

Sockets: Tipos de Sockets

3.3.- Qué son los sockets

Los protocolos TCP y UDP utilizan el concepto de sockets para proporcionar los puntos extremos de la comunicación entre aplicaciones o procesos. La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, a este conector es a lo que se le denomina socket.

Los socket permiten que un proceso emita o reciba información con otro proceso, incluso estando en otra máquina.

Un socket queda definido por un par de IP (local y remota) , un protocolo de transporte (TCP o UDP) y un par de números de puertos (local y remoto).

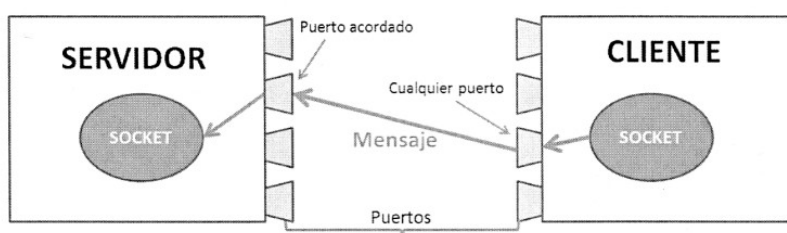
Para que dos aplicaciones puedan comunicarse entre si es necesario que se cumpla lo siguiente:

- Que un programa sea capaz de localizar al otro.
- Que ambas aplicaciones sean capaces de intercambiarse cualquier secuencia de octetos.

Para los procesos receptores de mensajes, su conector debe tener asociado dos campos:

- La dirección IP del host en el que la aplicación está corriendo.
- El puerto local a través del cual la aplicación se comunica y que identifica el proceso.

Todos los mensajes enviados a esa dirección IP y a ese puerto concreto llegarán al proceso receptor.



En la figura observamos que un proceso cliente (envía un mensaje) y un proceso servidor (recibe un mensaje) comunicándose mediante sockets. Cada socket tiene un puerto asociado, el proceso cliente debe de conocer el puerto y la IP del proceso servidor. Los mensajes al servidor le deben llegar al puerto acordado. El proceso cliente podrá enviar el mensaje por el puerto que quiera.

3.3.1.- Funcionamiento en general de un socket

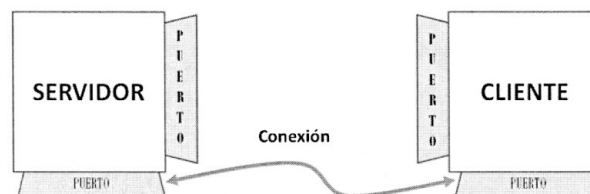
Un puerto es un punto de destino que identifica hacia qué aplicación o proceso deben dirigirse los datos. Normalmente en una aplicación cliente-servidor, el programa servidor se ejecuta en una máquina específica y tiene un socket que está unido a un número de puerto específico. El servidor queda a la espera “escuchando” las solicitudes de conexión de los clientes sobre ese puerto.

El programa cliente conoce el nombre de la máquina en la que se ejecuta el servidor y el número de puerto por el que escucha las peticiones. Para realizar una solicitud de conexión, el cliente realiza la petición a la máquina a través del puerto.



El cliente también debe identificarse ante el servidor por lo que durante la conexión se utilizará un puerto local asignado por el sistema.

Si todo va bien, el servidor acepta la conexión. Una vez aceptada, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que por un lado debe seguir atendiendo las peticiones de conexión mediante el socket original y por otro debe atender las necesidades del cliente que se conectó.



En el lado cliente, si se acepta la conexión, se crea un socket y el cliente puede utilizarlo para comunicarse con el servidor. Este socket utiliza un número de puerto diferente al usado para conectarse al servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo por sus respectivos sockets.

3.4.- Tipos de socket

Hay dos tipos de sockets en redes IP, los que usan el protocolo TCP y los que usan el protocolo UDP.

3.4.1.- Sockets orientados a conexión (TCP)

Los procesos que se van a comunicar deben establecer antes una conexión mediante un stream. Un stream es una secuencia ordenada de unidades de información (bytes, caracteres, etc.) que puede fluir en dos direcciones: hacia fuera de un proceso (de salida) o hacia dentro de un proceso (de entrada). Están diseñados para acceder a los datos de manera secuencial.

Una vez establecida la conexión, los procesos leen y escriben en el stream sin tener que preocuparse de las direcciones de internet ni de los números de puerto. El establecimiento de la conexión implica:

- Una petición de conexión desde el proceso cliente al proceso servidor.
- Una aceptación de la conexión del proceso servidor al proceso cliente.

En Java hay dos tipos de stream sockets que tienen asociadas las **clases Socket** para implementar el cliente y **ServerSocket** para el servidor.

3.4.2.- Sockets no orientados a conexión

Este tipo de conexión no es fiable y no se garantiza que la información enviada llegue a su destino. Los datagramas se transmiten desde un proceso emisor a otro receptor sin que se haya establecido previamente una conexión, sin acuse de recibo ni reintentos.

Cualquier proceso que necesite enviar o recibir mensajes debe crear primero un conector asociado a una dirección IP y a un puerto local. El servidor enlazará su conector a un puerto de servidor conocido por los clientes. El cliente enlazará su conector a cualquier puerto local libre. Cuando un receptor recibe un mensaje, se obtiene además del mensaje, la dirección IP y el puerto del emisor, permitiendo al receptor enviar la respuesta correspondiente al emisor.

Los sockets UDP se usan cuando una entrega rápida es más importante que una entrega garantizada, o en los casos en que se desea enviar tan poca información que cabe en un único datagrama. Se usan en aplicaciones para la transmisión de audio y vídeo en tiempo real donde no es posible el reenvío de paquetes retrasados.

