

# PROGRAMACIÓN DE COMUNICACIONES EN RED.

## SOCKETS.

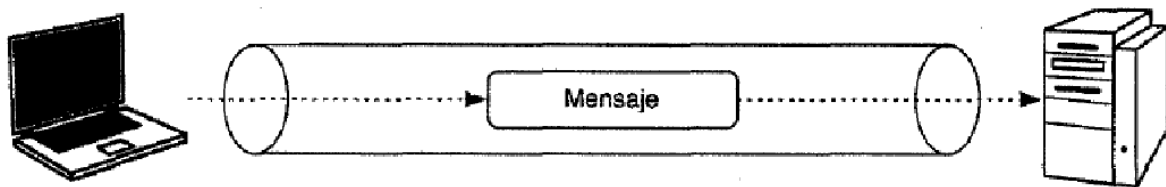
### Índice de contenido

1.SOCKETS.....	2
1.1.FUNCIONAMIENTO GENERAL DE UN SOCKET.....	2
1.2.TIPOS DE SOCKETS.....	3
1.2.1.SOCKETS ORIENTADOS A CONEXIÓN, SOCKETS TCP O STREAM SOCKETS..	3
1.2.2.SOCKETS NO ORIENTADOS A CONEXIÓN, SOCKETS UDP O DATAGRAM	
SOCKETS.....	4
1.3.CLASES PARA SOCKETS TCP.....	4
1.4.GESTIÓN DE SOCKETS TCP.....	6
1.5.CLASES PARA SOCKETS UDP.....	17
1.6.GESTIÓN DE SOCKETS UDP.....	21

## 1. SOCKETS.

Un **socket** (en inglés, literalmente, un "enchufe") representa el extremo de un canal de comunicación establecido entre un emisor y un receptor. Para establecer una comunicación entre dos aplicaciones, ambas deben crear sus respectivos sockets, y conectarlos entre sí. Una vez conectados, entre ambos sockets se crea una "tubería privada" a través de la red, que permite que las aplicaciones en los extremos envíen y reciban mensajes por ella. El procedimiento concreto por el cual se realizan estas operaciones depende del tipo de socket que se desee utilizar.

Para enviar mensajes por el canal de comunicaciones, las aplicaciones *escriben* (*write*) en su *socket*. Para recibir mensajes *leen* (*read*) de su *socket*.



Los protocolos TCP y UDP utilizan la abstracción 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 llamamos **socket**.

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.

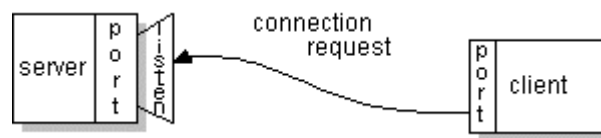
Así, todos los mensajes enviados a esa dirección IP y a ese puerto concreto llegarán al proceso receptor. Cuando un proceso cliente envía un mensaje y un proceso servidor recibe un mensaje, comunicándose mediante sockets, cada socket tiene un puerto asociado, el proceso cliente debe conocer el puerto 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.

Los procesos pueden utilizar un mismo conector tanto para enviar como para recibir mensajes. Cada conector se asocia con un protocolo concreto que puede ser **UDP** o **TCP**.

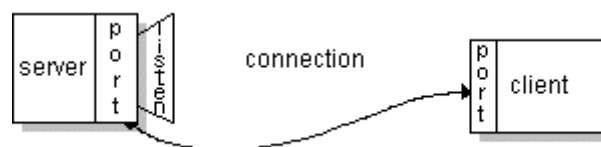
## 1.1.FUNCIONAMIENTO GENERAL DE UN SOCKET

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 del cliente, si se acepta la conexión, se crea un socket y el cliente puede utilizarlo para comunicarse con el servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo por sus respectivos sockets.

## 1.2. TIPOS DE SOCKETS

Hay dos tipos básicos de sockets en redes IP: los que utilizan el protocolo **TCP (stream sockets)**, orientados a conexión; y los que utilizan el protocolo **UDP (datagram sockets)**, no orientados a conexión.

### 1.2.1.SOCKETS ORIENTADOS A CONEXIÓN, SOCKETS TCP O STREAM SOCKETS

La comunicación entre las aplicaciones se realiza por medio del protocolo TCP. Por tanto es una conexión fiable en la que se garantiza la entrega de los paquetes de datos y el orden en que fueron enviados. TCP utiliza un esquema de acuse de recibo de los mensajes de tal forma que si el emisor

no recibe dicho acuse dentro de un tiempo determinado, vuelve a transmitir el mensaje.

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

Los sockets TCP se utilizan en la gran mayoría de las aplicaciones IP. Algunos servicios con sus números de puerto reservados son: FTP (puerto 21), Telnet (23), HTTP (80), SMTP (25).

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

### 1.2.2. SOCKETS NO ORIENTADOS A CONEXIÓN, SOCKETS UDP O DATAGRAM SOCKETS

En este tipo de sockets la comunicación entre las aplicaciones se realiza por medio del protocolo UDP. Esta conexión no es fiable y no garantiza que la información enviada llegue a su destino, tampoco se garantiza el orden de llegada de los paquetes que puede llegar en distinto orden al que se envía. 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; algunas aplicaciones como NFS (Network File System), DNS (Domain Name Server) o SNMP (Simple Network Management Protocol) usan este protocolo.

