PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Los servicios en red
Clases Socket y DatagramPacket
Sockets TCP y UDP

Sockets

Los Sockets son un **mecanismo de comunicación entre aplicaciones**, utilizado principalmente en redes de comunicación. Podemos entender los Sockets, en sí, como uno de los **extremos de una conexión bidireccional** entre dos **programas** que se están ejecutando en **red**. Por lo tanto, representan los **extremos del canal de comunicación** que necesitamos.

Los Sockets van a identificarse mediante una **dirección IP** y un **puerto** en el que transmitirán.

Podremos utilizar **dos tipos de Sockets**:

- Sockets TCP (orientados a conexión):
  - ➔ Este tipo de Sockets se van a utilizar para establecer una comunicación en red utilizando el protocolo TCP.
  - ➔ Esta conexión **no comenzará hasta** que los dos **Sockets** estén **conectados**.
  - ➔ La **forma de transmitir** información en una conexión con Sockets TCP será en **bytes**.
  - ➔ Para las aplicaciones que utilizan Sockets TCP, podremos utilizar las siguientes **clases en Java**: **Socket y ServerSocket**, que implementarán el cliente y el servidor de la conexión, respectivamente.
  - ➔ Vamos a utilizar la **clase Socket** tanto para **transmitir** como para **recibir datos**, mientras que la clase **ServerSocket** se encargará de implementar el servidor y su única tarea será **esperar** a que un **cliente** desee establecer una conexión con el servidor.
- Sockets UDP (no orientados a conexión):
  - ➔ Este tipo de Sockets se utilizará para establecer una comunicación en red utilizando el protocolo UDP.
  - ➔ La **forma de transmitir** información en una conexión con Sockets UDP será en **datagramas**.
  - ➔ Estos Sockets son mucho **más rápidos** que los TCP, aunque **menos seguros**.
  - ➔ Para las aplicaciones que utilizan Sockets UDP, podremos utilizar las **clases en Java DatagramSocket**.

Sockets en otros lenguajes de programación

Ya hemos visto las clases que nos proporciona el lenguaje de programación Java para la implementación de Sockets, pero ¿cuáles son en otros lenguajes?

En **C++** tenemos también la **clase socket**, en **Python** tenemos un **módulo** llamado también **socket** y en **C#** tenemos la **clase Sockets**.

Como podemos ver, los Sockets están presentes en cualquier lenguaje de hoy en día, con una **nomenclatura similar**, y ofreciendo la **misma funcionalidad**.

Servidor TCP

Pasos para crear aplicaciones que usen Sockets en Java:

- **Crear o abrir** los **Sockets** tanto en el **cliente** como en el **servidor**.
- **Crear los flujos de entrada o salida** tanto en el **cliente** como en el **servidor**.
- **Cerrar los flujos** y los **Sockets**.

Además de todo esto, como es posible que se produzcan **errores** en las transmisiones en red de información, será necesario llevar el control de excepciones mediante el **bloque try-catch**.

Vamos a crear una **clase llamada ServidorTCP**, que actuará como servidor de nuestra aplicación. En esta clase, vamos a crear un constructor en el que crearemos toda la funcionalidad que deseemos y, además, crearemos un método main en el que implementaremos nuestro servidor.

