Comunicaciones en red

# 1.- Conceptos básicos.

Las aplicaciones que se comunican con sockets, lor ealizan a través de una determinada dirección IP y un determinado puerto. Básicamente, las direcciones IP definen a qué ordenador van los daros y el puerto indica, dentro del ordenador, a qué programa van los datos.

Cada día es más frecuente que las empresas utilicen aplicaciones que se comunican por Internet para poder compartir información entre sus empleados. Por ejemplo, aplicaciones de gestión y facturación que permiten que varias tiendas puedan realizar de forma centralizada la facturación de toda la empresa, gestionar el stock, etc.

A continuación, vamos a ver los conceptos más importantes sobre redes, necesarios para más adelante poder programar nuestras aplicaciones. Para ello, primero veremos una pequeña introducción sobre el modelo TCP/IP, los tipos de conexiones que se pueden realizar, así como los modelos más importantes de comunicaciones.

## 1.1.- Recordando TCP/IP.

En 1969, la agencia ARPA (Advanced Research Projects Agency) del Departamento de Defensa de EE.UU inició un **proyecto de interconexión de ordenadores mediante redes telefónicas**. Al ser un proyecto desarrollado por militares en plena guerra fría, un principio básico de diseño era que la red debía **poder resistir la destrucción de parte de su infraestructura** (por ejemplo a causa de un ataque nuclear), de forma que dos nodos cualesquiera pudieran seguir comunicados siempre que hubiera alguna ruta que los uniera

**Esto se consiguió en 1972 con ARPANet**. La conmutación de paquetes unida al uso de topologías malladas mediante múltiples líneas punto a punto dio como resultado una red altamente fiable y robusta.

ARPAnet fue creciendo paulatinamente, y pronto **se hicieron experimentos utilizando otros medios de transmisión de datos, en particular enlaces por radio y vía satélite**; los protocolos existentes tuvieron problemas para interoperar con estas redes, por lo que se diseñó un nuevo conjunto o pila de protocolos, y con ellos una arquitectura.

Este nuevo conjunto y modelo se denominó **TCP/IP** (Transmission Control Protocol/Internet Protocol), nombre que provenía de los dos protocolos más importantes que componían la pila. A la nueva red, que se creó como consecuencia de la **fusión de ARPAnet con las redes basadas en otras tecnologías de transmisión, se la denominó Internet**.

La aproximación adoptada por los diseñadores del TCP/IP fue mucho más pragmática que la de los autores del modelo OSI.

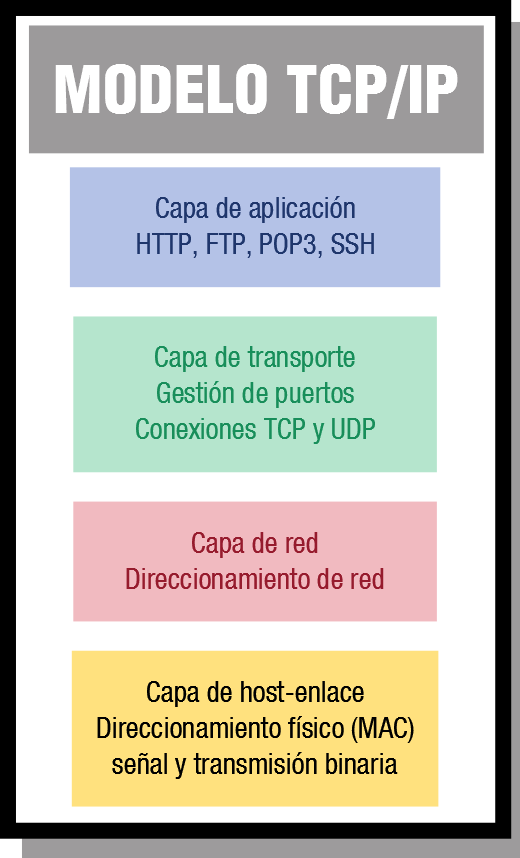
* **En OSI**, se emplearon muchos años en definir cuidadosamente una arquitectura de capas donde la función y servicios de cada una estaban perfectamente definidas, y sólo después se planteó desarrollar los protocolos para cada una de ellas
* **En TCP/IP** la operación fue a la inversa; primero se especificaron los protocolos, y luego se definió el modelo como una simple descripción de los protocolos ya existentes. Por este motivo el modelo TCP/IP es mucho más simple que el OSI.

También **por este motivo el modelo OSI se utiliza para describir otras arquitecturas**, como por ejemplo TCP/IP, mientras que el modelo **TCP/IP nunca suele emplearse para describir otras arquitecturas que no sean la suya propia**.

