UD3: Programación de comunicaciones en red

Segunda parte: Sockets

1. **Definición**
2. **Funcionamiento general**
3. **Tipos de sockets**
4. **Clases para sockets TCP.**
5. **Clases para sockets UDP.**
6. **Envío de objetos a través de sockets**
7. **Conexión de múltiples clientes. Hilos.**

1. Definición.

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 lo que llamamos socket**.**

Recordar que el proceso cliente envía el mensaje y el servidor recibe ese mensaje. Esta comunicación se hace mediante sockets

Para los procesos receptores de mensajes, su conector debe tener asociados 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.

Cada socket tiene un puerto asociado, el proceso cliente debe conocer el puerto y la dirección IP del proceso servidor. Los mensajes al servidor le deben llegar al puerto acordado. El cliente puede enviar el mensaje por el puerto que quiera.

2. Funcionamiento general.

La pila de protocolos IP y los sockets son las herramientas fundamentales sobre las cuales se desarrolla la práctica totalidad de aplicaciones distribuidas. Pero dependiendo de cuál sea el propósito y como vaya a funcionar internamente deberemos escoger un modelo de comunicaciones distinto.

Un modelo de comunicaciones es una arquitectura general que especifica cómo se comunican entre sí los diferentes elementos de una aplicación distribuida. Los más usados en la actualidad son:

- modelo cliente/servidor. Es el más sencillo y más común de los usados en la actualidad. Hay un proceso central llamado servidor que ofrece uno o más servicios a uno o más procesos cliente. Es en el modelo que más vamos a incidir a lo largo del tema, ya que los sockets stream nos ayudarán a implementarlo fácilmente.

- modelo de comunicaciones en grupo. En este modelo no existen roles diferenciados de tal manera que los procesos colaboran en un trabajo en común. Los mensajes se transmiten mediante lo que se denomina radiado, es decir, los mensajes se envían de manera simultánea a los distintos miembros del grupo.
- modelo hibrido y redes peer to peer (P2P).Los modelos híbridos son mezcla de los dos modelos anteriores. Las redes P2P son el ejemplo más potente de estos modelos(Ejemplo Spotify). Están formadas por un grupo de elementos distribuidos que colaboran con un objetivo en común. Cada elemento puede desempeñar los roles de cliente o de servidor. Así, al tener más de un servidor, el sistema es mucho más tolerante a fallos ya que puede seguir funcionando aunque algún elemento se desconecte. Además, el hecho de que cualquier elemento puede actuar como servidor permite repartir la carga de forma equilibrada entre todos los elementos, haciendo que el sistema sea aún más robusto.

Ahora vamos a empezar por conocer el funcionamiento de un modelo de comunicaciones cliente/servidor.

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. 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. Tipos de sockets

Hay dos tipos básicos de sockets en redes IP, los que utilizan el protocolo TCP, orientados a conexión y los que utilizan el protocolo UDP, no orientados a conexión

- **Sockets orientados a conexión**

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 enviar el mensaje. Son comunicaciones en las que es necesario que los procesos establezcan una conexión antes de intercambiar información, para lo que usan un stream.

Un stream es una secuencia ordenada de unidades de información que puede fluir en dos direcciones: hacia afuera de un proceso (de salida) o hacia dentro de un proceso (de entrada). Están diseñados para acceder a los datos de forma secuencial.

Protocolo del cliente:

1. Creación del "*socket cliente*". Crea el socket y le asigna un puerto, este se puede dejar a elección del SO.
2. Conexión del socket. Se localiza el socket del proceso servidor y se crea el canal de comunicación. El proceso cliente debe conocer la dirección IP del servidor y el puerto por el que escucha.
3. Envío y recepción de mensajes.
4. Cierre de la comunicación.

