FACULTAD DE INGENIERIA
INGENIERIA WEB II
Sockets - Thread - Streams - Téorico - Ejemplos y Actividades
Docente: Ing. Carlos Simes

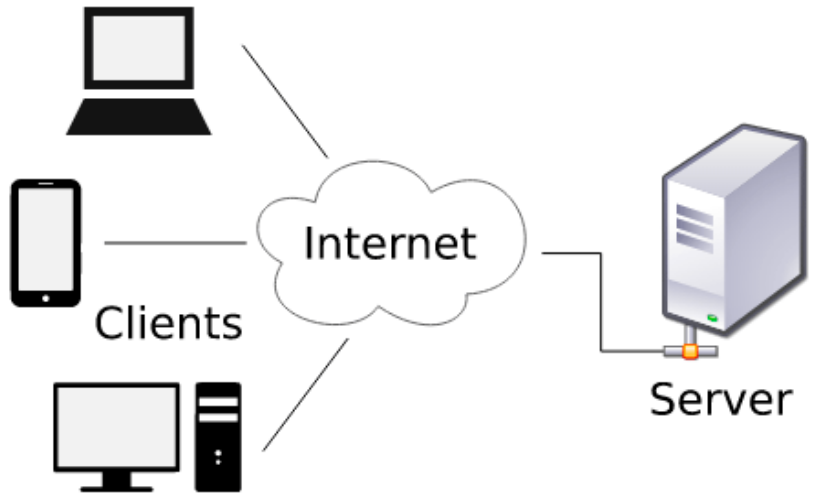
Introducción:

Antes de avocarnos a la realización de nuestro primer servidor web, vamos a analizar algunas cuestiones técnicas que conceptualmente son importantes para comprender lo que haremos.

En una aplicación web, tenemos dos partes que necesitan conectarse el cliente (una pagina web y en un desarrollo más complejo, todo un conjunto de paginas HTML + Css + Angular o algún otro frame, que se despliegan en el navegador web y un conjunto de recursos remotos que se encuentran en un sitio distante y que son provistos por un servidor.

Para hacerlo más entretenido, vean este video de 5 minutos para introducirse en el tema:

INTRODUCCIÓN A LA ARQUITECTURA CLIENTE SERVIDOR



Para que estas dos partes, el cliente y servidor puedan comunicarse entre sí e intercambiar información va a ser necesario establecer resolver 4 cuestiones básicas.

Debemos establecer:

1. Como se comunicará el cliente con el servidor
2. De qué manera se enviará y recibirá la información (flujo de los datos)
3. Que protocolo manejarán el cliente y el servidor para comunicarse y entenderse
4. Como hará el servidor para procesar los pedidos y las respuestas para múltiples clientes, ya que or defecto es la concepción de un Webapp.

Para resolver estos items, va a depender del lenguaje de programación y de la arquitectura que adoptemos para construir nuestra app.

Como en nuestro caso, vamos a programar nuestro servidor usando el lenguaje java, la solución a estas cuestiones vendrá dada por los siguientes componentes:

Sockets: *para establecer el enlace entre el cliente y el servidor.*

Streams: *para gestionar el flujo de envío y recepción de la información.*

Threads: *para implementar un servicio multiusuario de manera que nuestro servidor pueda atender múltiples pedidos simultáneamente.*

Protocolo HTTP: *para definir la forma en que el cliente "conversará" con nuestro servidor*
Vamos a desarrollar en las próximas páginas, estos temas.

Lo primero que se nos ocurre pensar es que en la maquina que contiene el web server, además de tener dicha aplicación, hay otras aplicaciones ejecutándose como bases de datos, sistemas de backups, servicios de correo, etc.

Cuando el cliente envíe un pedido por un recurso a nuestro web server (entendiendo como webserver a la aplicación que provee los recursos solicitados, no el artefacto físico que denominamos servidor), esta solicitud debe llegar exactamente a esa aplicación, pues cualquier otra no sabría qué hacer con ella y por lo tanto, no obtendríamos ninguna respuesta.

Por otro lado, nuestro webserver deberá tener un mecanismo para recibir los pedidos y posteriormente procesarlos convenientemente. Esta recepción deberá ser en un punto exacto que signifique la entrada y salida de la información.

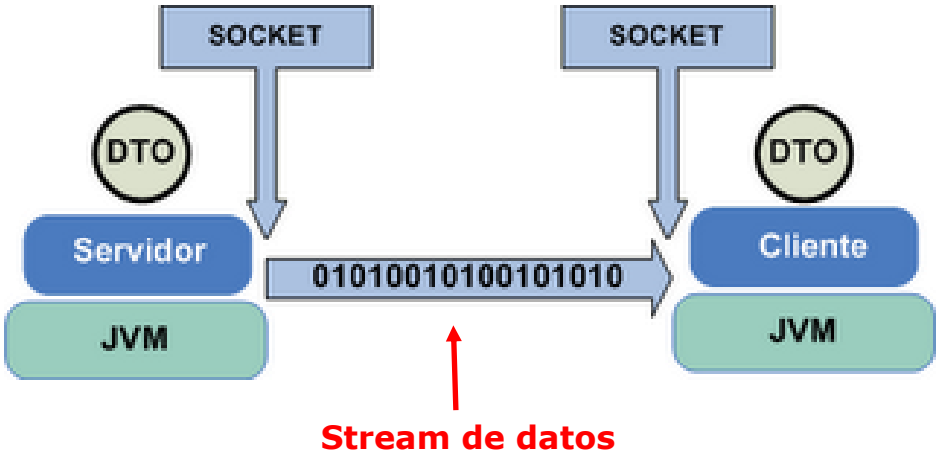
Para resolver esta primera cuestión utilizaremos lo que se denomina **Socket**

*Los **sockets** son un mecanismo que nos permite establecer un enlace entre dos programas que se ejecutan independientes el uno del otro (generalmente un programa cliente y un programa servidor).*

Java por medio de la librería **java.net** nos provee dos clases para lograr esto: **Socket** para implementar la conexión desde el lado del cliente y **ServerSocket** que nos permitirá manipular la conexión desde el lado del servidor.

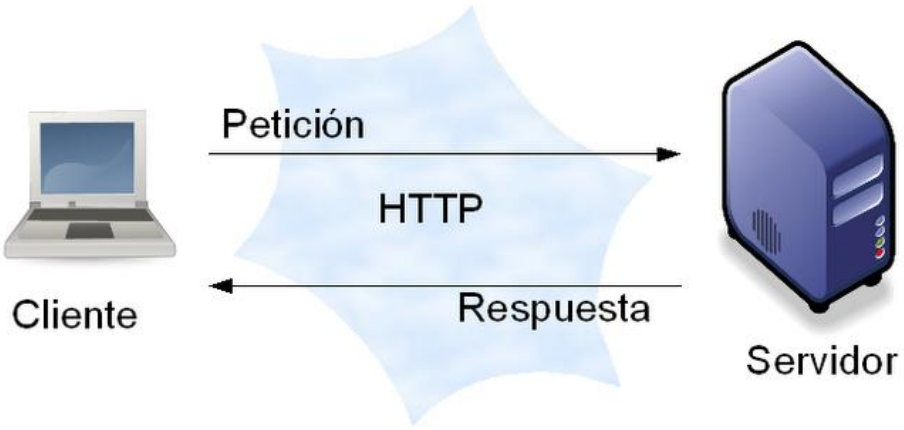
Aclaración: *En nuestro caso particular, nuestro cliente es web, por lo tanto no está programado en java y no será necesario, del lado cliente usar socket, pues recurriremos a otro mecanismo (**Formularios y escritura de URLs**) que explicaremos más adelante, pero como java sirve no solo para crear web apps, si quisiéramos hacer por ejemplo, un servicio de chat en una intranet, donde tanto los clientes como el servidor sean aplicaciones java ejecutándose en diferentes nodos de la red, entonces deberíamos usar para nuestros clientes la clase **Socket**.*