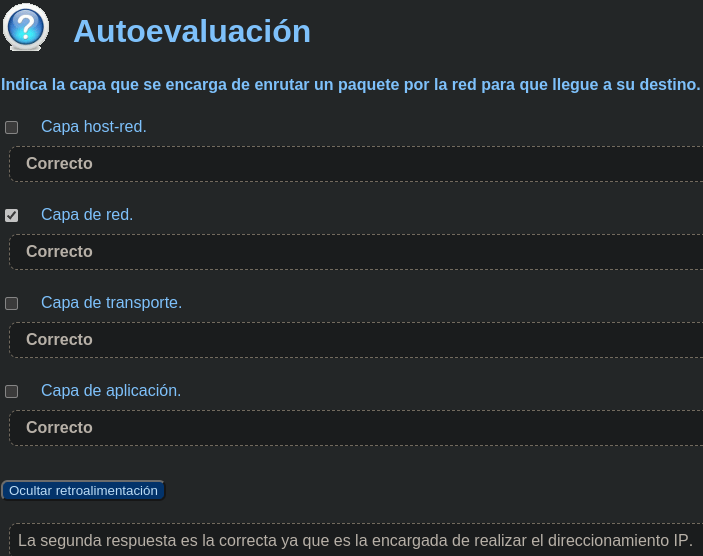
**Debes conocer**: <https://es.wikipedia.org/wiki/Modelo_TCP/IP>

### 1.1.1.- Recordando TCP/IP (II).

El modelo TCP/IP tiene solo 4 capas:

* **La capa host-red**: comunica el ordenador con el medio que conecta el equipo a la red.
  + Convierte la información en impulsos físicos (eléctricos, magnéticos o luminosos) y además, debe permitir las conexiones entre los ordenadores de la red.
  + Se realiza un **direccionamiento físico utilizando las direcciones MAC**.
* **La capa de red**: eje de la arquitectura TCP/IP. Permite que los equipos envíen paquetes en cualquier red y viajen de forma independiente a su destino (que podría estar en una red diferente).
  + Se encarga de que los **paquetes llegan a su destino**.
  + Los paquetes podrían llegar incluso en un orden diferente al que se enviaron. Corresponde a las capas superiores reordenarlos.
  + Esta capa define un **formato de paquete y protocolo oficial** llamado **IP** (Internet Protocol).
  + En esta capa se realiza el direccionamiento lógico o direccionamiento por IP.
* **La capa de transporte**: es la **capa que permite que los equipos lleven a cabo una conversación**. Se crearon **dos protocolos para esta capa**:
  + **TCP** (Transmission Control Protocol): protocolo orientado a conexión y fiable
  + **UDP** (User Datagram Protocol): protocolo UDP es un protocolo no orientado a conexión y no fiable
  + **Direccionamiento por puertos** (la información se envía a la aplicación adecuada que tenga abierto dicho puerto).
  + Gracias a la capa anterior, los paquetes **viajan de un equipo origen a un equipo destino**.
* **La capa de aplicación**: Esta capa engloba las **funcionalidades de las capas de sesión**, presentación y aplicación del modelo OSI. Incluye todos los protocolos de alto nivel relacionados con las aplicaciones que se utilizan en Internet (por ejemplo HTTP, FTP, TELNET).

**Para saber más**: <https://view.officeapps.live.com/op/view.aspx?src=http%3A%2F%2Fcentros.edu.xunta.es%2Fiesoteropedrayo.ourense%2Fdptos%2Ftec%2F_private%2Fredes.ppt&wdOrigin=BROWSELINK>



## 1.2.- Conexiones TCP/UDP.

La **capa de transporte** cumple la función de **establecer las reglas** necesarias para establecer una conexión entre dos dispositivos.

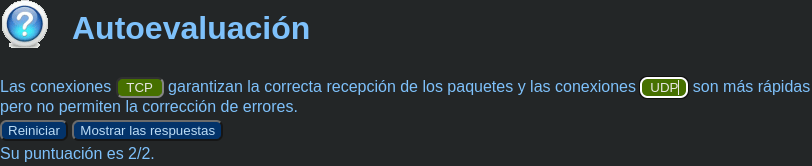
Desde la capa anterior, la capa de red, **la información se recibe en forma de paquetes desordenados** y la **capa de transporte** debe ser capaz de manejar dichos paquetes y **obtener un único flujo de datos**. Es decir, en esta capa se deben ordenar los paquetes y resolver los posibles errores de transferencia, aunque emisor y receptor no estén directamente conectados.