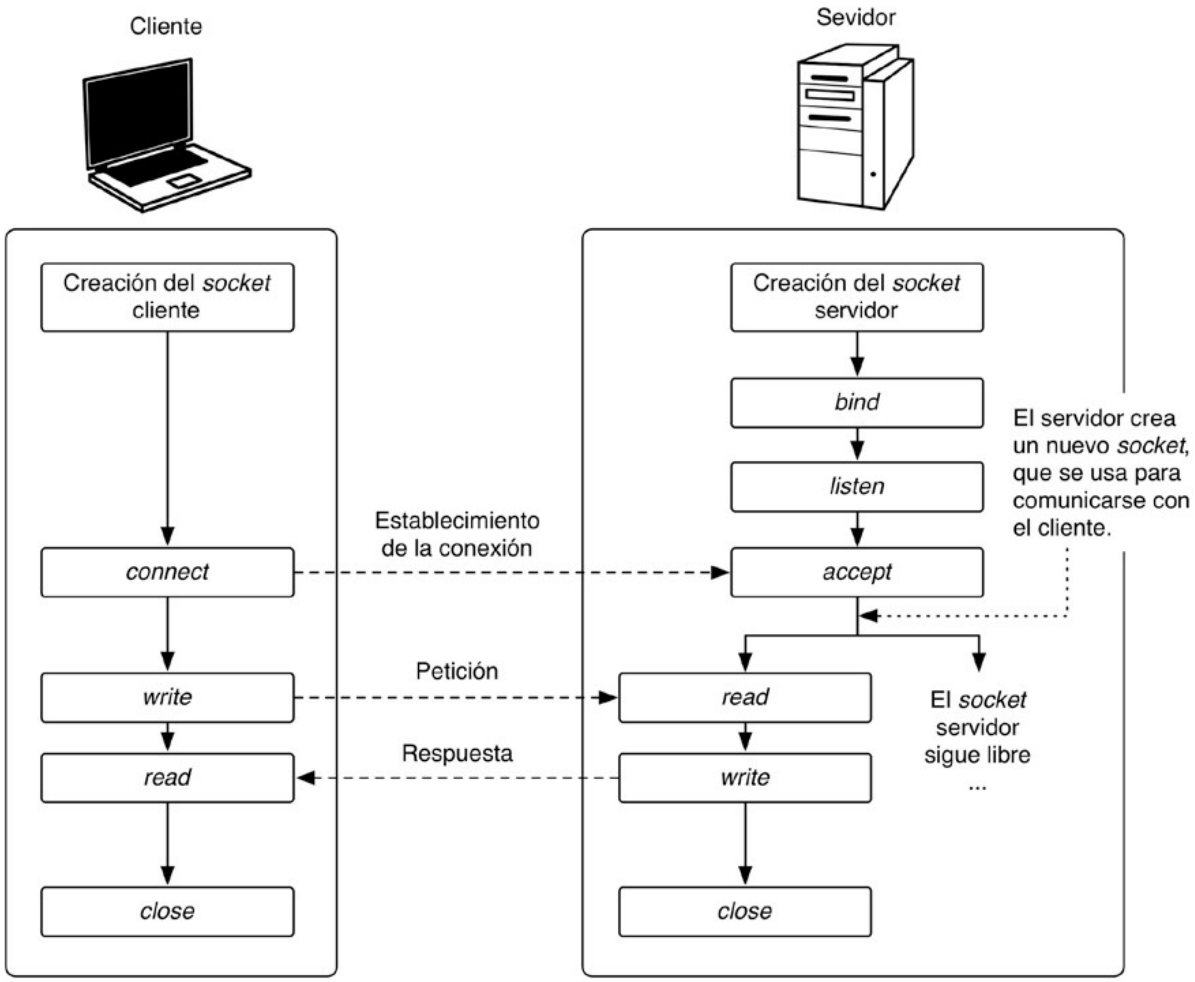
Protocolo del servidor:

1. Creación del "*socket servidor*".
2. Asignación de dirección y puerto. Esta operación conocida como blind asigna una dirección IP y un número de puerto concreto al socket servidor. Lógicamente, la dirección IP asignada debe ser la de la máquina que lo contiene.
3. Escucha. Se debe configurar el proceso para que escuche por ese puerto. Así la operación *listen* hace que el socket servidor quede preparado para aceptar conexiones por parte del proceso cliente.
4. Aceptación de conexiones. Una vez el socket servidor está listo para aceptar conexiones, el proceso servidor tiene que esperar hasta que un proceso cliente se conecte (operación *connect*). Cuando llega una petición de conexión se crea un nuevo socket dentro del proceso servidor, y es este el que queda conectado con el socket del proceso cliente, estableciendo la comunicación entre ambos (operación *accept*()). El socket servidor queda libre, por lo que puede seguir escuchando a la espera de nuevas conexiones.

5. Envío y recepción de mensajes. Una vez establecida la conexión, el proceso servidor puede enviar y recibir mensajes mediante operaciones de escritura y lectura sobre su nuevo socket. OJO: ¡no se usa el socket servidor para esta tarea!
6. Cierre de la conexión.

En cualquier momento, cualquiera de los procesos (cliente o servidor) puede cerrar su socket, destruyendo el canal y terminando así la comunicación.

Java incorpora las clases *Socket* (para implementar el cliente) y *ServerSocket* (para el servidor) para este tipo de conexiones.



- **Sockets no orientados a conexión**

En este tipo de sockets la comunicación entre las aplicaciones se realiza por medio del protocolo UDP. No es necesario establecer una comunicación punto a punto y no existe ningún flujo de control en el intercambio de información. Por lo tanto, en su utilización no está garantizada la fiabilidad ni el orden en que lleguen los paquetes. Por tanto los paquetes pueden estar duplicados, perdidos o llegar en un orden diferente en el que se enviaron. único

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 datagrama. Se suelen usar en aplicaciones para la transmisión de audio y vídeo en tiempo real dónde no es posible el reenvío de paquetes retrasados.

En este caso, no hay distinciones entre el proceso cliente y el servidor. Los pasos a seguir en ambos procesos son los siguientes:

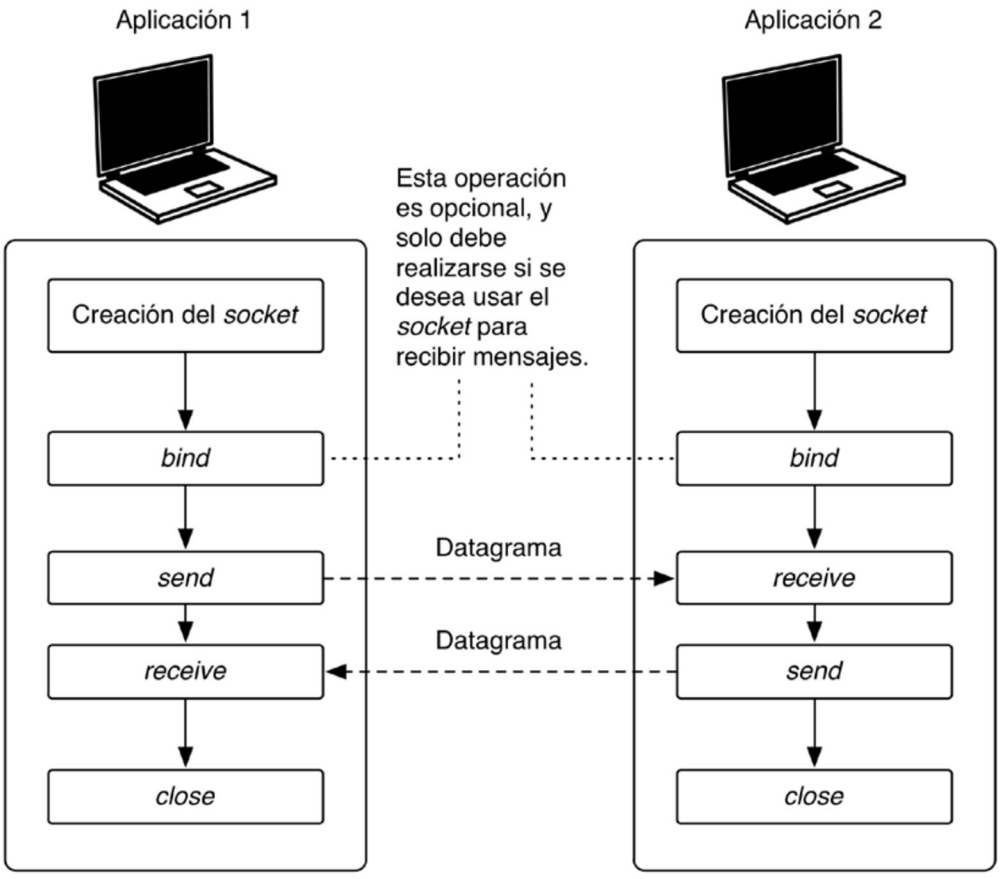
a) Creación del socket.