La dinámica será como se ve:



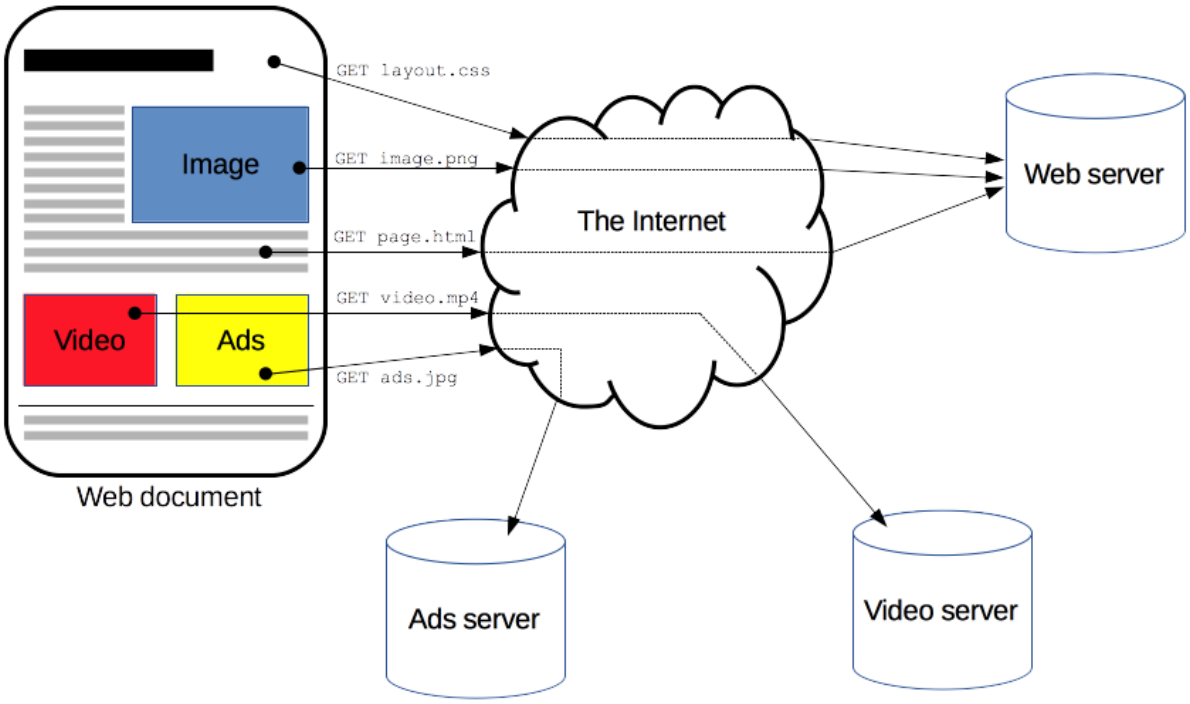
A su vez el cliente y el servidor deben establecer el modo en que se comunicaran, para ello emplearán algún tipo de protocolo, el más utilizado es el **protocolo HTTP**. (Protocolo de Transferencia de Hipertexto).

Un protocolo es un conjunto de reglas usadas para el intercambio de mensajes a través de una red, estos protocolos viajan en grupos denominados paquetes. Los protocolos se pueden dividir en varias categorías, la más estudiada es la OSI (open system interconnection).



HTTP es la base de cualquier intercambio de datos en la Web, y un protocolo de estructura cliente-servidor, esto quiere decir que una petición de datos es iniciada por el elemento que recibirá los datos (el cliente), normalmente un navegador Web.

Así, una página web completa resulta de la unión de distintos sub-documentos recibidos, como, por ejemplo: un documento que especifique el estilo de maquetación de la página web (CSS), el texto, las imágenes, vídeos, scripts, etc...



Ahora vamos a profundizar un poco más estos temas y veremos las tecnologías que resuelven los puntos planteados anteriormente.

SOCKETS EN JAVA

La programación en red siempre ha sido dificultosa, el programador debía de conocer la mayoría de los detalles de la red, incluyendo el hardware utilizado, los distintos niveles en que se divide la capa de red, las librerías necesarias para programar en cada capa, etc.

Pero, la idea simplemente consiste en obtener información desde otra máquina, aportada por otra aplicación software. Por lo tanto, de cierto modo se puede reducir al mero hecho de leer y escribir archivos, con ciertas salvedades.

Aquí un Videotutorial sobre Sockets

QUE SON LOS SOCKETS ?

El sistema de Entrada/Salida de Unix y adoptado también por otros sistemas operativos, sigue el paradigma que normalmente se designa como **Abrir-Leer-Escribir-Cerrar.**
Antes de que un proceso de usuario pueda realizar operaciones de entrada/salida, debe hacer una llamada a Abrir (**open**) para indicar, y obtener los permisos del fichero o dispositivo que se desea utilizar.

Una vez que el fichero o dispositivo se encuentra abierto, el proceso de usuario realiza una o varias llamadas a Leer (read) y Escribir (write), para la lectura y escritura de los datos.

El proceso de lectura toma los datos desde el objeto y los transfiere al proceso de usuario, mientras que el de escritura los transfiere desde el proceso de usuario al objeto. Una vez concluido el intercambio de información, el proceso de usuario llamará a Cerrar (**close**) para informar al sistema operativo que ha finalizado la utilización del fichero o dispositivo.

Un proceso tiene un conjunto de descriptores de entrada/salida desde donde leer y por donde escribir. Estos descriptores pueden estar referidos a ficheros, dispositivos, o canales de comunicaciones llamados **sockets**.

El ciclo de vida de un socket, está determinado por tres fases :

1. Creación, apertura del socket

2. Lectura y Escritura, recepción y envío de datos por el socket

3. Destrucción, cierre del socket

Los destinatarios de los mensajes se especifican como **direcciones de socket**; cada dirección de socket es un identificador de comunicación que consiste en una **dirección Internet y un número de puerto**.

Las operaciones de intercambio de datos entre el cliente y el servidor se basan en pares de sockets.

Se intercambia información transmitiendo datos a través de mensajes que circulan entre un socket en un proceso y otro socket en otro proceso.
Cuando los mensajes son enviados, se encolan en el socket hasta que el protocolo de red los haya transmitido.
Cuando llegan, los mensajes son encolados en el socket de recepción hasta que el proceso que tiene que recibirlos haga las llamadas necesarias para recoger esos datos. *(Esto es válido cuando tanto el cliente como el servidor están implementados en java. Para las aplicaciones web, esto es válido solo para el webserver desarrollado en java, el cliente adopta un mecanismo diferente para el envío y recepción de datos).*

El lenguaje Java fue desarrollado por la empresa Sun MicroSystems hacia el año 1990, mediante la creación de un grupo de trabajo en cuya cabeza estaba James Gosling.

Este grupo de trabajo fue ideado para desarrollar un sistema de control de electrodomésticos y de PDAs o asistentes personales (pequeños ordenadores) y que además tuviese la posibilidad de interconexión a redes de ordenadores.
Todo ello implicaba la creación de un hardware polivalente, un sistema operativo eficiente (SunOS) y un lenguaje de desarrollo (Oak). El proyecto concluyó dos años más tarde con un completo fracaso que condujo a la disolución del grupo.