**Tipos de conexiones**:

* **TCP (Transmission Control Protocol)**: es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino.
  + El emisor fragmenta el flujo de bytes en mensajes y pasa cada uno a la capa de red.
  + El receptor reensambla los mensajes recibidos para formar el flujo de salida.
  + TCP se encarga del control de flujo para asegurar que un emisor rápido no pueda saturar a un receptor lento.
* **UDP (User Datagram Protocol)**: es un protocolo sin conexión, para aplicaciones que no necesitan la asignación de secuencia ni el control de flujo y que desean utilizar los suyos propios.
  + **Se usa en aplicaciones en las que la velocidad es más importante que la entrega precisa**, como las transmisiones de voz o de vídeo (y más si es en tiempo real, donde no sería posible realizar retransmisiones con TCP por los estrictos requisitos de retrardo que se tiene en estos casos).

De esta forma, a la hora de programar nuestra aplicación, **deberemos elegir el protocolo que queremos utilizar seǵun nuestras necesidades**: TCP o UDP.

**Para saber más**: [TCP](https://es.wikipedia.org/wiki/Protocolo_de_control_de_transmisi%C3%B3n), [UDP](https://es.wikipedia.org/wiki/Protocolo_de_datagramas_de_usuario).



## 1.3.- Puertos de comunicación.

Con la **capa de red** se consigue que la **información vaya de un equipo origen a un equipo destino** a través de su dirección IP.

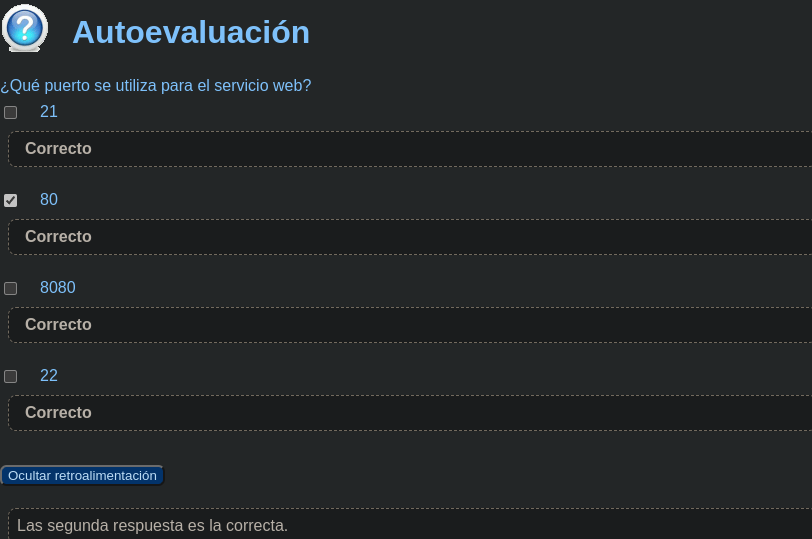
El método que se emplea para establecer a qué aplicación se conectará es el de **definir los puertos** de escucha de solicitudes de conexión.

Aunque muchos de los puertos se asignan de manera arbitraria, ciertos puertos se reservan, por convenio, a ciertas aplicaciones particulares o servicios de carácter universal. Según la IANA (Internet Asssigned Numbers Authority):

* **Puertos conocidos [0, 1023]**: puertos reservados a aplicaciones de uso estándar como:
  + 21 -> FTP (File Transfer Protocol)
  + 22 -> SSH (Secure SHell)
  + 53 -> DNS -> Domain Name Services
  + 80 -> HTTP -> (Hypertext Transfer Protocol)
* **Puertos registrados [1024, 49151]**: puertos asignados por IANA para un servicio específico o aplicaciones. Estos puertos pueden ser utilizados por los usuarios libremente.
* **Puertos dinámicos [49152, 65535]**: Este rango de puertos no puede ser registrado y su uso se establece para conexiones temporales entre aplicaciones.

**Cuando se desarrolla** una aplicación que utilice un puerto de comunicación, **optaremos por utilizar puertos comprendidos entre el rango 1024-49151**.

**Para saber más**: durante el desarrollo de este módulo y de otros de este ciclo, necesitarás conocer cuáles son los puertos relacionados con cada una de las aplicaciones. [Service Name and Transport Protocol Port Number Registry (iana.org)](https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml)



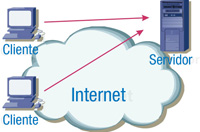
## 1.4.- Nombres en Internet.

Los **equipos informáticos se comunican entre sí mediante direcciones IP**. Sin embargo, **nosotros preferimos utilizar nombres**, porque son **más fáciles de recordar** y porque ofrecen la flexibilidad de poder cambiar la máquina en la que están alojados (cambiaría entonces la dirección IP) sin necesidad de cambiar las referencias a él.

El **sistema DNS basado en dominios**, es aquel en el **que se dispone de uno o más servidores encargados de resolver los nombres** de los equipos pertenecientes a su ámbito. Esto constituye un sistema jerárquico que permite una administración focalizada y, también, descentralizada y un mecanismo de resolución eficiente.