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

### 1.3. CLASES PARA SOCKETS TCP

El paquete **java.net** proporciona las clases **ServerSocket** y **Socket** para trabajar con sockets TCP. TCP es un protocolo orientado a conexión, por lo que para establecer una comunicación es necesario especificar una conexión entre un par de sockets. Uno de los sockets, el cliente, solicita una conexión, y el otro socket, el servidor, atiende las peticiones de los clientes. Una vez que los

dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

## Clase ServerSocket

La clase **ServerSocket** 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 (pueden lanzar la excepción *IOException*):

CONSTRUCTOR	MISIÓN
<b>ServerSocket()</b>	Crea un socket de servidor sin ningún puerto asociado
<b>ServerSocket(int port)</b>	Crea un socket de servidor, que se enlaza al puerto especificado
<b>ServerSocket(int port, int máximo)</b>	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
<b>ServerSocket(int port, int máximo, InetAddress direc)</b>	Crea un socket de servidor en el puerto indicado, especificando un máximo de peticiones y conexiones entrantes y la dirección IP local

Algunos métodos importantes son:

MÉTODOS	MISIÓN
<b>Socket accept()</b>	El método <i>accept()</i> 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 <b>Socket</b> , a través del cual se establecerá la comunicación con el cliente. Tras esto, el <b>ServerSocket</b> sigue disponible para realizar nuevos <i>accept()</i> . Puede lanzar <i>IOException</i>
<b>close()</b>	Se encarga de cerrar el <b>ServerSocket</b>
<b>int getLocalPort()</b>	Devuelve el puerto local al que está enlazado el <b>ServerSocket</b>

El siguiente 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 a otro cliente
//realizar acciones con cliente2
Servidor.close(); //cierro socket servidor
```

## Clase Socket

La clase `Socket` implementa un extremo de la conexión TCP. Algunos de sus constructores son (pueden lanzar la excepción *IOException*):

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

Algunos métodos importantes son:

MÉTODOS	MISIÓN
<code>InputStream getInputStream()</code>	Devuelve un <b>InputStream</b> que permite leer bytes desde el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <i>IOException</i>
<code>OutputStream getOutputStream()</code>	Devuelve un <b>OutputStream</b> que permite escribir bytes sobre el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <i>IOException</i>
<code>close()</code>	Se encarga de cerrar el socket
<code>InetAddress getInetAddress()</code>	Devuelve la dirección IP y puerto a la que el socket está conectado. Si no lo está devuelve null
<code>int getLocalPort()</code>	Devuelve el puerto local al que está enlazado el socket, -1 si no está enlazado a ningún puerto
<code>int getPort ()</code>	Devuelve el puerto remoto al que está conectado el socket, 0 si no está conectado a ningún puerto

El siguiente ejemplo crea un socket cliente y lo conecta al host local al puerto 6000 (tiene que haber un **ServerSocket** escuchando en ese puerto). Después visualiza 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):

```
String Host = "localhost";
int Puerto = 6000;//puerto remoto

// ABRIR SOCKET
Socket Cliente = new Socket(Host, Puerto);//conecta

InetAddress i= Cliente.getInetAddress ();
System.out.println ("Puerto local: "+ Cliente.getLocalPort());
System.out.println ("Puerto Remoto: "+ Cliente.getPort());
System.out.println ("Host Remoto: "+ i.getHostName() .toString());
System.out.println ("IP Host Remoto: "+ i.getHostAddress() .toString());
Cliente.close();// Cierra el socket
```

La salida que se genera es la siguiente:

Puerto local: 8784

Puerto remoto: 6000

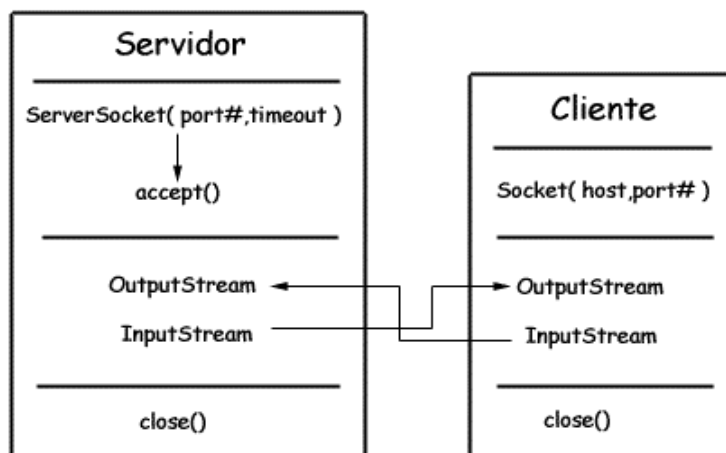
Host Remoto: localhost

IP Host Remoto: 127.0.0.1

## 1.4. GESTIÓN DE SOCKETS TCP

El modelo de sockets más simple sería el siguiente:

- 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 máquina 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 de **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();
}
```

```
}
```

Necesitamos 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 ("máquina", numeroPuerto);
} catch (IOException io) {
    io.printStackTrace();
}
```

Donde *máquina* es el nombre de la máquina 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 streams 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 sus métodos son: *readInt()*, *readDouble()*, *readLine()*, *readUTF()*, etc.

### Creación de streams de salida



En el **programa servidor** podemos usar **DataOutputStream** para escribir los mensajes que queramos 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 el) {
    el.printStackTrace();
}
DataOutputStream flujoSalida = new DataOutputStream(salida);
```

En el **programa cliente** podemos 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()*, etc.

### Cierre de sockets

El orden de cierre de los sockets 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();
}
```

A continuación se muestra un ejemplo de un **programa servidor** que recibe un mensaje de un cliente y lo muestra por pantalla; después envía un mensaje al cliente.

```

import java.io.*;
import java.net.*;
public class ejemplo1Servidor {
    public static void main(String[] arg) throws IOException {
        int numeroPuerto = 6000; // Puerto
        try{
            ServerSocket servidor = new ServerSocket(numeroPuerto) ;
            Socket clienteConectado = null;
            System.out.println("Esperando al cliente ..... ");
            clienteConectado = servidor.accept();

            //CREO FLUJO DE ENTRADA DEL CLIENTE
            InputStream entrada = null;
            entrada = clienteConectado.getInputStream();
            DataInputStream flujoEntrada = new DataInputStream(entrada);

            //EL CLIENTE ME ENVIA UN MENSAJE
            System.out.println("Recibiendo del CLIENTE: \n\t"+flujoEntrada.readUTF());

            //CREO FLUJO DE SALIDA AL CLIENTE
            OutputStream salida = null;
            salida = clienteConectado.getOutputStream();
            DataOutputStream flujoSalida = new DataOutputStream(salida);

            //ENVIO UN SALUDO AL CLIENTE
            flujoSalida.writeUTF("Saludos al cliente del servidor");