Servidor TCP simple

```
public class ServidorTCP {
	/**
	 * En esta clase, vamos a crear una variable que indicará
	 * el puerto donde escuchará nuestro servidor, por ejemplo,
	 * el puerto 6000.
	 */
	private final int PUERTO = 6000;

	public ServidorTCP() {
		try {
			/**
			 * Para implementar el servidor, crearemos un
			 * objeto de la clase ServerSocket con el puerto
			 * deseado.
			 */
			ServerSocket skServidor =
					new ServerSocket(PUERTO);
			/**
			 * Una vez hecho esto, deberemos crear un bucle
			 * infinito para atender a todos los clientes que
			 * se conecten al servidor, obteniéndolos mediante
			 * el método accept(), que devolverá el cliente
			 * conectado.
			 *
			 * Escucho a los clientes
```

```
             */

            while (Boolean.TRUE) {
                  Socket skCliente = skServidor.accept();
                  /**
                   * El siguiente paso será crear los flujos de
                   * entrada y de salida para poder llevar a
                   * cabo la comunicación, mediante los métodos
                   * readUTF() y writeUTF(), respectivamente.
                   */
                  // Obtengo el flujo de entrada del cliente
                  // (Mensajes que recibe del cliente)
                  InputStream aux2 = skCliente.getInputStream();
                  DataInputStream flujorecibir =
                              new DataInputStream(aux2);
                  // Obtengo el flujo de salida del cliente
                  // (Mensajes que envía al cliente)
                  OutputStream aux = skCliente.getOutputStream();
                  DataOutputStream flujoenviar =
                              new DataOutputStream(aux);
                  // Manda un mensaje al cliente
                  flujoenviar.writeUTF("Hola cliente");
                  // Por último, deberemos de cerrar el Socket.
                  // Cierro el socket servidor
                  skCliente.close();
            }
      } catch (IOException e) {
            System.out.println("Error en el servidor: "
      + e.getMessage());
      }
   }
}
```

Cliente TCP

Una vez creado nuestro servidor, vamos a pasar a crear un cliente, el cual tendrá que:

1. Crear un **Socket** y **conectar con el servidor** mediante su **IP y puerto**.
2. Crear los **flujos** de entrada o salida.
3. Una vez terminada la comunicación, **cerrar los flujos y los Sockets**.

De igual forma que pasaba con el servidor, es posible que se produzcan errores en las transmisiones en la red de comunicación, será necesario, por tanto, llevar el **control de excepciones** mediante el bloque try-catch.

Vamos a crear una **clase** llamada **ClienteTCP**, que actuará como uno de los **clientes que se conectarán al servidor** de nuestra aplicación. En esta clase, vamos a crear un **constructor** en el que montaremos toda la **funcionalidad** que necesitemos, y además, implementaremos un método **main** en el que **crearemos nuestro cliente**.

```
public class ClienteTCP {
	/**
	 * crearemos una variable que indicará el puerto donde
	 * escuchará nuestro servidor, por ejemplo, el puerto
	 * 6000 y otra con el host donde se aloja el servidor
	 * (en nuestro caso, como será en la misma máquina,
	 * será localhost).
	 */
	private static final String HOST = "localhost";
	private static final int PUERTO = 6000;

	public ClienteTCP() {
		try {
			/**
			 * Para crear el cliente, crearemos un objeto de
			 * la clase Socket con el host y el puerto deseado.
			 */
			Socket skCliente =
					new Socket(HOST, PUERTO);
			/**
			 * Una vez hecho esto, el servidor nos aceptará,
			 * por lo que el siguiente paso será crear los
			 * flujos de entrada y de salida para poder llevar
			 * a cabo la comunicación, mediante los métodos
			 * readUTF() y writeUTF() respectivamente.
			 */
```

```
                // Obtengo el flujo de entrada del cliente creado
                // (Mensajes que recibe el cliente del servidor)
                InputStream aux = skCliente.getInputStream();
                DataInputStream flujo =
                            new DataInputStream(aux);
                /**
                 * Por último, deberemos de cerrar el Socket.
                 */
                // Cierro el socket
                skCliente.close();

            } catch (IOException ex) {
                System.out.println("Error -> " + ex.toString());
            }

        }

}
```

Ejemplo de aplicación con Sockets TCP

¿Cómo crear una aplicación de comunicación en red?

Vamos a ver cómo podemos crear una aplicación de comunicación en red mediante el protocolo TCP, utilizando sockets TCP. Para ello vamos a crear dos clases, una que va a representar el servidor y otra que va a representar el cliente.

