Tema 3.- Programación de Comunicaciones en Red.

Objetivos

- Comunicación entre aplicaciones.
- Roles cliente y servidor.
- Elementos de programación de aplicaciones en red. Librerías.
- Sockets.
- Creación de sockets.
- Enlazado y establecimiento de conexiones.
- Utilización de sockets para la transmisión y recepción de información.
- Programación de aplicaciones cliente y servidor.
- Utilización de hilos en la programación de aplicaciones en red.

Contenidos

1.- Comunicación entre aplicaciones.

En la actualidad, muchos de los procesos que se ejecutan en una computadora requieren obtener o enviar información a otros procesos que se localizan en una computadora diferente.

Para lograr esta comunicación se utilizan los protocolos de comunicación **TCP** y **UDP**. Los protocolos TCP y UDP usan puertos para asignar datos entrantes a un proceso.

El protocolo TCP:

- Establece un **conducto de comunicación punto a punto** entre dos computadoras, es decir, cuando se requiere la transmisión de un flujo de datos entre dos equipos, el protocolo TCP establece un conducto exclusivo entre dichos equipos por el cual los datos serán transmitidos y este perdurará hasta que la transmisión haya finalizado.
- **Garantiza que los datos** enviados de un extremo de la conexión **lleguen al otro extremo y en el mismo orden en que fueron enviados**.
- Es conocido como un **protocolo orientado a conexión**.

El protocolo UDP:

- **Es un protocolo no orientado a la conexión.**
- La forma de trasmitir los datos **no garantiza** en primera instancia su **llegada al destino**, e incluso si este llegara al destino final, **tampoco garantiza la integridad de los datos**.
- UPD hace la transmisión de los datos **sin establecer un conducto de comunicación** exclusiva como lo hace TCP.
- Utiliza **datagramas**, los cuales contienen una porción de la información y que son enviados a la red en espera de ser capturados por el equipo destino. Cuando el destino captura los datagramas debe reconstruir la información, para esto debe ordenar la

información que recibe ya que la información transmitida no viene con un orden específico, además se debe tener conciencia de que no toda la información va a llegar.
- Su funcionamiento **se puede comparar con el servicio postal**.

Una red de ordenadores está formada por clientes y servidores. Un ordenador tiene una conexión física de red y los datos llegan a través de esa conexión. Puesto que los datos pueden estar destinados a diferentes aplicaciones, el ordenador sabe a qué aplicación enviar los datos mediante el uso de puertos. La aplicación servidor vincula un socket a un número de puerto específico y el cliente puede comunicarse con el servidor enviándole peticiones a través de ese puerto.

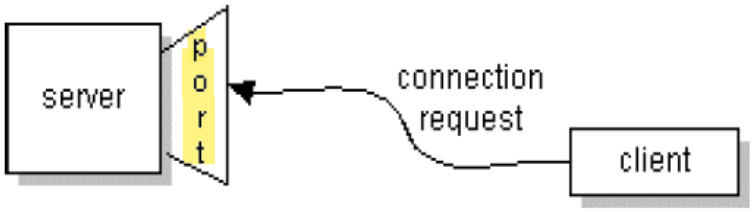
Así pues, si a través de una conexión física se recibe una petición por parte de un cliente, la manera de identificar qué proceso debe atender dicha petición es mediante el uso de puertos, que permiten tanto a TCP como a UDP dirigir los datos a la aplicación correcta de entre todas las que se están ejecutando en la máquina.

Un Socket se puede definir como un extremo de un enlace de **comunicación bidireccional** entre dos programas que se comunican por la red. Se identifican por IP y puerto de la máquina. Existen tanto en TCP como en UDP.

Roles Cliente-Servidor.

Un **Servidor** es una computadora específica y tiene un *socket* que responde en un puerto específico. Únicamente espera, escuchando a través del *socket* a que un cliente haga una petición.

Un **Cliente** conoce el nombre de host de la máquina en la cual el servidor se encuentra ejecutando y el número de puerto en el cual el servidor está conectado. Para realizar una petición de conexión, intenta encontrar al servidor en la máquina servidora en el puerto especificado.

