clase HttpServletRequest encapsula los datos enviados por el cliente al servidor. Ya veremos la forma de acceder a estos datos. Y el objeto de la clase HttpServletResponse encapsula los datos que el servidor enviará al cliente. Este objeto proporciona dos formas para retomar los datos:
  - una, mediante un objeto PrintWriter, mediante su método getWriter, permite devolver texto (cadenas de caracteres)
  - otra, mediante un objeto ServletOutputStream, mediante su método getOutputStream, permite devolver datos binarios (cadenas de bytes).

- El método destroy es el último método invocado justo antes de destruir el servlet. Normalmente se utiliza para liberar los recursos adquiridos; por ejemplo, borrar ficheros temporales.

Un servlet http puede responder a múltiples clientes simultáneamente. Quiere esto decir que los métodos que hacen el trabajo en el servlet para los clientes, pueden requerir ser sincronizados dependiendo de los recursos que tengan que ser compartidos. Esto puede hacerse bien por los métodos clásicos de sincronización de hilos ya vistos en la Unidad III.

Un servlet sencillo

El servlet que se implementa a continuación muestra el mensaje "Hola mundo cruel" en el explorador desde el que es invocado.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HolaMundo extends HttpServlet{
  // Manipular las posibles peticiones enviadas por el cliente:
  // utilizando el atributo method=get o method=post.
  protected void procesarPeticion(HttpServletRequest request,
    HttpServletResponse response, int tipo)throws ServletException, IOException{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet HolaMundo</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<font size=7>");
    out.println("¡¡¡Hola mundo cruel!!!");
    out.println("<font size=5>");
    out.println("<br>");
    if(tipo == 1)out.println(" via metodo Get(Default)");
    else         out.println(" via metodo Post");
    out.println("</font>");
    out.println("</body>");
    out.println("</html>");
    out.close();
  }

    // Manipular la petición enviada por el cliente
    // utilizando el atributo method=get.
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)throws ServletException, IOException{
        procesarPeticion(request, response,1);
  }

    // Manipular la petición enviada por el cliente
    // utilizando el atributo method=post.
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response)throws ServletException, IOException{
      procesarPeticion(request, response,2);
  }

  // Devuelve una descripción breve.
  public String getServletInfo(){
    return "Servlet HolaMundo";
  }
}
```

¿Cómo se ejecuta este servlet? Cuando el servlet del ejemplo anterior recibe por primera vez en la sesion una petición de un cliente, el servidor lo carga para

su ejecución, instante en el que se ejecuta el método init que lo inicia (por omisión, el método init invocado es el de su super clase). Cuando el método init finaliza, el servlet está en condiciones de atender a las peticiones de los clientes; en el ejemplo anterior, para cada una de las peticiones se ejecuta el método procesarPeticion que es invocado por doGet o por doPost. El servlet permanecerá cargado hasta que el servidor decida destruido, ejecutando, por ejemplo, el método destroy.

Veamos el metodo procesarPeticion. Este método almacena en el objeto out de la clase PrintWriter los datos que el servlet HolaMundo enviará al cliente que realizó la petición (el explorador). Este objeto forma parte del objeto response y es obtenido a través de su método getWriter. Los datos se almacenan en out utilizando su método println. Obsérvese que los datos almacenados se corresponden con una página HTML, la que será enviada al explorador.

Además de los datos explícitos enviados, se envía otra información de protocolo HTTP como son las cabeceras de respuesta generadas por el servidor (ídem en la petición; el cliente envía datos más las cabeceras de petición generadas por él). Estas cabeceras de respuesta incluyen una línea como la siguiente:

Content-type: text/html

Esta línea identifica el tipo del documento; en este caso se trata de un documento HTML. Cuando el cliente reciba esta información, sabrá cómo interpretar el tipo de documento que tiene que visualizar. Para establecer este valor procesarPeticion, utiliza el método setContentType del objeto response.

Software necesario para ejecutar un servlet

Como se instala Tomcat, ver Apéndice A, al final de esta Unidad
Como se ejecuta un servlet, ver Apéndice B, idem.

Queremos ejecutar HolaMundo. Entonces, en nuestro explorador web tipeamos:

http://localhost:8080/servlet/HolaMundo

Y la respuesta ES:

INCLUIR PROCESOS ESCRITOS EN JAVA

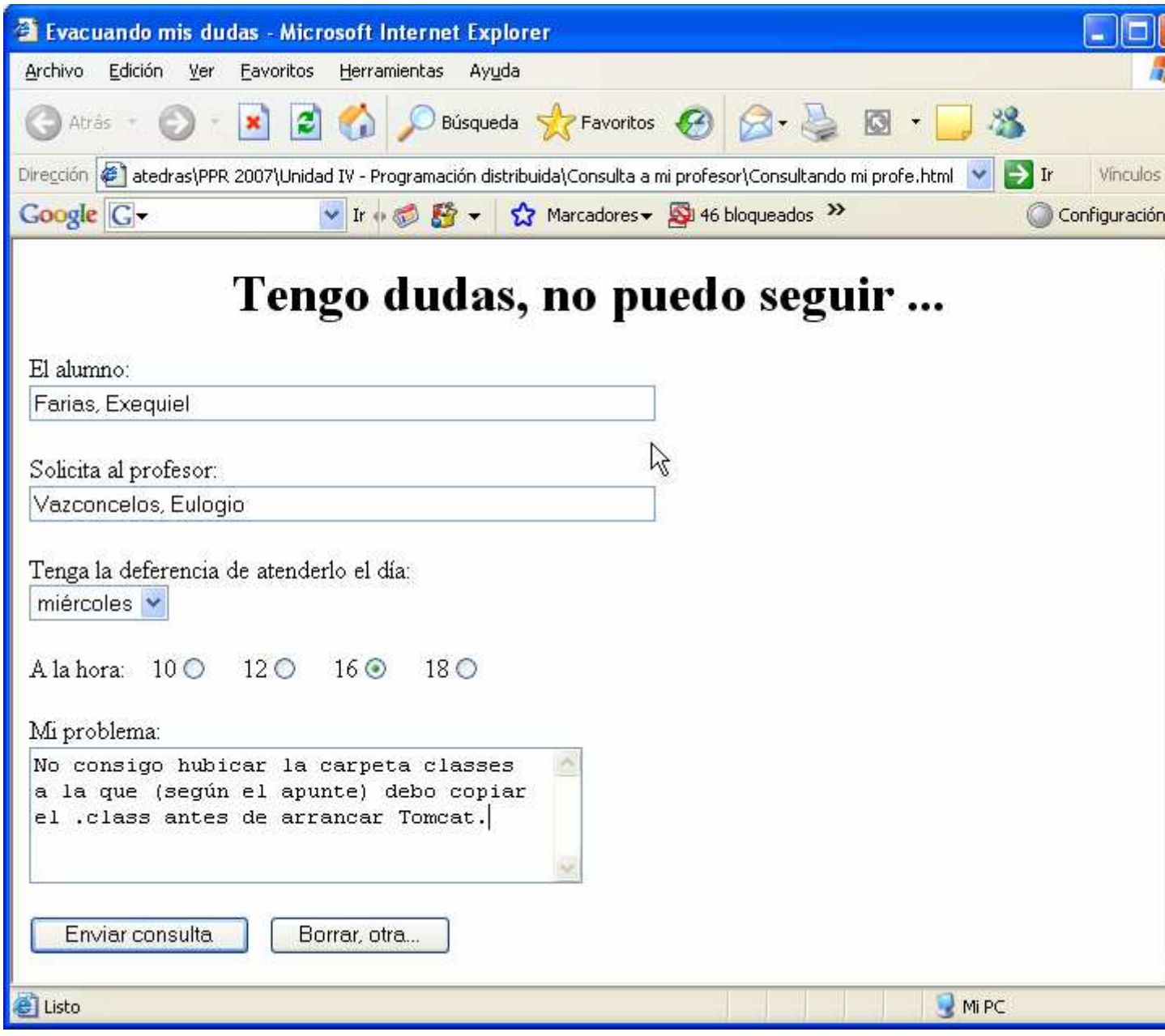
Los servlets, además de generar páginas HTML, pueden utilizar toda la potencia del lenguaje Java para realizar cualquier proceso que se requiera. Un servlet es un componente capaz de generar contenido dinámico, por ejemplo, un saludo múltiple y la fecha y hora del servidor.

```
import java.io.*; import javax.servlet.*; import javax.servlet.http.*;
import java.util.*;
public class FechaHora extends HttpServlet{
   // Manipular la petición enviada por el cliente
   // utilizando el atributo method=get.
  protected void doGet(HttpServletRequest request,
```

```
    HttpServletResponse response)throws ServletException, IOException{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet Saludo</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<center>");
    Random rnd = new Random();
    int tamaño;
    for (int i = 1; i <= 7; ++i){
      tamaño = rnd.nextInt(7)+1;
      out.println("<font size="+tamaño+" color=blue>");
      out.println("¡¡¡Hola mundo!!!"+tamaño);
      out.println("<br>");
    }
    out.println("<br>");
    Calendar fechahora = new GregorianCalendar();
    tamaño = rnd.nextInt(7)+1;
    out.println("<font size="+tamaño+">");
    out.println("<font color=green>");
    out.println("Son las  " +
                fechahora.get(Calendar.HOUR_OF_DAY) + ":" +
                fechahora.get(Calendar.MINUTE) + ":" +
                fechahora.get(Calendar.SECOND) );
    out.println("<br>");
    out.println("<font color=gray>");
    out.println("Del día " +
                fechahora.get(Calendar.DAY_OF_MONTH) + "/" +
                (fechahora.get(Calendar.MONTH)+1) + "/" +
                fechahora.get(Calendar.YEAR) );
    out.println("</font>");
    out.println("</center>");
    out.println("</body>");
    out.println("</html>");
    out.close();
  }
  public String getServletInfo(){return "Servlet Saludo";}
}
```

PROCESAR FORMULARIOS

Hasta ahora, los servlets que hemos diseñado sólo enviaban información al explorador. No obstante, los clientes Web pueden mostrar formularios que permitan solicitar datos a los usuarios, que serán enviados al servidor para, por ejemplo, ser almacenados en una base de datos para posteriores peticiones, o para otros procesos, o bien para dar al usuario una respuesta ajustada a la petición. Y vamos a aprovechar un formulario que ya mostramos en la página 15 de esta Unidad, "Tengo dudas, no puedo seguir…"



Vamos a codificar el servlet ConsProfe que responde a este formulario.

