MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN DE SERVICIOS Y PROCESOS
TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Los servicios en red. Sockets I

10

ÍNDICE

OBJETIVOS

Profundizar en el uso de la clase Socket

Conocer el uso de la clase DatagramPacket

Programar aplicaciones en red usando Sockets TCP

Programar aplicaciones en red usando Sockets UDP

/ 1. Introducción y contextualización práctica

En esta unidad, vamos a ver cómo podemos crear programas que se comuniquen entre sí mediante Sockets.

Vamos a ver las dos clases principales para crear aplicaciones, que se comunican mediante los dos protocolos de comunicación primordiales: TCP y UDP. Estas son las clases Sockets y DatagramPacket, respectivamente.

Seguidamente, vamos a ver cómo podemos crear una aplicación que realice comunicaciones en red con TCP, creando la parte del servidor y del cliente.

Por último, estudiaremos también cómo podemos crear una aplicación que realice comunicaciones en red con UDP, creando, igualmente, la parte del servidor y del cliente.

En este tema, puedes observar que están apareciendo muchos conceptos que ya hemos adelantado en unidades anteriores, pero que ahora vamos a poner en práctica.

Escucha el siguiente audio, en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

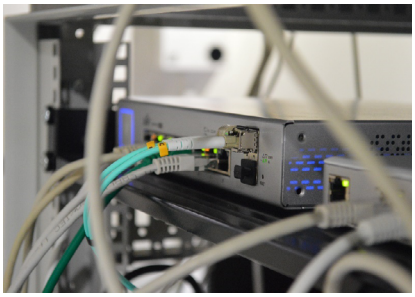

Fig. 1. Comunicaciones en red.

Audio Intro. "Analizando aplicaciones"
https://bit.ly/2YRIDH6

/ 2. Definición de Sockets

Definiremos los Sockets como 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*.

Audio 1. "Sockets en otros lenguajes de programación"
https://bit.ly/3bobZlq

/ 3. Programación de un servidor TCP

Para crear aplicaciones que usen *Sockets* en Java, tenemos que:

- **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.

En esta clase, vamos a crear una variable que indicará el puerto donde escuchará nuestro servidor, por ejemplo, el puerto 6000.

Para implementar el servidor, crearemos un objeto de la clase *ServerSocket* con el puerto deseado. 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.

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. Por último, deberemos de cerrar el Socket.

```
public ServidorTCP() {
    try {
        ServerSocket skServidor = new ServerSocket(PUERTO);

        // Escucho a los clientes
        while(Boolean.TRUE) {
            Socket skCliente = skServidor.accept();
            // 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");
            // Cierro el socket servidor
            skCliente.close();
        }
    } catch (IOException e) {
        System.out.println("Error en el servidor: " + e.getMessage());
    }
}
```

Fig. 2. Servidor TCP simple.

Puedes acceder a un proyecto completo de programación de un cliente y un servidor en el apartado «Recursos del tema».

/ 4. Caso práctico 1: "¿Usar Sockets TCP o UDP?"

Planteamiento: Pilar y José se disponen a realizar un nuevo ejercicio que les ha enviado su profesor. Este ejercicio consiste en realizar una aplicación cliente-servidor en la que el servidor ofrezca una funcionalidad muy simple: debe poder proporcionar al cliente su letra del DNI a partir del número de su DNI.

José se queda pensativo y le comenta a Pilar que recuerda haber hecho algo parecido en la asignatura de *Programación*, y que el cálculo de la letra era muy sencillo, pero lo que no tiene claro es cómo realizar la comunicación entre cliente y servidor, si con TCP o UDP.

Nudo: ¿Qué piensas al respecto? ¿Crees que es mejor utilizar los *Sockets* TCP o los *Sockets* UDP a la hora de revolver el ejercicio?

Desenlace: Una cosa muy normal a la hora de empezar a trabajar con elementos nuevos es decidir cuál de ellos usar, en el caso de que con ambos se pueda alcanzar el mismo resultado. En ese sentido, José está confundido porque se puede llegar al mismo objetivo utilizando *Sockets TCP* o *Sockets UDP*.

Ya conocemos las ventajas y desventajas de utilizar cada uno de los protocolos tras haberlos estudiando en unidades pasadas, por lo que, remitiéndonos a ellas, podremos extrapolar la información a este caso. Por lo tanto, podremos decidirnos con cierta facilidad.

Para empezar a trabajar con estos protocolos, 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. Podemos concluir, por tanto, que, para este caso tan simple, será mucho más rápido y seguro TCP (aunque, por definición, sea más lento, con tan poca información a intercambiar, no va a existir a penas diferencia) que UDP. La forma de comunicación, en este ejercicio, es muy simple. Una vez el cliente se conecte con el servidor, deberá enviar su número de DNI, el servidor lo recibirá y realizará el cálculo de la letra, devolviéndola al cliente mediante su flujo de salida correspondiente:

```
1  // El servidor espera un cliente
2  cliente = obtenerCliente()
3  // Una vez obtenido el cliente
4  // obtiene sus flujos de comunicación
5  entrada = obtenerFlujoEntrada()
6  salida = obtenerFlujoSalida()
7
8  // Obtiene el número del DNI
9  numero = entrada.obtenerMensaje()
10
11 // Calcula la letra
12 letra = calcularLetra(numero)
13
14 // Envía la letra al cliente
15
16 salida.enviarMensaje(letra)
```

Fig. 3. Pseudocódigo que resuelve el ejercicio práctico.