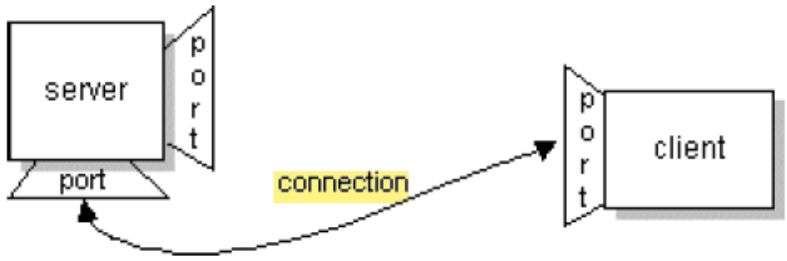


Si todo va bien, **el servidor acepta la conexión**. Además obtiene un nuevo *socket* de comunicación, sobre un puerto diferente, para poder seguir atendiendo al *socket* original para peticiones de conexión mientras atiende las necesidades del cliente que se conectó.

Por la parte del cliente, **si la conexión es aceptada**, se crea un *socket* que puede usar para comunicarse con el servidor.

El ***socket* en el cliente no está utilizando el número de puerto usado para realizar la petición al servidor.** En lugar de éste, el cliente asigna un número de puerto local a la máquina en la

cual está siendo ejecutado. Ahora el cliente y el servidor **pueden comunicarse escribiendo o leyendo en o desde sus respectivos *sockets*.**



El **Servidor e**stá ejecutándose y esperando a que otro quiera conectarse a él. Nunca da "el primer paso" en la conexión. Es el que "sirve" información al que se la pida.

El **Cliente e**s el programa que da el "primer paso" en la conexión. En el momento de ejecutarlo o cuando lo necesite, intenta conectarse al servidor. Es el que solicita información al servidor.

La clase InetAddress de Java nos proporciona objetos que se pueden utilizar para manipular tanto direcciones IP como nombres de dominio. Dispone de los siguientes métodos:

- **getByName()** devuelve objeto InetAdress a partir del nombre
- **getHostAddress()** devuelve la dirección IP del objeto.
- **getHostName()** obtiene el nombre de host para el objeto
- **getLocalHost()** obtiene información del equipo donde se ejecuta

Ejemplo1: Obtener la dirección IP y el nombre de Host del dominio www.google.es

```
import java.net.*;
public class Ejemplo1 {
    public static void main(String[] args) {
        try{
            // Obtenemos el objeto InetAddress y asignamos www.google.es
            InetAddress host = InetAddress.getByName("www.google.es");
            // Obtenemos el Host
            System.out.println("Host: "+host);
             // Obtenemos la IP
            System.out.println("IP: "+ host.getHostAddress());
             // Obtenemos el nombre
            System.out.println("Nombre: "+host.getHostName());
        } catch( UnknownHostException e){
            System.out.println("Host Exception");
            System.out.println(e.toString());
        }
    }
}
```

Salida:

```
run:
Host: www.google.es/172.217.168.163
IP: 172.217.168.163
Nombre: www.google.es
```

2.- Programación de aplicaciones en red con Java.

El paquete java.net de la plataforma Java proporciona clases para Sockets TCP: La clase **Socket**, la cual implementa una de las partes de la comunicación bidireccional entre un programa Java y otro programa en la red y la clase **ServerSocket** que implementa la funcionalidad de servidor:

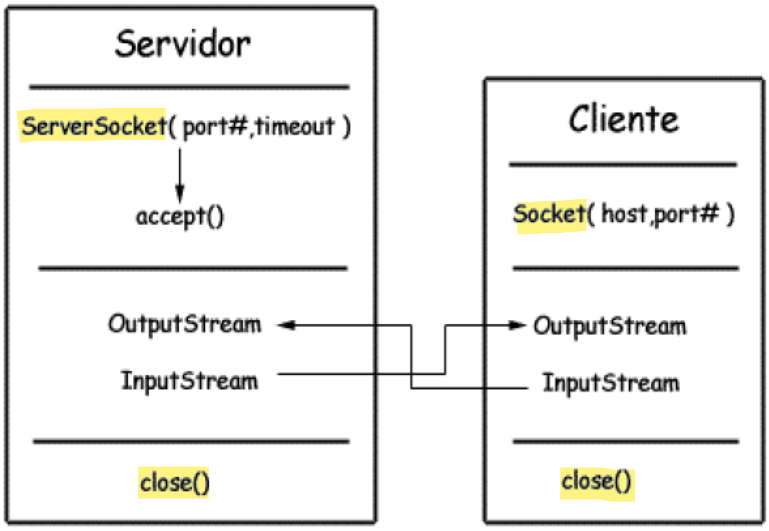
- **Socket**: Es el objeto básico en toda comunicación a través de internet, bajo el protocolo TCP. Esta clase proporciona métodos para la E/S a través de streams que hacen la lectura y escritura a través de sockets muy sencilla.
- **ServerSocket**: Es un objeto utilizado en las aplicaciones servidor para escuchar las peticiones que realicen los clientes conectados a ese servidor. Este objeto no realiza el servicio, sino que crea un objeto Socket en función del cliente para realizar toda la comunicación a través de él.