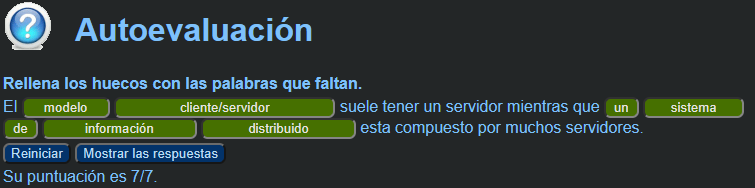
A la hora de comunicarse con un equipo, puedes hacerlo directamente a través de su dirección IP, o puede poner su entrada DNS. Si pone la entrada DNS, servidor de nombres que utilice en su conexión a Internet resolverá dicho nombre convirtiéndolo a IP por ti.

## 1.5.- Modelos de comunicaciones.

El **modelo de comunicación que ha revolucionado** los sistemas informáticos es el **modelo cliente/servidor**. Este está compuesto por un servidor que ofrece una serie de servicios y unos clientes que acceden a dichos servicios a través de la red. **Ejemplo**: WWW. El servidor aloja la página web, y el cliente es el navegador web del usuario.

Otro modelo ampliamente utilizado son los **Sistemas de Información Distribuido**. Un Sistema de Información Distribuido está compuesto por un conjunto de equipos que interactúan entre sí y pueden trabajar a la vez como cliente y servidor.

Desde el punto de vista externo, es **igual que un sistema cliente-servidor**, ya que **el cliente ve al Sistema Distribuido como una entidad**.



# 2.- Sockets TCP.

**Caso práctico**: dos tipos de conexiones: TCP -> orientadas a conexión, y se utilizan para enviar datos de aplicaciones. UDP -> no orientadas a conexión y se utilizan para enviar la información muy rápido, sin control de pérdida de paquetes. En Java, para trabajar con estos protocolos, se hace uso de sockets.

Un **socket** es un **punto de información por el cual un proceso puede recibir o enviar información**. Permite la comunicación entre procesos de diferentes equipos de una red.

El protocolo TCP es un protocolo orientado a la conexión que permite que **un flujo de bytes** originado en una máquina **se entregue sin errores** en cualquier máquina destino.

A la hora de crear un socket **hay que tener claro el tipo de socket que se quiere crear** (TCP o UDP). En esta sección vamos a aprender a utilizar los sockets TCP y en el siguiente punto veremos los sockets UDP.

Al utilizar **sockets TCP**, el servidor utiliza un **puerto** por el que recibe las diferentes peticiones de los clientes. Normalmente, el puerto del servidor es un puerto bajo [1-1023].



**Cuando el cliente realiza la conexión con el servidor**, a partir de ese momento se crea un **nuevo socket** que será en el encargado de permitir el envío y recepción de datos entre el cliente/servidor. El **puerto** se crea **de forma dinámica** y se encuentra en el **rango 49152 y 655335**.

De esta forma, **el puerto por donde se reciben las conexiones de los clientes queda libre** y cada comunicación tiene su propio socket.



El paquete **java.net** de Java proporciona:

* **Clase *Socket***: permite la comunicación por red. Permite a un cliente conectarse a un servidor para enviar y recibir información.
* **Clase *ServerSocket***: permite a un servidor escuchar y recibir las peticiones de los clientes por la red.