```
import java.io.*; import javax.servlet.*; import javax.servlet.http.*;
import java.util.*;
public class ConsProfe extends HttpServlet{
    // Manipular la petición enviada por el cliente
    // utilizando el atributo method=post.
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException{
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Responder al solicitante
        out.println("<html)") ;
        out.println("<title)Concertar una horario de consulta</title)");
        out.println("<br>");
        out.println("Mañana miercoles, de tarde, de 14:00 a 18:00 hs");
        out.println("<br>");
        out.println("Nos vemos");
        out.println("</html)");
        out.close();
    }

    // Devuelve una descripción breve.
    public String getServletInfo(){
        return "Servlet <ConsProfe";
    }
}
```

Y lo que aparece en el navegador del alumno

Nota: *no se usaron la etiquetas html head, body… No sale del todo bien…* Vamos a mejorar esto.

Tipos de peticiones

Anteriormente dijimos que el método service de HttpServlet se llama cada vez que el servidor recibe una petición para el servlet; éste método examina el tipo de petición HTTP (GET, POST, PUT, DELETE, TRACE, OPTIONS o HEAD) con el fin de llamar al método adecuado para atenderla; cuando desarrollamos servlets http, normalmente sólo utilizamos los métodos doGet y doPost. Los otros métodos son más bien utilizados por programadores muy familiarizados con la programación HTTP.

Petición HTTP GET

En este caso, los datos del formulario son enviados al servidor a continuación de la dirección URL especificada por el atributo action de form; por ejemplo, cuando el usuario haga click en el botón Enviar datos del formulario mostrado anteriormente, el navegador abrirá una conexión HTTP al puerto 8080 de la máquina localhost y enviará una petición con la siguiente información (suponiendo que se han introducido los datos mostrados en la figura anterior):

```
http://localhost:8080/servlet/ConsProfe
      ?alumno=xxxxxxxxxxxxxxxxx&
```

```
profesor=xxxxxxxxxxxxxxxx&
d%EDa   =Miercoles&
hora=10&
miDuda =xxxxxxxxxxxxxxxxxx
```

Observamos a continuación del URL los parámetros alumno, profesor, día, hora y miDuda van colocados después del símbolo? y separados por el símbolo &. Los parámetros y sus valores están compuestos por caracteres ASCII alfanuméricos más los caracteres. (punto), - (guión), * (asterisco) y _ (subrayado); los espacios en blanco son sustituidos por + y otros caracteres como las letras acentuadas por %dd (dd: código en hexadecimal del carácter).

Resumiendo, con el tipo de petición HTTP GET la información del formulario se envía **añadiéndola** al URL indicado por action, el tamaño de la cadena enviada está limitado, al estar en ASCII es totalmente legible, por lo tanto, no se le considera un modo seguro, ya que puede ser utilizada malintencionadamente como destino de un enlace.

Petición HTTP POST

Cuando el tipo de petición HTTP es POST, los datos son enviados al servidor en el cuerpo de la petición. Es el método usual de enviar los datos de un formulario, **después** del URL indicado en action.

Elegir el método para enviar la información no es dificil.

- Si los datos son pocos y no confidenciales, GET.
- Si son largos, privados o importantes, POST.

LEER LOS DATOS ENVIADOS POR EL CLIENTE

Un servlet lee los datos enviados en la petición realizada por un cliente mediante el método **getParameter** del objeto **HttpServletRequest** que los métodos **doGet** y **doPost** reciben como parámetro.

getParameter devuelve en un objeto de tipo String el valor del parámetro especificado como argumento. Los nombres de los parámetros son sensibles a minúsculas y mayúsculas. Por ejemplo, cuando el usuario hace click en el botón EnviarDatos del formulario "Evacuando mis dudas" (pg 16), se enviarán al servidor los parámetros alumno, profesor, día, hora y miDuda con sus valores; entonces, la sentencia siguiente devolverá el valor asociado al parámetro profesor:

```
String valor = request.getParameter("profesor");
```

Si el cliente envía el parámetro sin asignarle un valor, entonces getParameter devuelve una cadena vacía y si no envió el parámetro, devuelve nuU.

Para leer el conjunto de valores asignados a un parámetro, se utiliza el método **getParameterValues** que devuelve una array de objetos tipo String con todos los valores del parámetro especificado como argumento; si éste no existe, devuelve un array null.
Por ejemplo, piense en la pregunta "medios de transporte que habitualmente utiliza" que tiene como respuestas {"automovil", "tren", "avion", "bicicleta"}. Esta pregunta se puede implementar con tantas casillas de verificación como tipos de transportes posibles, todas con el mismo nombre, por ejemplo, "transporte". El código siguiente mostraría los tipos de transportes seleccionados:

```
String[] valor = request.getParameterValues("transporte");
for (int i = O; i < valor.length; ++i) out.println(valor[i]);
```

Para obtener los nombres de todos los parámetros, disponemos del método **getParameterNames** que retorna una enumeración (objeto **Enumeration**) de objetos String con los nombres de los parámetros. Por ejemplo, el siguiente segmento de código muestra los nombres de los parámetros que han sido pasados al servidor junto con sus valores:

```
String param, valor;
Enumeration nombresParams = request.getParameterNames();
while (nombresParams.hasMoreElements()){
      param = (String)nombresParams.nextElement();
      valor = request.getParameter(param);
      out.println(param + ": " + valor);
```

El método hasMoreElements de Enumeration permite iterar mientras haya más elementos en la enumeración. Sucesivas llamadas al método nextElement de la misma clase van devolviendo el siguiente elemento de la serie de datos.

Para ejemplificar todo lo que estamos diciendo, vamos a modificar el servlet **ConsProfe** para que almacene en un fichero de texto las peticiones realizadas por los distintos alumnos que desean concretar una consulta con alguno de sus profesores. Y vamos a registrar estas consultas en un fichero de texto, por si nos insteresa algún tipo de estudio estadístico mas adelante. Al nuevo servlet le llamaremos **ConsultaProfesor.**

Recordadando AED, 1er año, para crear un fichero de texto, basta con definir un flujo de la clase FileWriter vinculado con ese fichero, y enviar información al mismo utilizando el método println. Como este método es proporcionado por la clase PrintWriter, lo que haremos será crear un flujo de esta clase que utilice el flujo FileWriter anterior como intermediario para escribir en el fichero. El constructor de la clase FileWriter que utilizaremos tiene dos argumentos: el primero permite especificar el nombre del fichero y el segundo, si es true indica que cada vez que se acceda al fichero para escribir, los datos serán añadidos al final (si el fichero no existe, se crea). El código siguiente muestra cómo escribir en un fichero consultas.txt parejas "nombre del parámetro" - "valor":

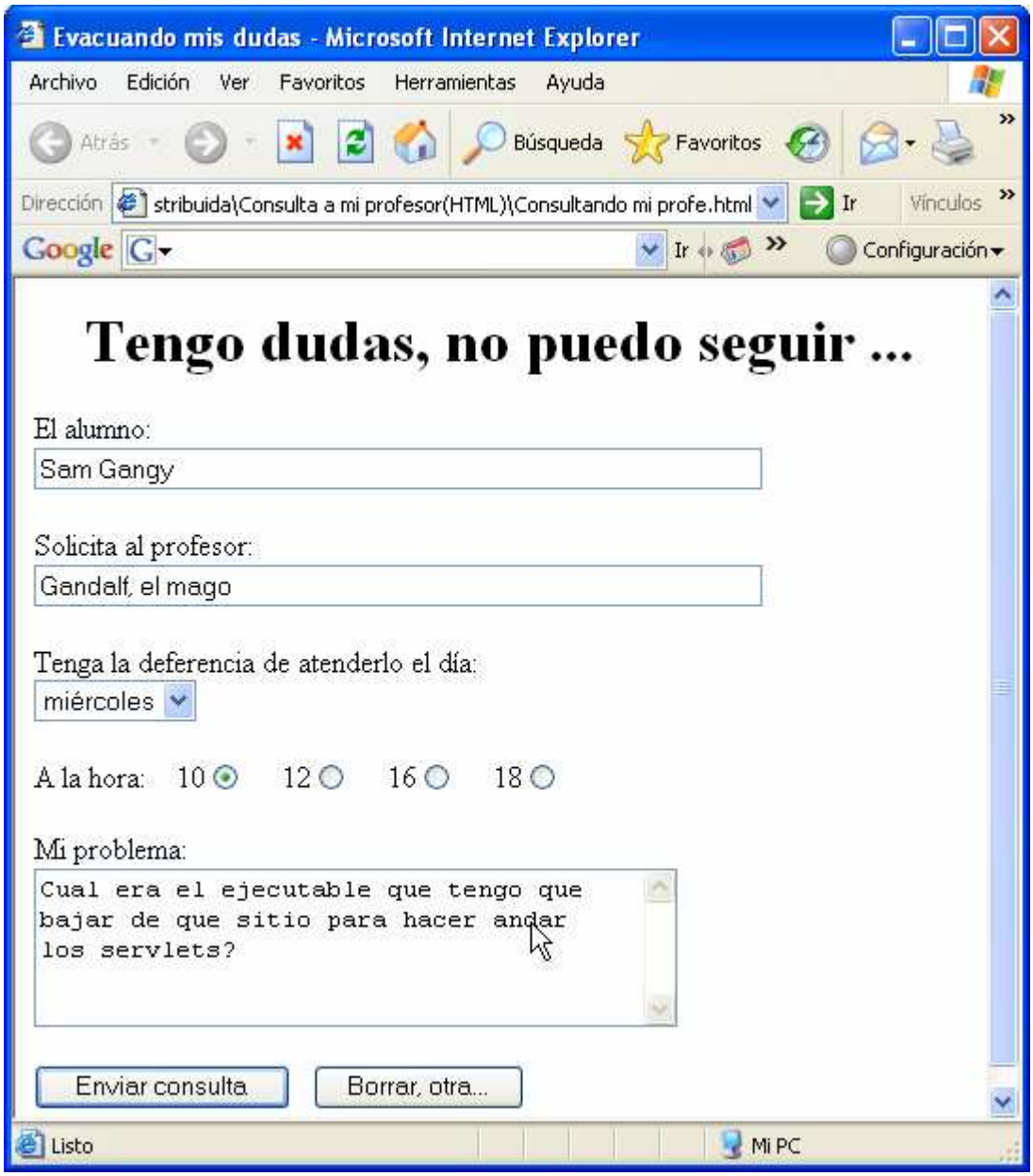
```
FileWriter fw = new FileWriter("consultas.txt". true);
PrintWriter fichConsultas = new PrintWriter(fw);
// ...
fichConsultas.println(param + ": " + valor);
```

También, cada vez que el servidor reciba una petición responderá al cliente indicándole si la petición fue registrada, o si ocurrió un error, en cuyo caso tendrá que volver a realizarla. El código correspondiente a este servlet sigue:

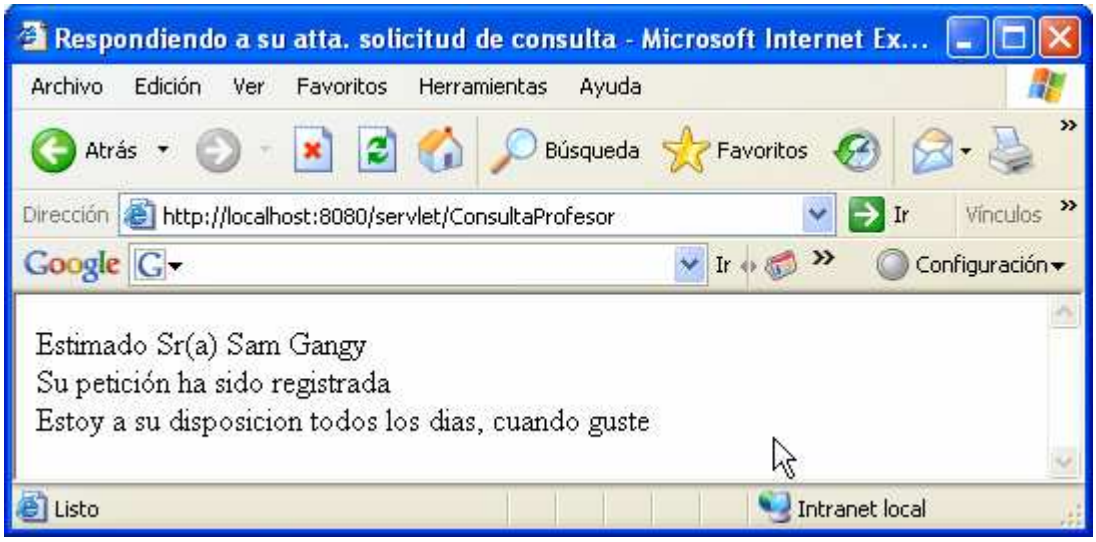
```
import java.io.*; import javax.servlet.*; import javax.servlet.http.*;
import java.util.*;

public class ConsultaProfesor extends HttpServlet{
    // Daremos un poco de variedad a los mensajes con que el servlet responde
    String[] msgs = {"Gustosamente lo atenderemos el dia del parcial",
                     "Al volver de mis vacaciones tendremos  el placer ",
                     "Estoy a su disposicion todos los dias, cuando guste",
                     "Le recomiendo la lectura del material de la catedra",
                     "Esas dudas suyas se deben a falta de base. Repase ...",
                     "Su pregunta me sorprende. En realidad, no deberia ...",
                     "Sus dudas corresponden a mi clase de mañana. No falte"};
    String mensaje; int indice;
    // utilizando el atributo method=post.
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException{