            //CERRAR STREAMS y SOCKETS
            entrada.close();
            flujoEntrada.close();
            salida.close();
            flujoSalida.close();
            clienteConectado.close();
            servidor.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    } // main
} // fin

```

El **programa cliente**, en primer lugar envía un mensaje al servidor y después recibe un mensaje del servidor visualizándolo en pantalla:

```
import java.io.*;
import java.net.*;
public class ejemplo1Cliente {
    public static void main(String[] args) throws Exception {
        String Host = "localhost";
        int Puerto = 6000; //puerto remoto
        try{
            System.out.println("PROGRAMA CLIENTE INICIADO .... ");
            Socket Cliente = new Socket(Host, Puerto);

            //CREO FLUJO DE SALIDA AL SERVIDOR
            DataOutputStream flujoSalida = new DataOutputStream(Cliente.getOutputStream());

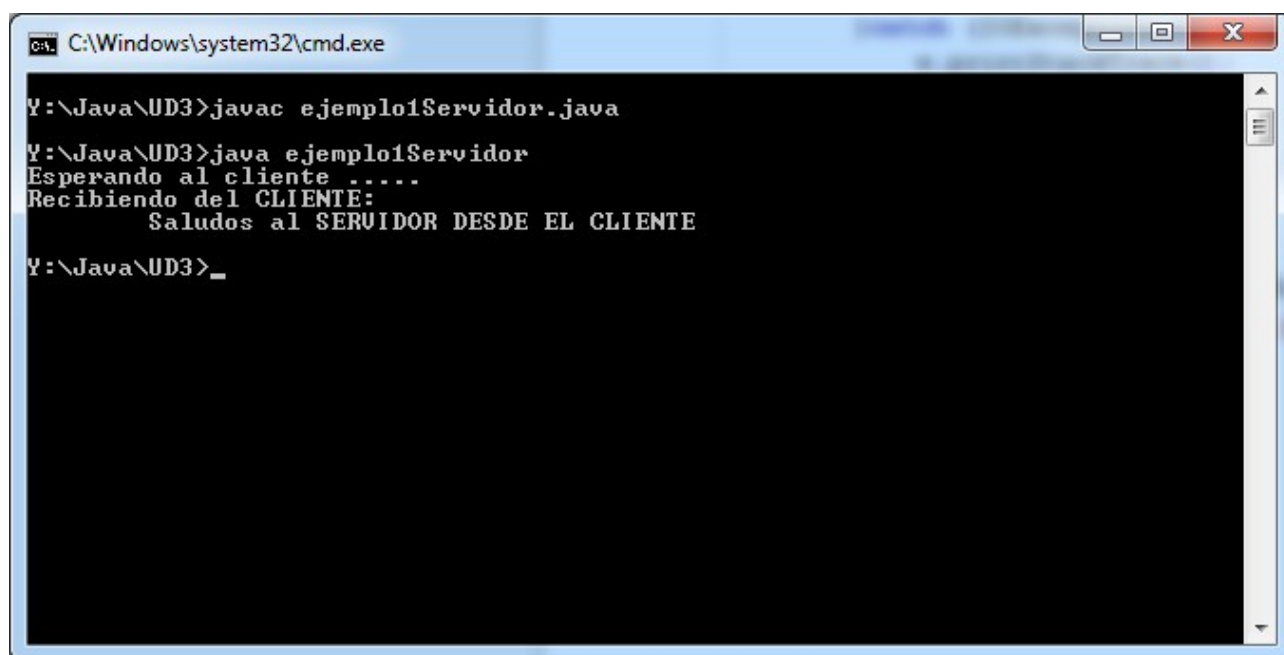
            //ENVIO UN SALUDO AL SERVIDOR
            flujoSalida.writeUTF("Saludos al SERVIDOR DESDE EL CLIENTE");

            //CREO FLUJO DE ENTRADA AL SERVIDOR
            DataInputStream flujoEntrada = new DataInputStream(Cliente.getInputStream());

            //EL SERVIDOR ME ENVIA UN MENSAJE
            System.out.println("Recibiendo del SERVIDOR: \n\t"+flujoEntrada.readUTF());

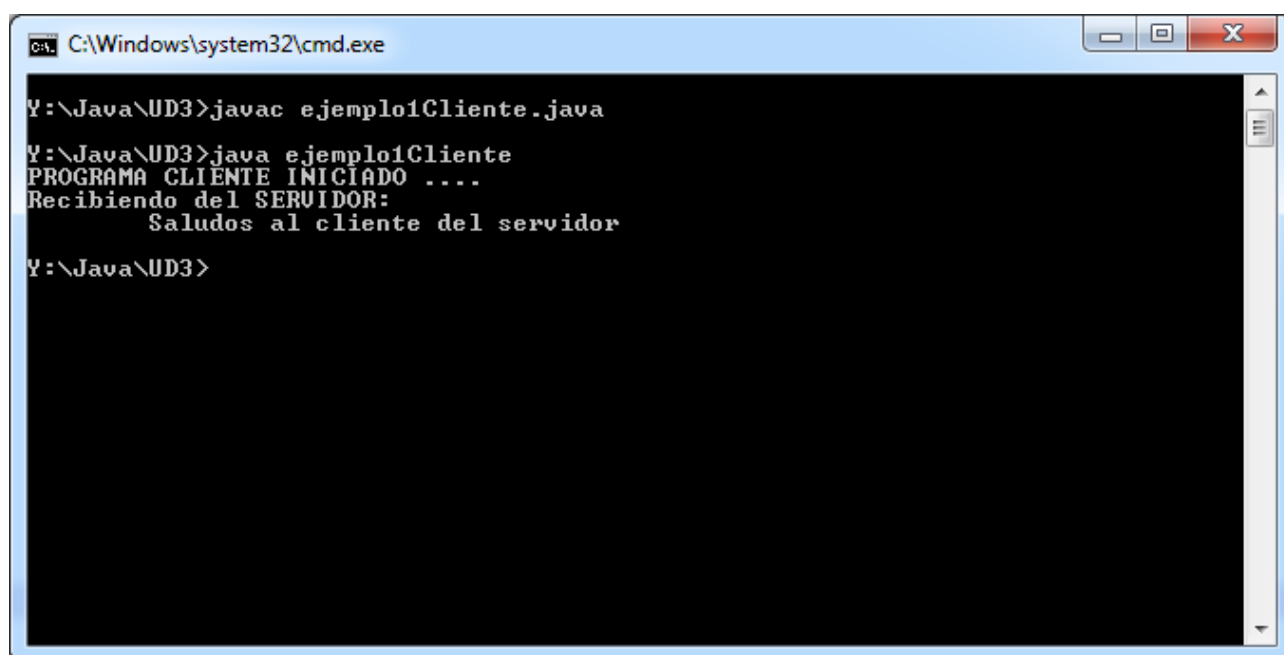
            //CERRAR STREAMS y SOCKETS
            flujoEntrada.close();
            flujoSalida.close();
            Cliente.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    } // main
} //
```

La compilación y ejecución se muestra debajo. En una ventana se ejecuta el programa servidor y en otra se ejecuta el programa cliente. **Es importante que se ejecute el servidor antes que el cliente.**