**Para saber más**: [clase Socket](https://www.youtube.com/shorts/Ikccidn25pw).

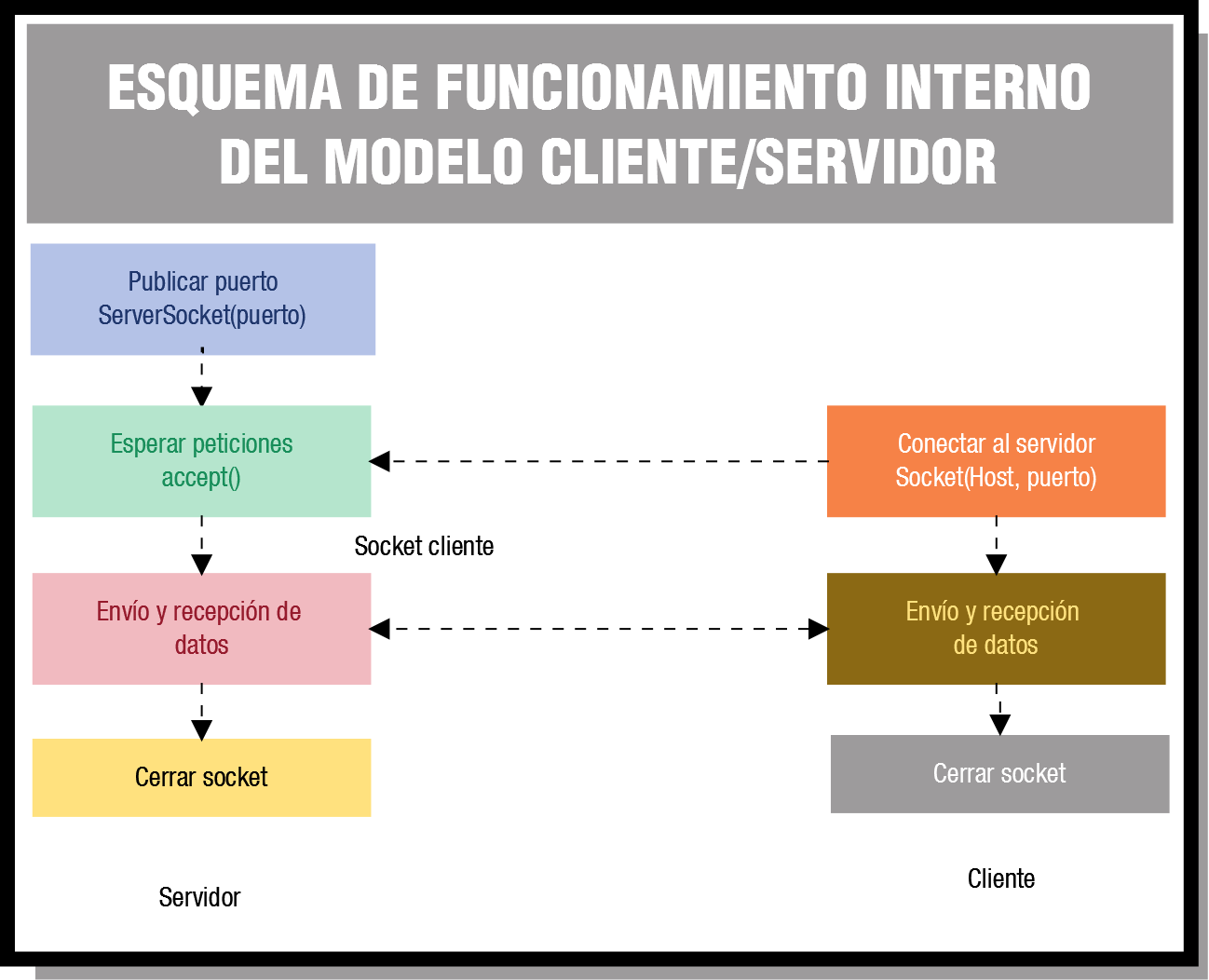
Texto

Descripción generada automáticamente

## 2.1.- Servidor.

Pasos que realiza el servidor para llevar a cabo una comunicación:

* **Publicar puerto**: indicando el puerto por donde se van a recibir las conexiones.
* **Esperar peticiones**: el servidor queda a la espera de recibir clientes. Una vez se conecte un cliente, se crea el socket del cliente por donde se envían y reciben datos.
* **Envío y recepción de datos**: necesario crear un flujo (stream) de entrada y otro de salida. Cuando el servidor recibe una petición, esta la procesa y le envía el resultado al cliente.
* Una vez finalizada la comunicación, se **cierra el socket del cliente**.



import java.io.\* ;

import java.net.\* ;

class Servidor {

static final int Puerto = 2000;

public Servidor( ) {

try {

// Inicio la escucha del servidor en un determinado puerto

ServerSocket skServidor = new ServerSocket(Puerto);

System.out.println("Escucho el puerto " + Puerto );

// Espero a que se conecte un cliente y creo un nuevo socket para el cliente

Socket sCliente = skServidor.accept();

// ATENDER PETICIÓN DEL CLIENTE

// Cierro el socket

sCliente.close();

}

} catch( Exception e ) {

System.out.println( e.getMessage() );

}

public static void main( String[] arg ) {

}

new Servidor();

}

}

## 2.2.- Cliente.

Pasos que realiza el cliente para llevar a cabo una comunicación:

* **Conectarse con el servidor**: el cliente utiliza la función Socket para indicarse con un determinado servidor **a un puerto específico**. Una vez realizada la conexión, se crea el socket por donde se realizará la comunicación.
* **Envío y recepción de datos**: necesario crear un flujo (stream) de entrada y otro de salida.

import java.io.\*;

import java.net.\*;

class Cliente {

static final String Host = "localhost";

static final int Puerto=2000;

public Cliente( ) {

try{

// Me conecto al servidor en un detrminado puerto

Socket sCliente = new Socket( Host, Puerto );

// TAREAS QUE REALIZA EL CLIENTE

// Cierro el socket

sCliente.close();

} catch( Exception e ) {

System.out.println( e.getMessage() );

}

}

public static void main( String[] arg ) {

new Cliente();

}

}

Imagen que contiene Texto

Descripción generada automáticamente

## 2.3.- Flujo de Entrada y Salida.

Una vez establecida la conexión entre el cliente y el servidor se inicializa la variable del tipo Socket que en el ejemplo se llama sCliente. **Para poder enviar o recibir datos a través del socket es necesario establecer un stream** (flujo) **de entrada o de salida** según corresponda.