Además el paquete java.net la plataforma Java proporciona clases para Sockets UDP: La clase **DatagramSocket** permite crear Sockets UDP y la clase **DatagramPacket** que representa un paquete datagrama con información:

- **DatagramSocket**: Esta clase puede ser utilizada para implementar datagramas no fiables (**Sockets UDP**), no ordenados. Aunque la comunicación por estos sockets es muy rápida porque no hay que perder tiempo estableciendo la conexión entre cliente y servidor.
- **DatagramPacket**: Clase que representa un paquete datagrama conteniendo información de paquete, longitud, direcciones y números de puerto.

El modelo de socket más simple es:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout en segundos), a que el cliente establezca la conexión.
- Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método **accept()**.
- El cliente establece una conexión con la máquina host a través del puerto que se designe.
- El cliente y servidor se comunican con manejadores **InputStream y OutputStream**.



3.- Creación de Sockets TCP.

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. Además el servidor necesita crear un objeto *socket* desde el **ServerSocket** para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
int numeroPuerto = 6000;// Puerto
ServerSocket servidor =  new ServerSocket(numeroPuerto);
System.out.println("Esperando al cliente.....");
Socket clienteConectado = servidor.accept();
```

Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. Para las aplicaciones que se desarrollen, debemos asegurarnos de seleccionar un puerto por encima del 1023.

En el programa cliente, se abre el socket de la forma:

```
      Socket cliente = new Socket("maquina",numeroPuerto);
```

Donde maquina es el nombre de la máquina en donde estamos intentando abrir la conexión y numeroPuerto es el número de puerto del servidor que está escuchando las peticiones de los clientes.

```
String Host = "localhost";
int Puerto = 6000;//puerto remoto
System.out.println("Programa Cliente Iniciado....");
Socket cliente = new Socket(Host, Puerto);
```

Cierre de Sockets.

Para cerrar los sockets de cliente y servidor utilizamos el método close(). En el servidor debemos cerrar el ServerSocket llamado **servidor** y el Socket llamado **clienteConectado**.

```
clienteConectado.close();
servidor.close();
```

En el cliente debemos cerrar el Socket llamado **cliente**.

```
cliente.close();
```

Comunicación Cliente-Servidor.

a) Streams de Entrada.

En el **Cliente** se puede utilizar la clase **DataInputStream** para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe:

```
DataInputStream entrada = new DataInputStream(cliente.getInputStream());
System.out.println(entrada.readUTF());
```

En el Servidor también usaremos DataInputStream, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado.

```
DataInputStream entrada = new DataInputStream(clienteConectado.getInputStream());
System.out.println(entrada.readUTF());
```

La clase **DataInputStream** permite la lectura de líneas de texto y tipos de datos primitivos. Dispone de métodos para leer todos esos tipos como: ***read(), readUTF(), readChar(), readInt(), readDouble() y readLine()***.

b) Streams de Salida.

En el **Cliente** podemos crear un stream de salida para enviar información al socket del servidor utilizando la clase **DataOutputStream**:

```
DataOutputStream salida = new DataOutputStream(cliente.getOutputStream());
salida.writeUTF("Conexión establecida. Bienvenido!!!");
```

En el **Servidor** también podemos usar cualquiera de las 2 clases para enviar información al cliente. Por ejemplo:

```
DataOutputStream salida = new DataOutputStream (clienteConectado.getOutputStream());
salida.writeUTF("Conexión establecida. Bienvenido!!!");
```

La clase **DataOutputStream** y **PrintStream** permite la escritura datos. Dispone de métodos para escribir todos esos tipos como: ***write(), writeUTF(),writeChar(),writeInt(), writeDouble() y println()***.

c) Cerrar Flujo de datos entre Cliente y Servidor.

Por último debemos cerrar los flujos de datos en el cliente y en el servidor con el método close(). Es importante destacar que el orden de cierre es relevante. Es decir, se deben cerrar primero los streams relacionados con un socket antes que el propio socket, ya que de esta forma evitamos posibles errores de escrituras o lecturas sobre descriptores ya cerrados.

En el cliente:

```
//Cerramos los flujos de datos
entrada.close();
salida.close();
//Cerramos los sockets
cliente.close();
```

En el servidor:

```
//Cerramos los flujos de datos
entrada.close();
salida.close();
//Cerramos los sockets
clienteConectado.close()
servidor.close();
```

Código Completo Ejemplo2Servidor:

```
package ejemplo2servidor;
import java.io.*;
import java.net.*;
public class Ejemplo2Servidor {
    public static void main(String[] args) throws IOException {
        // Creamos socket en el servidor y nos ponemos a la espera....
        int numeroPuerto = 6000;// Puerto
        ServerSocket servidor =  new ServerSocket(numeroPuerto);
        System.out.println("Esperando al cliente.....");
        Socket clienteConectado = servidor.accept();
        // Enviamos mensaje al cliente con un DataOutputStream
        DataOutputStream salida =
            new DataOutputStream(clienteConectado.getOutputStream());
        salida.writeUTF("Conexión Establecida. Bienvenido!!!");
        // Cerramos Socket y SocketServer en el servidor y flujos de datos
        System.out.println("Conexión Cerrada");
        salida.close();
        clienteConectado.close();
        servidor.close();
        }
    }
```

Salida Ejemplo2Servidor:

```
run:
Esperando al cliente.....
Conexión Cerrada
BUILD SUCCESSFUL (total time: 1 second)
```

Código Completo Ejemplo2Cliente:

```
package ejemplo2cliente;
import java.io.*;
import java.net.*;
public class Ejemplo2Cliente {
    public static void main(String[] args) throws  Exception {
        // Creamos Socket en el cliente
        String Host = "localhost";
        int Puerto = 6000;//puerto remoto
        System.out.println("Programa Cliente Iniciado....");
        Socket cliente = new Socket(Host, Puerto);
        // Recuperamos mensaje del Servidor con un DataInputStream
        DataInputStream entrada =
            new DataInputStream(cliente.getInputStream());
        System.out.println(entrada.readUTF());
        // Cerramos Socket en el cliente y flujos de datos
        System.out.println("Conexión Cerrada");
        entrada.close();
        cliente.close();
    }
}
```

Salida: Ejemplo2Cliente

```
run:
Programa Cliente Iniciado....
Conexión Establecida
Conexión Cerrada
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo3: Crear un servidor que pueda recibir 3 clientes. Para ello creamos un proyecto llamado **PServidorSockets** y añadimos una clase **Servidor** con el siguiente contenido:

```
package pservidorsocket;
import java.io.*;
import java.net.*;
public class Servidor {
    static final int PUERTO = 5003;
    private ServerSocket skServidor;
    public void crearSocket() throws IOException {
        skServidor = new ServerSocket(PUERTO);
        System.out.println("Servidor dice: Escucho el puerto " + PUERTO);
        for (int numCli = 0; numCli < 3; numCli++) {
            Socket skCliente = skServidor.accept(); // Crea objeto
            System.out.println("Servidor dice: Sirvo al cliente " + numCli);
            // envío de datos al cliente
            DataOutputStream flujo =
                 new DataOutputStream(skCliente.getOutputStream());
            flujo.writeUTF("Hola cliente " + numCli);
            skCliente.close();
        }
        System.out.println("Servidor dice: Demasiados clientes por hoy");
    }
}
```

En la clase llamada principal, en el método **main** creamos un objeto del servidor, ejecutando el método crearSocket():

```
package pservidorsocket;
import java.io.IOException;
public class PServidorSocket {
    public static void main(String[] args) throws IOException {
        Servidor s = new Servidor();
        s.crearSocket();
    }
}
```