b) Asignación de dirección y puerto. En el caso de que se desee usar el socket para recibir mensajes, es importante que la dirección IP y el número de puerto del socket estén claramente especificados. De lo contrario los emisores no serán capacees de localizar al receptor. Al igual que en el otro caso, podemos usar la operación blind.

c) Envío y recepción de mensajes.

d) Cierre de la conexión.

En Java la comunicación con UDP se implementa mediante dos clases: *DatagramPacket* y *DatagramSocket*.



4. Clases para sockets TCP

4.1. La clase *socket*.

Algunos de sus constructores son:

Constructor	Descripción
Socket()	Crea un socket sin ningún puerto asociado. Se usa para procesos clientes.
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)	Premite 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 especificados. Puede lanzar UnknownHostException, IOException.

Algunos de los métodos importantes son:

Método	Descripción
InputStream getInputStream()	Devuelve un objeto de la clase InputStream que permite leer bytes desde el socket utilizando los mecanismos de streams. El socket debe estar conectado. Puede lanzar IOException.
OutputStream getOutputStream()	Devuelve un objeto de la clase OutputStream que permite escribir bytes desde el socket utilizando los mecanismos de streams. El socket debe estar conectado. Puede lanzar IOException.
close()	Se encarga de cerrar el socket
InetAddress getInetAddress()	Devuelve la dirección IP y puerto 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.
Connect(SocketAddress addr)	Establece la conexión con la dirección y puerto destino

Veamos un ejemplo de la clase que nos crea un socket cliente

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

public class Cliente{
    static final String SERVIDOR = ”localhost”;
    static final int PUERTO = 6000;

    //Constructor
    public Cliente(){
        try{
          Socket SocketC = new Socket(SERVIDOR, PUERTO);
          InputSream is = SocketC.getInputStream();
          DataInputStream dis = new DataInputStream(is);
          System.out.println(dis.readUTF());
          SocketC.close();
        }catch(Exception e){
          System.out.println(e.getMessage());
        }
    }