**Para enviar datos**, tenenos la siguiente sentencia:

**DataOutputStream flujo\_salida= new DataOutputStream(sCliente.getOutputStream());**

**Y para recibir**, tenemos esta sentencia:

**DataInputStream flujo\_entrada = new DataInputStream(sCliente.getInputStream());**

A partir de este momento, puede enviar información de las siguiente manera:

|  |  |
| --- | --- |
| **Emisor** | **Receptor** |
| **flujo\_salida.writeUTF( "Enviar datos");** | **String datos=new String();**  **Datos=flujo\_entrada.readUTF();** |

**Para saber más**: además de utilizar las funciones ***writeUTF*** y ***readUTF***, es posible recibir información de otras maneras. Para más información, consulta los siguientes enlaces: [DataInputStream (Java SE 11 & JDK 11 ) (oracle.com)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/DataInputStream.html), [DataOutputStream (Java SE 11 & JDK 11 ) (oracle.com)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/DataOutputStream.html)

Texto

Descripción generada automáticamente

## 2.4.- Ejemplo.

Para continuar con el ejemplo anterior y poder utilizar los sockets para enviar información, vamos a realizar un ejemplo en el que el servidor va a aceptar tres clientes (de forma secuencial no concurrente) y le va a indicar el nº de cliente que es.

**Servidor**:

import java.io.\* ;

import java.net.\* ;

class Servidor {

static final int Puerto=2000;

public Servidor( ) {

try {

ServerSocket skServidor = new ServerSocket(Puerto);

System.out.println("Escucho el puerto " + Puerto );

for ( int nCli = 0; nCli < 3; nCli++) {

Socket sCliente = skServidor.accept();

System.out.println("Sirvo al cliente " + nCli);

OutputStream aux = sCliente.getOutputStream();

DataOutputStream flujo\_salida= new DataOutputStream( aux );

flujo\_salida.writeUTF( "Hola cliente " + nCli );

sCliente.close();

}

System.out.println("Ya se han atendido los 3 clientes");

} catch( Exception e ) {

System.out.println( e.getMessage() );