A continuación creamos otro proyecto llamado **PClienteSocket** y añadimos la siguiente clase **Cliente**:

```
package pclientesocket;
import java.io.*;
```

```
import java.net.*;
public class Cliente {
    static final String HOST = "localhost";
    static final int PUERTO = 5003;
    public void crearSocket() throws IOException {
        Socket skCliente = new Socket(HOST, PUERTO);
        //recepción de datos del servidor
        DataInputStream                          flujo                          =
             new DataInputStream(skCliente.getInputStream());
        System.out.println("Cliente recibe:" + flujo.readUTF());
        skCliente.close();
    }
}
```

En la clase llamada principal, en el método **main** creamos un objeto del cliente y ejecutará su método crearSocket():

```
package pclientesocket;
import java.io.*;
public class PClienteSocket {
    public static void main(String[] args) throws IOException{
        for (int i = 0; i < 3; i++) {
            Cliente c = new Cliente();
            c.crearSocket();
        }
    }
}
```

Salida PServidorSocket:

```
run:
Servidor dice: Escucho el puerto 5003
Servidor dice: Sirvo al cliente 0
Servidor dice: Sirvo al cliente 1
Servidor dice: Sirvo al cliente 2
Servidor dice: Demasiados clientes por hoy
BUILD SUCCESSFUL (total time: 3 seconds)
```

Salida PClienteSocket:

```
run:
Cliente recibe:Hola cliente 0
Cliente recibe:Hola cliente 1
Cliente recibe:Hola cliente 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

4.- Creación de Sockets UDP.

En el protocolo **UDP**: no hay "conexión" entre cliente y servidor.

El emisor debe indicar explícitamente la dirección IP y el puerto del destino para cada paquete.

El servidor debe extraer la dirección IP y el puerto del emisor del paquete UDP: los datos pueden recibirse desordenados o incluso perderse.

La clase **DatagramPacket** proporciona constructores para crear instancias a partir de los datagramas recibidos y para crear instancias de datagramas que van a ser enviados.

Podemos crean una instancia de datagrama que se van a enviar compuesta por una cadena de bytes que almacena el mensaje, la longitud del mensaje y la dirección de Internet y el número de puerto local del conector destino:

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

También podemos crear una instancia de datagrama recibido especificando la cadena de bytes en la que alojar el mensaje y la longitud de la misma:

```
DatagramPacket(byte[] buf, int length)
```

Dentro de esta clase hay métodos para obtener los diferentes componentes de un datagrama, tanto recibido como enviado:

- **getData()**: para obtener el mensaje contenido en el datagrama.
- **getAddress()** : para obtener la dirección IP.
- **getPort()**: para obtener el puerto.

La clase **DatagramSocket** maneja sockets para enviar y recibir datagramas UDP. Proporciona tres constructores:

- **DatagramSocket():** constructor sin argumentos que permite que el sistema elija un puerto entre los que estén libres y selecciona una de las direcciones locales.
- **DatagramSocket(int port):** constructor que toma un número de puerto como argumento, apropiado para los procesos que necesitan un número de puerto (servicios).
- **DatagramSocket(int port, InetAddress laddr):** constructor que toma como argumentos el número de puerto y una determinada dirección local.

Dispone de varios métodos, destacamos los más utilizados:

- **send(DatagramPacket p)** y **receive(DatagramPacket p)**: Estos métodos sirven para transmitir datagramas entre un par de conectores. El argumento de *send* es una instancia de DatagramPacket conteniendo el mensaje y el destino. El argumento de *receive* es un DatagramPacket vacío en el que colocar el mensaje, su longitud y su origen. Ambos métodos pueden lanzar excepciones **IOException**.
- **setSoTimeout(int timeout)**: Este método permite establecer un tiempo de espera límite. Cuando se fija un límite, el método *receive* se bloquea durante el tiempo fijado y después lanza una excepción **InterruptedIOException.**
- **connect(InetAddress address, int port)**: Este método se utiliza para conectarse a un puerto remoto y a una dirección Internet concretos, en cuyo caso el conector sólo podrá enviar y recibir mensajes de esa dirección.

Ejemplo: Programa con un cliente y un servidor mediante protocolo UDP. El servidor, que se queda a la escucha de peticiones, y el cliente envía una única petición y finaliza. El servidor se queda esperando por nuevas peticiones.

Código Servidor:

```
package servidorudp;
import java.net.*;
import java.io.*;
public class ServidorUDP {
    public static void main(String[] args) {
        DatagramSocket socketUDP = null;
        try {
            socketUDP = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            System.out.println("Esperando paquetes...");
            while (true){
                // Construimos el DatagramPacket para recibir peticiones
                DatagramPacket paquetePeticion = new DatagramPacket(buffer,
                     buffer.length);

                // Leemos un paquete del DatagramSocket
                socketUDP.receive(paquetePeticion);
                // Obtenemos información del host que envia el paquete
                System.out.print("Datagrama recibido de "
                    +paquetePeticion.getAddress());
                System.out.println(" por el puerto "
                    +paquetePeticion.getPort());

                // Construimos el Datagrama de respuesta
                DatagramPacket paqueteRespuesta =
                    new DatagramPacket(paquetePeticion.getData(),
                    paquetePeticion.getLength(), paquetePeticion.getAddress(),
                    paquetePeticion.getPort());
                // Enviamos respuesta que es un eco
                socketUDP.send(paqueteRespuesta);
            }
        }catch(SocketException e){
            System.out.println("Socket: "+e.getMessage());
        }catch(IOException e){
            System.out.println("IO: "+e.getMessage());
        }
        socketUDP.close();
    }
}
```

Código Cliente:

```
package clienteudp;
import java.net.*;
import java.io.*;
public class ClienteUDP {
```

```
    public static void main(String[] args) {
        DatagramSocket socketUDP = null;
        String mensaje = "Hola Mundo!!!";
        String mensajeRecibido;
        byte[] mensajeBytes = mensaje.getBytes();
        InetAddress hostServidor;
        int puertoServidor;
        DatagramPacket paquetePeticion;
        DatagramPacket paqueteRespuesta;
        try {
            socketUDP = new DatagramSocket();
            hostServidor = InetAddress.getByName("localhost");
            puertoServidor = 6789;

            // Construimos un paquete para enviar el mensaje al servidor
            paquetePeticion =
                new DatagramPacket(mensajeBytes, mensaje.length(),
                    hostServidor, puertoServidor);

            // Enviamos el paquete
            socketUDP.send(paquetePeticion);

            // Cosntruimos el paquete que contendrá la respuesta del Servidor
            byte[] buffer = new byte[1000];
            paqueteRespuesta = new DatagramPacket(buffer, buffer.length);
            socketUDP.receive(paqueteRespuesta);

            // Enviamos la respuesta del servidor a la salida estandar
            mensajeRecibido = new String(paqueteRespuesta.getData());
            System.out.println("Respuesta: "+ mensajeRecibido);

            // Cerramos el socket
            socketUDP.close();
        }catch(SocketException e){
            System.out.println("Socket: "+e.getMessage());
        }catch(IOException e){
            System.out.println("IO: "+e.getMessage());
        }
    }
}
```

Salida ServidorUDP:

```
run:
Esperando paquetes...
Datagrama recibido de /127.0.0.1 por el puerto 59376
Datagrama recibido de /127.0.0.1 por el puerto 59377
Datagrama recibido de /127.0.0.1 por el puerto 59378
Datagrama recibido de /127.0.0.1 por el puerto 59379
```

Salida ClienteUDP:

```
run:
Respuesta: Hola Mundo!!!
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.- Sockets Multicast.

La clase MulticastSocket permite enviar paquetes a multiples destinos simultáneamente. Para poder recibir el paquete es necesario establecer un **grupo multicast**, que es un grupo de direcciones IP que comparten el mismo número de puerto. Cuando se envía un mensaje a un grupo multicast, todos los que pertenecen al grupo reciben en mensaje. El emisor no conoce el número de integrantes 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 van de 224.0.0.1 a 239.255.255.255

Constructores de la clase MulticastSocket (pueden lanzar IOException):

- **MulticastSocket()**. Construye un socket multicast dejando al sistema elegir el puerto de los que están libres.
- **MulticastSocket(int port).** Construye un socket multicast y lo conecta al puerto local especificado.