En la clase **ServidorTCP** vamos a tener lo siguiente:

- Una **variable int** que va a ser el **puerto** por el que va a escuchar el servidor y en el que tendrán que hablar los clientes. Por ejemplo, el puerto 6000.
- Y una variable MAX_CLIENTES que valdrá 3, que serán los clientes que va a dar soporte a nuestro servidor.

En el constructor de nuestra clase tenemos dentro de un **bloque try-catch** lo siguiente:

1. Mediante la clase ServerSocket hemos creado un socket (skServidor) al servidor con el puerto en el que escuchamos.
2. Ahora con un bucle for desde cero hasta el máximo de clientes **esperaremos a que se conecte un cliente** mediante el método accept, el cual será devuelto en un objeto del tipo socket.
3. Ahora vamos a obtener el flujo de salida del cliente por el que podremos enviarle mensajes.
4. Y una vez hecho esto mediante el método writeUTF podremos **enviar el mensaje** que queramos,
5. y cerramos el socket del cliente ya que está **servido**.
6. Dentro de la clase servidorTCP tenemos un método main que creará un servidor.

```
public class ServidorTCP {
    private static final int PUERTO = 6000;
    private static final int MAX_CLIENTES = 3;
    public ServidorTCP() {
        try {
            // Creo el socket servidor que escucha
            // en el puerto 6000
            ServerSocket skServidor =
                    new ServerSocket(PUERTO);
            System.out.println("Escucho el puerto "
                    + PUERTO);
            // Escucho a un cierto número de clientes
            for (int numCli = 0; numCli < MAX_CLIENTES; numCli++)
            {
                // Escucho a un cliente
                Socket skCliente = skServidor.accept();
                // Cuando escucha un cliente da un aviso
                System.out.println("Sirvo al cliente "
                + numCli);
```

```
                // Obtiene el flujo de salida del cliente
                // (Mensajes que envía al cliente)
                OutputStream aux = skCliente.getOutputStream();
                DataOutputStream flujo =
                          new DataOutputStream(aux);
                // Manda un mensaje al cliente
                flujo.writeUTF("Hola cliente " + numCli);
                // Cierro el socket servidor
                skCliente.close();
            }
            System.out.println("Demasiados clientes por hoy.");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
    // Método main del servidor
    public static void main(String[] args) {
        new ServidorTCP();
    }
}
```

En la clase **ClienteTCP** tendremos la representación de un cliente que se conectará a nuestro servidor TCP.

Esta clase tiene:

- una **variable** del tipo **String** que es el **host** donde se va a conectar nuestro cliente, que en este caso tiene el valor "**localhost**".
- Una **variable** del tipo **int** que es el **puerto**, que tiene como valor 6000, que es el mismo puerto en el que escucha nuestro servidor.

Dentro del constructor de nuestro cliente y dentro de un **bloque try-catch** para poder gestionar todas las acciones que se nos presenten. Tenemos lo siguiente:

1. Creamos un socket con la clase socket que se conectará a un puerto y a un host. Aquí será donde se conectará al servidor.
2. Una vez conectado y aceptado por el servidor tendremos que obtener el flujo de entrada del cliente, que serán los mensajes que recibe el cliente, es decir, los mensajes enviados por el servidor.
3. Una vez hecho esto podremos obtener el mensaje que se nos envíe por el servidor mediante el método readUTF, el cual devolverá un String.
4. Y cerraremos el socket ya que está servido.
5. Dentro de la clase cliente TCP tenemos un método main que ejecuta un cliente.

```java
public class ClienteTCP {

    private static final String HOST = "localhost";
    private static final int PUERTO = 6000;
    public ClienteTCP() {

        try {
            // Creo el socket cliente que escucha en la
            // máquina localhost y en el puerto 6000
            Socket skCliente = new Socket(HOST, PUERTO);
            // Obtengo el flujo de entrada del cliente creado
            // (Mensajes que recibe el cliente del servidor)
            InputStream aux = skCliente.getInputStream();
            DataInputStream flujo = new DataInputStream(aux);
            // Saco por pantalla el mensaje recibido del servidor
            System.out.print("Mensaje recibido del servidor: ");
            System.out.println(flujo.readUTF());
            // Cierro el socket
            skCliente.close();
        } catch (IOException ex) {
            System.out.println("Error en el cliente: " + ex.toString());
        }
    }
    // Método main del cliente
    public static void main(String[] args) {
        // Creo un cliente
        new ClienteTCP();
    }
}
```