    public static void main(String args[]){
        new Cliente();
    }
}
```

Esta clase declara un objeto de tipo socket que utilizará para la comunicación con el servidor. El cliente debe especificar la máquina en la que se encuentra el servidor y el puerto por el que escucha. En este ejemplo, tanto la aplicación cliente como la servidora se encuentran en la misma máquina (localhost).

Una vez creado el socket, lo asocia a un objeto *InputStream* mediante el método *getInputStream()*, ya que la información irá del servidor al cliente. El objeto *InputStream* se asocia a un objeto *DataInputStream* para poder utilizar su método *readUTF()*, que obtendrá la información enviada por el servidor como si fuera un fichero. Una vez recibida la información, el cliente la imprime en la salida estándar.

Finalmente, el cliente cierra el objeto *socket* mediante su método *close()*.

4.2. La 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:

Constructor	Descripción
ServerSocket()	Crea un socket de servidor sin ningún puerto asociado.
ServerSocket(int port)	Crea un socket de servidor, que se enlaza al puerto especificado.
ServerSocket(int port, int máximo)	Crea un socket de servidor y lo enlaza con el número de puerto local especificado. El parámetro *máximo* especifica el número máximo de peticiones de conexión que se pueden mantener en cola.
ServerSocket(int port, int máximo, InetAddress direcc)	Crea un socket de servidor en el puerto indicado especificando un máximo de peticiones y conexiones entrantes y la dirección IP local.

Pueden lanzar IOEXception.

Algunos métodos importantes son:

Métodos	Descripción
Socket accept()	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.
Close()	Se encarga de cerrar el ServerSocket.
Int getLocalPort()	Devuelve el puerto local al que está enlazado el ServerSocket.

Veamos un ejemplo:

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

class Servidor{
    static final int PUERTO = 6000;

    //Constructor
    public Servidor(){
        try{
          ServerSocket SocketS = new ServerSocket(PUERTO);
          System.out.println("Servidor escuchando por el puerto "
          + PUERTO);
        for(int i=0; i<3;i++){
          //Crea el objeto socket cliente
          Socket SocketC = SocketS.accept();
          System.out.println("Escuchando al cliente " + i);
          OutputStream os = SocketC.getOutputStream();
          DataOutputStream dos = new DataOutputStream(os);
          dos.writeUTF("Hola cliente " + i);
          SocketC.close();
        }
        System.out.println("El servidor termina");
        SocketS.close();
        }catch(Exception e){
          System.out.println(e.getMessage());
        }
    }

    public static void main(String args[]){
        new Servidor();
    }
}
```

El servidor utiliza un objeto *SocketServer* para escuchar las peticiones por el puerto 6000. Para atender estas peticiones utiliza el método *accept()*. Este método crea un objeto *socket* cuya referencia guarda en *SocketC*, y es el que se utiliza para realizar la comunicación individual con cada cliente.

Para ejecutar la aplicación, debe ejecutarse primero la aplicación servidor, que se queda bloqueado a la espera de recibir peticiones de clientes.

A continuación se ejecuta una instancia de cada cliente, crearemos cuatro, y como el servidor sólo atiende a 3 peticiones, el cuarto recibirá una excepción al no poder conectarse con el servidor.

El resultado de ejecutar el servidor será:

```
Servidor escuchando por el puerto 6000
Escuchando al cliente 0
Escuchando al cliente 1
Escuchando al cliente 2
El Servidor termina.
```

5. Clases para sockets UDP

Recordar que en los sockets UDP no es necesario establecer una conexión entre cliente y servidor, 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 de destino	Nº del puerto de destino

5.1. La 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.

Algunos constructores son:

Constructor	Descripción
DatagramPacket(byte[] buf, int length)	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje (buf) y la longitud de la misma
DatagramPacket(byte[] buf, int legth, InetAddress addrss, int port)	Constructor para el envío de datagramas. Se especifica la cadena de bytes a enviar(buf), la longiutd (length), el número de puerto de destino (port) y el host especificado en la dirección addrss

Algunos de los métodos son:

Método	Descripción
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 la 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.