        // Tipo de la respuesta que será enviada al cliente
        response.setContentType("text/html");
```

```
        // Obtener el objeto 'PrintWriter' para devolver la respuesta
        PrintWriter out = response.getWriter();

        // Registrar la consulta del alumno
        try{
            // Abrir el fichero para el registro de la consulta
            FileWriter fw = new FileWriter("Consultas.txt", true);
            PrintWriter fichConsultas = new PrintWriter(fw);

            // Tomar los datos recibidos del cliente y escribirlos
            // en el fichero. Se finaliza cada registro con el literal
            // "<FIN>" para su posterior identificación.
            Enumeration nombresParams = request.getParameterNames();
            while (nombresParams.hasMoreElements()){
                String param = (String)nombresParams.nextElement();
                String valor = request.getParameter(param);
                fichConsultas.println(param + ": " + valor);
            }
            fichConsultas.println("<FIN>");
            // Cerrar el fichero fichConsultas
            fichConsultas.close();
            fw.close();

            // Respondiendo al alumno
            out.println("<html>");
            out.println("<title>Respondiendo a su atta. solicitud de
                            consulta</title>");
            String nome = request.getParameter("alumno");
            out.println("Estimado Sr(a) "+ nome);
            out.println("<br>");
            out.println("Su petición ha sido registrada");
            out.println("<br>");
            indice = (int)(7*Math.random());
            mensaje = msgs[indice];
            out.println(mensaje);
            out.println("<br>");
            out.println("</html>");
        }
        catch(IOException e){
            out.println("Hubo problemas cursando su solicitud.");
            out.println("<br>Por favor, inténtelo otra vez.");
        }
        out.close();
    }
    // Cerrar el flujo
    // Devuelve una descripción breve.
    public String getServletInfo(){return "Servlet Tutorías";}
}
```

Ante una consulta como esta



La respuesta



DESCRIPTOR DE DESPLIEGUE

Un descriptor de despliegue es un fichero de texto basado en XML cuyos elementos describen:

- Como utilizar los componentes en un entorno específico.
- Parámetros de iniciación
- opciones de seguridad.

Para aplicaciones Java que se ejecutan en un servidor de aplicaciones (servlets, JSP, etc.), este fichero:

- Recibe el nombre de **web.xml**
- Se localiza en la carpeta WEB-INF.

Por ejemplo, para la aplicación ConsultaProfesor anterior, el descriptor de despliegue incluirá:

- el nombre y la descripción del servlet
- la clase del servlet
- los parámetros de iniciación
- alguna otra información necesaria para su ejecución

Por ejemplo:

```
<?xml version="l.O" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
      PUBLIC "-//Sun Mierosystems, Ine.//DTD Web Applieation 2.3//EN"
      "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
      <display-name>Servlet ConsultaProfesor</display-name>
      <servlet>
            <servlet-name>ConsultaProfesor</servlet-name>
            <servlet-elass>ConsultaProfesor</servlet-elass>
            <init-param>
                  <param-name>carpeta</param-name>
                  <param-value>D:/Servlets/Ejemplos PPR</param-value>
            </init-param>
      </servlet>
      <servlet-mapping>
            <servlet-name>ConsultaProfesor</servlet-name>
            <url-pattern>/servlet/ConsultaProfesor</url-pattern>
      </servlet-mapping>
      <welcome-file-list>
            <welcome-file>ConsultaProfesor.html</welcome-file>
      </welcome-file-list>
</web-app>
```

La etiqueta <display-name> especifica la información que mostrará un administrador de aplicaciones Web para identificar esa aplicación.

Las etiquetas <servlet> y </servlet> delimitan los datos relativos al servlet.

Las etiquetas <init-param> y </init-param> delimitan los parámetros de iniciación y sus valores.

La etiqueta <url-pattern> especifica el patrón utilizado para resolver los URL. La porción de URL después de http://servidor:puerto/localización es comparada con el <url-pattern> , si coincide el servlet será invocado.

La etiqueta <welcome-file> especifica el fichero HTML o JSP que se utilizará por omisión para arrancar el servicio. Según esto, para mostrar el formulario bastaría con escribir en el explorador el URL siguiente (obsérvese que se omite el nombre del fichero ConsultaProfesor.html):

http://localhost:8080/ConsultaProfesor

Nota: Donde será que por default se grabó **consultas.txt**, ya que no lo indicamos en el método init()?. El buscador de Windows me dice: D:\tomcat\Tomcat5.5. Y que es esta cadena? Pues el valor guardado en la variable de entorno TOMCAT_HOME.

INICIACIÓN DE UN SERVLET

Después de que el contenedor Web carga y crea un objeto de la clase del servlet y antes de que el servlet responda al cliente, se ejecuta su método **init**. Esto quiere decir que cuando necesitemos realizar una iniciación personalizada, tendremos que redefinir este método heredado de **HttpServlet**. Por ejemplo, el fichero Consultas.txt ¿donde se guardó?. Si queremos controlar esta situación, tendremos que definir parámetros de iniciación y leerlos en el método init. Fijándonos en el fichero web.xml escrito en el apartado anterior observamos el parámetro carpeta al que se le ha asignado el valor **"D:/Servlets/Ejemplos PPR"**. ¿Cómo obtenemos el valor de este parámetro desde init? Pues, invocando al método **getInitParameter**. Este método devuelve en un objeto de tipo String el valor del parámetro especificado como argumento. Según esto, vamos a añadir a la clase **ConsultaProfesor** que define nuestro servlet, el método **init** escrito como se nuestra a continuación:

```
public class ConsultaProfesor extends HttpServlet{
      String carpeta = null;
      public void init(ServletConfig config) throws ServletException{
            super.init(config);
            carpeta = getlnitParameter("carpeta");
            // parámetro establecido en el fichero web.xml que se
            // localiza en la carpeta WEB-INF del servidor
            if (carpeta == null)
                  System.err.println("No se especificó la carpeta de destino");

      }
      // Codificacion anterior …
}
```

El método init tiene un parámetro de tipo ServletConfig que es utilizado por el contenedor del servlet para pasar información al servlet durante su iniciación.

Cuando se ejecute el método anterior, se almacenará en el objeto carpeta de tipo String el valor "D:/Servlets/Ejemplos PPR", que se corresponde con la carpeta donde deseamos guardar el fichero "ConsultapROFESOR.txf'. Según esto modifique el primer parámetro del flujo de la clase FileWriter vinculado con ese fichero como se indica a continuación:

```
FileWriter fw = new FileWriter(carpeta + "/consultas.txt", true);
```

Ahora, cuando ejecuta el servlet, las peticiones serán almacenadas en D:/Servlets/Ejemplos PPR/Consultas.txt.

SEGUIMIENTO DE UNA SESiÓN

HTTP es un protocolo sin estado: cada petición es tratada de forma independiente por el servidor. Por lo tanto, el servidor no sabe si una serie de peticiones provienen del mismo o de diferentes clientes y si además, están relacionadas entre sí. Esto presenta un sin fin de problemas para deterninadas aplicaciones; imaginemos, estemos haciendo una compra a través de una página Web y añadamos

uno y otro artículo a nuestro carrito, entre petición y petición ¿cómo recordará el servidor lo que hemos comprado? ¿Cómo identificará mi compra de la de otro cliente? Igualmente, cuando realicemos otra petición, por ejemplo, al servicio que nos pennite dar nuestro número de tarjeta de crédito y la dirección de envío, ¿cómo recordará el servidor lo que hemos comprado? Pues bien, las sesiones se usan para identificar y guardar infonnación individual de cada cliente.

Una sesión se inicia después que el cliente realiza una petición al servidor para utilizar alguno de sus servicios y expira cuando pase un tiempo sin que el cliente realice otra petición (veinte o treinta minutos), o bien cuando se cierra el cliente. Esta es la manera más habitual de finalizar una sesión, aunque también es posible cancelarla ejecutando el código adecuado (invalidate de HttpSession).

Existen varias soluciones a este problema:

- cookies,
- reescritura del URL
- campos ocultos en los formularios.
- Usando objeto HttpSession (HttoServletRequest.getSession())

Veamos la primera.

Cookies

Una **cookie** (Galleta, biscocho, dice el diccionario…) es una pequeña cantidad de información (no más de 4K) que un servlet puede crear y almacenar en la máquina cliente y, posterionnente, consultar a través de la API de cookies de los servlets.

Las cookies son una de las soluciones más utilizada para realizar el seguimiento de una sesión. Utilizando este mecanismo el contenedor Web envía una cookie al cliente (por ejemplo, con un identificador de sesión), la cual será retornada sin modificar por el cliente automáticamente por cada nueva petición que éste haga al servidor, asociando la petición del servicio de modo inequívoco con una sesión.

En el caso del carrito de la compra, podemos utilizar cookies para almacenar el identificador del cliente para poder identificar en el servidor los artículos comprados por éste; de esta fonna, cada petición subsiguiente podrá obtener información de la anterior. Muy bueno, pero, a pesar de que Java proporciona la clase **Cookie** en el paquete **javax.servlet.http** para que podamos trabajar con cookies, existen ciertos aspectos que deben ser controlados y que no son simples:

- Extraer la ***cookie*** que almacena el identificador de sesión entre todas las cookies, ya que puede haber varias.
- Seleccionar un tiempo de expiración apropiado para la ***cookie***.
- Asociar la información del servidor con el identificador de sesión

Una cookie se compone de dos partes: un nombre y un valor; el nombre la identifica entre todas las demás cookies almacenadas en el cliente y el valor es un dato asociado con la cookie.

Un servlet para crear una cookie simplemente tiene que invocar al constructor de la clase **Cookie** pasando como argumentos su nombre y el valor asociado:

```
Cookie miCookie = new Cookie("nombre-cookie", "valor-asociado");
```

Una vez creada una cookie, el servet debe añadirla a las cabeceras de la respuesta HTTP. Para ello, hay que invocar al método **addCookie** del objeto **HttpServletResponse** pasando como argumento la cookie que se desea añadir:

```
response.addCookie(miCookie);
```

El método addCookie no afecta a las cookies que ya existan en el cliente. El tamaño de una cookie está limitado a 4 Kb. Y la mayoría de los exploradores limitan el espacio total de almacenamiento a 2 Mb., por lo que este método puede hacer que se eliminen otras cookies. La fecha de expiración de una cookie es sólo una sugerencia; es el explorador el que decide cuándo eliminar las cookies dependiendo del espacio de almacenamiento.

Para leer una cookie, hay que invocar al método **getCookies** del objeto **HttpServletRequest**. Este método devuelve un array de objetos **Cookie**:

```
Cookie cookies[] = request.getCookies();
if (cookies == null) out.println("No hay cookies");
```

Cuando un cliente realiza una petición a un servidor, le envía las cookies propias de ese servidor, no todas. Las cookies que un cliente almacena para un servidor sólo pueden ser devueltas a ese mismo servidor. Por lo tanto, los *servlets* que se ejecutan dentro de un servidor comparten las *cookies*.

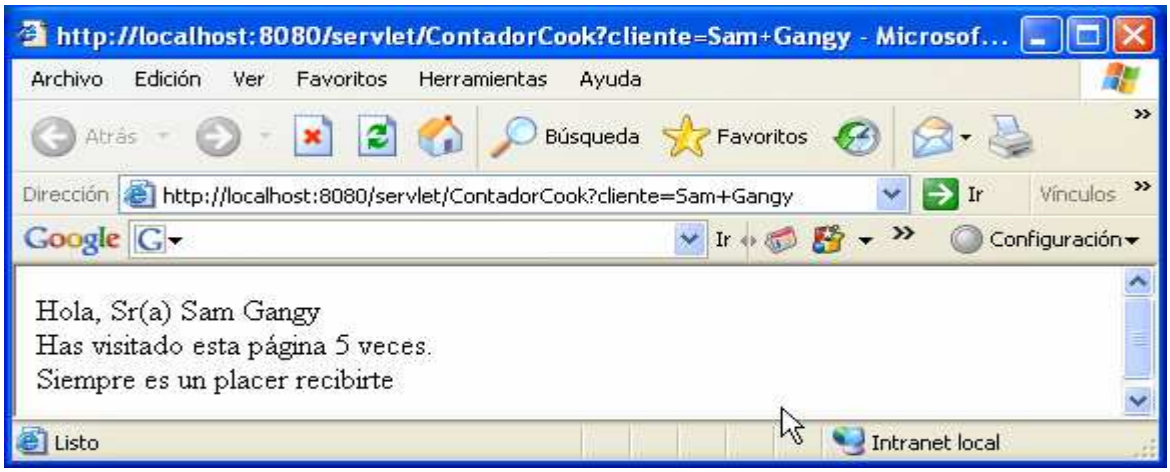
El servlet puede recorrer el array de cookies y recuperar los atributos mediante los métodos **getName** y **getValue**. El primero devuelve un objeto String con el nombre de la cookie y el segundo otro con el valor asociado:

```
String nombre, valor;

for (int i = 0; i < cookies.length; ++i)
      nombre = cookies[i].getName();
      valor = cookies[i].getValue();
// Operaciones
}
```

El siguiente ejemplo muestra un servlet ContadorCook que escribe una cookie para informar el número de veces que este servicio es accedido; puede atender a tantos clientes como se quiera.

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

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

    // Obtener el valor actual de la cookie "contador.cook" buscando
    // entre las cookies recibidas.
    String sCuenta  = null;
    String cliente  = null;
           cliente  = request.getParameter("cliente");
    Cookie[] cookies = request.getCookies();
    if (cookies != null){ // si hay cookies...
        for (int i = 0; i < cookies.length; i++){
            // Buscar la cookie "contador.cook"
            if (cookies[i].getName().equals("contador.cook")){
                // y obtener el valor asociado
                sCuenta = cookies[i].getValue();
                break;
            }
        }   //  for
    }   //  if (cookies != null)

    // Incrementar el contador para esta página. El valor es
    // guardado en la cookie con el nombre "contador.cook".
```

```
    // Después, asegurarse de enviársela al cliente con la
    // respuesta (response).
    Integer objCuenta = null; // contador
    if (sCuenta == null) // si no se encontró "contador.cook"
        objCuenta = new Integer(1);
    else                 // si se encontró "cuenta.cook"
        objCuenta = new Integer(Integer.parseInt(sCuenta)+1);
    // Crear una nueva cookie con la cuenta actualizada
    Cookie c = new Cookie("contador.cook", objCuenta.toString());
    // Añadir la cookie a las cabeceras de la respuesta HTTP
    response.addCookie(c);

    // Responder al cliente
    out.println("<html>");
    out.println("Hola, Sr(a) "+cliente);
    out.println("<br>");
    out.println("Has visitado esta página " + objCuenta.toString() +
                ((objCuenta.intValue() == 1) ? " vez." : " veces."));
    out.println("<br>");
    out.println("Siempre es un placer recibirte");
    out.println("</html>");

    // Cerrar el flujo
    out.close();
  }

  // Devuelve una descripción breve.
  public String getServletInfo(){return "Servlet ContadorCook";}
}
```



Hemos hecho las visitas alternadamente. Verificamos que el server no pierde la "pista" de sus clientes. Recordemos que habiamos visto que abre un hilo para cada uno, esto lo hace en forma transparente para el programador.
El ejemplo anterior no hace ningun registro del cliente que hace la petición. Tomamos su nombre, solo claridad de este material. Si arrancamos otra sesión de explorador web, no reconocerá a Sam Gangy.
Sin embargo es posible identificar al cliente desde el que un usuario hace la petición. Las cookies también pueden utilizarse para almacenar información que permita identificar a un usuario. Si lo que está haciendo el usuario es una compra de uno o más artículos a través de un sitio Web, podemos utilizar una cookie para almacenar el identificador del cliente, con el fin de identificarlo en este sitio y llevar así un seguimiento de sus compras hasta finalizar.

Combinando documentos HTML, Servers, Cookies …

La aplicación que vemos a continuación consta de:

- Un documento HTML, que solicita datos del cliente y lo conecta con el servlet Atención.
- El servlet Atención le ofrece un combo de tres articulos, informandole de su monto total. Usa Cookie. El usuario puede:
  - Volver atraz, (Para optar por otro combo)
  - Ir al servlet Despacho (Para pedir el envio)

- El servlet Despacho solo sirve para:
  - Volver atraz, (Para optar por otro combo)
  - Ir al servlet Envio (No implementado, Tomcat nos lo dirá)

El documento Html

Source Editor [aComprar]

```
1  <html>
2    <head>
3      <title>Atendiendo clientes queridos...</title>
4    </head>
5    <body>
6      <hl align="center">Tengo hambre!!! ... no puedo m...</hl>
7      <form action="http:\\localhost:8080/servlet/Atencion">
8        El hambriento: <input type="text" name="hambriento" size="20">  
9          Solicita al Hiper: <input type="text" name="hiper" size="20">
10       <br><br><font size=5>Tenga la deferencia de atenderlo :
11       <font size=6>¡¡¡Urgente!!!</font>
12       <br><br><input type="submit" value="Atender ya!">
13         <input type="reset" value="Desistir...">
14     </form>
15   </body>
16 </html>
17
```

14:12 INS

La página que vemos:

Clickeando **Atender ya**, el servlet invocado es **Atención**, su codigo:

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

public class Atencion extends HttpServlet{
    private String hungry, hipMerc;

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

        // obtenemos datos del formulario
        hungry = request.getParameter("hambriento");
        hipMerc= request.getParameter("hiper");
        // Preparamos informacion acerca de comidas
        Comidas comida = new Comidas();
        // Obtener identificador de sesión actual (getValue()) buscando
        // por el nombre del cliente (getName()) entre las cookies recibidas.
        String idSesion = null;
        Cookie[] cookies = request.getCookies();
        if (cookies != null){
            for (int i = 0; i < cookies.length; i++){
                if (cookies[i].getName().equals(hungry)){
                    idSesion = cookies[i].getValue();
                    break;
                }
            }   //  for
        }   //  if (cookies ...

        // Si el identificador de sesión no fue enviado, debemos generarlo.
        if (idSesion == null){ // Es la primera conexion del cliente
            idSesion = generarIdSesion(); // le generamos su idSesion
```

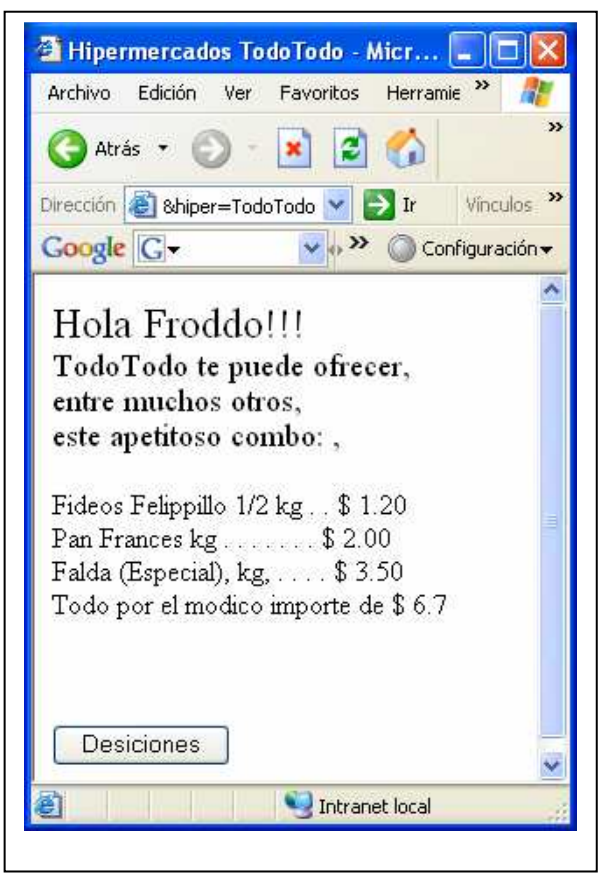
```
            Cookie c = new Cookie(hungry, idSesion); // su Cookie
            response.addCookie(c);  // y la incorporamos al objeto response
        }

        //  Generando html para el cliente
        out.println("<head>");
        out.println("<title>Hipermercados " + hipMerc + "</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<font size="+5+">");
        out.println("Hola " + hungry + "!!!<br>");
        out.println("<font size=" + 4 + ">");
        out.println(hipMerc + " te puede ofrecer,<br>");
        out.println("entre muchos otros,<br>");
        out.println("este apetitoso combo: ,<br><br>");

        int i, j;
        float total = 0;
        Float precioIt;
        String auxStr;
        out.println("<font size=" + 3 + ">");
        for (i = 0; i < 3; i++){
            j = (int)(10*Math.random());
            out.println(comida.carta[j]+"<br>");
            auxStr = new String(comida.carta[j]);
            auxStr = auxStr.substring(31);
            auxStr = auxStr.trim();
            precioIt = new Float(auxStr);
            total+= precioIt.floatValue();
        }
        out.println("Todo por el modico importe de $ "+total);
        out.println("</font>");

        //  Vamos al servlet Despacho, donde le
        //  preguntaremos si continua o finaliza
        out.println("<form action=/servlet/Despacho?&idSesion=" + idSesion
        //          + "tuPedido="+tuPedido
                    + " method=get>");
        out.println("<br><br>");
        out.println("<input type=submit value=\"Decisiones\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

    private static String generarIdSesion(){
        // Garantizar un id único
        String uid = new java.rmi.server.UID().toString();
        // Codificar cualquier carácter especial
        return java.net.URLEncoder.encode(uid);
    }
}   //  public class Atención
```

El servlet Atención usa **class Comidas**, hela aquí:

```
class Comidas{
    String[] carta =
    {"Amburguesas, cja 10 Un, . . . $10.00",
     "Fideos Macarron, pk 1 kg, . . $ 1.99",
     "Cerveza BadHein, env 780cc  . $ 2.80",
     "Falda (Especial), kg, . . . . $ 3.50",
     "Bife lomo tern, kg, . . . . . $12.50",
     "Pan Frances kg . . . . . .  $ 2.00",
```