}

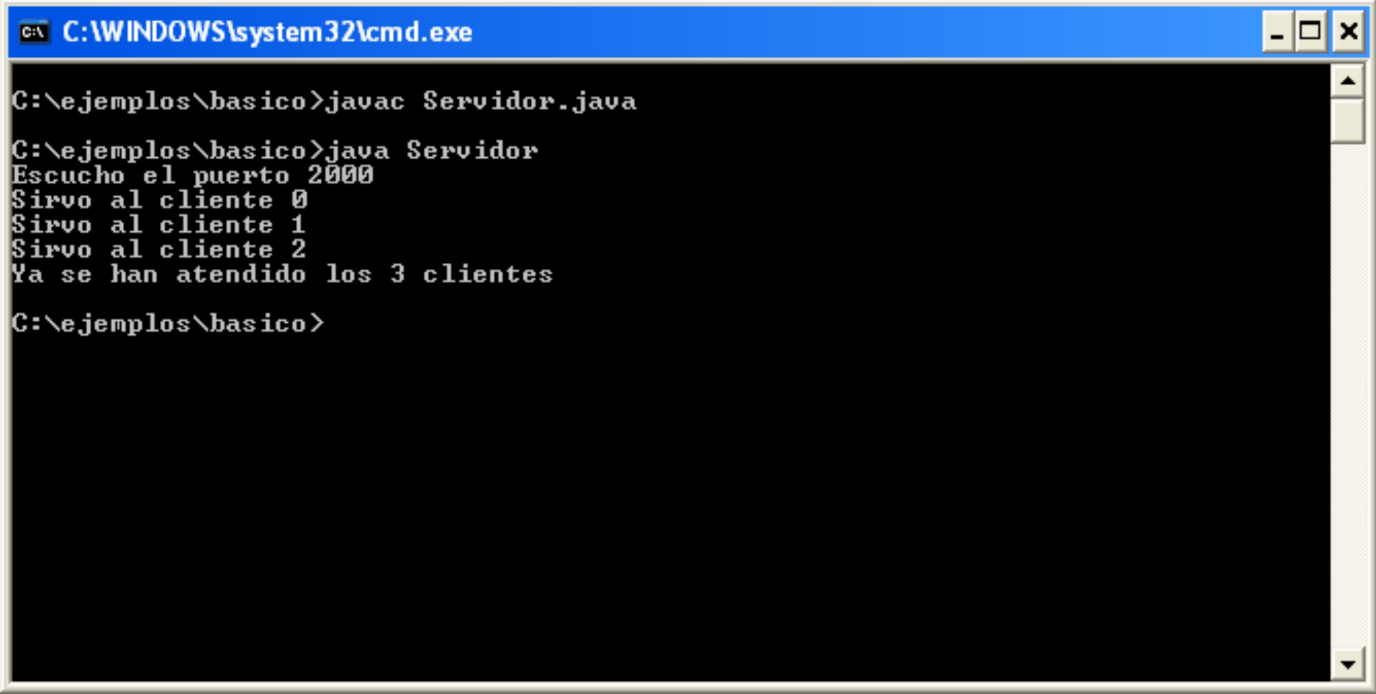
}

public static void main( String[] arg ) {

new Servidor();

}

}



**Cliente**:

import java.io.\*;

import java.net.\*;

class Cliente {

static final String HOST = "localhost";

static final int Puerto=2000;

public Cliente( ) {

try{

Socket sCliente = new Socket( HOST , Puerto );

InputStream aux = sCliente.getInputStream();

DataInputStream flujo\_entrada = new DataInputStream( aux );

System.out.println( flujo\_entrada.readUTF() );

sCliente.close();

} catch( Exception e ) {

System.out.println( e.getMessage() );

}

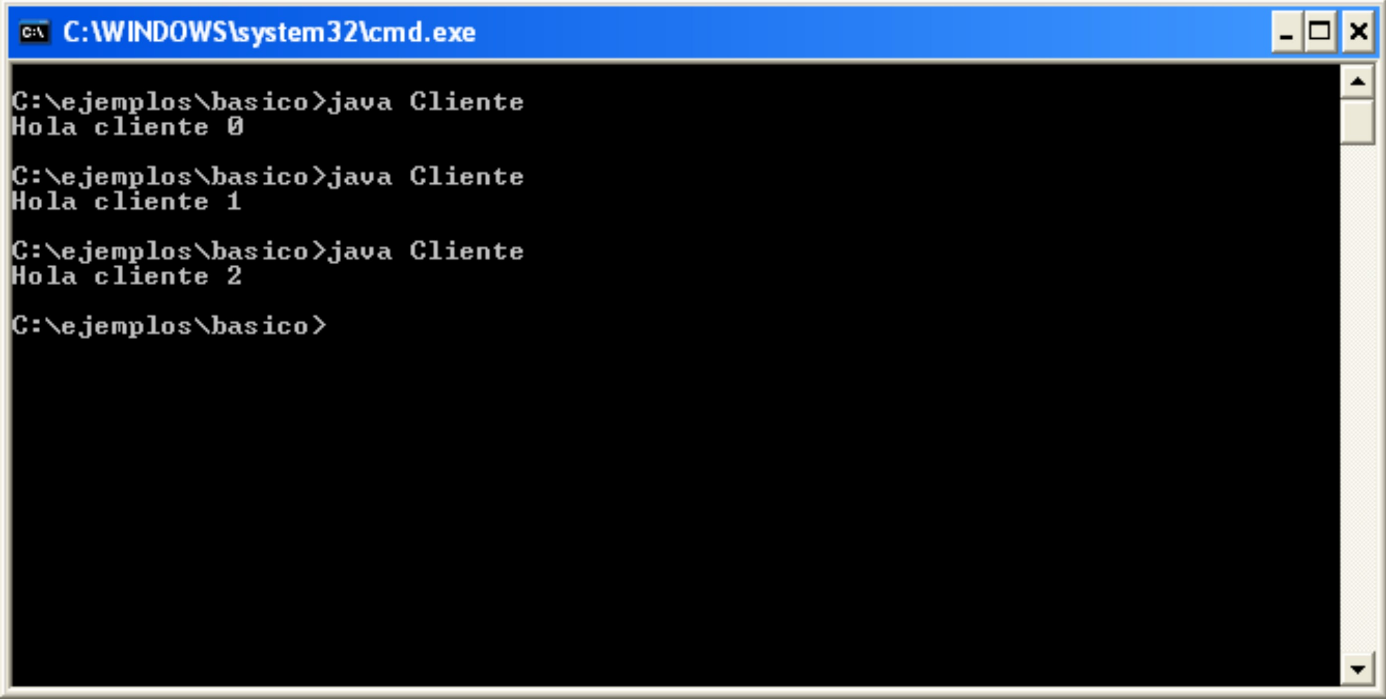
}

public static void main( String[] arg ) {

new Cliente();

}

}



# 3.- Sockets UDP.

En el caso de utilizar sockets UDP no se crea una conexión (como en el caso de TCP), sino que básicamente, enviamos y recibimos mensajes a través de una dirección IP y un puerto. Estos **mensajes se gestionan individualmente**, y **no se garantiza la recepción o el envío** del mensaje como si ocurre en TCP.