Pero el desarrollo del proyecto relativo al lenguaje **oak** siguió adelante gracias entre otras cosas a la distribución libre del lenguaje por Internet mediante la incipiente, por aquellos años, **World Wide Web**.

De esta forma el lenguaje alcanzó cierto auge y un gran número de programadores se encargaron de su depuración así como de perfilar la forma y usos del mismo.

El nombre de Java, surgió durante una de las sesiones de brain storming que se celebraban por el equipo de desarrollo del lenguaje. Hubo que cambiar el nombre debido a que ya existía otro lenguaje con el nombre de oak.

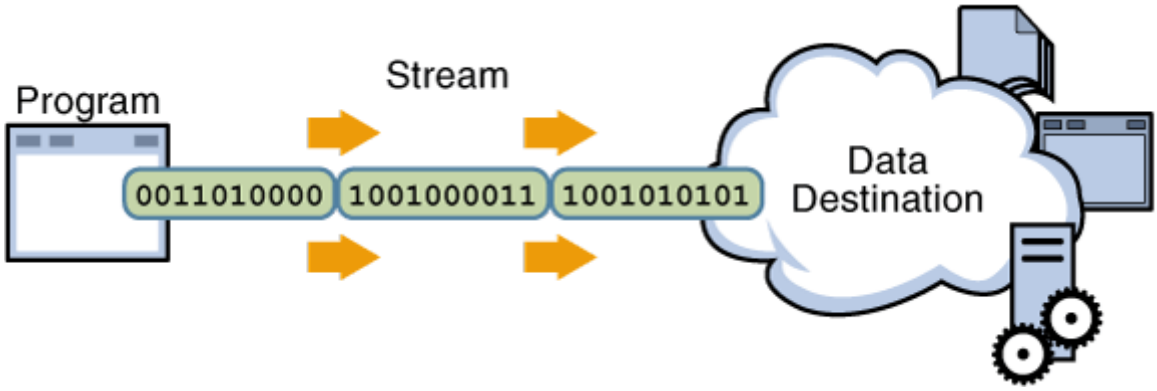
Sun MicroSystems lanzó las primeras versiones de Java a principios de 1995, y se han ido sucediendo las nuevas versiones durante estos últimos años, fomentando su uso y extendiendo las especificaciones y su funcionalidad.

Una de las características más importantes de Java es su capacidad y, a la vez, facilidad para realizar aplicaciones que funcionen en red. La mayoría de los detalles de implementación a bajo nivel están ocultos y son tratados de forma transparente por la JVM (Java Virtual Machine). Los programas son independientes de la arquitectura y se ejecutan indistintamente en una gran variedad de equipos con diferentes tipos de microprocesadores y sistemas operativos.

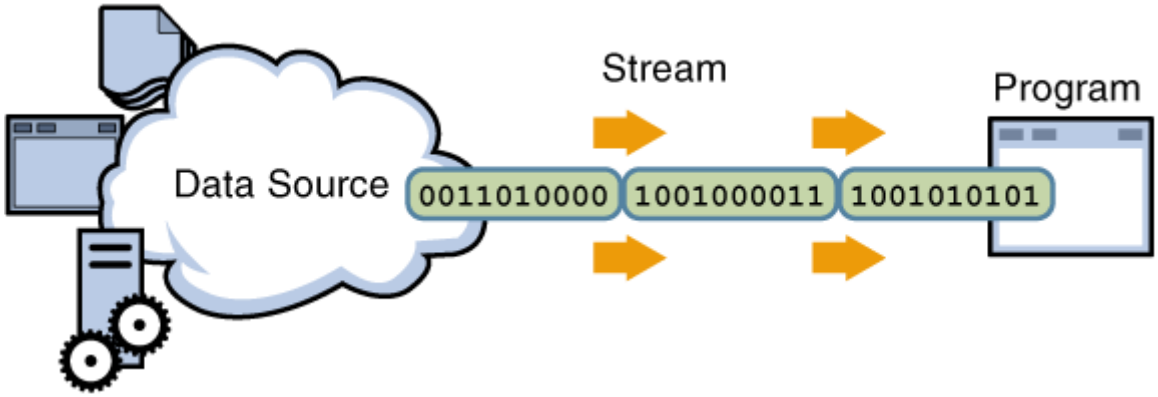
2 CLASES PARA LAS COMUNICACIONES DE RED EN JAVA: java.net

En las aplicaciones en red es muy común el **paradigma cliente-servidor**. El servidor es el que **espera** las conexiones del cliente (en un lugar claramente definido) y el cliente es el que lanza las peticiones a la maquina donde se está ejecutando el servidor, y al lugar donde está esperando el servidor (**el puerto(s) específico que atiende**).

Una vez establecida la conexión, ésta es tratada como un **Stream** (flujo de bytes) típico de entrada/salida. **Los streams nos permitirán realizar el intercambio de datos entre el cliente y el servidor**.



El cliente solicita un recurso al servidor el cual llega al server como un stream al socket en el cual el server está escuchando (puerto).



El servidor responde con un stream de datos que se pueda interpretar como HTML, ya que el receptor es un web browser.

Cuando se escriben programas Java que se comunican a través de la red, se está programando en la capa de aplicación.

Típicamente, no se necesita trabajar con las capas TCP y UDP, en su lugar se puede utilizar las clases del paquete java.net. Estas clases proporcionan comunicación de red independiente del sistema.

A través de las clases del paquete java.net, los programas Java pueden utilizar TCP o UDP para comunicarse a través de Internet.

Las clases **URL, URLConnection, Socket, y SocketServer** utilizan TCP para comunicarse a través de la Red.

Las clases **DatagramPacket y DatagramServer** utilizan UDP.

TCP proporciona un canal de **comunicación fiable punto a punto**, lo que utilizan para comunicarse las aplicaciones cliente-servidor en Internet.

Las clases **Socket y ServerSocket** del paquete **java.net** proporcionan un canal de comunicación independiente del sistema utilizando TCP, cada una de las cuales implementa el lado del cliente y el servidor respectivamente.

Así el paquete java.net proporciona, entre otras, las siguientes clases:

Socket: Implementa un extremo de la conexión TCP.

ServerSocket: Se encarga de implementar el extremo **Servidor** de la conexión en la que se esperarán las conexiones de los clientes.

DatagramSocket: Implementa tanto el servidor como el cliente cuando se utiliza UDP.

DatagramPacket: Implementa un datagram packet, que se utiliza para la creación de servicios de reparto de paquetes sin conexión.

InetAddress: Se encarga de implementar la dirección IP.

La clase **Socket** del paquete java.net es una implementación independiente de la plataforma de un cliente para un enlace de comunicación de dos vías entre un cliente y un servidor.

Utilizando la clase **java.net.Socket** en lugar de tratar con código nativo, los programas Java pueden comunicarse a través de la red de una forma independiente de la plataforma.

El entorno de desarrollo de Java incluye un paquete, **java.io**, que contiene un juego de canales de entrada y salida que los programas pueden utilizar para leer y escribir datos.

Las clases **InputStream y OutputStream** del paquete java.io son superclases abstractas que definen el comportamiento de los canales de I/O de tipo **stream** de Java. java.io también incluye muchas subclases de **InputStream y OutputStream** que implementan tipos específicos de canales de I/O.