```
      "Polenta PrestPres 1/2 kg .  . $ 1.60",
      "Fideos Felippillo 1/2 kg . .  $ 1.20",
      "Tinto FacuRelli, 1 lt . . . . $ 2.80",
      "Salch. VienAus, pk 200 grs, . $ 2.70"};
      // 23456789 123456789 123456789 12345
      String pedido ="";

      public String[] getComidas(){return carta;}
      public String   getComida(int cual){return carta[cual];}
      public void addItem(String item){pedido+=item+", <br>";}
      public String getPedido(){return pedido;}
}    //  class Comidas
```

Y la página que se genera para el cliente,
titulada **Hipermercados TodoTodo**



El boton **decisiones** llama al servlet **Despacho** (abajo),
quien presenta la siguiente página

```
import java.util.*; import java.net.URLEncoder;
public class Despacho extends HttpServlet{
    private int ind; private String idSesion;
    private String itemPed;
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Necesito datos del cliente
        idSesion = request.getParameter("idSesion");

        //  Generando html para el cliente
        out.println("<head>");
        out.println("<title>Despacho</title>");
```

```
        out.println("</head>");
        out.println("<body>");

        out.println("<font size="+5+">");
        out.println("Despacho<br>");

        out.println("<font size="+3+">");
        out.println("Tenemos muchos otros combos<br>");
        out.println("Para verlos, boton <-- Atraz<br>");

        out.println("<form action=\"/servlet/Envio\"" +
                    " method=get>");
        out.println("<br><br>");
        out.println("<input type=submit value=\"Enviar combo\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Finamente, si estando en la página generada por el servlet **Despacho** optamos por clickear sobre **Enviar Combo**, llamamos al servlet **Envio**, a quien no hemos implementado. Entonces Tomcat nos dice:

Java Server Pages (JSP)

JSP es un acrónimo de **Java Server Pages**, (Páginas de Servidor Java). Es una tecnología orientada a crear **páginas Web** con **programación en Java**.

Las páginas **JSP** están compuestas de código **HTML/XML** mezclado con etiquetas especiales (órdenes) y con trozos de código escritos en Java (scriptlets - secuencias de órdenes). Por lo tanto, podremos escribirlas con nuestro editor HTML/XML habitual. Una **JSP** no está llena de llamadas al método println como hemos estado viendo hasta ahora.

El éxito de esta tecnología radica en que una JSP puede:

- Procesar la petición incluyendo ella misma la lógica de negocio, o bien
- Separar la lógica de negocio de la capa de presentación. Esto se puede hacer mediante llamadas a componentes construidos con tecnología de servlets, EJB o con alguna otra tecnología. *(como las que se utilizan hoy en día: Apache **Struts**, **Spring Frameworks**, **JBoss Seam**, etc, "Que son frameworks que ayudan a seperar bien las distintas capas y proveen ya muchos de los servicios para distintas capas de aplicación, que sino lo deberíamos codificar nosostros mismos"*

El motor de las páginas JSP, está basado en los servlets de Java.

¿CÓMO TRABAJA UNA PÁGINA JSP?

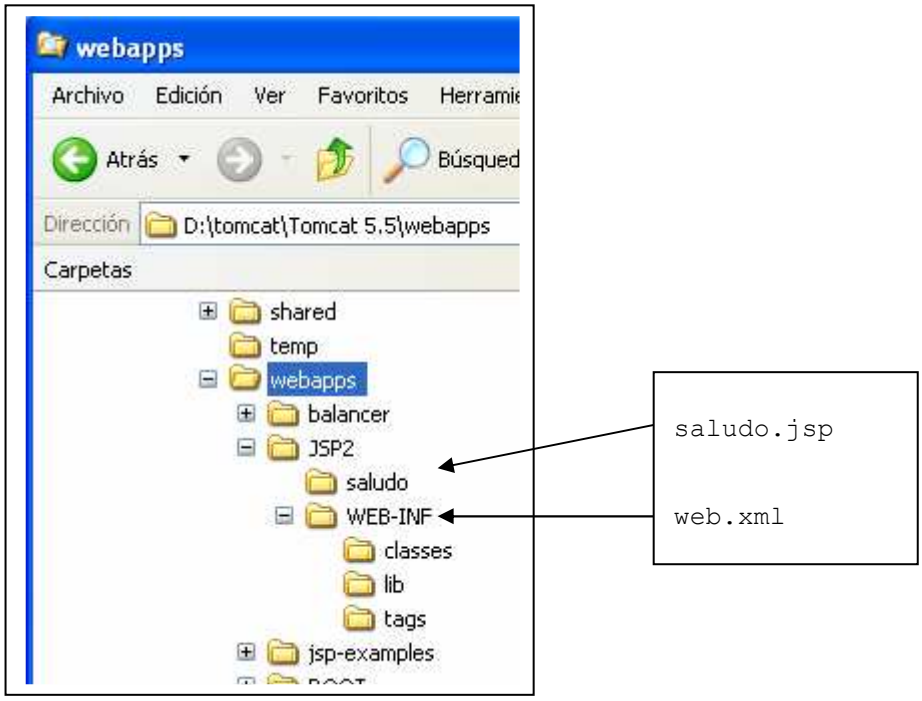
En la tecnología **JSP** creamos páginas de forma parecida a como se crean en **ASP** (Active Server Page de Microsoft) o **PHP** (Personal Home Page), otras dos tecnologías de páginas ejecutadas en el servidor. El código se almacena en ficheros con extensión .jsp que incluyen, dentro de la estructura de etiquetas HTML, las sentencias Java a ejecutar. Por ejemplo, supongamos que hemos escrito el siguiente código en el fichero Saludo.jsp:

```
<%@page contentType="text/html"%>
<html>
<head><title>Ejemplo JSP</title></head>
      <body bgcolor="ffffcc">
            <center>
            <h2>Hola mundo</h2>
                  <%java.util.Date hoy = new java.util.Date();%>
                  La fecha de hoy es: <%=hoy%>
            </center>
      </body>
</html>
```

Este ejemplo muestra HTML tradicional y un fragmento de código Java (***un scriptlet***). La etiqueta **<%** identifica el comienzo del scriptlet y **%>** el final.
En el scriplet <%java.util.Date hoy = new java.util.Date();%> tenemos una **sentencia Java** donde la referencia a objeto Date **hoy** es instanciada con el retorno del comando new, la fecha actual.
Para ejecutar esta página desde un explorador (o cualquier otra página JSP), hay que instalarla en el servidor de aplicaciones análogamente a como lo hacíamos con los servlets.

Y como se hace esto?

1. En la jerarquía de carpetas < TOMCAT_HOME>*webapps**añada MiCarpeta*, donde *MiCarpeta* va a ser **JSP2**, por ejemplo, y construya bajo ella la jerarquía de carpetas saludo,WEB-INF, classes, lib, tags. Pegue en las carpetas indicadas los archivos web.xml y saludo.jsp. Le debe quedar algo así:



2. Edite en la carpeta WEB-INF el descriptor de despliegue web.xml indicado a continuación. Este archivo indica dónde se localiza el esquema XML para la especificación Servlet 2.4 que a su vez utiliza el esquema XML para la especificación Java ServerPage 2.0.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
      xmlns=http://java.sun.com/xml/ns/j2ee
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi :schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
      version="2.4">
</web-app>
```

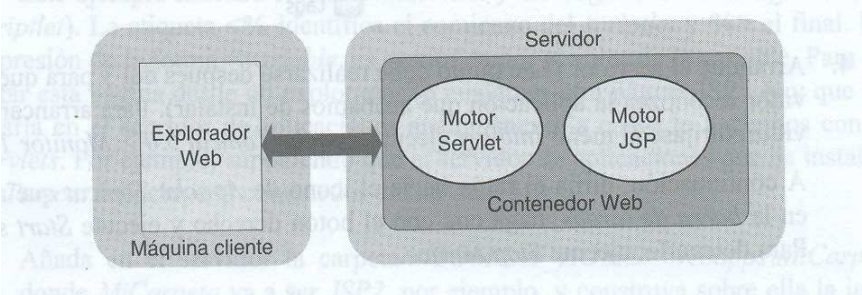
4. Arranque el Tomcat, como ya lo hemos hecho cuando estudiamos servlets.

5. Ejecute la página JSP. Para ello, inicie el explorador y escriba el URL según la sintaxis siguiente:

http://localhost:8080/JSP2/saludo/Saludo.jsp

Que ocurre ahora ? Distinguimos 2 situaciones:

- **Primera vez o página fuente modificada (motor JSP)**
  - El motor JSP traduce el fuente .jsp a fuente .java
- **Veces siguientes (Motor Servlet)**
  - El fuente java es compilado, se genera el .class; A partir de este instante el motor de servlets actuará igual que cuando ejecuta cualquier otro servlet



Nota: *dada la extensión de lo desarrollado a partir de las JSP a partir de este punto nos limitaremos a algunos conceptos, con un mínimo de detalles técnicos. Consideramos que su desarrollo en código es tema de otra asignatura de programación completa*

Ciclo de vida de una página JSP

Volviendo al ejemplo anterior, el fichero .java generado por Tomcat a partir del fichero Saludo.jsp se localiza en la carpeta work\Catalina\llocalhost\l... de la instalación de Tomcat.

La clase generada a partir de la página JSP se deriva de HttpJspBase, que a su vez implementa la interfaz Servlet. En dicha clase existirá un metodo **_JspService** que esencialmente contiene las líneas de la página JSP correspondientes al HTML dinámico que se generará, y que **no se puede sobrescribir**. En cambio, si es posible escribir código de iniciación y de finalización añadiendo los métodos _jsp**Init** y _jsp**Destroy**, respectivamente. Todos estos métodos tienen la misma función que la descrita para los métodos respectivos de los servlets.

Objetos implícitos

La tecnología JSP utiliza una serie de objetos que también están disponibles para su uso durante la implementación de una página JSP, son los siguientes:

. **pageContext**. Se obtiene mediante getPageContext y es util para obtener otros objetos; metodos de este objeto me permiten obtener otros, algunos ya vistos cuando estudiamos servlets: **request**, **response**, **application**, **config**, **session**, **out**, **page**, **exception**.

Ámbito de los atributos

En el capítulo de "servlets" vimos que un objeto HttpSession se creaba en el servidor y era asociado automáticamente con la petición actual. También vimos que este objeto tenía una estructura de datos que permitía almacenar un número de claves y sus valores asociados. Una clave es un atributo de la sesión, cuyo valor se puede leer con el método getAttribute y se puede fijar con el método setAttribute del objeto HttpSession (la sesión). Pues bien, los objetos ServletContext y PageContext también ofrecen atributos que pueden ser fijados y leídos de la misma forma, mientras una petición de un servicio se está procesando.

Además, los atributos pueden también utilizarse para asociar datos con varios objetos en el contenedor Web, afirmación que introduce el concepto de **ámbito de un atributo.**

El ámbito de un atributo determina la duración y visibilidad de los atributos de los objetos HttpSession, ServletContext o PageContext de la capa Web. Los cuatro tipos de ámbito, de más a menos restrictivos, son: página (page), petición (request), sesión (session) y aplicación (application).

Las páginas JSP pueden acceder a los atributos definidos en un ámbito utilizando los métodos de la interfaz PageContext. En este contexto, podemos fijar y leer atributos utilizando los métodos setAttribute y getAttribute. Por ejemplo, la línea siguiente fija en el objeto implícito pageContext el atributo identificado por nombre con el valor y ámbito especificados:

```
pageContext.setAttribute(nombre, valor, ámbito);
```

En cada uno de los cuatro ámbitos, los métodos para fijar y leer atributos son, respectivamente, setAttribute y getAttribute y el ámbito del atributo queda determinado por la interfaz utilizada para acceder al mismo.

Ámbito de aplicación

Los atributos fijados y leídos utilizando la interfaz ServletContext a través del objeto implícito application tienen ámbito de aplicación; esto quiere decir que son visibles por todos los componentes de la capa Web durante todo el período de vida de la aplicación. Esto es, si la aplicación se desinstala (undeployed) o se reinicia, los atributos se pierden.

Ámbito de sesión

Los atributos fijados y leídos utilizando la interfaz HttpSession a través del objeto implícito session tienen ámbito de sesión; esto quiere decir que son visibles sólo por el cliente HTTP que inició la sesión, para múltiples peticiones y hasta que la sesión finalice. Esto es, si la sesión finaliza automáticamente porque transcurrió el tiempo fijado por el servidor, si es cancelada manualmente, si la aplicación se desinstala (undeployed) o se reinicia, los atributos se pierden.

Ámbito de petición

Los atributos fijados y leídos utilizando la interfaz HttpServletRequest a través del objeto implícito request tienen ámbito de petición; esto quiere decir que son visibles sólo por el componente de la aplicación que está sirviendo la petición HTTP, y sólo hasta que ésta sea servida; esto es, los atributos para una única petición se pierden antes de que la respuesta sea enviada al cliente. Cuando una petición que está siendo procesada por un componente de la capa Web es reenviada a otro, los atributos son también visibles para este segundo, y así sucesivamente.

Ámbito de página

Los atributos fijados y leídos utilizando la interfaz PageContext a través del objeto implícito page tienen ámbito de página; esto quiere decir que son visibles sólo en el contexto de una única página JSP, y hasta que la página sea procesada. En particular, no son visibles a través de peticiones reenviadas.

Cuándo utilizar uno u otro ámbito ?

En general, hay que intentar utilizar el ámbito más específico posible al contexto en el que estemos trabajando. Esto es, si un dato es utilizado sólo en una página, utilice el ámbito de página; si es utilizado en más de una página, utilice el ámbito de petición; si es utilizado a través de múltiples peticiones, utilice el ámbito de sesión; y si es utilizado a través de distintas sesiones, utilice el ámbito de aplicación.

Ejemplo

En la página **Cuenta.jsp** mostrada a continuación, *(y que realiza la misma funcion que vimos cuando estudiamos Cookies)* se puede observar cómo se hace un seguimiento del número de veces que un cliente accede a esta página a través de distintas sesiones. Para ello, suponemos que dicha cuenta es el valor del atributo atrCuenta del objeto implícito **application**. La primera vez que el cliente acceda a esta página el valor de atrCuenta será null, porque aún no ha sido fijado este atributo; en este caso, lo fijaremos con el valor 1. En los siguientes accesos, se leerá el valor del atributo atrCuenta, se incrementará en una unidad y se volverá a fijar. Finalmente, se envía la respuesta al cliente para que muestre al usuario el valor del contador.

```
<%@page contentType="text/html"%>
   <html>
      <head><title>Página JSP cuenta</title></head>
```

```
    <body>
          <!--Incrementar el contador para esta página. El valor es
           guardado en el objeto implícito application con el nombre
           "atrCuenta".
          -->
          <%
            Integer cuenta = (Integer)application.getAttribute("atrCuenta");
            String s = " vez.";
            if (cuenta == null) cuenta = new Integer(1);
            else{
                cuenta = new Integer(cuenta.intValue() + 1);
                s = " veces.";
            }
            application.setAttribute("atrCuenta", cuenta);
          %>
          <h1>Demostración de seguimiento a nivel de aplicación</h1>
          <!-- Visualizar la cuenta para esta página -->
          Has visitado esta página <%=cuenta%> <%=s%>
    </body>
  </html>
```

Este ejemplo muestra HTML tradicional, un fragmento de código java delimitado por las etiquetas <% y %> (un scriptlet), expresiones de la forma =variable para acceder al valor de la variable y comentarios delimitados por <!-- y -->.

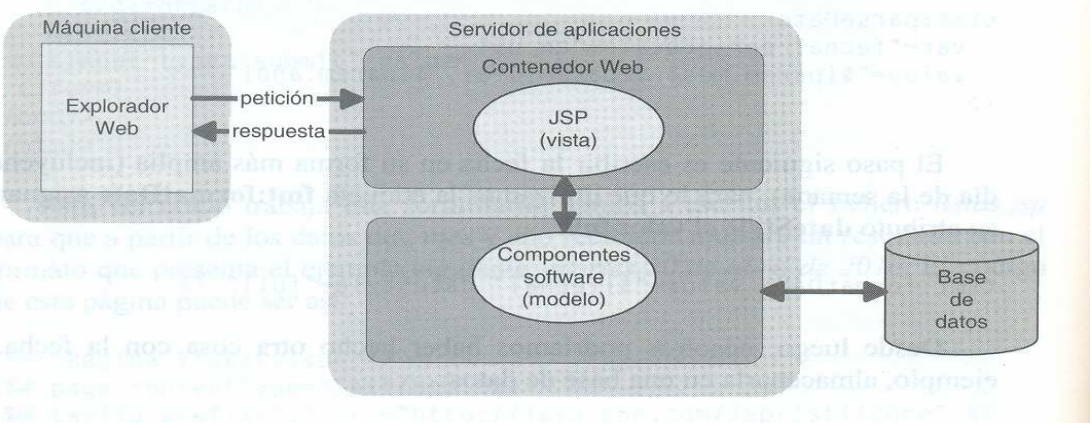
El atributo atrCuenta tiene ámbito de aplicación. Como ejercicio puede sustituirlo sucesivamente por los ámbitos de sesión, petición y página, y comparar los resultados. Para las pruebas, abra varias ventanas del explorador simultáneamente. (Como se hizo en el estudio de Cookies)

Finalizando introducción JSP … **APLICACIONES WEB UTILIZANDO JSP**

La especificación JSP utiliza básicamente dos arquitecturas para la construcción de aplicaciones Web utilizando páginas JSP: arquitectura modelo 1 y modelo 2. Ambas arquitecturas, como veremos a continuación, difieren en el sitio donde el proceso tiene lugar.

Modelo 1

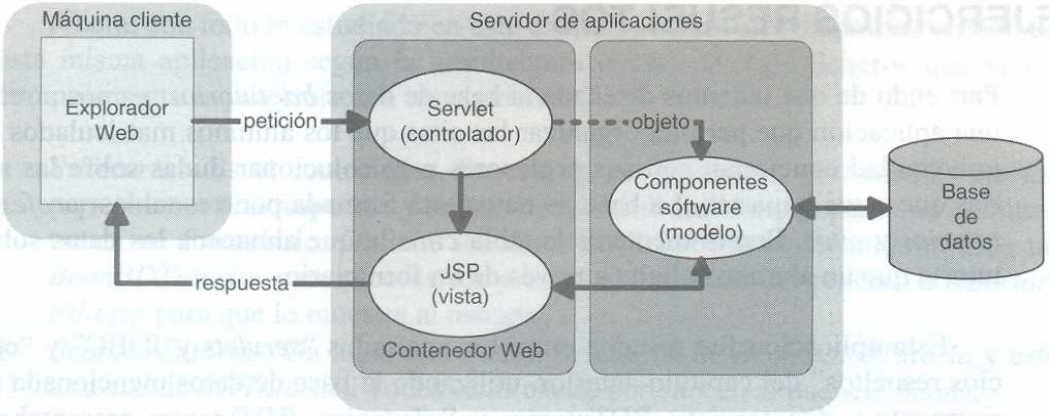
En la arquitectura modelo 1, según se puede observar en la figura siguiente, la petición procedente del explorador Web es enviada directamente a la página JSP, la cual es responsable de procesarla y de responder al cliente. No obstante, la presentación está separada del procesamiento, ya que todo el acceso a los datos es llevado a cabo por componentes software (beans) incluyendo las bibliotecas de etiquetas. Es una arquitectura fácil de implementar; la página JSP estará diseñada en base a una plantilla formada por el contenido estático y llamadas a componentes software que permitan extraer el contenido dinámico. Todo ello conformará el resultado que hay que enviar al cliente.



La arquitectura modelo 1 es adecuada para aplicaciones simples, pero en aplicaciones complejas puede conducir a una cantidad significativa de proceso embebido en la página JSP, especialmente cuando hay que ejecutar bastantes peticiones, lo que la define como una arquitectura no escalable (muchas peticiones pueden requerir muchos recursos del servidor, lo que hará disminuir el rendimiento de la aplicación Web). Otra desventaja de esta arquitectura es que cada página JSP debe ser responsable de gestionar el estado de la aplicación, de la identificación y de la seguridad.

Modelo 2

La arquitectura modelo 2, según se puede observar en la figura siguiente, integra la utilización de servlets y páginas JSP. Las páginas JSP son utilizadas para la capa de presentación, y los servlets para la de procesamiento. El servlet actúa como un controlador responsable de procesar las peticiones y de crear cualquier componente software (beans) necesario para la página JSP; también es responsable de decidir a qué página JSP se envía la petición. La página JSP recupera los objetos creados por el servlet y extrae el contenido dinámico para insertado en una plantilla, la que se enviará al explorador.



La ventaja de esta arquitectura es que no hay procesamiento dentro de la capa de presentación; ésta simplemente es responsable de recuperar cualquier objeto que haya sido creado previamente por el controlador y extraer el contenido dinámico e insertarlo en la plantilla estática que se enviará al cliente para su presentación. Consecuentemente, esta separación favorece el trabajo en equipo entre desarrolladores y diseñadores de páginas. Otra ventaja de este modelo es que el controlador presenta un único punto de entrada en la aplicación, además de responsabilizarse de gestionar el estado de la aplicación, de la identificación, de la seguridad y de la presentación, lo que facilita su mantenimiento.

Finalmente, decir que esta arquitectura fomenta el uso del **patrón de diseño modelo - vista - controlador (MVC)**, donde el modelo lo forman los componentes que controlan los datos que utiliza la aplicación, la vista los que presentan los datos al cliente, y el controlador lo forman los componentes responsables de la gestión de eventos y de coordinar las actividades del modelo y de la vista.

La ventaja de utilizar la arquitectura MVC es que no hay lógica de procesamiento en el componente de presentación (la vista); éste es responsable simplemente de extraer los datos necesarios utilizando los componentes precisos (por ejemplo, componentes JavaBean o tags). La desventaja es la dificultad para implementar en el modelo la notificación a la vista para su actualización de los cambios que se vayan sucediendo en el mismo. Por eso, para aplicaciones sencillas es bastante habitual utilizar el modelo 1. En este caso, como sabemos, sería la página JSP la que procesaría la petición y enviaría la respuesta al cliente.

A continuación y un poco a modo de cierre de esta Unidad y de la asignatura, incorporo aquí el trabajo realizado por el ayudante alumno Salvador Celia. Es la misma aplicación **Hipermercados TodoTodo**, (pgs 76/80), pero ahora usando JSP y aplicando algunas optimizaciones por el descriptas en su mail.

Date: Mon, 25 Jun 2007 02:34:01 +0000
From: Salvador Celía <salvadorcelia@hotmail.com>
To: tymos@fcm.unc.edu.ar
Subject: Buenas!

Aca le adjunto el ejercicio del HipermercadoTodoTodo, adaptado en un JSP y con algunas modificaciones, variantes para aprovechar al maximo el potencial de los JSP's. Bueno tambien le adjunto un .doc donde detallo una breve explicacion de la librería JSTL que uso para lograr iterar una lista de menues disponibles en la clase Carta.java. Bueno esto y un .pdf que le incluyo (hasta la pagina 7 inclusive), creo que deberian ser includidos en el apunte como para hacer entender un poco mas la pontecialidad de los JSP's (sus directivas) que es lo que falta para mi en el apunte.

Bueno, le cuento que ya estoy mejor y comenze a trabajar, asi que por eso solamente pude terminar este ejercicio hoy, y vere que puedo hacer mas adelante porque ahora me pongo a estudiar para finales y con el tema del trabajo no tengo mucho tiempo!

Saludos y nos estamos comunicando cualquier cosa!

Salvador A. Celía

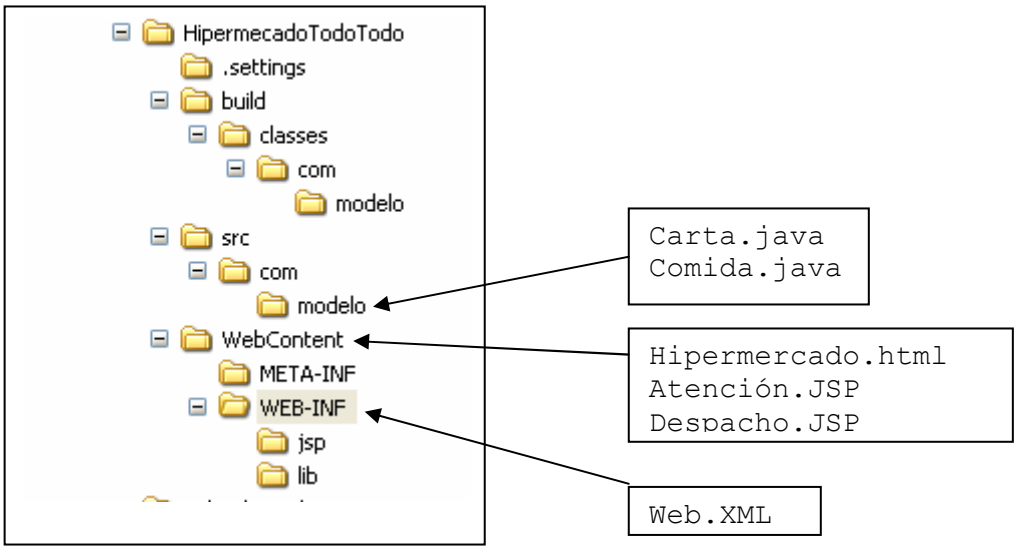
Attachment 1: HipermecadoTodoTodo.rar (513KB) Delete WebDisk	
Type: application/octet-stream Encoding: base64	Download

Attachment 2: JavaServer Pages Standard Tag Library (JSTL).doc (62KB) Delete Preview WebDisk	
Type: application/msword Encoding: base64	Download

Attachment 3: Introduccion JSP.pdf (942KB) Delete WebDisk	
Type: application/pdf Encoding: base64	Download

Delete all non-text attachment(s)

Attachment 1 - HipermecadoTodoTodo.rar (513KB) (Estructura)



Y sus contenidos son (en el mismo orden de los cuadros de texto)

Carta.java