Para utilizar sockets UDP en java tenemos la clase **DatagramSocket** y para recibir o enviar los mensajes se utiliza clase **DatagramPacket**. Cuando se recibe o envía un paquete se hace con la siguiente información: **mensaje, longitud del mensaje, equipo y puerto**.

**Para saber más**: manuales oficiales de [***DatagramSocket***](https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html) y [***DatagramPocket***](https://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html).

## 3.1.- Receptor.

En el caso de querer iniciar el socket en un determinado puerto se realiza de la siguiente forma:

**DatagramSocket sSocket = new DatagramSocket(puerto);**

Una vez iniciado el socket, ya podemos **recibir mensajes** usando la clase ***DatagramPacket***.

**byte [] cadena = new byte[1000] ;**

**DatagramPacket mensaje = new DatagramPacket(cadena, cadena.length);**

**sSocket.receive(mensaje);**

**String datos=new String(mensaje.getData(),0,mensaje.getLength());**

**System.out.println("\tMensaje Recibido: " +datos);**

**sSocket.close();**

## 3.2.- Emisor.

Una aplicación emisora de mensajes UDP debe inicializar primero la estructura ***DatagramSocket***.

**DatagramSocket sSocket = new DatagramSocket();**

Ahora debe crear el mensaje de tipo ***DatagramPacket***, al que debe indicar:

* **Mensaje** a enviar.
* **Longitud** del mensaje.
* **Equipo** al que se le envía el mensaje.
* **Puerto** destino.

**Ejemplo**:

**InetAddress Equipo = InetAddress.getByName("localhost"); // Obtiene la direccion del equipo al que se le envía el mensaje a través de su nombre**

**DatagramPacket mensaje = new DatagramPacket(mensaje,longitud\_mensaje, Equipo, Puerto);**

Una vez creado el mensaje, lo enviamos y después, cerramos el socket:

**sSocket.send(mensaje);**

**sSocket.close();**

Texto

Descripción generada automáticamente

## 3.3.- Ejemplo.

A continuación, para aprender a programar Sockets UDP se va a realizar un ejemplo sencillo donde intervienen dos procesos:

* **ReceptorUDP**. Inicia el puerto 1500 y muestra en pantalla todos los mensajes que llegan a él.
* **EmisorUDP**. Permite enviar por líneas de comandos mensajes al receptor por el puerto 1500.

**ReceptorUDP.java**

import java.net.\*;

import java.io.\*;

public class ReceptorUDP {

public static void main(String args [] ) {

// Sin argumentos

if (args.length != 0) {

System.err.println("Uso: java ReceptorUDP");

}

else try{

// Crea el socket

DatagramSocket sSocket = new DatagramSocket(1500);

// Crea el espacio para los mensajes

byte [] cadena = new byte[1000] ;

DatagramPacket mensaje = new DatagramPacket(cadena, cadena.length);

System.out.println("Esperando mensajes..");

while(true){

// Recibe y muestra el mensaje

sSocket.receive(mensaje);

String datos=new String(mensaje.getData(),0,mensaje.getLength());

System.out.println("\tMensaje Recibido: " +datos);

}

} catch(SocketException e) {

System.err.println("Socket: " + e.getMessage());

} catch(IOException e) {

System.err.println("E/S: " + e.getMessage()); }

}

}

**EmisorUDP.java**

import java.net.\*;

import java.io.\*;

public class EmisorUDP {

public static void main(String args [] ) {

// Comprueba los argumentos

if (args.length != 2) {

System.err.println("Uso: java EmisorUDP maquina mensaje");

}

else try{

// Crea el socket

DatagramSocket sSocket = new DatagramSocket();

// Construye la dirección del socket del receptor

InetAddress maquina = InetAddress.getByName(args[0]);

int Puerto = 1500;

// Crea el mensaje

byte [] cadena = args[1].getBytes();

DatagramPacket mensaje = new DatagramPacket(cadena,args[1].length(), maquina, Puerto);

// Envía el mensaje

sSocket.send(mensaje);

// Cierra el socket

sSocket.close();

} catch(UnknownHostException e) {

System.err.println("Desconocido: " + e.getMessage());

} catch(SocketException e) {

System.err.println("Socket: " + e.getMessage());

} catch(IOException e) {

System.err.println("E/S: " + e.getMessage());

}

}

}