Ejecución de la aplicación:

Primero deberemos ejecutar el **servidor**:

En la clase ServidorTCP pulsamos botón derecho y seleccionamos “run file”. Y veremos que el servidor escucha en el puerto 6000.

Y ahora ejecutaremos los clientes de la misma forma. Botón derecho en la clase cliente TCP y “run file”. Aquí vemos que el servidor nos ha enviado este mensaje:

Escucho el puerto 6000

Sirvo al cliente 0

Volvemos a ejecutar otra vez, y tenemos el cliente 1.

Volvemos a ejecutar otra vez, y tenemos al cliente 2.

Terminando el servidor su ejecución ya que solamente admite 3 clientes:

Escucho el puerto 6000

Sirvo al cliente 0

Sirvo al cliente 1

Sirvo al cliente 2

Demasiados clientes por hoy.

Sockets UDP
Clases Datagrampacket y Datagramsocket

Los Sockets UDP serán algo más sencillos de implementar que los TCP, ya que, si recordamos, el protocolo UDP no es un protocolo orientado a conexión, por lo que simplemente deberemos crear el Socket y enviar o recibir mensajes.

Para poder enviar o recibir datos a través de una conexión UDP deberemos utilizar la **clase DatagramPacket** y crear un **Socket UDP** mediante la **clase DatagramSocket**. La documentación de las clases la podemos encontrar en:

http://docs.oracle.com/javase/8/docs/api/java/net/DatagramPacket.html

https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html

La clase DatagramPacket va a representar un **datagrama**, pudiendo **enviar o recibir paquetes**, también denominados datagramas. Un datagrama no es más que un **array de bytes** enviado a una **dirección IP** y a un **puerto UDP** concreto.

Un **datagrama** tiene la siguiente forma:

datagrama

Array de bytes + Longitud + IP destino + Puerto destino

Primero implementamos el elemento DatagramSocket y luego lo agregamos al elemento contenedor o paquete DatagramPacket:

Métodos de la clase DatagramPacket:

- getData(): Este método devuelve el **array de bytes** que contiene los datos.
- getLenght(): Este método devuelve la **longitud del mensaje** a enviar o recibir.
- getPort(): Devuelve el **puerto de envío/recepción** del paquete.
- getAddress(): Devuelve la **dirección del host** remoto de envío/recepción (dirección IP).

Métodos la clase DatagramSocket:

- send(datagrama): Permite **enviar** datagramas.
- receive(datagrama): **Recibe** datagramas.
- close(): **Cierra** el Socket.
- getLocalPort(): Devuelve el **puerto donde está conectado** el Socket.
- getPort(): Devuelve el puerto **de donde procede** el Socket.

Servidor UDP

Cuando utilizamos Sockets UDP, no necesitamos realizar ningún tipo de conexión. Por este motivo, la **diferenciación entre clientes y servidores** será un poco **más complicada**.

Vamos a distinguir al servidor UDP como el que **espera un mensaje de un cliente** y lo responde, y consideraremos al cliente como el que **inicia la comunicación**.

Al igual que con el protocolo TCP, es posible que se produzcan errores en las transmisiones en red de información. Por ello, también será necesario llevar el **control de excepciones** mediante el bloque try-catch.