Para implementar en Java este tipo de sockets se utilizan las clases `DatagramSocket` y `DatagramPacket`.

3.5.- Clases para sockets TCP

El paquete `java.net` proporciona las clases `ServerSocket` y `Socket` para trabajar con sockets TCP.

3.5.1.- Clase ServerSocket

Esta clase se utiliza para implementar el extremo de la conexión que corresponde al servidor, donde se crea un conector en el puerto de servidor que escucha las peticiones de conexión de los clientes

Algunos de los constructores de esta clase son los siguientes. Hay que tener en cuenta que pueden lanzar la excepción `IOException`.

CONSTRUCTOR	MISIÓN
<code>ServerSocket()</code>	Crea un socket de servidor sin ningún puerto asociado.
<code>ServerSocket(int port)</code>	Crea un socket de servidor, que se enlaza al puerto especificado.
<code>ServerSocket(int port, int máximo)</code>	Crea un socket de servidor y lo enlaza con el número de puerto local especificado. El parámetro <i>máximo</i> especifica, el número máximo de peticiones de conexión que se pueden mantener en cola.
<code>ServerSocket(int port, int máximo, InetAddress direc)</code>	Crea un socket de servidor en el puerto indicado, especificando un máximo de peticiones y conexiones entrantes y la dirección IP local.

Algunos de los métodos de esta clase son:

MÉTODOS	MISIÓN
<code>Socket accept ()</code>	El método accept() escucha una solicitud de conexión de un cliente y la acepta cuando se recibe. Una vez que se ha establecido la conexión con el cliente, devuelve un objeto de tipo Socket , a través del cual se establecerá la comunicación con el cliente. Tras esto, el ServerSocket sigue disponible para realizar nuevos accept() . Puede lanzar IOException .
<code>close ()</code>	Se encarga de cerrar el ServerSocket .
<code>int getLocalPort ()</code>	Devuelve el puerto local al que está enlazado el ServerSocket .

Ejemplo:

Crea un socket de servidor y lo enlaza al puerto 6000, visualiza el puerto por el que se esperan las conexiones y espera que se conecten 2 clientes.

```
int Puerto = 6000; // Puerto
ServerSocket Servidor = new ServerSocket(Puerto);
System.out.println("Escuchando en " + Servidor.getLocalPort() );
```

```
Socket cliente1=Servidor.accept(); //esperando a un cliente
//realizar acciones con cliente1
```

```
Socket cliente2=Servidor.accept(); //esperando otro cliente
//realizar acciones con cliente2
```

```
Servidor.close(); //Se cierra el socket del servidor.
```

3.5.2.- Clase Socket

La clase Socket implementa un extremo de la conexión TCP . Algunos de sus constructores son los siguientes:

CONSTRUCTOR	MISIÓN
Socket()	Crea un socket sin ningún puerto asociado.
Socket (InetAddress address, int port)	Crea un socket y lo conecta al puerto y dirección IP especificados.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Permite además especificar la dirección IP local y el puerto local a los que se asociará el socket.
Socket (String host, int port)	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar <i>UnknownHostException</i> , <i>IOException</i>

Estos constructores pueden lanzar la excepción IOException

Métodos más importantes de la clase Socket

MÉTODOS	MISIÓN
InputStream getInputStream ()	Devuelve un InputStream que permite leer bytes desde el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <i>IOException</i> .
OutputStream getOutputStream ()	Devuelve un OutputStream que permite escribir bytes sobre el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <i>IOException</i> .

MÉTODOS	MISIÓN
close ()	Se encarga de cerrar el socket.
InetAddress getInetAddress ()	Devuelve la dirección IP a la que el socket está conectado. Si no lo está devuelve null.
int getLocalPort ()	Devuelve el puerto local al que está enlazado el socket, -1 si no está enlazado a ningún puerto.
int getPort ()	Devuelve el puerto remoto al que está conectado el socket, 0 si no está conectado a ningún puerto.

Ejercicio 1: Socket Servidor_Cliente Basico:

Crear una clase TCPServidorBasico enlazado en el puerto 6000

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

public class TCPServidorBasico {

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

        int Puerto = 6000; // Puerto
        ServerSocket Servidor = new ServerSocket(Puerto);

        System.out.println("Escuchando en " + Servidor.getLocalPort());
        Socket cliente1= Servidor.accept(); //esperando a un cliente
        //realizar acciones con cliente1
    }
}
```

```
Socket cliente2 = Servidor.accept(); //esperando a otro cliente
//realizar acciones con cliente2