```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac ejemplo1Servidor.java
Y:\Java\UD3>java ejemplo1Servidor
Esperando al cliente .....
Recibiendo del CLIENTE:
    Saludos al SERVIDOR DESDE EL CLIENTE
Y:\Java\UD3>_
```

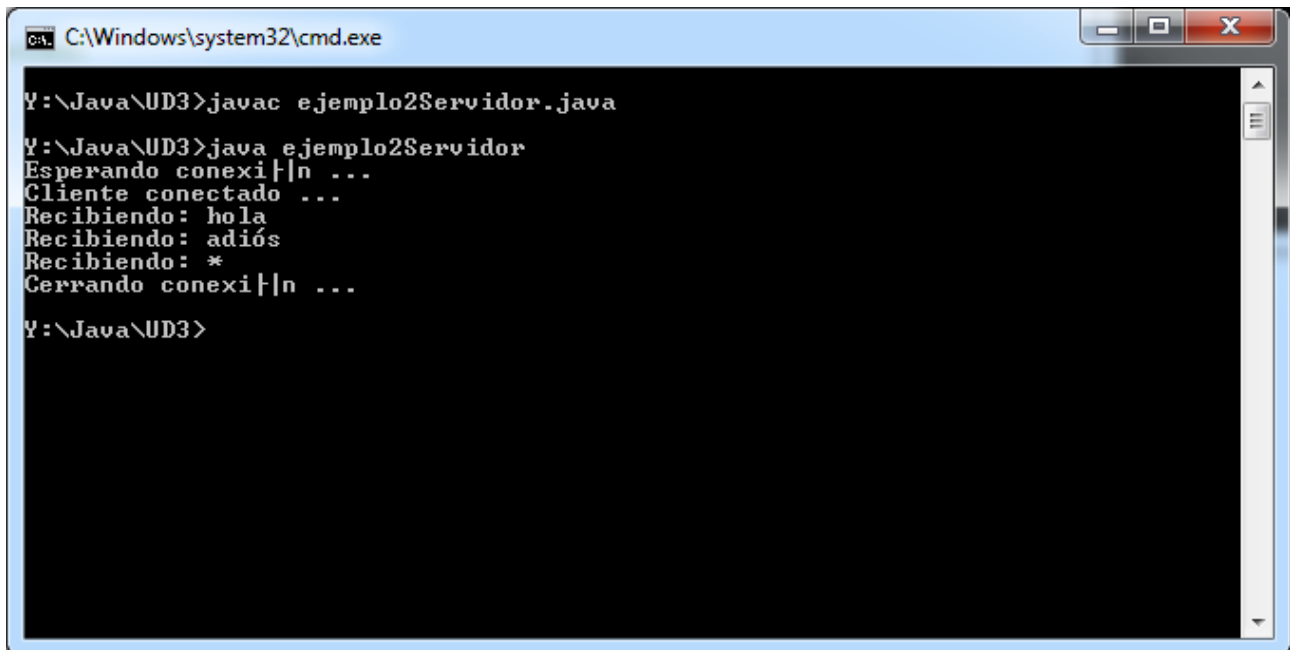


```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac ejemplo1Cliente.java
Y:\Java\UD3>java ejemplo1Cliente
PROGRAMA CLIENTE INICIADO ....
Recibiendo del SERVIDOR:
    Saludos al cliente del servidor
Y:\Java\UD3>
```

En el siguiente ejemplo el programa cliente envía el texto tecleado en su entrada estándar al servidor (en un puerto pactado) escribiendo en el socket; el servidor lee del socket y devuelve de nuevo al cliente el texto recibido escribiendo en el socket; el programa cliente lee del socket lo que le envía el servidor de vuelta y lo muestra en pantalla. El programa servidor finaliza cuando el cliente termine la entrada por teclado o cuando recibe como cadena un asterisco; el cliente finaliza cuando se detiene la entrada de datos mediante las teclas Ctrl+C o Ctrl+Z.

La salida sería algo así:

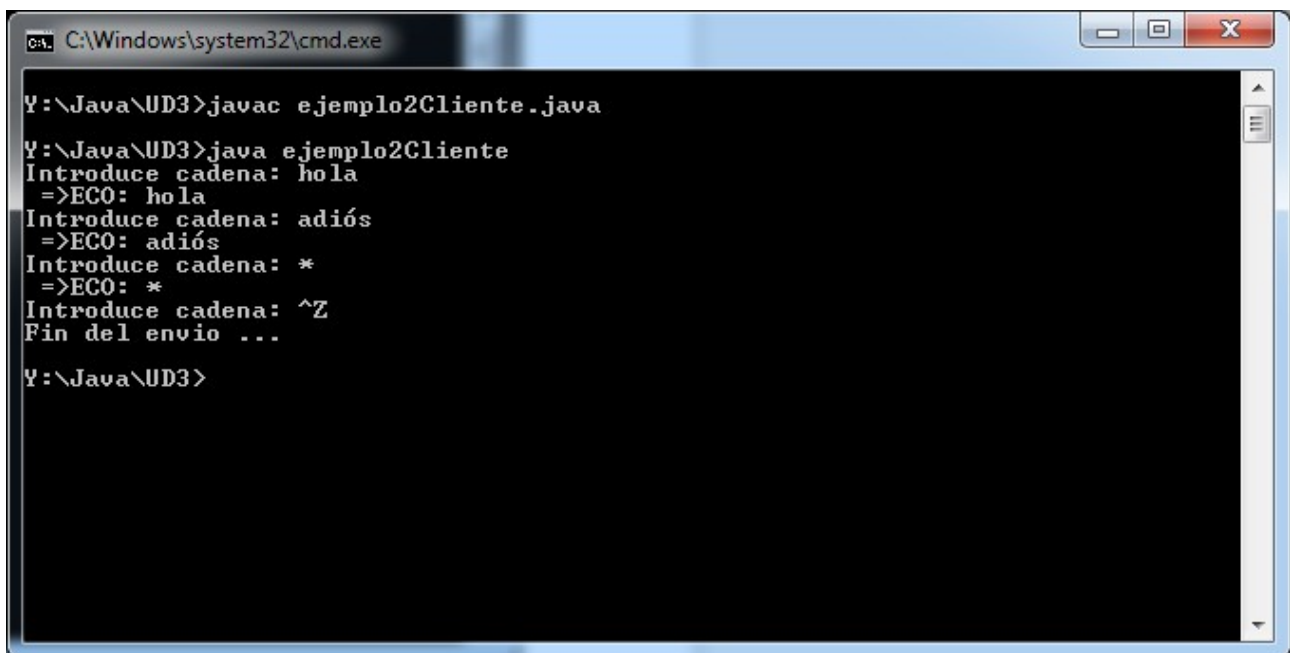