Vamos a crear una **clase llamada ServidorUDP**, que actuará como el servidor de nuestra aplicación. En esta clase, vamos a crear un constructor en el que montaremos toda la **funcionalidad** que deseemos, y además, implementaremos un método main, en el que **ubicaremos nuestro servidor**.

En esta clase, vamos a crear una variable que indicará el **puerto** donde escuchará nuestro servidor, por ejemplo, el puerto 6789.

Para crear el servidor, utilizaremos las **clases DatagramSocket y DatagramPacket**, para poder crear el **Socket** y los **mensajes datagramas** intercambiados. También deberemos crear un **array del tipo byte** para los datos.

Una vez hecho esto, simplemente deberemos **esperar la recepción** de un mensaje, mediante el método **receive**, pudiendo responder a la misma utilizando el método **send**.

En este caso, **no deberemos cerrar ninguna conexión**, ya que no existe.

Servidor UDP simple

```
public class ServidorUDP {
	private static final int PUERTO = 6789;

	public ServidorUDP() {
		try {
			DatagramSocket socketUDP = new DatagramSocket(PUERTO);
			byte[] bufer = new byte[1000];
			System.out.println("Escucho €el puerto " + PUERTO);
			while (true) {
				// Construimos el DatagramPacket para recibir peticiones
				DatagramPacket peticion =
						new DatagramPacket(bufer, bufer.length);
				// Leemos una peticion del DatagramSocket
				socketUDP.receive(peticion);
				// Obtengo el mensaje del cliente
				String mensajerecibido =
						new String (peticion.getData());
				// Construimos el DatagramPacket para enviar la respuesta
				String mensajerespuesta = "Hola cliente " + mensajerecibido;
				DatagramPacket respuesta =
						new DatagramPacket (
								mensajerespuesta.getBytes(),
								mensajerespuesta.length(),
								peticion.getAddress(),
								peticion.getPort()
								);
				// Enviamos la respuesta, que es un eco
				 socketUDP.send(respuesta);
			}
		} catch (IOException e) {
			System.out.println("Error: " + e.getMessage());
		}
	}
}
```

Cliente UDP

Ya hemos concretado que, en una comunicación UDP, vamos a distinguir al **cliente** como el que **inicia la comunicación**.

Nuevamente, al igual que pasaba cuando utilizábamos el protocolo TCP, es posible que se produzcan errores en las transmisiones en red de información, por lo que también será necesario llevar el **control de excepciones** mediante el bloque try-catch.

Crearemos una clase llamada **ClienteUDP**, que actuará como un cliente de nuestra aplicación. Nuevamente, en esta clase, debemos crear un constructor con toda la **funcionalidad** que deseemos y, además, crearemos el main en el que **ubicaremos el cliente UDP**.

En esta clase, vamos a crear una variable int que indicará el **puerto** donde escuchará nuestro servidor; por ejemplo, el puerto 6789, y otra variable String que indicará el **host** al que nos vamos a conectar (en nuestro caso, **localhost**), ya que tanto el cliente como el servidor van a estar en la **misma máquina**.

Para crear el cliente, utilizaremos de nuevo las **clases DatagramSocket y DatagramPacket**, para poder crear el **Socket** y los **mensajes datagramas** intercambiados. Como ocurría en el servidor UDP, deberemos crear un array del tipo **byte para los datos**.

Una vez hecho esto, simplemente deberemos **enviar un mensaje al servidor** utilizando el método send y **recibir** un mensaje del mismo mediante el método receive.