2.1 DATAGRAM SOCKET (Servicio sin Conexión)

Es el más simple, lo único que se hace es enviar los datos, mediante la creación de un socket y utilizando los métodos de envío y recepción apropiados.

Se trata de un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad: los datos se envían y reciben en paquetes, cuya entrega no está garantizada; los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con datagramas UDP, es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

La clase principal es:

DatagramSocket

A) Constructores:

public DatagramSocket () throws SocketException

Se encarga de construir un socket para datagramas y de conectarlo al primer puerto disponible.

public DatagramSocket (int port) throws SocketException

Ídem, pero con la salvedad de que permite especificar el número de puerto asociado.

public DatagramSocket (int port, InetAddress ip) throws SocketException

Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket.

B) Métodos:

public void close()

Cierra el socket.

protected void finalize()

Asegura el cierre del socket si no existen más referencias al mismo.

public int getLocalPort()

Retorna el número de puerto en el host local al que está conectado el socket.

public void receive (DatagramPacket p) throws IOException

Recibe un DatagramPacket del socket, y llena el búfer con los datos que recibe.

public void send (DatagramPacket p) throws IOException

Envía un DatagramPacket a través del socket.

2.2 DATAGRAM PACKET

Un DatagramSocket envía y recibe los paquetes y un DatagramPacket contiene la información relevante. Cuando se desea recibir un datagrama, éste deberá almacenarse bien en un búfer o un array de bytes. Y cuando preparamos un datagrama para ser enviado, el DatagramPacket no sólo debe tener la información, sino que además debe tener la dirección IP y el puerto de destino, que puede coincidir con un puerto TCP.

public final class java.net.DatagramPacket extends java.lang.Object

A) Constructores:

public DatagramPacket(byte ibuf[], int ilength)

Implementa un DatagramPacket para la recepción de paquetes de longitud ilength, siendo el valor de este parámetro menor o igual que ibuf.length.

public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)

Implementa un DatagramPacket para el envío de paquetes de longitud ilength al número de puerto especificado en el parámetro iport, del host especificado en la dirección de destino que se le pasa por medio del parámetro iaddr.

B) Métodos:

public InetAddress getAddress ()

Retorna la dirección IP del host al cual se le envía el datagrama o del que el datagrama se recibió.

public byte[] getData()

Retorna los datos a recibir o a enviar.

public int getLength()

Retorna la longitud de los datos a enviar o a recibir.

public int getPort()

Retorna el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió.

2.3 STREAM SOCKET (Servicio Orientado a Conexión)

Es un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados. El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

Permite a las aplicaciones Cliente y Servidor, disponer de un **stream** que facilita la comunicación entre ambos, obteniéndose una mayor fiabilidad.

El funcionamiento es diferente al anterior ya que cada extremo se comportará de forma diferente, el servidor adopta un papel (inicial) pasivo y espera conexiones de los clientes. Mientras que el cliente adoptará un papel (inicial) activo, solicitando

conexiones al servidor.

En la parte del servidor se tiene:

public final class java.net.ServerSocket extends java.lang.Object

A) Constructores :

public ServerSocket (int port) throws IOException

Se crea un socket local al que se enlaza el puerto especificado en el parámetro port, si se especifica un 0 en dicho parámetro creará el socket en cualquier puerto disponible. Puede aceptar hasta 50 peticiones en cola pendientes de conexión por parte de los clientes.

public ServerSocket (int port , int count) throws IOException

Aquí, el parámetro count sirve para que puede especificarse, el número máximo de peticiones de conexión que se pueden mantener en cola.

*Hay que recordar, que es fundamental que el puerto escogido sea conocido por el cliente, en caso contrario, no se podría establecer la conexión.***B) Métodos :**

public Socket accept () throws IOException

*Sobre un **ServerSocket** se puede realizar una espera de conexión por parte del cliente mediante el método **accept().***
*Hay que decir, que este método es de bloqueo, el proceso espera a que se realice una conexión por parte del cliente para seguir su ejecución. Una vez que se ha establecido una conexión por el cliente, este método devolverá un objeto tipo **Socket**, a través del cual se establecerá la comunicación con el cliente.*

public void close () throws IOException

Se encarga de cerrar el socket.

public InetAddress getInetAddress ()

Retorna la dirección IP remota a la cual está conectado el socket. Si no lo está retornará null .

public int getLocalPort ()

Retorna el puerto en el que está escuchando el socket.

public String toString( )

Retorna un string representando el socket.

En la parte del cliente :

public final class java.net.Socket extends java.lang.Object

A) Constructores :

public Socket (InetAddress address, int port) throws IOException

Crea un StreamSocket y lo conecta al puerto remoto y dirección IP remota especificados.

public Socket (InetAddress address, int port , boolean stream) throws IOException

Ídem a la anterior incluyendo el parámetro booleano stream que si es true creará un StreamSocket y si es false un DatagramSocket (En desuso).

public Socket (String host, int port) throws UnKnownHostException, IOException

Crea un StreamSocket y lo conecta al número de puerto y al nombre de host especificados.

public Socket (String host , int port , boolean stream) throws IOException

Ídem al anterior incluyendo el parámetro booleano stream que si es true creará un StreamSocket y si es false un DatagramSocket (En desuso).

B) MÉTODOS :

public void close() throws IOException

Se encarga de cerrar el socket.

public InetAddress getInetAddress ()

Retorna la dirección IP remota a la que se conecta el socket.

public InputStream getInputStream () throws IOException

Retorna un input stream para la lectura de bytes desde el socket.

public int getLocalPort()

Retorna el puerto local al que está conectado el socket.

public OutputStream getOutputStream () throws IOException

Retorna un output stream para la escritura de bytes hacia el socket.

public int getPort ()
Retorna el puerto remoto al que está conectado el socket.

2.4 La Clase InetAddress

Esta clase implementa la dirección IP.

public final class java.net.InetAddress extends java.lang.Object

A) Constructores :

Para crear una nueva instancia de esta clase se debe de llamar a los métodos **getLocalHost(), getByname() o getAllByName()**

B) Métodos :

public boolean equals (Object obj)

Devuelve un booleano a true si el parámetro que se la pasa no es null e implementa la misma dirección IP que el objeto. Dos instancias de InetAddress implementan la misma dirección IP si la longitud del vector de bytes que nos devuelve el método getAddress() es la misma para ambas y cada uno de los componentes del vector de componentes es el mismo que el vector de bytes.

public byte[] getAddress ()

Retorna la dirección raw IP del objeto InetAddress.
Hay que tener en cuenta que el byte de mayor orden de la dirección estará en getAddress()[0] .

public static InetAddress[] getAllByName(String host) throws UnknownHostException

Retorna un vector con todas las direcciones IP del host especificado en el parámetro.

public static InetAddress getByName (String host) throws UnknownHostException

Retorna la dirección IP del nombre del host que se le pasa como parámetro, aunque también se le puede pasar un string representando su dirección IP

public String getHostName()

Retorna el nombre del host para esta dirección IP.

public static InetAddress getLocalHost() throws UnknownHostException

Retorna la dirección IP para el host local.

public int hashCode ()

Retorna un código hash para esta dirección IP.
public String toString ()

Retorna un String representando la dirección IP.