```
package com.modelo;
import java.util.ArrayList;
public class Carta {

	/*<Comida> ésta expresion usada de la forma que se muestra, permite
	 * asegurar que los datos que se van a almacenar van a ser compatibles
	 * con un determinado tipo, en este caso "Comida". A ésto se lo llama
	 * "Generics" y es utilzado en las últimas versiones del jdk(5.0 en
	 * adelante)*/

	private ArrayList carta=new ArrayList<Comida>();

	public Carta() {
		carta.add(new Comida("Amburguesas, cja 10 Un", 10.00f));
		carta.add(new Comida("Fideos Macarron, pk 1 kg", 1.99f));
		carta.add(new Comida("Cerveza BadHein, env 780cc", 2.80f));
		carta.add(new Comida("Falda (Especial), kg,", 3.50f));
		carta.add(new Comida("Bife lomo tern, kg", 12.50f));
		carta.add(new Comida("Pan Frances kg", 2.00f));
		carta.add(new Comida("Polenta PrestPres 1/2 kg", 1.60f));
		carta.add(new Comida("Fideos Felippillo 1/2 kg", 1.20f));
		carta.add(new Comida("Tinto FacuRelli, 1 lt", 2.80f));
		carta.add(new Comida("Salch. VienAus, pk 200 grs", 2.70f));

	}

	public ArrayList getCarta() {
		return carta;
	}

	public void setCarta(ArrayList carta) {
		this.carta = carta;
	}

	public Comida getComida(int index) {
		return (Comida)carta.get(index);
	}

}
```

Comida.java

```
package com.modelo;
public class Comida {
	private float precio;
	private String descripcion;

	public Comida(String desc, float precio){
		this.precio=precio;
		descripcion=desc;
	}

	public String getDescripcion() {
		return descripcion;
	}
	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}
	public float getPrecio() {
		return precio;
	}
	public void setPrecio(float precio) {
		this.precio = precio;
	}
}
```

Hipermercado.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>HIPERMERCADOS</title>
</head>
<body>
<h1 align="center">Tenes hambre?...</h1>
<form action="Atencion.jsp">
<table border="1" align="center">
<tr>
<td><h3 align="right">Tu Nombre</h3></td>
<td><input type="text" name="hambriento" size="20"></td>
</tr>
<tr>
<td><h3 align="right">Elegir Hypermercado</h3></td>
<td>
<select name="hiper">
<option selected="selected" value="TodoTodo">TodoTodo</option>
<option  value="Libertad">Libertad</option>
<option value="Carrefur">Carrefur</option>
</select>
</td>
</tr>

<tr>
<td colspan="2"><input type="submit" value="Atender ya" align="middle"></td>
</tr>
</table>

</form>
</body>
</html>
```

Atención.JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1" isELIgnored="false"%>
<%@page import="java.lang.String"%>
<%@page import="com.modelo.Carta"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core-rt" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%
      session = request.getSession(true);
      String hambriento =
((String)request.getParameter("hambriento")).toUpperCase();
      if(hambriento==null || hambriento.equals(""))
            hambriento="Sr.";
      session.setAttribute("hambriento",hambriento);
      String hiper = request.getParameter("hiper");
      session.setAttribute("hiper",hiper);
      Carta carta = new Carta();
      request.setAttribute("carta2",carta.getCarta());
%>
<html>
<head>

<script type="text/javascript">
<!--Para realizar el calculo del Menu!-->
function calcularMenu(form){

total=0E-32;
for(i=0; i<form.menu.length; i++){
if(form.menu[i].checked)
 total+=parseFloat(form.menu[i].value);
}

form.total.value="$"+total;
for(i=0; i<form.menu.length; i++)
 if(form.menu[i].checked)
      return true;
 return false;
}

function validarCheckBox(form){
for(i=0; i<form.menu.length; i++)
 if(form.menu[i].checked){
   return true;
   }
alert("Seleccionar algun alimento o presione el boton 'Atrás'")
return false;
}
</script>


<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>HIPERMERCADO - <%=hiper%></title>
</head>
<body>
<form action="Despacho.jsp" method="get" name="form" onsubmit="return
validarCheckBox(this)">
<h1 align="center"><font size="5"> Hola <%=hambriento%>!!!<br>
<font size="4"> "<%=hiper%>" te puede ofrecer<br>
entre muchos otros,<br>
este apetitoso combo: </font></h1>
<br>
```

```
<font size=" + 3 + ">
<table border="1" align="center">
      <tr>
            <td>
            <h3>Productos</h3>
            </td>
            <td>
            <h3>Precio</h3>
            </td>
      </tr>
<c:forEach var="comida" items='${requestScope.carta2}'>  <!-- Utilizamos
c:forEach el tag de JSTL que nos sirve para iterar la lista de menues que
tenemos! -->
    <tr>
    <td>${comida.descripcion}<!-- Aquí utlizamos codigo EL para poder acceder a
las propiedades del obejeto  --></td>
    <td>${comida.precio}</td>
    <td><input type="checkbox" name="menu" id='${comida.descripcion}'
value='${comida.precio}' onclick="calcularMenu(this.form);"/></td>
    </tr>
</c:forEach>
      <tr>
            <td>
      <b>Impote Total</b>
            </td>
            <td>
            <input id="total" type="text" name="total" align="right"
readonly="readonly">
            </td>
      </tr>
      <tr>
      <td colspan="2"><input type="submit" value="Despachar" align="right"></td>
      </tr>
</table>
</font>
</form>
<input type="button" value="Atrás"  onclick="javascript:window.history.back();"
align="left"/>
</body>
</html>
```

Despacho.JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script type="text/javascript">
function campoVacio(form){
var mensj1,mensj2,mensj3;
mensj1='<%=(String)session.getAttribute("hambriento")%>'; <!--Tomo de la session
del usuario el atributo 'hambriento'(el nombre ingresado por el usuario)-->
var band=true;

if(form.direccion.value == null || form.direccion.value == "")
 mensj2="\n-Debe colocar una dirección. ";
if(form.telefono.value == null || form.telefono.value == ""){
 mensj3="\n-Debe colocar un telefono. ";
} else if((isNaN(parseInt(form.telefono.value)))) <!--Compruebo si es un valor
numérico con la funcion isNaN()-->
           mensj3="\n-Debe colocar un telefono válido. ";
if(mensj2!=null){
alert("Sr. "+ mensj1 + mensj2);
```

```
band=false;
}else if(mensj3!=null){
alert("Sr. "+ mensj1 + mensj3);
band=false;
}else alert("Sr. "+ mensj1+"\n su pedido ya se ha solicitado.\n Y la demora es
de 30 minutos (aprox).\nGracias por su compra!");

return band;
}

</script>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>HIPERMERCADO - <%=(String) session.getAttribute("hiper")%> <!--Tomo de la
session del usuario el atributo 'hiper'(el hipermercado seleccionado por el
usuario)--></title>
</head>
<body>
<h1 align="center">Complete los Datos</h1>
<form action="Hipermercado.html" name="form" method="post" onSubmit="return
campoVacio(this)" >
<table align="center" border="1">
      <tr>
            <td>
            <h3>Escriba su direccion</h3>
            </td>
            <td><input type="text" name="direccion" /></td>
      </tr>
      <tr>
            <td>
            <h3 align="right">Telefono</h3>
            </td>
            <td><input type="text" name="telefono"></td>
      </tr>
      <tr>
            <td colspan="2"><input type="submit" value="Enviar"/></td>
      </tr>
</table>
</form>
<input type="button" value="Atras" onclick="javascript:window.history.back();"
align="left"/>
</body>
</html>
```

Web.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
      xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
      <display-name>HipermecadoTodoTodo</display-name>

      <servlet>
            <description></description>
            <display-name>Atencion2</display-name>
            <servlet-name>Atencion2</servlet-name>
            <servlet-class>clases.Atencion2</servlet-class>
      </servlet>
      <servlet>
            <description></description>
            <display-name>Despacho</display-name>
            <servlet-name>Despacho</servlet-name>
            <servlet-class>clases.Despacho</servlet-class>
      </servlet>
```