Servidor.close(); //cierro socket servidor
}

}
```

Crea la clase *TCPClienteBasico* donde se cree un socket cliente y lo conecta al host local al puerto 6000 (tiene que haber un *ServerSocket* escuchando en ese puerto). Visualizar el puerto local al que está conectado el socket, y el puerto, host y dirección IP de la máquina remota a la que se conecta (en este caso es el host local).

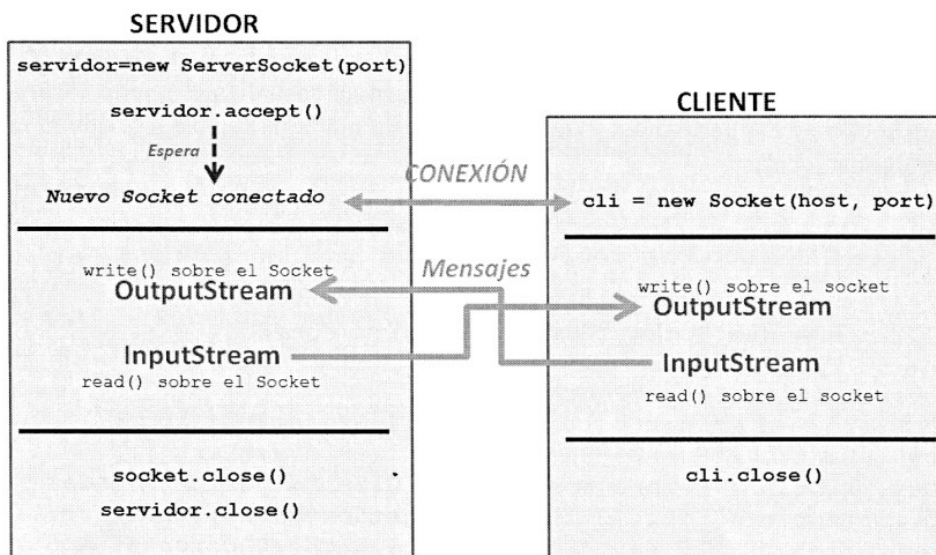
Nota: La clase servidor se debe de ejecutar antes de ejecutar la clase cliente.

El resultado de la salida debe ser similar a la siguiente:

```
Puerto local: 49745
Puerto Remoto: 6000
Nombre Host/IP: localhost/127.0.0.1
Host Remoto: localhost
IP Host Remoto: 127.0.0.1
```

3.5.3.- Gestión de sockets TCP

El modelo de socket más simple se muestra en la siguiente figura:



- El programa servidor crea un socket de servidor definiendo un puerto, mediante el método `ServerSocket(port)`, y espera mediante el método `accept()` a que el cliente solicite la conexión.
- Cuando el cliente solicita una conexión, el servidor abrirá la conexión al socket con el método `accept()`.

- El cliente establece una conexión con la maquina host a través del puerto especificado mediante el método `Socket(host, port)`.
- El cliente y el servidor se comunican con manejadores `InputStream` y `OutputStream`. El cliente escribe los mensajes en el `OutputStream` asociado al socket y el servidor leerá los mensajes del cliente del `InputStream`. Igualmente, el servidor escribirá los mensajes al `OutputStream` y el cliente los leerá del `InputStream`.

Apertura de sockets

En el **programa servidor** se crea un objeto **ServerSocket** invocando al método **ServerSocket()** en el que indicamos el número de puerto por el que el servidor escucha las peticiones de conexión de los clientes (se considera el tratamiento de excepciones)

```
ServerSocket servidor=null;
try{
    servidor = new ServerSocket(numeroPuerto);
} catch (IOException io) {
    io.printStackTrace();
}
```

Se necesita también crear un objeto `Socket` desde el `ServerSocket` para aceptar las conexiones, se usa el método `accept()`:

```
Socket clienteConectado=null;
try{
    clienteConectado=servidor.accept();
} catch (IOException io) {
    io.printStackTrace();
}
```

En el programa cliente es necesario crear un objeto `Socket`; el socket se abre de la siguiente manera:

```
Socket cliente;
try{
    cliente=new Socket ("maquina", numeroPuerto);
} catch (IOException io) {
    io.printStackTrace();
}
```

Donde “**maquina**” es el nombre de la maquina a la que nos queremos conectar y “**numeroPuerto**” es el puerto por el que el programa servidor está escuchando las peticiones de los clientes.

Hay puertos TCP de 0 a 65535. Los puertos en el rango de 0 a 1023 están reservados para servicios privilegiados; otros puertos de 1024 a 49151 están reservados para aplicaciones

concretas (por ejemplo el 3306 lo usa MySQL, el 1521 Oracle); por último de 49152 a 65535 no están reservados para ninguna aplicación concreta.

Creación de stream de entrada

En el programa servidor podemos usar **DataInputStream** para recuperar los mensajes que el cliente escriba en el socket, previamente hay que usar el método **getInputStream()** para obtener el flujo de entrada del socket del cliente:

```
InputStream entrada=null;
try{
    entrada=clienteConectado.getInputStream();
} catch (IOException e) {
    e.printStackTrace();
}
DataInputStream flujoEntrada = new DataInputStream (entrada);
```

En el programa cliente podemos realizar la misma operación para recibir los mensajes procedentes del programa servidor.

La clase **DataInputStream** permite la lectura de líneas de texto y tipos primitivos Java. Algunos de su métodos son:

- *readInt()*
- *readDouble()*
- *readLine()*
- *readUTF()*
-

Creación de streams de salida.

En el **programa servidor** podemos usar **DataOutputStream** para escribir los mensajes que queremos que el cliente reciba,, previamente hay que usar el método **getOutputStream()** para obtener el flujo de salida del socket del cliente:

```
OutputStream salida=null;
try{
    salida = clienteConectado.getOutputStream();
} catch (IOException e1) {
    e1.printStackTrace();
}
DataOutputStream flujoSalida=new DataOutputStream(salida);
```

NOTA: En el programa cliente tenemos que realizar la misma operación para enviar mensajes al programa servidor.

La clase `DataOutputStream` dispone de métodos para escribir tipos primitivos Java:

- `writeInt()`
- `writeDouble()`
- `writeBytes()`
- `writeUTF()`
-

Cierre de sockets

El orden de cierre de los socket es relevante, primero se han de cerrar los streams relacionados con un socket antes que el propio socket:

```
try{
    entrada.close();
    flujoEntrada.close();
    salida.close();
    flujoSalida.close();
    clienteConectado.close();
    servidor.close()
} catch (IOException e){
    e.printStackTrace();
}
```

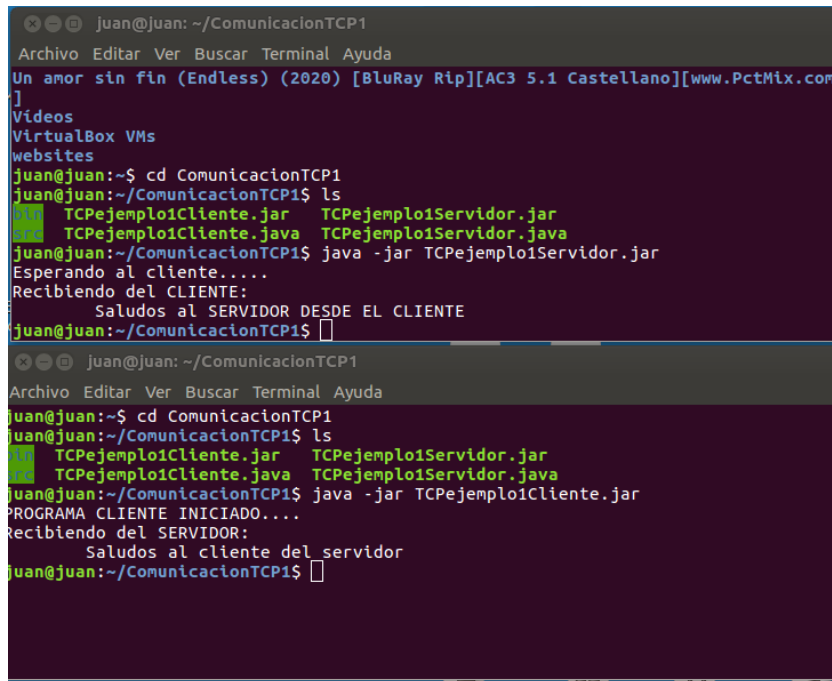
Ejemplo 2:Comunicación entre sockets.

Realiza una clase “`TCPejemplo1Servidor`” que recibe un mensaje de un programa cliente y lo muestra por pantalla; después envía un mensaje al cliente.

Realiza una clase “`TCPejemplo1Cliente`” que envíe un mensaje al programa servidor y después reciba un mensaje del servidor visualizándolo en pantalla; simplificar la obtención de los flujos de entrada y salida.

Convertir las clases en ficheros ejecutables y ejecutar ambas teniendo en cuenta que primero se debe de ejecutar el servidor.

En los mensajes indicar claramente vuestro nombre y quien es el cliente y el servidor.



```
juan@juan: ~/ComunicacionTCP1
Archivo Editar Ver Buscar Terminal Ayuda
Un amor sin fin (Endless) (2020) [BluRay Rip][AC3 5.1 Castellano][www.PctMix.com]
Videos
VirtualBox VMs
websites
juan@juan:~$ cd ComunicacionTCP1
juan@juan:~/ComunicacionTCP1$ ls
TCPejemplo1Cliente.jar  TCPejemplo1Servidor.jar
TCPejemplo1Cliente.java TCPejemplo1Servidor.java
juan@juan:~/ComunicacionTCP1$ java -jar TCPejemplo1Servidor.jar
Esperando al cliente....
Recibiendo del CLIENTE:
Saludos al SERVIDOR DESDE EL CLIENTE
juan@juan:~/ComunicacionTCP1$

juan@juan: ~/ComunicacionTCP1
Archivo Editar Ver Buscar Terminal Ayuda
juan@juan:~$ cd ComunicacionTCP1
juan@juan:~/ComunicacionTCP1$ ls
TCPejemplo1Cliente.jar  TCPejemplo1Servidor.jar
TCPejemplo1Cliente.java TCPejemplo1Servidor.java
juan@juan:~/ComunicacionTCP1$ java -jar TCPejemplo1Cliente.jar
PROGRAMA CLIENTE INICIADO....
Recibiendo del SERVIDOR:
Saludos al cliente del servidor
juan@juan:~/ComunicacionTCP1$
```

Clase Cliente

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

```
public class TCPejemplo1Cliente {
    public static void main(String[] args) throws Exception {
```

```
        System.out.println("PROGRAMA CLIENTE INICIADO....");
```

```
        // CREO FLUJO DE SALIDA AL SERVIDOR
```

```
        // ENVIO UN SALUDO AL SERVIDOR
```

```
        // CREO FLUJO DE ENTRADA AL SERVIDOR
```

```
        // EL SERVIDOR ME ENVIA UN MENSAJE
```

```
        // CERRAR STREAMS Y SOCKETS (Cuidado con el orden)
```

```
}// main  
}//
```

Clase Servidor

```
import java.io.*;  
import java.net.*;  
  
public class TCPejemplo1Servidor {  
    public static void main(String[] arg) throws IOException {  
        // Puerto  
  
        System.out.println("Esperando al cliente.....");  
  