Un ejemplo de utilización muy sencillo de esta clase es el siguiente, que se encarga de devolver la dirección IP de la máquina.

```
public class QuienSoy {

public static void main (String[] args) throws Exception {
	if (args.length !=1) {
		System.err.println("Uso: java QuienSoy NombreMaquina");
		System.exit(1);
	}
		InetAddress direccion= InetAddress.getByName(args[0]);
		System.out.println(direccion);

}
}
```

3 ENVIO Y RECEPCIÓN A TRAVÉS DE SOCKETS

El servidor creará un socket, utilizando **ServerSocket**, le asignará un **puerto** y una dirección, una vez haga el **accept** para esperar llamadas, se quedará bloqueado a la espera de las mismas. Una vez llegue una llamada el **accept** creará un **socket** para procesarla.

A su vez, cuando un cliente desee establecer una conexión (solo clientes java), creará un **socket** y establecerá una conexión al puerto establecido. Sólo es en este momento, cuando se da una conexión real y se mantendrá hasta su liberación mediante **close().**

Para poder leer y escribir datos, los sockets disponen de unos **stream** asociados, uno de entrada (**InputStream**) y otro de salida (**OutputStream**) respectivamente.

Para obtener estos **streams** a partir del **socket** utilizaremos :

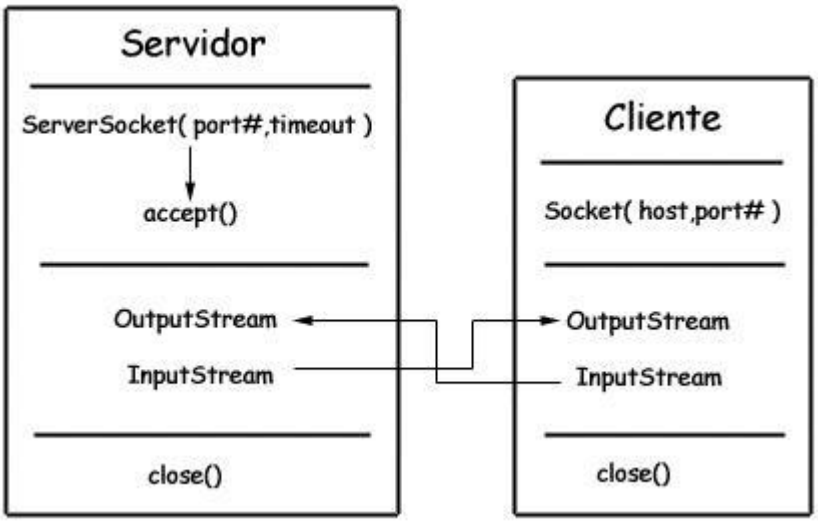
miSocket.getInputStream ()

Devuelve un objeto de tipo InputStream.

miSocket.getOutputStream ()

Devuelve un objeto de tipo OutputStream.

Para el envío de datos, puede utilizarse OutputStream directamente en el caso de que se quiera enviar un flujo de bytes sin búfer o también puede crearse un objeto de tipo stream basado en el OutputStream que proporciona el socket.



Esquema de conexión mediante sockets stream.

En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete

java.net.

El servidor establece un puerto y espera a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método **accept().**

El cliente establece una conexión con la máquina host a través del puerto que se designe en port#. El cliente y el servidor se comunican con manejadores **InputStream y OutputStream**

Si se está programando un cliente, el socket se abre de la forma:

Socket miSocket;

miSocket= new Socket(host, puerto);

Donde host es el nombre de la máquina sobre la que se está intentando abrir la conexión y puerto es el puerto (un número) que el servidor está atendiendo.

Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango **0-1023 están reservados**. Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp, http, etc. Por lo que, para aplicaciones de usuario, el programador deberá asegurarse de seleccionar un puerto por encima del **1023.**

Hasta ahora no se han utilizado excepciones; pero deben tener en cuenta la captura de excepciones cuando se está trabajando con sockets.
Así :

```
Socket miSocket;

try {

      miSocket= new Socket(host, puerto);

     } catch(IOException e) { System.out.println(e);

     } catch (UnknownHostException uhe) {

       System.out.println(uhe);

     }
```

En el caso de estar implementando un servidor, la forma de apertura del socket sería

como sigue :

```
Socket socketSrv;

try {

      socketSrv= new ServerSocket(puerto);

      } catch(IOException e) { System.out.println(e);

    }
```

Cuando se implementa un servidor se necesita crear un objeto Socket a partir del ServerSocket, para que éste continúe ateniendo las conexiones que soliciten potenciales nuevos clientes y poder servir al cliente, recién conectado, a través del Socket creado:

```
Socket socketServicio= null;

try {

socketServicio= socketSrv.accept(); } catch(IOException e) {

System.out.println(e);

}
```

4 CREACIÓN DE STREAMS

4.1 Creación de Streams de Entrada

En la parte cliente de la aplicación (esto es solo informativo, ya que nuestro cliente será web y no usaremos esta implementación), se puede utilizar la clase DataInputStream para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.

```
DataInputStream inSocket;

   try{
         inSocket= new DataInputStream(miSocket.getInputStream());

      } catch( IOException e ) { System.out.println( e );

      }
```

La clase DataInputStream permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: read(), readChar(), readInt(), readDouble() y readUTF(). Deberá utilizarse la función que se crea necesaria dependiendo del tipo de dato que se

espera recibir del servidor. En el lado del servidor, también se usará DataInputStream, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

```
DataInputStream inSocket;

try {

        inSocket=new DataInputStream(socketServicio.getInputStream());

        } catch(IOException e) {

        System.out.println(e);

     }
```

4.2 Creación de Streams de Salida

En el lado del cliente(esto es solo informativo, ya que nuestro cliente será web y no usaremos esta implementación), puede crearse un stream de salida para enviar información al socket del servidor utilizando las clases PrintStream o DataOutputStream:

```
PrintStream outSocket;

try {
outSocket= new PrintStream(miSocket.getOutputStream());
}
catch(IOException e) {
System.out.println(e);
}
```

La clase PrintStream tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos write y println() tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar DataOutputStream:

```
DataOutputStream outSocket;

   try{
        outSocket= new DataOutputStream(miSocket.getOutputStream());
        } catch(IOException e) { System.out.println( e );
}
```

La clase DataOutputStream permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea writeBytes(). En el lado del servidor, puede utilizarse la clase PrintStream para enviar información al cliente:

```
PrintStream outSocket;
```

```
  try {
      outSocket= new PrintStream(socketServicio.getOutputStream());
      } catch(IOException e) {
      System.out.println(e);
      }
```

Pero también puede utilizarse la clase DataOutputStream como en el caso de envío de información desde el cliente.

5 CIERRE DE SOCKETS

Siempre deben cerrarse los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación.

En el lado del cliente:

```
try {
      outSocket.close();
      inSocket.close();
      miSocket.close();
      }
      catch(IOException e)
      {
      System.out.println(e);
      }
```

Y en el lado del servidor:

```
try {
      outSocket.close();
      inSocket.close();
      socketServicio.close();
      socketSrv.close();
      } catch(IOException e)
      {
      System.out.println(e);
      }
```

6 DIFERENCIAS ENTRE SOCKETS STREAM Y DATAGRAMA

Ahora se presenta un problema, ¿qué protocolo, o tipo de sockets, debe utilizarse - UDP o TCP? La decisión depende de la aplicación cliente/servidor que se esté escribiendo. Se muestran, a continuación, algunas diferencias entre los protocolos para ayudar en la decisión.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tiene que establecerse esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp), que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión, esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

7 CLASES UTILES EN COMUNICACIONES

A continuación se introducen otras clases que resultan útiles cuando se está desarrollando programas de comunicaciones, aparte de las que ya se han visto. El problema es que la mayoría de estas clases se prestan a discusión, porque se encuentran bajo el directorio sun. Esto quiere decir que son implementaciones Solaris y, por tanto, específicas del Unix Solaris. Además su API no está garantizada, pudiendo cambiar. Pero, a pesar de todo, resultan muy interesantes y se comentarán un grupo de ellas, las que se encuentran en el paquete sun.net.