Cliente UDP simple

```
public class ClienteUDP {
	private static final int PUERTO = 6789;
	private static final String HOST = "localhost";

	public ClienteUDP() {
		try {
			String mensajeenviar = String.valueOf("Hola");
			// Creo el socket UDP
			DatagramSocket socketUDP =
					new DatagramSocket();
			byte[] mensaje = mensajeenviar.getBytes();
			// Obtengo la direccion del host
			InetAddress hostServidor = InetAddress.getByName (HOST);
			// Construimos un datagrama para enviar al servidor
			DatagramPacket peticion =
					new DatagramPacket (
							mensaje,
							mensajeenviar.length(),
							hostServidor,
							PUERTO);
			// Enviamos el datagrama
			socketUDP.send(peticion);
```

```
                    // Construimos el DatagramPacket que contendra la respuesta
                    byte[] bufer = new byte[1000];
                    // Construimos el DatagramPacket que contendra la respuesta
                    DatagramPacket respuesta =
                                new DatagramPacket(
                                            bufer,
                                            bufer.length);
                    socketUDP.receive(respuesta);
                    // Enviamos la respuesta del servidor a la salida estandar
                    System.out.println("Respuesta: " +
                    new String(respuesta.getData()));
                    // Cerramos el socket
                    socketUDP.close();

              } catch (IOException e) {
                    System.out.println("Error: " + e.getMessage());
              }
        }
}
```

Ejemplo de aplicación con Sockets UDP

Vamos a ver un ejemplo de una aplicación cliente-servidor programada con Sockets UDP.

Vamos a crear **dos clases**, una que va a representar al **servidor UDP** y otra que va a representar al **cliente UDP**.

En el **ServidorUDP** vamos a tener lo siguiente:

- ✔ La variable int PUERTO va a representar el puerto donde se conectará nuestro cliente, que será el mismo puerto que tendremos en el servidor.
- ✔ En el constructor de la clase es donde vamos a poner toda la **funcionalidad** de nuestra aplicación cliente. Si nos fijamos tenemos un **bloque try-catch** ya que todo el tratamiento con Sockets, tanto TCP como UDP, puede dar algún fallo y hay que tratarlos.
- ✔ Lo primero que hacemos será crearnos un datagram socket que será un socket de UDP. Lo crearemos con el puerto al que nos conectamos.
- ✔ Crear un array del tipo byte que nos va a permitir tanto el envío como la recepción de datagramas UDP.
- ✔ En un bucle infinito, para poder dar servicio a todos los clientes que se nos conecten, tendremos lo siguiente:
    1. Creamos un objeto de la clase DatagramPacket que nos permitirá realizar las operaciones. A él le vamos a pasar tanto el buffer como la longitud del mismo.
    2. Esperamos a recibir una petición de un cliente.
    3. Una vez recibida, mostraremos toda la información del mismo, tanto como el **mensaje**, el **puerto** de donde se recibe y el **host** de donde se recibe.
    4. Crearemos un mensaje respuesta y lo **enviaremos** mediante un objeto DatagramPacket.
    5. Tenemos un método main que será el encargado de ejecutar nuestro servidor.

```
public class ServidorUDP {

	private static final int PUERTO = 6789;

	public ServidorUDP() {
		try {
			DatagramSocket socketUDP = new DatagramSocket(PUERTO);
			byte[] bufer = new byte[1000];
			System.out.println("Escucho el puerto " + PUERTO);
			/**
			 *  Creo un bucle infinito para atender a todos los
			 *  clientes que se conecten sin límite
```

```
                 */
                while (true) {
                // Construimos el DatagramPacket para recibir peticiones
                      DatagramPacket peticion =
                                  new DatagramPacket(
                                              bufer,
                                              bufer.length);
                      // Leemos una petición del DatagramSocket
                      socketUDP.receive(peticion);
                      // Otengo el mensaje del cliente
                      String mensajerecibido =
                                  new String(peticion.getData());
                      System.out.print("Datagrama recibido del host: " +
                                  peticion.getAddress());
                      System.out.print(" desde el puerto remoto: " +
                                  peticion.getPort());
                      System.out.println(" con el mensaje: " +
                                  mensajerecibido);
                    // Construimos el DatagramPacket para enviar la respuesta
                      String mensajerespuesta = "Hola cliente " +
                      mensajerecibido;
                      DatagramPacket respuesta =
                                  new DatagramPacket(
                                              mensajerespuesta.getBytes(),
                                              mensajerespuesta.length(),
                                              peticion.getAddress(),
                                              peticion.getPort());
                      // Enviamos la respuesta, que es un eco
                      socketUDP.send(respuesta);
                }
          } catch (IOException e) {
                System.out.println("Error: " + e.getMessage());
          }
    }

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

En el **ClienteUDP** tenemos lo siguiente:

- ✔ Una variable int que guarda el puerto que va a ser el mismo que el cliente.
- ✔ Una variable String para conectar con el servidor que, como estamos en la misma máquina, va a ser localhost.
- ✔ Y una variable Random, ya que vamos a mandar números aleatorios al servidor.

1. Lo primero que haremos será en el constructor, en un bloque try-catch, crear un **número aleatorio** que será el mensaje que vamos a enviar.
2. Ahora, mediante un objeto de DatagramSocket, vamos a realizar la conexión.
3. Creamos el mensaje en un array de bytes.
4. Mediante el método getBytes de string, obtenemos la **URL del servidor**.
5. Creamos un DatagramPacket con la **petición** que será lo que vamos a enviar. En ella tenemos:
    - el mensaje,
    - la longitud,
    - el host donde está el servidor
    - y el puerto.
6. Lo enviamos mediante el método send.
7. Construimos el DatagramPacket que va a contener la respuesta, para recibirla del servidor mediante el método receive.
8. Y la mostramos por pantalla.
9. Aquí podemos ver que también tenemos un método main que será el encargado de ejecutar el cliente.