        // CREO FLUJO DE ENTRADA DEL CLIENTE  
  
        // EL CLIENTE ME ENVIA UN MENSAJE  
  
        // CREO FLUJO DE SALIDA AL CLIENTE  
  
        // ENVIO UN SALUDO AL CLIENTE  
  
        // CERRAR STREAMS Y SOCKETS  
  
    }// main  
}// fin
```

Ejemplo 3 Comunicaciones entre sockets

Cambiar host = “localhost” por la dirección de un compañero y probar el envío de mensaje.
Recordar que el servidor tiene que estar a la escucha antes de que el cliente envíe el mensaje.

3.6.- Clases para sockets UDP

Los sockets UDP son más simples y eficientes que los TCP pero sin embargo, no garantiza la entrega de paquetes. No es necesario establecer una “conexión” entre cliente y servidor, como en el caso de los sockets TCP, por ello cada vez que se envíen datagramas el emisor debe indicar explícitamente la dirección IP y el puerto del destino para cada paquete y el receptor debe extraer la dirección IP y el puerto del emisor del paquete.

El paquete del datagrama está formado por los siguientes campos:

CADENA DE BYTES CONTENIENDO EL MENSAJE	LONGITUD DEL MENSAJE	DIRECCIÓN IP DESTINO	Nº DE PUERTO DESTINO
---	-------------------------	-------------------------	-------------------------

El paquete `java.net` proporciona las clases `DatagramPacket` y `DatagramSocket` para implementar sockets UDP.

Clase `DatagramPacket`

Esta clase proporciona constructores para crear instancias de los datagramas que se van a recibir y de los datagramas que van a ser enviados:

CONSTRUCTOR	MISIÓN
<code>DatagramPacket(byte[] buf, int length)</code>	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje (<i>buf</i>) y la longitud (<i>length</i>) de la misma.
<code>DatagramPacket(byte[] buf, int offset, int length)</code>	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje, la longitud de la misma y el offset (<i>offset</i>) dentro de la cadena.
<code>DatagramPacket(byte[] buf, int length, InetAddress address, int port)</code>	Constructor para el envío de datagramas. Se especifica la cadena de bytes a enviar (<i>buf</i>), la longitud (<i>length</i>), el número de puerto de destino (<i>port</i>) y el host especificado en la dirección <i>address</i> .
<code>DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)</code>	Constructor para el envío de datagramas. Igual que el anterior pero se especifica un offset dentro de la cadena de bytes.

Ejemplo:

Vamos a usar el tercer constructor del cuadro para construir un datagrama de envío. El datagrama se enviará a la dirección IP de la máquina local, que lo estará esperando por el puerto 12345.

El mensaje está formado por la cadena “Enviando saludos!! “ que es necesario codificar en una secuencia de bytes y almacenar el resultado en una matriz de bytes. Después será necesario calcular la longitud del mensaje a enviar.

Con `InetAddress.getLocalHost()` obtenemos la dirección IP del host al que enviaré el mensaje, en este caso se trata del host local.

<i>mensaje: Enviando Saludos !!,</i>	<i>Longitud: 19</i>	<i>destino: 192.168.21 IP del host local</i>	<i>port: 12345</i>
--------------------------------------	---------------------	--	--------------------

```
int puerto = 12345; //puerto por el que se realiza la escucha.
```

```
InetAddress destino = InetAddress.getLocalHost(); //IP del host localizar
```

```
byte[] mensaje = new byte[1024]; // matriz de bytes
```

```
String Saludo = "Enviando un saludo soy el alumno ..... !!";
```

```
mensaje = Saludo.getBytes(); // codificamos el mensaje a bytes para enviarlo
```

```
// Construimos el datagrama a enviar con el tercer constructor.
```

```
DatagramPacket envio = new DatagramPacket(mensaje, mensaje.length, destino, puerto);
```

Nota: Para definir el destino de un host con una IP concreta, por ejemplo la 192.168.10.5, escribo lo siguiente:

```
InetAddress destino = InetAddress.getByName("192.168.10.5");
```

Algunos métodos de la clase DatagramPacket

MÉTODOS	MISIÓN
InetAddress getAddress ()	Devuelve la dirección IP del host al cual se le envía el datagrama o del que el datagrama se recibió.
byte[] getData()	Devuelve el mensaje contenido en el datagrama tanto recibido como enviado.
int getLength()	Devuelve la longitud de los datos a enviar o a recibir.
int getPort()	Devuelve el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió el datagrama.
setAddress (InetAddress addr)	Establece la dirección IP de la máquina a la que se envía el datagrama.
setData (byte [buf])	Establece el búfer de datos para este paquete.
setLength (int length)	Ajusta la longitud de este paquete.
setPort (int Port)	Establece el número de puerto del host remoto al que este datagrama se envía.

Ejemplo de datagrama de recepción

Usaremos el primer constructor de DatagramPacket para construir un datagrama de recepción.

El mensaje se aloja en la variable buffer, se obtiene la longitud y el mensaje contenido en el datagrama recibido, el mensaje se convierte a String . A continuación visualiza el número de puerto de la maquina que envía el mensaje y su dirección IP.

```
byte[] bufer = new byte [1024];  
DatagramPacket recibo = new DatagramPacket (buffer, bufer.length);  
  
int bytesRec = recibo.getLength(); //obtenemos la longitud del mensaje.  
String paquete = new String (recibo.getData() ); // obtenemos el mensaje recibido  
  
System.out.println ("Puerto de origen del mensaje : "+ recibo.getPort() );  
  