Métodos de la clase MulticastSocket (pueden lanzar IOException):

- **joinGroup(multicastAddress).** Permite al socket multicast unirse al grupo multicast.
- **leaveGroup(multicastAddress).** El socket multicast abandona el grupo multicast.
- **send(DatagramaPacket).** Envia un paquete a todos los miembros del grupo multicast.
- **receive(DatagramaPacket).** Recibe el paquete de un grupo multicast.

Ejemplo: Crear un servidor multicast que lee datos por teclado y los envía a todos los clientes que pertenezcan al grupo multicast hasta que se introduce un asterisco. El cliente visualiza el paquete que recibe del servidor hasta que recibe un asterisco.

Servidor Multicast:

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

public class MCservidor {
    public static void main(String args[]) throws Exception {
        // FLUJO PARA ENTRADA ESTANDAR
        BufferedReader in = new
                BufferedReader(new InputStreamReader(System.in));

        //Se crea el socket multicast.
        MulticastSocket ms = new MulticastSocket();

        int Puerto = 12345;//Puerto multicast
```

Se van a pedir textos por teclado y se va a enviar a todos los clientes. Para cuando el servidor escriba un *

```
        InetAddress grupo = InetAddress.getByName("225.0.0.1");//Grupo

        String cadena="";

        while(!cadena.trim().equals("*")) {
            System.out.print("Datos a enviar al grupo: ");
            cadena = in.readLine();
            // ENVIANDO AL GRUPO
            DatagramPacket paquete = new DatagramPacket (cadena.getBytes(),
                cadena.length(), grupo, Puerto);
            ms.send (paquete);
        }
        ms.close ();//cierro socket
        System.out.println ("Socket cerrado...");
    }
}
```

Cliente Multicast:

```
package mccliente;
import java.io.*;
import java.net.*;
public class MCcliente {
    public static void main(String args[]) throws Exception {
        //Se crea el socket multicast
        int Puerto = 12345;//Puerto multicast
        MulticastSocket ms = new MulticastSocket(Puerto);

        InetAddress grupo = InetAddress.getByName("225.0.0.1");//Grupo

        //Nos unimos al grupo
        ms.joinGroup (grupo);

        String msg="";

        // Mientras el mensaje no sea un asterisco
        while(!msg.trim().equals("*")) {
            byte[] buf = new byte[1000];
            //Recibe el paquete del servidor multicast
            DatagramPacket paquete = new DatagramPacket(buf, buf.length);
            ms.receive(paquete);

            msg = new String(paquete.getData());
            System.out.println ("Recibo: " + msg.trim());
        }
        ms.leaveGroup (grupo); //abandonamos grupo
        ms.close (); //cierra socket
        System.out.println ("Socket cerrado...");
    }
}
```

Salida MCservidor:

```
run:
Datos a enviar al grupo: Hola!!!
Datos a enviar al grupo: espero todo vaya bien.
Datos a enviar al grupo: este es el mensaje final*
Datos a enviar al grupo: *
Socket cerrado...
BUILD SUCCESSFUL (total time: 1 minute 7 seconds)
```

Salida en todos los MCcliente:

```
run:
Recibo: Hola!!!
Recibo: espero todo vaya bien.
Recibo: este es el mensaje final*
Recibo: *
Socket cerrado...
BUILD SUCCESSFUL (total time: 47 seconds)
```

6.- Conexión de múltiples clientes. Hilos.

Hasta ahora las programas servidores que hemos creado solo son capaces de atender un cliente en cada momento, pero lo más habitual es que el servidor pueda atender a muchos clientes simultáneamente. El multihilo nos va a permitir atender a múltiples clientes. Cada hilo atenderá a un cliente.

El esquema básico será construir un servidor con la clase **ServerSocket** y invocar el método **accept()** para esperar las peticiones de conexión de los clientes. Cuando el cliente se conecta el método accept() devuelve un objeto **Socket**, que se usará para crear un hilo que atenderá al cliente. Después se vuelve a invocar a accept() para esperar un nuevo cliente. Habitualmente la espera de conexiones se realiza dentro de un bucle infinito.

Todas las operaciones que sirven a un cliente quedan dentro de la clase hilo. El hilo permite que el servidor se mantenga a la escucha de peticiones y no interrumpa su proceso mientras los clientes son atendidos.

Ejemplo: Un servidor devuelve en mayúsculas las cadenas enviadas por los clientes hasta que recibe un asterisco. El proceso de tratamiento de la cadena se realiza en un hilo que se llamará HiloServidor. El programa cliente se conecta al servidor por el puerto 6000 y envía al servidor cadenas introducidas por teclado hasta que se recibe un asterisco.

Programa Servidor:

```
package servidor;
import java.io.*;
import java.net.*;
public class Servidor {
    public static void main(String args[]) throws IOException  {
        ServerSocket servidor;
        servidor = new ServerSocket(6000);
```

```
        System.out.println("Servidor iniciado...");
        while (true) {
            Socket cliente = new Socket();
            cliente=servidor.accept();//esperando cliente
            HiloServidor hilo = new HiloServidor(cliente);
            hilo.start();
        }
    }
}

class HiloServidor extends Thread {
    BufferedReader fentrada;
    PrintWriter fsalida;
    Socket socket = null;

    public HiloServidor(Socket s) throws IOException {// CONSTRUCTOR
        socket = s;
        // se crean flujos de entrada y salida
        fsalida = new PrintWriter(socket.getOutputStream(), true);
        fentrada = new
            BufferedReader(new InputStreamReader(socket.getInputStream()));
    }

    @Override
    public void run() {// tarea a realizar con el cliente
        String cadena = "";
        System.out.println("COMUNICO CON: " + socket.toString());
        while (!cadena.trim().equals("*")) {
            try {
                cadena = fentrada.readLine();
            } catch (IOException e) {
                System.out.println("IO: "+e.getMessage());
            } // obtener cadena
            fsalida.println(cadena.trim().toUpperCase());// enviar mayúscula
        }// fin while
        System.out.println("FIN CON: " + socket.toString());
        fsalida.close();
        try {
            fentrada.close();
        } catch (IOException e) {
            System.out.println("IO: "+e.getMessage());
        }
        try {
            socket.close();
        } catch (IOException e) {
            System.out.println("IO: "+e.getMessage());
        }
    }
}
```

Programa Cliente:

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

public class Cliente {
    public static void main(String[] args) throws IOException {
        String Host = "localhost";
        int Puerto = 6000;// puerto remoto
        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="";

        do{
            System.out.print("Introduce cadena: ");
            cadena = in.readLine();
            fsalida.println(cadena);
            eco=fentrada.readLine();
            System.out.println("  =>ECO: "+eco);
        } while(!cadena.trim().equals("*"));

        fsalida.close();
        fentrada.close();
        System.out.println("Fin del envío... ");
        in.close();
        Cliente.close();
    }
}
```

Salida Servidor:

```
run:
Servidor iniciado...
COMUNICO CON: Socket[addr=/127.0.0.1,port=62152,localport=6000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=62153,localport=6000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=62154,localport=6000]
FIN CON: Socket[addr=/127.0.0.1,port=62152,localport=6000]
FIN CON: Socket[addr=/127.0.0.1,port=62153,localport=6000]
FIN CON: Socket[addr=/127.0.0.1,port=62154,localport=6000]
```

Salida Cliente1:

```
run:
```

```
Introduce cadena: hola
  =>ECO: HOLA
Introduce cadena: cierro conexion con asterisco
  =>ECO: CIERRO CONEXION CON ASTERISCO
Introduce cadena: *
  =>ECO: *
Fin del envío...
BUILD SUCCESSFUL (total time: 1 minute 13 seconds)
```

Salida Cliente2:

```
run:
Introduce cadena: esto es una prueba
  =>ECO: ESTO ES UNA PRUEBA
Introduce cadena: cierro conexion con asterisco
  =>ECO: CIERRO CONEXION CON ASTERISCO
Introduce cadena: *
  =>ECO: *
Fin del envío...
BUILD SUCCESSFUL (total time: 1 minute 7 seconds)
```

Salida Cliente3:

```
run:
Introduce cadena: de servidor atendiendo a varios clientes
  =>ECO: DE SERVIDOR ATENDIENDO A VARIOS CLIENTES
Introduce cadena: cierro conexion con asterisco
  =>ECO: CIERRO CONEXION CON ASTERISCO
Introduce cadena: *
  =>ECO: *
Fin del envío...
BUILD SUCCESSFUL (total time: 1 minute 4 seconds)
```