```
public class ClienteUDP {

    private static final int PUERTO = 6789;
    private static final String HOST = "localhost";
    private Random aleatorio;

    public ClienteUDP() {
        try {
            /**
             *  Creo el mensaje a enviar al servidor
             *  con un número aleatorio
             */
            aleatorio = new Random();
            int numerocliente = aleatorio.nextInt(100) + 1;
            String mensajeenviar = String.valueOf(
                    numerocliente);
            // Creo el socket UDP
            DatagramSocket socketUDP =
```

```
                new DatagramSocket();
            byte[] mensaje = mensajeenviar.getBytes();
            // Obtengo la dirección del host
            InetAddress hostServidor = InetAddress.getByName(HOST);
            /**
             *  Construimos un datagrama para enviar el mensaje
             *  al servidor
             */
            DatagramPacket peticion
                    = new DatagramPacket(
                                mensaje,
                                mensajeenviar.length(),
                            hostServidor,
                            PUERTO);
            // Enviamos el datagrama
            socketUDP.send(peticion);
            /**
             *  Construimos el DatagramPacket que
             *  contendrá la respuesta
             */
            byte[] bufer = new byte[1000];
            DatagramPacket respuesta = new DatagramPacket(
                    bufer,
                    bufer.length);
            socketUDP.receive(respuesta);
            /**
             *  Enviamos la respuesta del servidor
             *  a la salida estandar
             */
            System.out.println("Respuesta: " +
             new String(respuesta.getData()));
            // Cerramos el socket
            socketUDP.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

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

Para ejecutar esta aplicación:

Primero deberemos ejecutar el servidor y luego los clientes.

Para ello, en el servidor, pulsamos botón derecho y seleccionamos “Run File”.

Y en el cliente, pulsamos botón derecho y seleccionamos “Run File”.

Tenemos que el servidor está escuchando en el puerto 6789.

Escucho el puerto 6789

Ahora podemos ejecutar un cliente de la misma forma. Botón derecho, “Run File”.

Respuesta: Hola cliente 21

En el servidor vemos que se ha recibido un datagram de esta dirección del puerto remoto este con un mensaje que es 21.

Escucho el puerto 6789
Datagrama recibido del host: /127.0.0.1 desde el puerto remoto: 60766 con el mensaje: 21

Y se ha devuelto la respuesta al cliente.