```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac ejemplo2Servidor.java

Y:\Java\UD3>java ejemplo2Servidor
Esperando conexi|n ...
Cliente conectado ...
Recibiendo: hola
Recibiendo: adiós
Recibiendo: *
Cerrando conexi|n ...

Y:\Java\UD3>
```



```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac ejemplo2Cliente.java

Y:\Java\UD3>java ejemplo2Cliente
Introduce cadena: hola
=>ECO: hola
Introduce cadena: adiós
=>ECO: adiós
Introduce cadena: *
=>ECO: *
Introduce cadena: ^Z
Fin del envio ...

Y:\Java\UD3>
```

El **programa servidor** es el siguiente:

```

import java.io.*;
import java.net.*;
public class ejemplo2Servidor {
    public static void main(String[] arg) throws IOException {
        int numeroPuerto = 6000; // Puerto
        try{
            ServerSocket servidor = new ServerSocket(numeroPuerto) ;
            String cad="";

            System.out.println("Esperando conexión ... ");
            Socket clienteConectado = servidor.accept() ;
            System.out.println("Cliente conectado ... ");

            // CREO FLUJO DE SALIDA AL CLIENTE
            PrintWriter fsalida = new PrintWriter(clienteConectado.getOutputStream(),true);

            // CREO FLUJO DE ENTRADA DEL CLIENTE
            BufferedReader fentrada = new BufferedReader(
                new InputStreamReader(clienteConectado.getInputStream()));

            while ((cad=fentrada.readLine())!= null)//recibo cad del cliente
            {
                fsalida.println(cad); //envio cadena al cliente
                System.out.println("Recibiendo: " + cad);
                if(cad.equals("")) break;
            }

            // CERRAR STREAMS y SOCKETS
            System.out.println("Cerrando conexión ... ");
            fentrada.close();
            fsalida.close();
            clienteConectado.close() ;
            servidor.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

En este ejemplo se han usado las clases **PrintWriter** para definir el flujo de salida al socket y **BufferedReader** para el flujo de entrada. Se han utilizado los métodos *readLine()* para leer una línea de texto y *println()* para escribirla.

El **programa cliente** es el siguiente:

```

import java.io.*;
import java.net.*;
public class ejemplo2Cliente {
    public static void main(String[] args) throws IOException {
        String Host = "localhost";
        int Puerto = 6000; // puerto remoto
        try{
            Socket Cliente = new Socket(Host, Puerto);

            // CREO FLUJO DE SALIDA AL SERVIDOR
            PrintWriter fsalida = new PrintWriter(Cliente.getOutputStream(), true);

            // CREO FLUJO DE ENTRADA AL SERVIDOR
            BufferedReader fentrada = new BufferedReader
                (new InputStreamReader(Cliente.getInputStream()));

            // FLUJO PARA ENTRADA ESTANDAR
            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

            String cadena, eco="";
            System.out.print("Introduce cadena: ");
            cadena = in.readLine(); //lectura por teclado
            while(cadena !=null) {
                fsalida.println(cadena); //envio cadena al servidor
                eco=fentrada.readLine(); //recibo cadena del servidor
                System.out.println(" =>ECO: "+eco);
                System.out.print("Introduce cadena: ");
                cadena = in.readLine(); //lectura por teclado
            }
            fsalida.close();
            fentrada.close();
            System.out.println("Fin del envio ... ");
            in.close();
            Cliente.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 1.5. CLASES PARA SOCKETS UDP

Los sockets UDP son más simples y eficientes que los TCP pero no está garantizada la entrega de paquetes. No es necesario establecer una "conexión" entre cliente y servidor, como en el caso de 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
--	-------------------------	----------------------	----------------------

Java implementa los datagramas sobre el protocolo UDP utilizando dos clases:

**DatagramPacket** contiene los datos y **DatagramSocket** es el mecanismo utilizado para enviar y recibir información.

### Clase DatagramPacket

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

CONSTRUCTOR	MISIÓN
<b>DatagramPacket(byte[] buf, int length)</b>	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
<b>DatagramPacket(byte[] buf, int offset, int length)</b>	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
<b>DatagramPacket(byte[] buf, int length, InetAddress addrss, int port)</b>	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>addrss</i>
<b>DatagramPacket(byte[] buf, int offset, int length, InetAddress addrss, int port)</b>	Igual que el anterior pero se especifica un offset dentro de la cadena de bytes

El siguiente ejemplo utiliza el tercer constructor para enviar un datagrama. El datagrama será enviado 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()* obtengo la dirección IP del host al que enviaré el mensaje, en este caso el host local: .

Mensaje: Enviando saludos!!	Longitud:19	Destino:X.X.X.X IP del host local	port: 12345
-----------------------------	-------------	--------------------------------------	-------------