System.out.println ("IP de origen :"+ recibo.getAddress().getHostAddress() );
```

Clase Datagram Socket

Da soporte a socket para el envío y recepción de datagramas UDP. Algunos de los constructores de esta clase , que pueden lanzar la excepción "SocketException", son:

CONSTRUCTOR	MISIÓN
DatagramSocket ()	Construye un socket para datagramas, el sistema elige un puerto de los que están libres.
DatagramSocket (int port)	Construye un socket para datagramas y lo conecta al puerto local especificado.
DatagramSocket (int port, InetAddress ip)	Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket.

Ejemplo de construcción de un socket para datagrama y no lo conecta a ningún puerto, el sistema elige el puerto:

DatagramSocket socket = new DatagramSocket ();

Para enlazar el socket a un puerto específico deberemos escribir lo siguiente:

DatagramSocket socket = new DatagramSocket (12345);

Algunos métodos importante de la clase DatagramSocket

MÉTODOS	MISIÓN
receive (DatagramPacket paquete)	Recibe un DatagramPacket del socket, y llena <i>paquete</i> con los datos que recibe (mensaje, longitud y origen). Puede lanzar la excepción IOException .
send (DatagramPacket paquete)	Envía un DatagramPacket a través del socket. El argumento <i>paquete</i> contiene el mensaje y su destino. Puede lanzar la excepción IOException .
close ()	Se encarga de cerrar el socket.
int getLocalPort ()	Devuelve el número de puerto en el host local al que está enlazado el socket, -1 si el socket está cerrado y 0 si no está enlazado a ningún puerto.
int getPort()	Devuelve el número de puerto al que está conectado el socket, -1 si no está conectado.
connect(InetAddress address, int port)	Conecta el socket a un puerto remoto y una dirección IP concretos, el socket solo podrá enviar y recibir mensajes desde esa dirección.
setSoTimeout(int timeout)	Permite establecer un tiempo de espera límite. Entonces el método <i>receive()</i> se bloquea durante el tiempo fijado. Si no se reciben datos en el tiempo fijado se lanza la excepción InterruptedIOException .

Siguiendo con el ejemplo anterior, una vez construido el datagrama, lo enviamos usando un DatagramSocket, en el ejemplo se enlaza al puerto 34567.

Mediante el método **send()** se envía el datagrama:

//Construimos el datagrama a enviar indicando el host de destino y su puerto

DatagramPacket envio = new DatagramPacket (mensaje, mensaje.length, destino, port);

Datagram socket = new DatagramSocket (34567);

socket.send(envio); //envio datagrama a destino y puerto

En el otro extremo, para recibir el datagrama usamos también un DatagramSocket. En primer lugar habrá que enlazar el socket al puerto por el que se va a recibir el mensaje. En este caso el puerto 12345.

Después se construye el datagrama para recepción y mediante el método receive() obtenemos los datos. Luego obtenemos la longitud, la cadena y visualizamos los puertos origen y destino del mensaje.

```
DatagramSocket socket = new DatagramSocket(12345);

//construyo datagrama a recibir
DatagramPacket recibo = new DatagramPacket(bufer, bufer.length);
System.out.println("Esperando Datagrama ..... ");

socket.receive(recibo); //recibo datagrama

int bytesRec = recibo.getLength(); //obtengo numero de bytes
String paquete = new String(recibo.getData()); //obtengo String

System.out.println("Número de Bytes recibidos: " + bytesRec);
System.out.println("Contenido del Paquete: " + paquete.trim());
System.out.println("Puerto origen del mensaje: " + recibo.getPort());
System.out.println("IP de origen: " + recibo.getAddress().getHostAddress());
System.out.println("Puerto destino del mensaje: " + socket.getLocalPort());

socket.close(); //cierro el socket
```

Ejercicio 4 : Envío y recepción de datagrama:

1. Crea la clase “UDPEnvioDatagrama”. Contendrá el main que deberá hacer lo siguiente: el puerto de escucha será el 12345, la dirección de destino será “localhost”, creará una matriz “mensaje de 2048 bytes para enviar el siguiente saludo: (“Enviando un saludo de nombre alumno”.
2. Codifica el mensaje a bytes para enviarlo.
3. Construye el datagrama con el mensaje y el puerto
4. Crea el socket para enviar el datagrama por el puerto 34567
5. Cierra el Socket
6. Crea la clase UDPreciboDatagrama, que contendrá solo el main para realizar lo siguiente:
7. Crea el array bufer de 2048 byte
8. Asocia un nuevo socket al puerto 12345
9. Construye el datagrama a recibir y escribe el mensaje “Esperando Datagrama”
10. Recibe el Datagrama
11. Obtener el número de bytes y la cadena del mensaje.
12. Escribir lo siguiente:
 - Número de bytes recibidos
 - Contenido del paquetes
 - Puerto de origen del mensaje
 - IP de origen (comprueba que es tu ip)

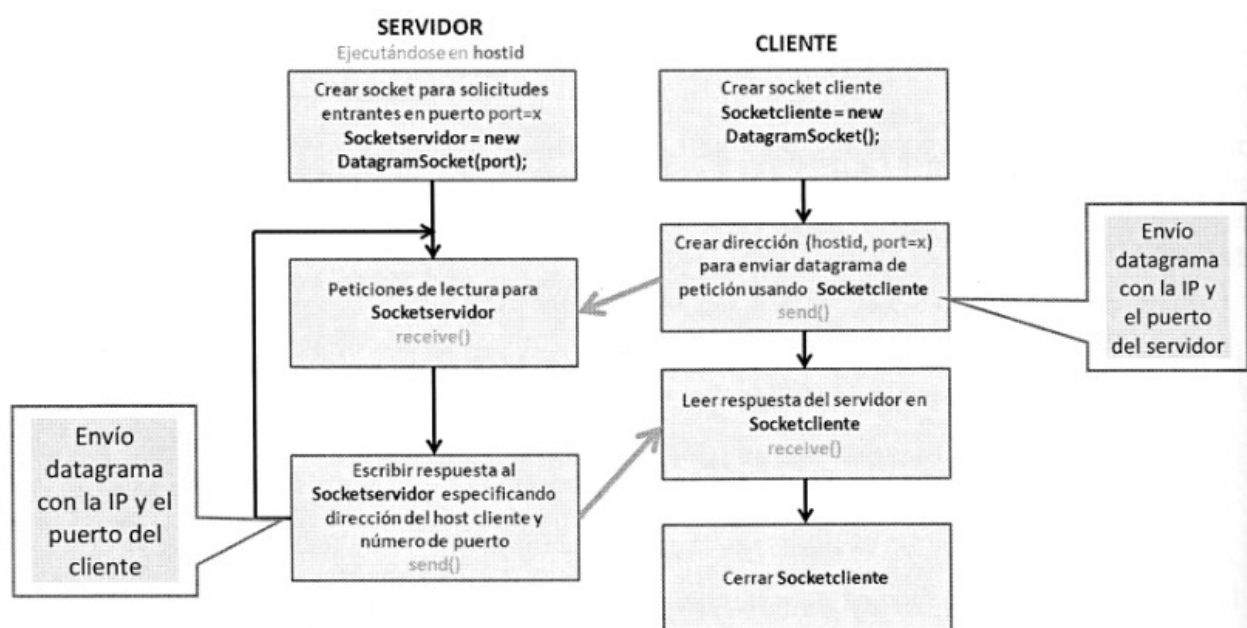
- Puerto destino del mensaje

Nota: Ejecutar primero la clase UDPreciboDatagrama y posteriormente la clase UDPEnvioDatagrama.

3.6.1.- Gestión de Sockets UDP

En los socket UDP no se establece conexión. Los roles cliente-servidor están un poco mas difusos que en el caso de TCP. Podemos considerar al servidor como el que espera un mensaje y responde; y al cliente como el que inicia la comunicación. Tanto uno como otro si desean ponerse en contacto necesitan saber en qué ordenador y en qué puerto está escuchando el otro.

En la siguiente figura se muestra el flujo de la comunicación entre cliente y servidor usando UDP, ambos necesitan crear un socket DatagramSocket.



- El **servidor** crea un socket asociado a un puerto local para escuchar peticiones de clientes. Permanece a la espera de recibir peticiones.
- El **cliente** creará un socket para comunicarse con el servidor. Para enviar datagramas necesita conocer su IP y el puerto por el que escucha. Utilizará el método `send()` del socket para enviar la petición en forma de datagrama.
- El **servidor** recibe las peticiones mediante el método `receive()` del socket. En el datagrama va incluido además del mensaje, el puerto y la IP del cliente emisor de la petición; lo que le permite al servidor conocer la dirección del emisor del datagrama. Utilizando el método `send()` del socket puede enviar la respuesta al cliente emisor.
- El **cliente** recibe la respuesta del servidor mediante el método `receive()` del socket.
- El **servidor** permanece a la espera de recibir más peticiones

Apertura y cierre de sockets

Para construir un socket datagrama es necesario instanciar la clase DatagramSocket tanto en el programa cliente como en el servidor. Para escuchar peticiones en un puerto UDP concreto pasamos al constructor el número de puerto por el que escucha las peticiones y la dirección "InetAddress" en la que se está ejecutando el programa, que normalmente es InetAddress.getLocalHost()

Ejemplo:

```
DatagramSocket socket = new DatagramSocket (12345 , InetAddress.getByName("localhost") );
```

Para cerrar el socket usamos el método close(): *socket.close()*

Envío y recepción de datagrama.

Para crear los datagramas de envío y recepción usamos la clase **DatagramPacket**

Para enviar usamos el método send () de DatagramSocket pasando como parámetro el DatagramPacket que creamos previamente.

```
DatagramPacket datagrama = new DatagramPacket (mensajeEnBytes, mensajeEnBytes.length,  
InetAddress.getByName("localhost"), PuertoDelServidor);
```

Donde:

- **mensajeEnBytes**: es el array de bytes
- **mensajeEnBytes.length**: La longitud del array
- **InetAddress.getByName("localhost")**: La maquina de destino
- **PuertoDelServidor**: puerto del destinatario

```
socket.send(datagrama)
```

Para recibir usamos el método receive() de DatagramSocket pasando como parámetro el DatagramPacket que creamos . Este método se bloquea hasta que se recibe un datagrama, a menos que se establezca un tiempo limite (timeout) sobre el socket.

```
DatagramPacket datagrama = new DatagramPacket (new byte[1024], 1024);  
socket.receive(datagrama);
```

Ejercicio 5: Servidor_Cliente (Realizar la comprobación en pareja, uno es el cliente y otro el servidor)

Nota: Ambos alumnos deben de implementar las dos clases

1. Crea la clase UDPServidor que recibe un datagrama enviado por un programa cliente. Este programa servidor debe permanecer a la espera hasta que le llega un paquete del cliente; en este momento visualiza:
 - a) El número de bytes recibidos

- b) El contenido del paquete
 - c) El puerto
 - d) La IP del cliente y el puerto local por el que recibe las peticiones.
2. Crea la clase UDPcliente que envíe un mensaje al servidor por el puerto 12345 por el que espera peticiones ; la IP del servidor es 192. 168.0.11 (comprobar la IP del compañero)

InetAddress destino = InetAddress.getByName("192.168.0.11")

Visualiza el nombre del host de destino y la dirección IP. También debe de imprimir el puerto local del socket y el puerto al que envía el mensaje.

Ejercicio 6: Servidor_Cliente 2 (Realizar la comprobación en pareja, uno es el cliente y otro el servidor)

Nota: Ambos alumnos deben de implementar las dos clases

En este ejemplo, el programa cliente envía un texto tecleado en su entrada estándar al servidor (en un puerto pactado), el servidor lee el datagrama y devuelve al cliente el número de apariciones de la letra “a” por ejemplo , en el texto. El programa cliente recibe el datagrama del servidor y muestra el número de repeticiones de la letra “a” . Para comenzar la prueba, debemos de ejecutar primero el programa servidor que permanecerá a la espera , y después (desde el otro equipo) ejecutamos el programa cliente.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPServidorEjemplo2 {

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

        // Asocio el socket al puerto 12345

        System.out.println("Servidor Esperando Datagrama ..... ");

        // creo el bufer para recibir el datagrama

        //recibo datagrama

        // obtengo String
```

```
//cuento el número de letras a

// DIRECCION ORIGEN DEL MENSAJE

// ENVIANDO DATAGRAMA AL CLIENTE

// paso entero a byte

// CERRAR STREAMS Y SOCKETS
}
}

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

import java.util.Scanner;

public class UDPclienteEjemplo2 {

    private static Scanner sc = new Scanner(System.in);

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

        // socket cliente

        // DATOS DEL SERVIDOR al que enviar mensaje

        // puerto por el que escucha

        // INTRODUCIR DATOS POR TECLADO

        // ENVIANDO DATAGRAMA AL SERVIDOR

        // RECIBIENDO DATAGRAMA DEL SERVIDOR

        // Obtener el número e caracteres

        // cerrar socket
```