MulticastSocket:

Clase utilizada para crear una versión multicast de las clase socket datagrama. Múltiples clientes/servidores pueden transmitir a un grupo multicast (un grupo de direcciones IP compartiendo el mismo número de puerto).

NetworkServer:

Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.

NetworkClient:

Una clase creada para implementar métodos y variables utilizadas en la creación de un cliente TCP/IP.

SocketImpl:

Es un interface que permite crear nuestro propio modelo de comunicación. Tendrán que implementarse sus métodos cuando se use. Si se va a desarrollar una aplicación con requerimientos especiales de comunicaciones, como pueden ser la implementación de un cortafuegos (TCP es un protocolo no seguro), o acceder a equipos especiales (como un lector de código de barras o un GPS diferencial), se necesita una de Socket específica

Bueno, hasta aquí un poco (bastante ... !!!!) de teoría sobre sockets, streams y comunicaciones. Todo esto lo verán con muchísimo más detalle y complejidad en otras asignaturas, la idea es tener una base mínima para comprender los ejemplos y poder realizar las actividades.
Así es que salgamos un poco de tanta teoría y manos a la obra !

<u>Ejemplo1:</u> Echo Server súper simple para explicar y entender todo el teórico sobre **sockets, threads y streams.**

```
package server;

// importamos todas las clases necesarias para crear nuestro server

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.StringTokenizer;

public class HTTPServer extends Thread {
       // nuestra clase extiende a thread para manejar varios hilos
       // Esto significa que a medida que vaya recibiendo pedidos, va a colocar cada uno de ellos
       // en un hilo diferente y así poderlos procesar independientemente.

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

        // creamos una instancia de ServerSocket que escucha en el puerto 5000
        // puede ser cualquiera que queramos por encima del 1024

        ServerSocket server = new ServerSocket(5000, 10, InetAddress.getByName("127.0.0.1"));

        System.out.println("Echo Server se inicio en el puerto 5000");
        while (true) {
            // usamos el socket del puerto 5000 para aceptar inputs, aceptamos conexiones
```

```
            Socket connected = server.accept();
            HTTPServer httpServer = new HTTPServer(connected);
            httpServer.start(); // arrancamos nuestro servidor
        }
    }

    private Socket cliente = null;

    // creamos los streams de entrada y salida de información a nuestro server
    private BufferedReader inCliente = null;
    private DataOutputStream outCliente = null;

    // le pasamos a nuestro Server un socket que manejará y procesará el pedido del cliente

    public HTTPServer(Socket cl) {
    cliente = cl;
    }

    public void run() {
    try {
        System.out.println("El Cliente " + cliente.getInetAddress() + ":"
        + cliente.getPort() + " está conectado");



       // capturamos el flujo de entrada que nos envía el cliente y lo vinculamos a un InputStream
        InputStream entrada=cliente.getInputStream();
        // Creamos un Flujo de salida y lo asociamos con el flujo de salida del socket
        PrintWriter salida=new PrintWriter(cliente.getOutputStream());
        // Creamos un Buffer de memoria donde procesar el flujo
        BufferedReader br=new BufferedReader(new InputStreamReader(entrada));

        String requestString = br.readLine(); //leemos la primera línea que nos llega
        String headerLine = requestString;

        /** algunas veces la entrada es nula  */
        if (headerLine == null)
        return;

        String NEW_LINE = "\r\n";
        System.out.println("He recibido la Siguiente Información desde la Web:");
        String respuestaServidor="Hola desde el servidor !";
        // Para que el Navegador entienda la respuesta que enviamos desde el server hay que enviar este e
ncabezado
            salida.println("HTTP/1.0 200 OK"); // todo está ok !!
            salida.println("Content-Type:text/html;charset=utf-8"); // tipo de contenido enviado
            salida.println("");
            // creamos una página html de respuesta al navegador

            salida.println("<!doctype 1.0>");
            salida.println("<html>");
            salida.println("<body>");
            salida.println("<H2>Bienvenido al Echo Servidor Web</H2>" );
            salida.println(respuestaServidor);
            salida.println("<div style='background-
color: honeydew; border: 2px solid'>Hemos recibido la siguiente información desde el Navegador" );
        while (br.ready()) {
        /** Leemos todo el pedido HTTP hasta el final.... */
```

```
        System.out.println(requestString);
        salida.println(requestString + "<br>");
        requestString = br.readLine();
        }
        salida.println("</div></body>");
        salida.println("</html>");
        salida.close();
        cliente.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
    }
}
```

Para probar este ejemplo, crearemos un proyecto java en Netbeans. Luego crearemos la clase HTTPServer, colocando todo el código que se muestra.

Ponemos a correr nuestro proyecto oprimiendo 'run'.
Si todo va bien, nuestro server estará recepcionando conexiones en el puerto 5000.

Abrimos el navegador web y en la barra de direcciones, escribimos

http://localhost:5000

Debemos obtener una respuesta similar a esto:

Hasta aquí el servidor es súper básico y no hace nada, vamos ahora a hacer uno un poco más complejo.
Para ello es necesario capturar la información que nos llega desde el cliente web y luego hacer algo con ella.

Ejemplo 2:

Webserver HTTP que implementa Get y un par de funciones.
Usaremos Sockets - Thread y Streams

La idea para este segundo ejemplo será poder capturar la información que nos envía nuestro cliente, procesarla de algún modo y poderle responder.

Como nuestro cliente es un web browser, la respuesta deberá ser HTML.
Para ello, vamos a implementar dentro de nuestro server, unas funciones que permitirán devolver páginas html al cliente.
Para recibir la información, asumiremos que nuestro cliente nos envía solicitudes usando el **protocolo de comunicación HTTP.**
Este protocolo tiene una serie de métodos básicos que permiten solicitudes de recursos a URLs.

El Protocolo de transferencia de hipertexto (en inglés, Hypertext Transfer Protocol, abreviado HTTP) es el protocolo de comunicación que permite las transferencias de información en la World Wide Web. HTTP fue desarrollado por el World Wide Web Consortium y la Internet Engineering Task Force, colaboración que culminó en 1999 con la publicación de una serie de RFC, siendo el más importante de ellos el RFC 2616 que especifica la versión 1.1. HTTP define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, proxies) para comunicarse. HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores.

Los métodos principales de este protocolo son:

GET
POST
PUT
DELETE

Estos métodos los podemos asociar con los clásicos Leer, Crear, Actualizar y Borrar que utilizamos para construir un CRUD (Create, Read, Update y Delete) en bases de datos.

Una solicitud usando el método GET tiene el siguiente formato:

GET /index.html HTTP/1.1

Como podemos ver el recurso solicitado se encuentra después de la primera "/"
Esto es importante pues, podemos implementar en nuestro server algún mecanismo que detecte la ubicación de la "/" y extraiga el contenido antes del token HTTP.
Dicho contenido será el recurso que nuestro cliente nos está solicitando y en caso que nos envíe información, la misma estará allí.