```
int port = 12345; //puerto al que envío
InetAddress destino = InetAddress.getLocalHost();//IP a la que envío
byte[] mensaje = new byte[1024]; //matriz de bytes
String Saludo = "Enviando Saludos! !";
mensaje = Saludo.getBytes();//codificarlo a bytes para enviarlo

//construyo el datagrama a enviar
DatagramPacket envio = new DatagramPacket(mensaje, mensaje.length, destino, port);
```

El siguiente ejemplo utiliza el primer constructor para recibir el mensaje de un datagrama, el mensaje se aloja en bufer, luego se verá como se recupera la información del mensaje:

```
byte[] bufer = new byte[1024];
```



```
DatagramPacket recibo = new DatagramPacket(bufer, bufer.length);
```

Algunos métodos importantes son:

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

El siguiente ejemplo obtiene la longitud y el mensaje del datagrama recibido, el mensaje se convierte a String. A continuación visualiza el número de puerto de la máquina que envía el mensaje y su dirección IP:

```
int bytesRec = recibo.getLength(); //obtengo longitud del mensaje
String paquete= new String(recibo.getData()); //obtengo mensaje
System.out.println ("Puerto origen del mensaje: " + recibo.getPort () );
System.out.println("IP de origen : " + recibo.getAddress().getHostAddress()1;
```

## Clase DatagramSocket

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

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

El siguiente ejemplo construye un socket para datagrama y no lo conecta a ningún puerto, el sistema elige el puerto:

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

<sup>1</sup> Recordemos que `getHostAddress` devuelve la IP de un objeto `InetAddress` en forma de cadena.

Para enlazar el socket a un puerto específico, por ejemplo al puerto 34567, escribimos:

```
DatagramSocket socket = new DatagramSocket(34567);
```

Algunos métodos importantes son:

MÉTODOS	MISIÓN
<b>receive (DatagramPacket paquete)</b>	Recibe un <b>DatagramPacket</b> del socket, y llena <i>paquete</i> con los datos que recibe (mensaje, longitud y origen). Puede lanzar <i>IOException</i>
<b>send (DatagramPacket paquete)</b>	Envía un DatagramPacket a través del socket. El argumento <i>paquete</i> contiene el mensaje y su destino. Puede lanzar <i>IOException</i>
<b>close()</b>	Se encarga de cerrar el socket
<b>int getLocalPort()</b>	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
<b>int getPort()</b>	Devuelve el número de puerto al que está conectado el socket, -1 si no está conectado
<b>connect(InetAddress address, int port)</b>	Conecta el socket a un puerto remoto y una dirección IP concretos, el socket solo podrá enviar y recibir mensajes desde esa dirección
<b>setSoTimeout(int timeout)</b>	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 <i>InterruptedIOException</i>

Siguiendo con el ejemplo inicial, una vez construido el datagrama lo enviamos usando un **DatagramSocket**, en el ejemplo se enlaza al puerto local 34567. Mediante el método *send()* se envía el datagrama:

```
//construyo datagrama a enviar indicando el host destino y puerto
DatagramPacket envio = new DatagramPacket(mensaje, mensaje.length, destino, port);
DatagramSocket socket = new DatagramSocket(34567);
socket.send(envio);//envio datagrama a destino y port
```

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 12345. Después se construye el datagrama para recibir 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);
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());;
```

2 Método que elimina los caracteres blancos iniciales y finales de la cadena, devolviendo una copia de la misma

```

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

```

La salida muestra algo así:

```

Número de Bytes recibidos: 19
Contenido del Paquete: Enviando saludos!!
Puerto origen del mensaje: 34567
IP de origen: 192.168.21.1
Puerto destino del mensaje:12345

```

## 1.6. GESTIÓN DE SOCKETS UDP

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

- 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, vimos anteriormente algunos ejemplos de cómo se usa.

Para escuchar peticiones en un puerto UDP concreto pasamos al constructor el número de puerto. El siguiente ejemplo crea un socket datagrama, le pasamos al constructor el número de puerto 34567 por el que escucha las peticiones y la dirección *InetAddress* en la que se está ejecutando el programa, que normalmente es *InetAddress.getLocalHost()*:

```
DatagramSocket socket = new DatagramSocket(34567, InetAddress.getByName("localhost"));
```

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

## Envío y recepción de datagramas

Para enviar y recibir datagramas usamos la clase **DatagramPacket**.

Para enviar usamos el método *send()* de **DatagramSocket** pasando como parámetro el **DatagramPacket** que acabamos de crear:

```
DatagramPacket datagrama = new DatagramPacket(
    mensajeEnBytes, // el array de bytes
    mensajeEnBytes.length, // su longitud
    InetAddress.getByName("localhost"), // máquina destino
    PuertoDelServidor); // puerto del destinatario
```

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

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

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

El siguiente ejemplo crea un **programa servidor** que recibe un datagrama enviado por un programa cliente. El programa servidor permanece a la espera hasta que le llega un paquete del cliente; en este momento visualiza: el número de bytes recibidos, el contenido del paquete, el puerto y la IP del programa cliente y el puerto local por el que recibe las peticiones:

```

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class servidorUDP {
    public static void main(String[] argv) throws Exception {
        byte[] bufer = new byte[1024]; // bufer para recibir el datagrama
        // ASOCIO EL SOCKET AL PUERTO 12345
        DatagramSocket socket = new DatagramSocket(12345);

        // ESPERANDO DATAGRAMA
        System.out.println("Esperando Datagrama .....");
        DatagramPacket recibo = new DatagramPacket(bufer, bufer.length);
        socket.receive(recibo); // recibo datagrama

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

        // VISUALIZO INFORMACIÓN
        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
    }
}