3.6.2.- MulticastSocket

La clase MulticastSocket es útil para enviar paquetes a múltiples destinos simultáneamente. Para poder recibir estos paquetes es necesario establecer un grupo multicast. Este grupo multicast no es más que un grupo de direcciones IP que comparten el mismo número de puerto. Cuando se envía un mensaje a un grupo multicast, todos los que estén en ese grupo recibirán el mensaje; la pertenencia al grupo es transparente al emisor, es decir, el emisor no conoce el número de miembros del grupo ni sus direcciones IP.

Un grupo multicast se especifica mediante una dirección IP de clase D y un número de puerto UDP estándar.

Las direcciones desde la 224.0.0.0 a la 239.255.255.255 están destinadas para ser direcciones de multicast. La dirección 224.0.0.0 está reservada y no debe ser utilizada.

La clase MulticastSocket tiene varios constructores que pueden lanzar la excepción IOException.

CONSTRUCTOR	MISIÓN
MulticastSocket ()	Construye un socket multicast dejando al sistema que elija un puerto de los que están libres.
MulticastSocket (int port)	Construye un socket multicast y lo conecta al puerto local especificado.

Algunos métodos importantes de la clase MulticastSocket ; estos métodos pueden lanzar la excepción IOException

MÉTODO	MISIÓN
void joinGroup(InetAddress mcastaddr)	Permite al socket multicast unirse al grupo de multicast.
void leaveGroup(InetAddress mcastaddr)	El socket multicast abandona el grupo de multicast.
void send(DatagramPacket p)	Envía el datagrama a todos los miembros del grupo multicast.
void receive(DatagramPacket p)	Recibe el datagrama de un grupo multicast

El esquema general para un servidor multicast que envía paquetes a todos los miembros del grupo es el siguiente:

1. Creamos el socket multicast; no hace falta especificar el puerto:
MulticastSocket ms = new MulticastSocket ();
2. Definimos el puerto multicast
int Puerto = 12345;
3. Creamos el grupo multicast
InetAddress grupo = InetAddress.getByName("225.0.0.1");

- ```
String mensaje = "Bienvenidos a grupo ";
```
4. Creamos el datagrama  
**DatagramPacket paquete = new DatagramPacket (mensaje.getBytes(),  
mensaje.length(), grupo, Puerto);**
  5. Enviamos el paquete al grupo  
**mensaje.send(paquete);**
  6. Cerramos el socket  
**mensaje.close();**

Para que un cliente se una al grupo multicast primero crea un MulticastSocket asociado al mismo puerto que el servidor y luego invoca al método joinGroup(). El cliente multicast que recibe los paquetes que le envía el servidor tiene la siguiente estructura:

1. Se crea un socket multicast en el puerto 12345  
**MulticastSocket mensaje = new MulticastSocket (12345);**
2. Se configura la IP del grupo al que nos conectaremos  
**InetAddress grupo = InetAddress.getByName("225.0.0.1");**
3. Se une al grupo  
**mensaje.joinGroup (grupo);**
4. Recibe el paquete del servidor multicast  
**byte[] bufer= new byte[1000];**  
**DatagramPacket recibido = new DatagramPacket (bufer, bufer.length);**  
**mensaje.receive(recibido);**
5. Abandona el grupo multicast  
**mensaje.leaveGroup (grupo);**
6. Se cierra el socket  
**mensaje.close()**

### Ejercicio 7: Multicast

Crea dos clases, la clase Mcservidor y la clase Mccliente. La clase Mcservidor tiene que leer datos por el teclado y los envía a todos los clientes que pertenezcan al mismo grupo multicast, el proceso debe de terminar cuando se introduce un "\*"

La clase cliente debe de visualizar el paquete que recibe del servidor, su proceso finaliza cuando recibe el "\*".

**NOTA:** Probar la aplicación con al menos dos compañeros (dos clientes) ejecutando el ejercicio en una consola (convertir el fichero .java en ejecutable ".jar")

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

public class Mcservidor {
 public static void main(String args[]) throws Exception {
 // FLUJO PARA ENTRADA ESTANDAR

 //Se crea el socket multicast.

 //Puerto multicast
```

```
//Grupo

// ENVIANDO AL GRUPO

//cierro socket

System.out.println ("Socket cerrado...");

import java.io.*;
import java.net.*;
public class MCcliente {
 public static void main(String args[]) throws Exception {
 //Se crea el socket multicast

 //Puerto multicast

 //Grupo

 //Nos unimos al grupo

 //Recibe el paquete del servidor multicast

 //abandonamos grupo

 //cierra socket

 System.out.println ("Socket cerrado...");
```