/ 5. Programación de un cliente TCP

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

- **Crear un *Socket* y conecta**r con el servidor mediante su IP y puerto.
- Crear los flujos de entrada o salida.
- 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.

En esta clase, 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*).

Para crear el cliente, crearemos un objeto de la clase Socket con el *host* y el puerto deseado.

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.

Por último, deberemos de cerrar el *Socket*.

```
private static final String HOST = "localhost";
private static final int PUERTO = 6000;

public ClienteTCP() {
    try {
        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);
        // Cierro el socket
        skCliente.close();
    } catch (IOException ex) {
        System.out.println("Error -> " + ex.toString());
    }
}
```

Fig. 4. Cliente TCP simple.

Vídeo 1. "Ejemplo de aplicación con Sockets TCP"
https://bit.ly/2QlHuwP

/ 6. 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:

Enlaces de interés...

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:

Array de bytes	Longitud	IP destino	Puerto destino

Tabla. 1. Datagrama.

La clase *DatagramPacket* tiene los siguientes métodos:

- **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.

La clase *DatagramSocket* tiene los siguientes métodos:

- **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.

/ 7. Programación de un 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.

```
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 petición del DatagramSocket
            socketUDP.receive(peticion);
            // Otengo 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());
    }
}
```

Fig. 5. Servidor UDP simple.

/ 8. Caso práctico 2: "Sincronizar dos aplicaciones"

Planteamiento: A nuestros amigos Pilar y José les acaba de llegar una nueva tarea. Esta tarea consiste en realizar una aplicación que dará un determinado servicio (que aún no conocen), a varios clientes; no obstante, antes de poder ofrecer este servicio, es un requisito indispensable que la aplicación pueda sincronizarse con cada uno de los clientes que se conecten a ella con el objetivo de llevar un registro del tiempo que cada cliente está ocupando el servidor.

«¿Sincronizar dos aplicaciones?, ¿cómo podemos hacer eso?», le pregunta Pilar a José, a lo que él le responde que no se le ocurre ninguna forma.

Nudo: ¿Crees que es posible sincronizar dichas aplicaciones? ¿Cómo piensas que puede realizarse esto?

Desenlace: Es muy normal que dos aplicaciones tengan que sincronizarse. Esto se hace, precisamente, para llevar a cabo un cálculo del tiempo que cada aplicación cliente está ocupando la aplicación servidor, tal y como se pide en la tarea.

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.

A continuación, puedes observar un ejemplo sencillo de cómo podría llevarse a cabo la sincronización 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
    // sincronización realizada
sino
    // enviar error al usuario
```

Fig. 6. Sincronización simple entre cliente y servidor.

/ 9. Programación de un 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 que indicará el puerto donde escuchará nuestro servidor; por ejemplo, el puerto 6789, y otra 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*.

```
public ClienteUDP() {
    try {
        String mensajeenviar = String.valueOf("Hola");
        // 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());
    }
}
```

Fig. 7. Cliente UDP simple.

/ 10. Resumen y resolución del caso práctico de la unidad

Vídeo 2. "Ejemplo de aplicación con Sockets UDP"
https://bit.ly/32IbEWN

A lo largo de esta unidad, hemos aprendido a crear programas que se comuniquen entre sí **mediante *Sockets*.**

En primer lugar, hemos estudiado cuáles son las dos clases principales para poder llegar a crear aplicaciones que se comuniquen mediante los dos protocolos principales de comunicación: TCP y UDP. Estas son las clases ***Sockets*** y ***DatagramPacket***, respectivamente.

Hemos aprendido, también, cómo podemos crear una aplicación que realice comunicaciones en red mediante el **protocolo TCP**, creando tanto la parte del servidor como la del cliente. De igual forma, hemos expuesto la forma de crear una aplicación que realice comunicaciones en red mediante el **protocolo UDP**, igualmente, implementando tanto la parte del servidor como la del cliente.

Resolución del caso práctico de la unidad

Sin duda, este es uno de los encargos más importantes que han recibido nuestros amigos. También es el que requiere mayor responsabilidad, ya que, de su análisis, va a resultar quedarse con una aplicación u otra. Puede pensarse que comprobar si una aplicación es más eficiente que otra es una tarea muy sencilla, pero no lo es en absoluto. Para comprobar esto, 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.

/ 11. Bibliografía

Colaboradores de Wikipedia. (2020, 14 junio). *Socket de Internet*. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Socket_de_Internet

Ejemplo conexión TCP cliente/servidor en Java | Disco Duro de Roer. (2018, 9 abril). discoduroderoer. https://www.discoduroderoer.es/ejemplo-conexion-tcp-clienteservidor-en-java/

Ejemplo conexión UDP cliente/servidor en Java | Disco Duro de Roer. (2018, 11 abril). discoduroderoer. https://www.discoduroderoer.es/ejemplo-conexion-udp-clienteservidor-en-java/

Gómez, O. (2016). *Programación de Servicios y Procesos Versión 2.1*. https://github.com/OscarMaestre/ServiciosProcesos/blob/master/_build/latex/ServiciosProcesos.pdf

Sánchez, J. M. y Campos, A. S. (2014). *Programación de servicios y procesos.* Alianza Editorial.