Un **formulario HTML**, tiene la capacidad de efectuar solicitudes HTTP usando los métodos GET Y POST. Para ello la estructura básica es:

```
<form method="GET" action="url a la que solicitamos/enviamos el recurso" >
   <input type="text" name=" variable enviada " value=" valor enviado"/>
    <input type="submit" value="Enviar Datos">
</form>
```

Los pedidos de tipo Get envían la información como un Stream de caracteres sin codificar que no tienen seguridad de ningún tipo.
Por este motivo, en términos generales, solo se usa el método Get para solicitar

información ya existente en el servidor, pero no para enviar información sensible como contraseñas, datos de tarjetas de crédito, etc.
Por ejemplo, podemos usar Get para solicitar un listado de productos, o de clientes o verificar si un usuario existe en nuestra base de datos, pero nunca lo usaremos para enviar **usuario=pepe & password=123456**, pues esta información es fácilmente visible y capturable por alguien malintencionado.
Veremos métodos más seguros como **POST** en las próximas clases.

Por ejemplo si tenemos una página web que contiene una formulario como:

```
<form action="http://localhost:5000/Hola" method="get">
      Ingresa tu Nombre <input type="text" name="nombre" id=""><br>

      <input type="submit" value="Saludar">
  </form>
```

Lo que estamos haciendo es enviar una variable llamada nombre un recurso a la url

http://localhost:5000/Hola

Como dentro del formulario hemos colocado un input cuyo name es **nombre**, lo que el usuario escriba dentro de ese input se enviará al server usando una estructura atributo = valor, por ejemplo si el usuario escribe **Juan**, entonces se crea una estructura **nombre=Juan.**

Cuando se oprima el botón enviar se generará una solicitud HTTP de tipo **GET** a la url indicada que tendrá el siguiente formato:

GET /Hola?nombre=Juan HTTP/1.1

Las variables enviadas al recurso Hola van después del signo '?'. Si son varias, se separan con el símbolo '&'. Sabiendo esto podemos usar nuestros conocimientos y herramientas de java para procesar las solicitudes GET.

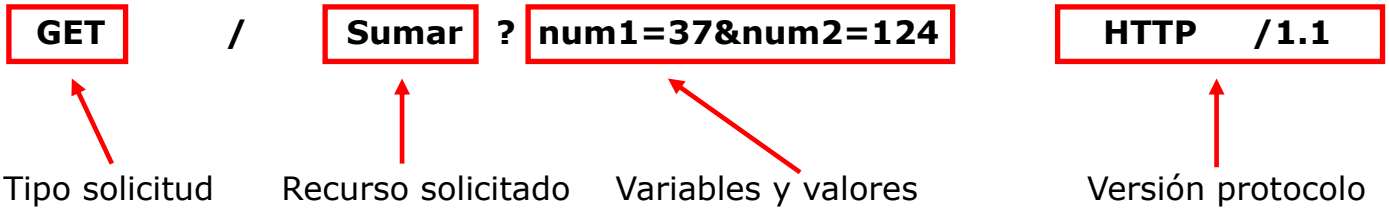
Como podemos ver nuestro pedido está comprendido entre la primera "/" y la segunda "/". Si podemos implementar un método en nuestro server que extraiga el contenido entre las barras y elimine el token HTTP, nos quedará la solicitud del cliente que es este caso es un recurso denominado Hola al cual le están pasando la variable **nombre** cuyo valor es **Juan**.

Igualmente si tuviésemos un formulario:

```
<form action="http://localhost:5000/Sumar" method="get">
   Ingresa el Primer Numero <input type="number" name="num1" id=""><br>
   Ingresa el Segundo Numero <input type="number" name="num2" id=""><br>
    <input type="submit" value="Sumar">
</form>
```

estaríamos enviando a la url **http://localhost:5000/Sumar**

al recurso Sumar las variables **num1 y num 2** con los valores que el usuario haya ingresado.
El pedido GET que recibirá nuestro server tendría este formato:



Teniendo en cuenta todos estos detalles, ahora mejoraremos nuestro server inicial. Por ahora solo realizaremos pedidos GET, las demás solicitudes las veremos más adelante.

```
package webserver;

/**
 *
 * @author CARLOMAGNO
 */


import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.StringTokenizer;

// nuestra clase Server extenderá a Thread. De esa forma cada solicitud que
// reciba se ejecutará en un hilo independiente para poder atender múltiples
// clientes.

public class HTTPServer extends Thread {
    public static void main(String args[]) throws Exception
{
    // creamos un server en el puerto 5000
    ServerSocket server = new ServerSocket(5000, 10,InetAddress.getByName("127.0.0.1"));

    System.out.println("HTTPServer se inicio en el puerto 5000");


    while (true)
    { // Lazo infinito para aceptar solicitudes desde los clientes
        Socket connected = server.accept();
        HTTPServer httpServer = new HTTPServer(connected);
        httpServer.start(); // método heredado de Thread . iniciamos un hilo ...
        }
    }

    private Socket cliente = null; // creamos un socket llamado cliente
    private BufferedReader inCliente = null; // Buffer de recepción de entrada
    private DataOutputStream outCliente = null; // flujo de salida
```

```
public HTTPServer(Socket cl) {
cliente = cl;
}

//implementamos el método run del Thread, ya que por defecto este método
// está vacío

public void run() {
try {
    System.out.println("El Cliente " + cliente.getInetAddress() + ":"
    + cliente.getPort() + " está conectado");

// asignamos los flujos de entrada y salida ....
    inCliente = new BufferedReader(new
    InputStreamReader(cliente.getInputStream()));

    outCliente = new DataOutputStream(cliente.getOutputStream());

    String requestString = inCliente.readLine();
    String headerLine = requestString;

    /** leemos la primera linea del encabezado recibido el que puede ser nulo */
    if (headerLine == null)
    return;
    // descomponemos el header en tokens
    StringTokenizer tokenizer = new StringTokenizer(headerLine);

// si el pedido es GET, entonces es el primer token del encabezado

    String httpMethod = tokenizer.nextToken();

/* si es GET, los parámetros del pedido vienen inmediatamente después
de GET */

    String httpQueryString = tokenizer.nextToken();

    System.out.println("El Pedido HTTP es ....");
    while (inCliente.ready()) {
    /** Leemos todo el pedido HTTP hasta el final.... */
    System.out.println(requestString);
    requestString = inCliente.readLine();
    }

    if (httpMethod.equals("GET")) {


// vemos si han enviado parámetros. Si solo está la barra volvemos a
// home

    if (httpQueryString.equals("/")) {
        /** volvemos a home page */
        homePage();
    } else if (httpQueryString.startsWith("/Hola")) {

        /** vamos a la página Hola */
        HolaPage(
            httpQueryString.substring(httpQueryString.lastIndexOf('/') + 1,
            httpQueryString.length()));
```

```
        } else if (httpQueryString.startsWith("/Sumar")) {
            String Parametro=httpQueryString.substring(httpQueryString.lastIndexOf('/') + 2,
            httpQueryString.length());
            String [] Parametros=Parametro.split("&");
            // parametros es un array que contiene num1= xx , num2= yy
            float num1=Float.parseFloat(Parametros[0].substring(Parametros[0].indexOf("=")+1));
            float num2=Float.parseFloat(Parametros[1].substring(Parametros[1].indexOf("=")+1));
            float resultado=num1+num2;
            SumarPage(resultado);

        } else {
            sendResponse(404, "<b>El Recurso solicitado no se encuentra disponible</b>");
        }
        } else {
        sendResponse(404, "<b>El Recurso solicitado no se encuentra disponible</b>");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    }


    public void sendResponse(int statusCode, String responseString) throws Exception {

    String HTML_START = "<html><title>Servidor HTTP</title><body>";
    String HTML_END = "</body></html>";
    String NEW_LINE = "\r\n";

    String statusLine = null;
    String serverdetails = Headers.SERVER + ": Servidor HTTP en Java";
    String contentLengthLine = null;
    String contentTypeLine = Headers.CONTENT_TYPE + ": text/html" + NEW_LINE;

    if (statusCode == 200)
        statusLine = Status.HTTP_200;
    else
        statusLine = Status.HTTP_404;

    statusLine += NEW_LINE;
    responseString = HTML_START + responseString + HTML_END;
    contentLengthLine = Headers.CONTENT_LENGTH + responseString.length() + NEW_LINE;

    outCliente.writeBytes(statusLine);
    outCliente.writeBytes(serverdetails);
    outCliente.writeBytes(contentTypeLine);
    outCliente.writeBytes(contentLengthLine);

outCliente.writeBytes(Headers.CONNECTION + ": close" + NEW_LINE);

    /** adding the new line between header and body */
    outCliente.writeBytes(NEW_LINE);

    outCliente.writeBytes(responseString);

    outCliente.close();
    }
```

```
    public void homePage() throws Exception {
    StringBuffer responseBuffer = new StringBuffer();
    responseBuffer.append("<b>HTTPServer Página Principal</b><BR><BR>");
    responseBuffer.append("<b>Para acceder a la Página Hola use:
    http://localhost:5000/Hola/TuNombre Aqui</b><BR>");
    responseBuffer.append("<b>Para acceder a la Página Sumar use:
    http://localhost:5000/Sumar/num1=xxx&num2=yyy</b><BR>");


    sendResponse(200, responseBuffer.toString());
    }


    public void HolaPage(String name) throws Exception {
    StringBuffer responseBuffer = new StringBuffer();
    responseBuffer.append("<style> h2 {background-
color: honeydew; border:2px solid; width:80%; margin:auto } </style>");
    responseBuffer.append("<h2 > Hola: ").append(name).append("</h2><BR>");
    sendResponse(200, responseBuffer.toString());
    }

    public void SumarPage(float valor) throws Exception {
    StringBuffer responseBuffer = new StringBuffer();

    responseBuffer.append("<style> h2 {background-
    color: lightblue; border:2px solid; width:80%; margin:auto } </style>");
    responseBuffer.append("<h2>  El Resultado de la Suma es: ").append(
    String.valueOf(valor)).append("</h2><BR>");
    sendResponse(200, responseBuffer.toString());
    }
    // constantes del protocolo HTTP
    private static class Headers
    {
        public static final String SERVER = "Server";
        public static final String CONNECTION = "Connection";
        public static final String CONTENT_LENGTH = "Content-Length";
        public static final String CONTENT_TYPE = "Content-Type";
    }

   // constantes de status
    private static class Status
    {
        public static final String HTTP_200 = "HTTP/1.1 200 OK";
        public static final String HTTP_404 = "HTTP/1.1 404 Recurso no Encontrado";
    }
}
```