```

El **programa cliente** envía un mensaje al servidor (máquina *destino*, en este caso es la máquina local; localhost) al puerto 12345 por el que espera peticiones. Visualiza el nombre del host de destino y la dirección IP. También visualiza el puerto local del socket y el puerto al que envía el mensaje:

```

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class clienteUDP{
    public static void main(String[] args) throws Exception{
        InetAddress destino = InetAddress.getLocalHost();
        int port = 12345; //puerto al que envio el datagrama
        byte[] mensaje = new byte[1024];

        String Saludo="Enviando Saludos!!";
        mensaje = Saludo.getBytes(); //codifico String a bytes

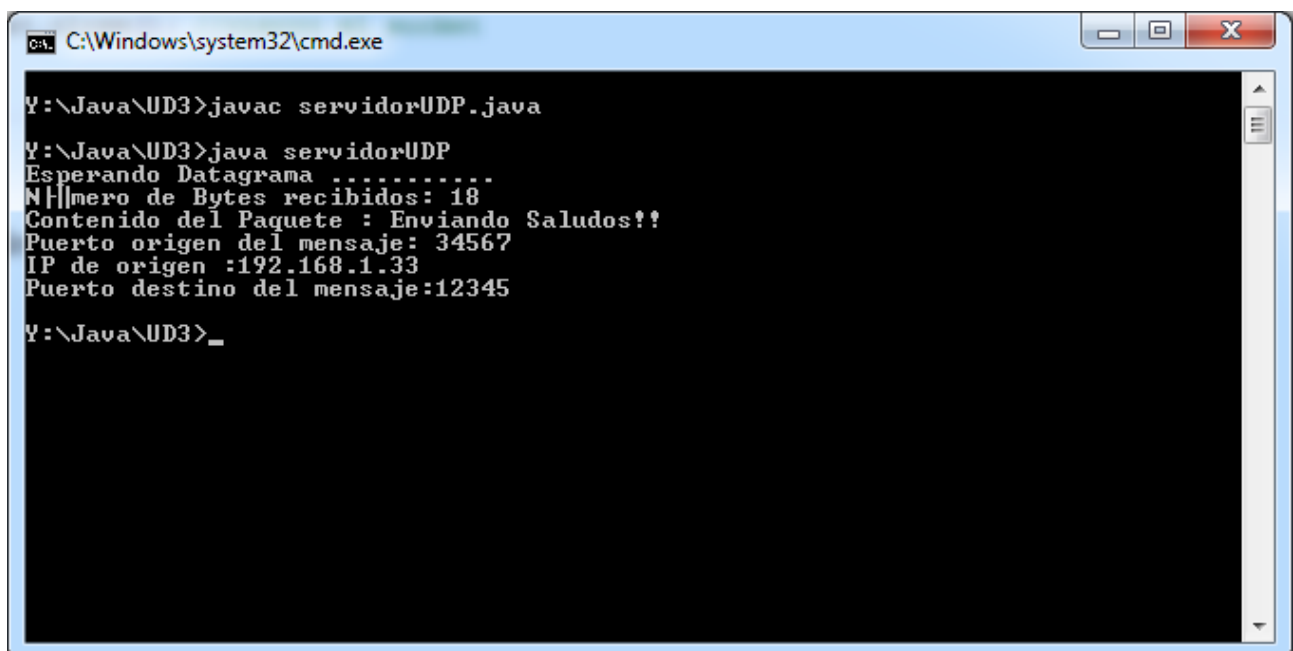
        //CONSTRUYO EL DATAGRAMA A ENVIAR
        DatagramPacket envio = new DatagramPacket(mensaje, mensaje.length, destino, port);
        DatagramSocket socket = new DatagramSocket(34567); //Puerto local

        System.out.println("Enviando Datagrama de longitud: "+mensaje.length);
        System.out.println("Host de destino : "+ destino.getHostAddress());
        System.out.println("IP Destino : "+ destino.getHostAddress());
        System.out.println("Puerto local del socket : " +socket.getLocalPort());
        System.out.println ("Puerto al que envio: " + envio.getPort());

        //ENVIO DATAGRAMA
        socket.send(envio);
        socket.close(); //cierro el socket
    }
}

```

La ejecución de los programas cliente y servidor es la siguiente:

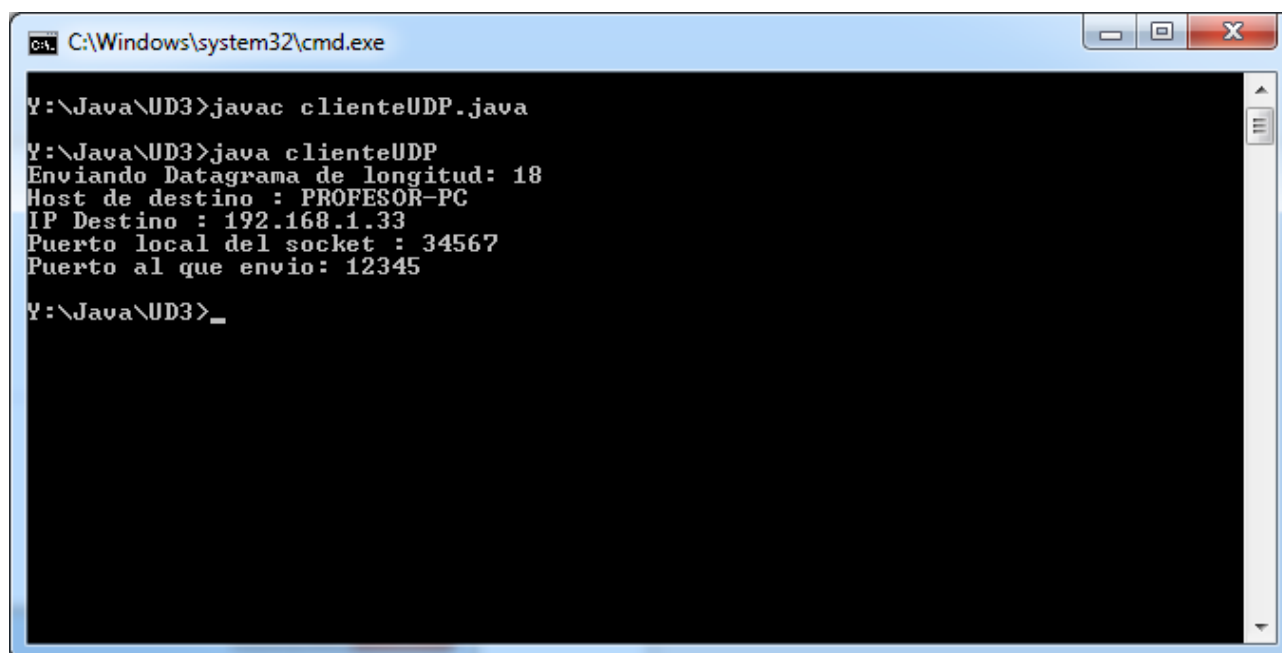


```

C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac servidorUDP.java
Y:\Java\UD3>java servidorUDP
Esperando Datagrama .....
Número de Bytes recibidos: 18
Contenido del Paquete : Enviando Saludos!!
Puerto origen del mensaje: 34567
IP de origen :192.168.1.33
Puerto destino del mensaje:12345
Y:\Java\UD3>_