Veamos un ejemplo:

```
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(); // codifico el mensaje a bytes para poder enviarlo

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

Este ejemplo utiliza el segundo 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 calculamos 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.

5.2. La clase DatagramSocket.

Da soporte a sockets para el envío y recepción de datagramas UDP.

Algunos de sus constructores son:

Constructor	Descripción
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 el puerto y la dirección local a la que se va a asociar el socket.

Estos constructores pueden lanzar la excepción SocketException.

Veamos algunos métodos importantes:

Métodos	Descripción
receive(DatagramPacket paquete)	REcive un DatagramPacket del socket y llena paquete con los datos que recibe (mensaje, longitud y origen). Puede lanzar IOException.
send(DatagramPacket paquete)	Envía un DatagramPacket a través del socket. El argumento paquete contiene el mensaje y su destino. Puede lanzar 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 addrs, 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 time)	Permite establecer un tiempo de espera límite. Entonces el método *receive*() se bloquea durante el tiempo fijado. Si no se reciben datos en el tiempo fijado se lanza la excepción *InterruptedIOException*

Veamos un sencillo ejemplo de un proceso que recibe un mensaje y luego lo envía usando sockets datagram.

```
import java.io.*;
import java.net.*;
public class ReciboEnvio {
        public static void main(String[] args){
                try {
                        System.out.println("Creando socket datagrama");
                        InetSocketAddress addrs =new InetSocketAddress("localhost",5555);
                        DatagramSocket datagramSocket = new DatagramSocket(addrs);

                        System.out.println("Recibiendo mensaje");

                        byte[] mensaje =new byte[25];
                        DatagramPacket datagrama1 =new DatagramPacket (mensaje,25);
                        datagramSocket.receive(datagrama1);

                        System.out.println("mensaje recibido: "+ new String(mensaje));

                        System.out.println("Enviando mensaje");

                        InetAddress addr = InetAddress.getByName("localhost");
                        DatagramPacket datagrama2 = new DatagramPacket(mensaje, mensaje.length, addr, 5556);
                        datagramSocket.send(datagrama2);
                        datagramSocket.close();
                        System.out.println("Terminé");

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

OBS: En los sockets UDP no se establece conexión, por lo que los roles cliente-servidor están un poco más difusos que en el caso TCP. Podemos considerar al servidor como el que espera un mensaje y al cliente como el que inicia la conversación. Tanto uno como otro necesitan saber que ordenador y puerto tiene el otro. Ambos necesitan crear un socket DatagramSocket.

5.3. Clase multicastSocket

La clase MulticastSocket es útil para enviar paquetes a múltiples destinos simultáneamente. Para poder recibir estos paquetes 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 de multicast, todos los que pertenezcan a 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 direcciones multicast. La dirección 224.0.0.0 está reservada y no debe ser utilizada.

Los constructores de esta clase pueden lanzar la excepción IOException. Algunos de ellos son:

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

Algunos métodos son:

Métodos	Descripción
joinGroup(InetAddress mcastaddrs)	Permite al socket multicast unirse al grupo de multicast
leaveGroup(InetAddress mcastaddrs)	El socket multicasta abandona el grupo multicast
send(DatagramPacket p)	Envía el datagrama a todos los miembros del grupo multicast.
receive(DatagramPacket p)	Recibe el datagrama de un grupo multicast.

6. Envío de objetos a través de sockets

Hasta ahora hemos visto como intercambiar cadenas de caracteres entre programas cliente y servidor. Pero los streams soportan diversos tipos de datos como son los bytes, datos primitivos, caracteres localizados y objetos.

6.1. Objetos en sockets TCP

Las clases ObjectInputStream y ObjectOutputStream nos periten enviar objetos a través de sockets TCP. Utilizaremos los métodos readObject() para leer el objeto del stream y writeObject() para escribir el objeto al stream. Usaremos el constructor que admite un InputStream y un OutputStream.