Respuesta: Hola cliente 21

Si volvemos a ejecutar otro cliente, pasará lo mismo. Solo que en este caso es 91.

en la consola del servidor tenemos:
Datagrama recibido del host: /127.0.0.1 desde el puerto remoto: 60767 con el mensaje: 91

Y en la consola del cliente tenemos:
Respuesta: Hola cliente 91

Y así con todos los clientes que queramos.

Usar Sockets TCP o UDP

Lo más recomendable será usar los **Sockets TCP**, ya que ofrecen **garantía de llegada** de los paquetes; además, los **envía y recibe en orden**, por lo que no deben realizarse operaciones de reestructuración de los mismos.

Será **mucho más rápido y seguro TCP** (aunque, por definición, sea más lento, con tan poca información a intercambiar, no va a existir apenas diferencia) que UDP.

La forma de comunicación, es muy simple. Una vez el cliente se conecte con el servidor, deberá enviar sus datos, el servidor lo recibirá y realizará la funcionalidad programada, devolviéndola al cliente mediante su flujo de salida correspondiente:

Pseudocódigo que resuelve el ejercicio práctico

```
// El servidor espera un cliente
cliente = obtenerCliente()
// Una vez obtenido el cliente obtiene sus flujos de comunicación
entrada = obtenerFlujoEntrada()
salida = obtenerFlujoSalida()
// Obtiene el número del DNI
numero = entrada.obtenerMensaje()
// Realizar funcionalidad
resultado = calcularResultado (dato)
// Envía el resultado al cliente
salida.enviarMensaje (dato)
```

Sincronizar dos aplicaciones que usen sockets

La sincronización de aplicaciones se hace, precisamente, **para** llevar a cabo un **cálculo del tiempo que cada aplicación cliente está ocupando** la aplicación servidor.

Una forma muy sencilla de realizar la implementación de la sincronización, tanto del cliente que se conecta como del servidor es través del **envío de mensajes vía red**.

Podemos hacer que el **cliente** envíe una **petición de conexión** al **servidor** y que este, al recibirla, le **envíe** al cliente, de vuelta, la **hora del sistema**, que será la **hora en la que se han conectado** uno al otro y que, por lo tanto, se han **sincronizado**.

Una vez hecho esto, cuando el **cliente decida desconectarse**, el **servidor** podrá volver a obtener la **hora del sistema**, sabiendo así el tiempo que el cliente ha estado conectado a él.

Esto se podría realizar tanto mediante Sockets TCP o UDP.

Ejemplo sencillo de cómo podría llevarse a cabo la **sincronización entre cliente y servidor** mediante **pseudocódigo**:

```
// Cuando el cliente se conecte al servidor
        hora = cliente.obtenerHoraConexion ()
// Una vez obtenida la hora de conexión del cliente
// el servidor comprueba que la hora es correcta
        horaser = obtenerHoraSistema ()
        diferencia = horaser - hora
// Dejamos un pequeño margen por la actuación del servidor
        si diferencia < 0.05
// sincronizacion realizada
        sino
// enviar error al usuario
```

Eficiencia en las aplicaciones que usan sockects

Para comprobar si una aplicación es más eficiente que otra se pueden tener en cuenta varios factores, y los más importantes serían:

- Tiempo de respuesta de cada una de ellas: Ya que unas milésimas de diferencia pueden significar mucho tiempo cuando hay muchas peticiones. Para ello, habría que medir el tiempo que tarda cada una de ellas en dar el resultado esperado.
- Tráfico de red que genera cada una, siendo la **mejor** opción la que **menor tráfico** genere, ya que saturará menos la red.
- Espacio de memoria que ocupen durante su ejecución. Habría que analizar qué RAM necesita cada una durante su funcionamiento.
- Recursos de CPU requeridos: A **menor uso de CPU** durante la ejecución, **mejor rendimiento** presentarán.