```
	<servlet-mapping>
		<servlet-name>Atencion2</servlet-name>
		<url-pattern>/Atencion2</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>Despacho</servlet-name>
		<url-pattern>/Despacho</url-pattern>
	</servlet-mapping>
	<welcome-file-list>
		<welcome-file>index.html</welcome-file>
		<welcome-file>index.htm</welcome-file>
		<welcome-file>index.jsp</welcome-file>
		<welcome-file>default.html</welcome-file>
		<welcome-file>default.htm</welcome-file>
		<welcome-file>default.jsp</welcome-file>
	</welcome-file-list>
	<jsp-config>
		<taglib>
			<taglib-uri>http://java.sun.com/jsp/jstl/fmt</taglib-uri>
			<taglib-location>/WEB-INF/fmt.tld</taglib-location>
		</taglib>
		<taglib>
			<taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>
			<taglib-location>/WEB-INF/c.tld</taglib-location>
		</taglib>
		<taglib>
			<taglib-uri>http://java.sun.com/jsp/jstl/core-rt</taglib-uri>
			<taglib-location>/WEB-INF/c-rt.tld</taglib-location>
		</taglib>
		<taglib>
			<taglib-uri>http://java.sun.com/jsp/jstl/sql</taglib-uri>
			<taglib-location>/WEB-INF/sql.tld</taglib-location>
		</taglib>

		<taglib>
			<taglib-uri>http://java.sun.com/jsp/jstl/x</taglib-uri>
			<taglib-location>/WEB-INF/x.tld</taglib-location>
		</taglib>
	</jsp-config>
</web-app>
```

Attachment 2 - JavaServer Pages Standard Tag Library (JSTL)

1. **Objetivo:**
   - **Simplificar y agilizar el desarrollo de aplicaciones web**
2. **3ra iteración después de servlets y JSPs**
3. **Sirven para la generación dinámica de páginas web**
4. **Asumimos que ya has instalado Tomcat 5 en tu máquina, si no consíguelo de:**
   - **http://apache.rediris.es/jakarta/tomcat-5/v5.5.7/bin/jakarta-tomcat-5.5.7.zip**
5. **Bajar JSTL 1.1 de:**
   - **http://apache.rediris.es/jakarta/taglibs/standard/binaries/jakarta-taglibs-standard-1.1.2.zip**
6. **JSTL 1.1 es una pequeña mejora de JSTL 1.0 creada para alinear JSTL con JSP 2.0.**
7. **Antes había una versión de cada librería dependiendo de si utilizabamos expresiones EL o Java, ahora es la misma librería.**
   - **Ha cambiado el nombre de los identificadores de las librerías, se ha añadido un elemento del path /jsp a todos ellos**
8. **Para cualquier aplicación web desde la cual quieres usar JSTL, copiar los ficheros .tld al directorio WEB-INF de tu aplicación web.**

9 **Edita el web.xml de tu aplicación web añadiendo las siguientes entradas**

```
<taglib>
 <taglib-uri>http://java.sun.com/jsp/jstl/fmt</taglib-uri>
 <taglib-location>/WEB-INF/fmt.tld</taglib-location>
</taglib>

<taglib>
 <taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>
 <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>

<taglib>
 <taglib-uri>http://java.sun.com/jsp/jstl/sql</taglib-uri>
 <taglib-location>/WEB-INF/sql.tld</taglib-location>
</taglib>

<taglib>
 <taglib-uri>http://java.sun.com/jsp/jstl/x</taglib-uri>
 <taglib-location>/WEB-INF/x.tld</taglib-location>
</taglib>
```

1 **Estas entradas permiten a tu aplicación web usar las librerías de etiquetas JSTL que usan el lenguaje de expresiones. La posición de estas entradas tiene importancia.**
2 **Las páginas JSTL son también páginas JSP. JSTL es un superconjunto de JSP.**
3 **JSTL provee un conjunto de cinco librerías estándar:**
   - **Core**
   - **Internationalization/format**
   - **XML**
   - **SQL y**
   - **Funciones**
4 **Además JSTL define un nuevo lenguaje de expresiones llamado EL, que ha sido luego adoptado por JSP 2.0**
5 **Una etiqueta JSTL corresponde a una acción; llamándolas acción nos indica que añaden comportamiento dinámico a una, de otra manera, página estática.**
6 **El lenguaje de expresiones EL simplemente define un poderoso mecanismo para expresar expresiones simples en una sintáxis muy sencilla.**
   - **Es algo entre JavaScript y Perl.**
   - **Su combinación con las etiquetas de las 4 librerías antes mencionadas proveen mucha flexibilidad y poder para el desarrollo de páginas dinámicas.**
7 **En EL las expresiones están delimitadas por ${ }.**
8 **Algunos ejemplos del uso de EL son:**
   - **${anExpression}**
   - **${aList[4]}**
   - **${aList[someVariable]} → acceso a un elemento de una colección**
   - **${anObject.aProperty} → acceso a la propiedad de un objeto**
   - **${anObject["aPropertyName"]} → entrada en un mapa con propiedad aPropertyName**
   - **${anObject[aVariableContainingPropertyName]}**
9 **Existen una serie de variables implícitas definidas en EL:**
   - **pageContext: el contexto del JSP actual**
   - **pageScope, requestScope, sessionScope, and applicationScope: colecciones de mapas que mapean nombres de variables en esos contextos a valores**

- **param and paramValues: parámetros pasados con la petición de la página, lo mismo que en JSP**
- **header and headerValues: cabeceras pasadas en la petición de la página**
- **cookie: mapa que mapea nombres de cookies a los valores de las mismas**

Librería	URI	Prefijo Librería
Core	http://java.sun.com/jsp/jstl/core	c
Internationalization I18N formateo	http://java.sun.com/jsp/jstl/fmt	fmt
SQL/DB support	http://java.sun.com/jsp/jstl/sql	sql
Procesamiento XML	http://java.sun.com/jsp/jstl/xml	x
Functions	http://java.sun.com/jsp/jstl/functions	fn

1. **La siguiente directiva ha de incluirse al comienzo de la página:**

```
<%@ taglib prefix="c" uri=http://java.sun.com/jsp/jstl/core %>
```

1. **Para utilizar una etiqueta de una librería simplemente se ha de preceder con el prefijo de la librería utilizada:**

```
<c:out value="${anExpression}"/>
```

1. **Permiten llevar a cabo las siguientes acciones:**
   - **Visualizar/asignar valores y manejar excepciones**
   - **Control de flujo**
   - **Otras acciones de utilidad**
2. **Javadoc de JSTL APIs es disponible en:**

http://www.jcp.org/aboutJava/communityprocess/final/jsr052/

Apéndice A

INSTALACION DEL CONTENEDOR DE SERVLET/JSP TOMCAT 5

Tomcat 5.X es un servidor de aplicaciones que implementa las tecnologías Java Servlet 2.4 y JavaServer Pages 2.0. Puede obtenerlo de la dirección de Internet **http://tomcat.apache.org**

Ya en la página, a la izquierda, downloads, Opte por Tomcat 5.x

Más abajo, Binary Distributions,
opte por core,
Windows Service Installer (pgp, md5)

Indique el contenedor donde desea bajar el instalador, es un ejecutable self-extracting, algo como apache-tomcat-5.5.23.exe

Suponiendo que ya tiene instalado J2SE 5.0, en cualquiera de sus versiones, la instalación sobre Windows de Tomcat 5.X es muy sencilla:
1. Verifique que la variable de entorno JAVA_HOME esté con su valor correcto. Debe contener la ruta del jdk o jre que UD esta usando. Por ejemplo, si tiene:

C:\Archivos de programa\Java\jre1.5.0_11\bin …

JAVA_HOME debe contener C:\Archivos de programa\Java

Para hacer esta verificación, <Mi PC>, <Panel de Control>, <Sistema>, <Caracteristicas avanzadas>, <Variables de entorno>

Opte por JAVA_HOME, en la parte superior de la ventanilla aparecen 2 campos de texto, en el superior tenemos el nombre, en el inferior podemos corregir su valor.

1. Ejecute el fichero apache-tomcat-5.5.23.exe que acaba de descargar

2. Siga los pasos indicados durante la instalación.

3. Una vez instalado, asegúrese de que la biblioteca <JAVA_HOME>/(jdk o jre)/lib/**tools.jar** está en <TOMCAT_HOME>/common/lib para que las aplicaciones Web se puedan compilar; si no estuviera, cópiela.

4. Asimismo, para poder utilizar la biblioteca de etiquetas JSTL 1.1, debe copiar en <TOMCAT_HOME>/common/lib los ficheros **jstl.jar y standard.jar**. Estos puede obtenerlos en la dirección **http://jakarta.apache.org/site/binindex.cgi** (Dounloads - Taglibs > Standard 1.1 Taglibs > 1.1.2.zip).

5. Copie el fichero <TOMCAT_HOME>\common\lib**servlet-api.jar** en

<JAVA_ HOME>\jre\lib\ext

6. Edite el fichero <TOMCAT_HOME>\conj\web.xml y asegúrese que **no este comentarizado** el código que sigue.

```
<servlet>
      <servlet-name>invoker</servlet-name>
      <servlet-class>
            org.apache.catalina.servlets.lnvokerServlet
      </servlet-class>
      <init-param>
            <param-name>debug</param-name>
            <param-value>O</param-value>
      </init-param>
```

```
      <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
      <servlet-name>invoker</servlet-name>
      <url -pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

Este código, por seguridad, viene comentado. Es necesario, al menos en la fase de desarrollo, poder invocar invocar la ejecución de servlets anónimos de la forma:

http://servidor[:puertoJ/localizacion/servlet/nombre_servlet

Es importante notar que el administrador de aplicaciones de Tomeat sólo podrá ser utilizado cuando el servidor esté arrancado. El procedimiento a seguir para correr aplicaciones servlet lo vemos a continuación.

Apéndice B

EJECUTAR UN SERVLET EN EL SERVIDOR

Para ejecutar un servlet como el del ejemplo anterior, teniendo instalados los paquetes J2SE y Tomcat 5, siga los pasos siguientes.

1. Compile el servlet. Para ello utilice su IDE habitual.
   Obtiene un fichero nombreServlet.class
2. Añada el fichero nombreServlet.class resultante a la carpeta
   <TOMCAT_HOME>\webapps\ROOT\WEB-INF\classes.

3. Levante Tomcat. Para ello, menú Inicio, Todos los programas, <Apache Tomcat 5.0>, <Monitor Tomcat>. Verá que se añade el icono Apache Tomcat en el área de notificación de la barra de tareas. Haga clic con el botón derecho del ratón sobre este icono y ejecute la orden **Start service** del menú contextual que se visualiza. (Cuando desee detenerlo, ejecute **Stop service**).

4. Una vez arrancado Tomcat, si desea verificar si está funcionando correctamente abra el explorador web y busque por la siguiente dirección:

   http://localhost:8080

   El explorador debe mostrar la página de Tomcat.

5. Ejecute el servlet. Para ello, en su navegador Web escriba el URL según la sintaxis siguiente:

http://servidor[:puerto]/servlet/nombreServlet

Por ejemplo: **http://localhost:8080/servlet/HolaMundo**

Modificaciones en los .class

En la forma que estamos trabajando, Tomcat **no se entera** automáticamente de los cambios que UD. va introduciendo en las clases que está trabajando. Para que ello suceda, antes de volver a probar los servlets o las clases que ellos usan, debemos ejecutar 2 pasos:

- Añadir los .class resultantes a la carpeta <TOMCAT_HOME>\webapps\ROOT\WEB-INF\classes.

- Haga clic con el botón derecho del ratón sobre el icono Apache Tomcat en el área de notificación de la barra de tareas y ejecute las ordenes **Stop service**/**Start service** del menú contextual que se visualiza. Esto es para que Tomcat refresque su vision de los .class a procesar