Para preparar el flujo de salida para escribir objetos escribimos:

ObjectOutputStream outObjeto = new ObjectOutputStream (socket.getOutputStream());

Para preparar el flujo de entrada para leer objetos escribimos:

ObjectInputStream inObjeto = new ObjectInputStream (socket.getInputStream());

Las clases a las que pertenecen estos objetos deben implementar la interfaz Serializable. Por ejmplo, sea la clase Persona con dos atributos, nombre y edad, 2 constructores y los métodos set y get correspondientes.

```
Import java.io.Serializable;
@SuppressWarnings("serial")
//Esta anotación se utiliza para evitar un error en tiempo de compilación al implementar la interfaz java.io.Serializable
public class Persona implements Serializable {
        String nombre;
        int edad;
        public Persona (String nombre, int edad){
                super();
                this.nombre=nombre;
        this.edad =edad
        }
        public Persona(){
                Super();}
        public String getNombre(){
                return nombre;
                }
        public void setNombre (String nombre){
                this.nombre = nombre;
                }
        public int getEdad(){
                return edad;
                }
        public void setEdad(int Edad){
```

```
                this.edad=edad
                }
}
```

Podemos intercambiar objetos persona entre un cliente y un servidor usando sockets TCP. Por ejemplo el programa servidor crea un objeto persona, dándole valores y se lo envía al programa cliente. El programa cliente realiza los cambios oportunos en el objeto y se lo devuelve modificado al servidor.

El programa servidor es el siguiente:

```
import java.io.*;
import java.net.*;
public class ServidorObjeto {
        public static void main(String[] arg) throws IOException, ClassNotFoundException{
                int numeroPuerto=6000;
                ServerSocket servidor = new ServerSocket (numeroPuerto);
                System.out.println("Esperando al cliente..");
                Socket cliente= servidor.accept();

                //preparamos el flujo de salida para objetos
                ObjectOutputStream outObjeto = new ObjectOutputStream (cliente.getOutputStream());

                //Se preparaper. un objeto y se envía
                Persona per =new Persona ("juan", 22);
                outObjeto.writeObject(per); //enviando el objeto
                System.out.println("Envío: "+per.getNombre()+ "*"+ per.getEdad());

                //Obtenemos un stream para leer objetos
                ObjectInputStream inObjeto = new ObjectInputStream(cliente.getInputStream());
                Persona dato =(Persona) inObjeto.readObject();
                System.out.println("Recibo: "+dato.getNombre() + "*"+dato.getEdad());

                //cerramos los streams y los sockets
                outObjeto.close();
                inObjeto.close();
                cliente.close();
                servidor.close();
                }
}
```

El programa cliente es el siguiente:

```
import java.io.*;
import java.net.*;
public class ClienteObjeto{
        public static void main (String[] arg)throws IOException, ClassNotFoundException{
                Sring host= “LocalHost”;
                int puerto = 6000; // puerto remoto
                System.out.println(“Programa cliente iniciado…”);
                Socket cliente = new Socket (host, puerto);

                //flujo de entrada para objetos
                ObjectInputStream perEnt = new ObjectInputStream (cliente.getInputStream());

                //se recibe un objeto
                Persona dato= (Persona) perEnt.readObject(); //recibo el objeto
                System.out.println(“Recibo: ”+dato.getNombre+ “*”+ dato.getEdad());

                //modifico el objeto
                dato.setNombre(“Juan Pérez”);
                darto.setEdad(25);

                //flujo de salida para objetos
                ObjectOutputStream perSal = new ObjectOutputStream(cliente.getOutputStream());

                //Se envía el objeto
                persal.writeObject(dato);
                System.out.println(“Envío: ”+dato.getNombre() +”*” + dato.getEdad());
                //cerramos streams y sockets
                perEnt.close();
                persal.close();
                cliente.close();
        }
}
```

6.2. Objetos en sockets UDP

Para intercambiar objetos en sockets UDP utilizaremos las clases ByteArrayOutputStream y ByteArrayInputStream. Se necesita convertir el objeto a un array de bytes. Por ejemplo, para convertir un objeto Persona en un array de bytes escribimos las siguientes líneas:

```
Persona persona = new Persona (“Marina”, 22);
//convertimos el objeto a bytes
ByteArrayOutputStream bs = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bs);
out.writeObject(persona); //escribimos el objeto persona en el stream
out.close(); //cerramos el stream
byte[] bytes =bs.toByteArray(); // objeto en bytes
```