Atención: Si copian y pegan el código anterior, tengan en cuenta que en algunas líneas por cuestiones de formato del Word, aquellas muy extensas, se han trasladado al renglón siguiente, sin embargo en el código java estos saltos de línea se interpretan como error y deberán mover la línea inferior y completar la superior.

Una vez que hayan implementado el código, ejecuten el programa y luego para probar que todo funciona Ok, pueden usar esta página web de prueba.

```
<!DOCTYPE html>
```

```html
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Prueba de nuestro Web Server</title>
</head>

<body>
    <div>
         <h2>Probando la funcion Hola de nuestro Web Server con Formularios ...</h2>
        <form action="http://localhost:5000/Hola" method="get">
            Ingresa tu Nombre <input type="text" name="nombre" id=""><br>
            <input type="submit" value="Saludar">
        </form>
    </div>
    <div>
        <h2>Probando la funcion Sumar de nuestro Web Server con Formularios ...</h2>
       <form action="http://localhost:5000/Sumar" method="get">
           Ingresa el Primer Numero <input type="number" name="num1" id=""><br>
           Ingresa el Segundo Numero <input type="number" name="num2" id=""><br>
           <input type="submit" value="Sumar">
       </form>
   </div>
</body>
</html>
```

Cuando se esté ejecutando el server, abran la página en el navegador, ingresen datos y prueben la respuesta del servidor.

Actividad Nº 1:

En base a lo visto anteriormente podemos determinar que una manera de solicitar recursos a un servidor es usando una notación URL donde en la estructura de la misma se incluya la acción esperada y posteriormente los parámetros necesarios para efectuar la acción. Por ejemplo si tenemos una app que efectúa suma de números y la acción está definida como "Sumar", la URL para acceder a dicha función sería:

../api/Sumar/?numero1=xxx&numero2=yyy

*Este mecanismo de reescritura de la URL, es muy utilizado cuando queremos implementar en el server algo que más adelante llamaremos API. No siempre nuestras aplicaciones web querrán acceder al servidor web haciendo solicitudes usando formularios. Es más la mayoría de las veces serán consultas en segundo plano llamadas **"callbacks"** y que se enviarán mediante objetos especiales capaces de conectarse con el servidor **"por detrás"** de la capa de interfaz de la aplicación.*

Un ejemplo de esto sería, por ejemplo si tenemos una lista de productos y de pronto queremos mostrar el detalle del producto al hacer clic en su imagen. En una aplicación real, los detalles de ese producto particular están en una base de datos, por lo que deberemos consultar al servidor para que nos envíe esa información, pero no tenemos ningún formulario para ingresar los datos del producto solicitado y además si lo tuviéremos sería muy engorroso usar ese mecanismo. Para ello se usan los

"callbacks asincrónicos" *basados en AJAX que veremos más adelante.*

Volviendo a lo que veníamos desarrollando, el server, tal como se muestra en el ejemplo, debe ser capaz de descomponer la URL para extraer la acción, los parámetros y devolver una página con el resultado de la acción efectuada.

Las páginas de respuesta están implementadas como funciones, que generan el HTML que deben enviar como respuesta. En aplicaciones más complejas, podríamos tener clases externas al server que generen dichas páginas y se importen dentro de la clase server cuando sea necesario, o mejor aún, páginas HTML completas a las que redireccionamos la respuesta cuando sea necesario. Veremos como se hace eso en las próximas clases.

Siguiendo estos conceptos, agregue al ejemplo anterior funcionalidades para resolver todas las operaciones matemáticas.

Las acciones serán: "sumar", "restar", "multiplicar", "dividir". Se operará solo con dos números.
El server debe responder tanto si se lo accede vía URL desde el navegador, como si se emplea una página html que envíe los datos por formulario usando GET.

Actividad Nº2: Construya una aplicación de Servidor que permita recibir un número y devuelva la tabla de multiplicar de dicho número hasta el 12.

Actividad Nº3: Construya una aplicación de Servidor que permita el logueo de usuarios. Para ello se usará la Ruta ../login/usuario=alguien&pwd=clave.
De momento los usuarios y los password estarán en un ArrayList en el código.

Actividad Nº4: Siguiendo con la actividad anterior, ahora incorpore la posibilidad de agregar a la aplicación de servidor nuevos usuarios (se agregarán al ArrayList). Para ello la ruta será ../agregar/usuario=pepe&pwd=clave.
Si el usuario ya está registrado, no se registrará mostrando mensaje.