```



```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac clienteUDP.java

Y:\Java\UD3>java clienteUDP
Enviando Datagrama de longitud: 18
Host de destino : PROFESOR-PC
IP Destino : 192.168.1.33
Puerto local del socket : 34567
Puerto al que envio: 12345

Y:\Java\UD3>_
```

En primer lugar, desde una consola ejecutamos el programa servidor, y una vez iniciado abrimos otra consola y ejecutamos el programa cliente.

En el siguiente 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 texto en mayúscula. El programa cliente recibe un datagrama del servidor y muestra información del mismo en pantalla (IP, puerto del servidor y el texto en mayúscula). El programa servidor finaliza cuando recibe como cadena un asterisco. Para comenzar la ejecución primero ejecutamos el programa servidor (desde la consola) que permanecerá a la espera, y después (desde otra consola) ejecutamos el programa cliente varias veces.

El **programa cliente** es el siguiente:

```

import java.io.*;
import java.net.*;
public class clienteUDP2 {
    public static void main(String args[]) throws Exception {
        //FLUJO PARA ENTRADA ESTANDAR
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        DatagramSocket clientSocket = new DatagramSocket();//socket cliente
        byte[] enviados = new byte[1024];
        byte[] recibidos = new byte[1024];

        // DATOS DEL SERVIDOR al que enviar mensaje
        InetAddress IPServidor = InetAddress.getLocalHost() ;// localhost
        int puerto = 9876; //,puerto por el que escucha

        //INTRODUCIR DATOS POR TECLADO
        System.out.print("Introduce mensaje: ");
        String cadena = in.readLine();
        enviados = cadena.getBytes();

        //ENVIANDO DATAGRAMA AL SERVIDOR
        System.out.println("Enviando " + enviados.length + " bytes al      servidor.");
        DatagramPacket envio = new DatagramPacket(enviados, enviados.length, IPServidor, puerto);
        clientSocket.send(envio) ;

        // RECIBIENDO DATAGRAMA DEL SERVIDOR
        DatagramPacket recibo = new DatagramPacket(recibidos, recibidos.length);
        System.out.println("Esperando datagrama .... ");
        clientSocket.receive(recibo);
        String mayuscula = new String(recibo.getData());

        //OBTENIDENDO INFORMACIÓN DEL DATAGRAMA
        InetAddress IPOrigen = recibo.getAddress();
        int puertoOrigen = recibo.getPort();
        System.out.println("\tProcedente de: " + IPOrigen + ":" + puertoOrigen);
        System.out.println("\tDatos: " + mayuscula.trim());
        clientSocket.close();//cerrar socket
    }
}

```

El programa servidor es el siguiente:



```

import java.io.*;
import java.net.*;
public class servidorUDP2 {
    public static void main(String args[]) throws Exception {
        //Puerto por el que escucha el servidor: 9876
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] recibidos = new byte[1024];
        byte[] enviados = new byte[1024];
        String cadena;

        while(true) {
            System.out.println ("Esperando datagrama .... ");
            //RECIBO DATAGRAMA
            recibidos = new byte[1024];
            DatagramPacket paqRecibido = new DatagramPacket(recibidos, recibidos.length);
            serverSocket.receive (paqRecibido);
            cadena = new String(paqRecibido.getData());

            //DIRECCION ORIGEN
            InetAddress IPOrigen = paqRecibido.getAddress();
            int puerto = paqRecibido.getPort();
            System.out.println ("\tOrigen: " + IPOrigen + ":" + puerto);
            System.out.println ("\tMensaje recibido: " + cadena.trim());

            //CONVERTIR CADENA A MAYÚSCULA
            String mayuscula = cadena.trim().toUpperCase();
            enviados = mayuscula.getBytes();

            //ENVIO DATAGRAMA AL CLIENTE
            DatagramPacket paqEnviado = new DatagramPacket(enviados, enviados.length, IPOrigen, puerto);
            serverSocket.send(paqEnviado);

            //Para terminar
            if(cadena.trim().equals("")) break;
        }
        serverSocket.close();
        System.out.println ("Socket cerrado ... ");
    }
}

```

Y esta sería la salida:

```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac servidorUDP2.java
Y:\Java\UD3>java servidorUDP2
Esperando datagrama .....
    Origen: /192.168.1.33:64875
    Mensaje recibido: uno
Esperando datagrama .....
    Origen: /192.168.1.33:64876
    Mensaje recibido: dos
Esperando datagrama .....
    Origen: /192.168.1.33:64877
    Mensaje recibido: *
Socket cerrado ...
Y:\Java\UD3>_
```

```
C:\Windows\system32\cmd.exe

Y:\Java\UD3>javac clienteUDP2.java
Y:\Java\UD3>java clienteUDP2
Introduce mensaje: uno
Enviando 3 bytes al servidor.
Esperando datagrama ....
    Procedente de: /192.168.1.33:9876
    Datos: UNO

Y:\Java\UD3>java clienteUDP2
Introduce mensaje: dos
Enviando 3 bytes al servidor.
Esperando datagrama ....
    Procedente de: /192.168.1.33:9876
    Datos: DOS

Y:\Java\UD3>java clienteUDP2
Introduce mensaje: *
Enviando 1 bytes al servidor.
Esperando datagrama ....
    Procedente de: /192.168.1.33:9876
    Datos: *

Y:\Java\UD3>
```