Para convertir los bytes recibidos por el datagrama en un objeto Persona escribimos:

```
//Recibo datagrama
byte[] recibidos = new byte[1024];
DatagramPacket paqRecibido = new DatagramPacket(recibidos, recibidos.length);
socket.receive(paqRecibido); //recibo el datagrama
//convertimos bytes a objeto
ByteArrayInputStream b = new ByteArrayInputStream(recibidos);
ObjectInputStream in = new ObjectInputStream(b);
Persona persona = (Persona) in.readObject(); //obtengo el objeto
in.close();
```

7. Conexión de múltiples clientes. Hilos.

Hasta ahora los programas que hemos creado solo pueden atender a un cliente en cada momento, pero lo más típico es que un programa servidor pueda atender a muchos clientes simultáneamente. Esto quiere decir que, si al servidor le llega una nueva petición mientras está atendiendo a un cliente, se deben cumplir las siguientes condiciones:

- La nueva petición no debe interferir con la que está en curso. Dicho de otra forma, los clientes deben percibir que operan con el servidor ellos solos, independientemente de cuántos clientes haya.
- La nueva petición debe ser atendida lo antes posible, incluso de manera simultánea a la que está ya en curso. Esto evita que unos clientes tengan que esperar por otros.

La solución para poder atender a múltiples clientes está en el multihilo, cada cliente será atendido por un hilo.

El esquema básico en sockets TCP sería construir un único servidor con la clase *ServerSocket* e invocar al método *accept*() para esperar las peticiones de conexión de los clientes. Cuando un cliente se conecta, el método *accept*() devuelve un objeto Socket, éste se usará para crear un hilo cuya misión es atender a este cliente. Después se vuelve a invocar a *accept*() para esperar a un nuevo cliente. Habitualmente la espera de conexiones se hace dentro de un bucle infinito:

```
public class servidor{
        public static void main(String args[]) throws IOException{
                ServerSocket servidor = new ServerSocket (6000);
                System.out.println("Servidor iniciado…");
                while(true){
                        Socket cliente = new socket();
                        cliente = servidor.accept(); //esperamos al cliente
                        HiloServidor hilo = new HiloServidor(cliente);
                        hilo.start(); // se atiende al cliente
                        }
                }
}
```

Todas las operaciones que sirven a un cliente en particular 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.

Por ejemplo, supongamos que el cliente envía una cadena de caracteres al servidor y el servidor se la devuelve en mayúsculas, hasta que recibe un asterisco que finalizará la comunicación con el cliente. El proceso de tratamiento de la cadena se realiza en un hilo, en este caso se llama HiloServidor.

NOTA: se han eliminado los bloques try-catch para simplificar.

```
import java.io.*;
import java.net.*;
public class HIloServidor extends Thread{
        BufferedReader fentrada;
        PrintWriter fsalida;
        Socket socket = null;
        public HiloServidor (Socket s ){ //constructor
        socket = s;
        // se crean flujos de entrada y salida
        fsalida = new PrintWriter (socket.getOutputStream(),true);
        fentrada = new BufferedReader (new InputStreamReader(socket.getInputStream()));
        }
        public void run(){ //tarea a realizar con el cliente
                String cadena= "";
                while (!cadena.trim().equals("*")){
                System.out.println ("Comunico con: "+socket.toString());
                cadena =fentrada.readLine(); //obtenemos la cadena
                fsalida.println(cadena.trim().toUpperCase()); //enviamos en mayúscula
                }
                system.out.println("fin con: "+ socket.toString());
                fsalida.close();
                fentrada.close();
                socket.close();
                }
                }
```

Como programa cliente podemos ejecutar un programa que se conecte con el servidor en el puerto 6000 y le enviará cadenas introducidas por teclado, cuando le envíe un asterisco el servidor finalizará la comunicación con el cliente.