PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Los servicios en red
Clientes HTTP, FTP, Telnet y SMTP
Sockets con hilos

Programación de un cliente HTTP

La implementación del protocolo HTTP o HTTPS se basa en un proceso sencillo que implica una serie de **solicitudes y respuestas** a esas solicitudes por parte tanto del **cliente**, como del **servidor**:

- En primer lugar, un **cliente** establecerá una conexión con un servidor, enviando para ello un **mensaje de solicitud** con los datos pertinentes.
- Cuando el **servidor** reciba dicho mensaje, le **responderá con un mensaje** muy similar, conteniendo éste el resultado de la operación solicitada por el cliente.

El lenguaje de programación **Java** dispone de dos clases que nos van a permitir programar aplicaciones donde tengamos **tanto servidores como clientes HTTP**. Estas clases son:

- La clase URL: Esta clase nos va a permitir **representar una dirección** de una página web, por ejemplo, https://www.google.es, de forma que el programa pueda realizar operaciones con ella.

  Si quieres saber más sobre esta clase puedes consultar su documentación oficial en:

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

- La clase URLConnection: Esta clase es la que nos va a permitir **realizar operaciones con la dirección** web que hemos creado mediante URL.

  Podremos lanzar **operaciones tipo GET** y obtener las respuestas de éstas de una forma muy sencilla.

  Si quieres saber más sobre esta clase puedes consultar su documentación oficial en:

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

Al utilizar estas clases estamos programando un **servicio HTTP de alto nivel**, esto quiere decir que todas las operaciones que harán posible la comunicación no se visualizarán, y para trabajar con ello, será tan sencillo como **programar un servidor o cliente** mediante el uso de **sockets**.

Procesamiento de peticiones HTTP

Los servidores HTTP deben procesar de alguna forma las peticiones de sus clientes. Este tipo de servidores van a dar un servicio **según el método que utilicen los clientes HTTP** al invocarlos.

Lo habitual es tener un **método llamado GET**, en el que según un **parámetro**, se pueda **indicar qué servicio** de los ofrecidos por el servidor **queremos**.

Programación de un cliente FTP

Las siglas del protocolo FTP significan File Transfer Protocol, o Protocolo de Transferencia de Ficheros, y es el protocolo que debemos usar siempre que deseemos **transferir ficheros entre un servidor y un cliente** o viceversa.

Los **pasos para crear un cliente FTP** son los siguientes:

- Realizar una **conexión** al servidor.
- **Comprobar** que la **conexión** que se ha realizado con éxito.
- **Validar el usuario** FTP que se ha conectado. En caso de que el usuario no sea válido deberemos abortar la conexión y enviar un **mensaje de error**.
- Realizar las **operaciones** pertinentes **con el servidor**.
- **Desconectar del servidor** una vez terminemos de realizar las operaciones requeridas.

No hay que olvidar que todo el proceso de conexiones y realización de operaciones puede generar excepciones, concretamente puede lanzar las **excepciones:**

- **SocketException,**
- **IOException**.

El lenguaje de programación **Java no dispone de clases específicas para el uso del protocolo FTP**, pero la fundación Apache creó la API:

org.apache.commons.**net.ftp**

para **trabajar con clientes y servidores FTP**.

Esta API tiene las siguientes clases que nos permiten operar de una manera sencilla con el protocolo FTP:

- Clase FTP:

  Esta es la clase que nos va a proporcionar todas las **funcionalidades básicas** para poder realizar un cliente FTP básico. Esta clase **hereda de SocketClient**.

- Clase FTPReplay:

  Esta es la clase que nos va a permitir **operar con los valores devueltos** por las consultas FTP **del servidor**.

- Clase FTPClient:

  Esta clase es la encargada de dar **soporte a las funcionalidades del cliente** FTP. **Hereda de SocketClient**.

- Clase FTPClientConfig:

Esta clase nos va a permitir realizar las **configuraciones** oportunas de los clientes FTP de una forma sencilla.

• Clase FTPSClient: Esta clase nos va a permitir utilizar el **protocolo FTPS**, que es la versión segura del protocolo FTP. Esta clase utiliza el **protocolo SSL** y **hereda de FTPClient**.

Comprobaciones de seguridad en red

La seguridad en los servidores es algo totalmente necesario, ya que puede exponerse código o documentos confidenciales. Con cualquier tipo de servicio deberíamos de tener unas mínimas comprobaciones de seguridad.

Para una mínima seguridad en el sistema podríamos realizar las siguientes **comprobaciones una vez nos llegue una petición de un cliente**:

• Comprobar que el cliente es un **cliente válido** del sistema. Para esto deberemos mantener el **listado de clientes** que tenga el sistema en una pequeña base de datos o, en su defecto, en un fichero.

• Comprobar que la operación que nos pide el cliente es una **operación válida** en el sistema. Para esto basta con saber qué operaciones hemos diseñado en nuestro sistema y cuáles son sus parámetros.

• Comprobar que el cliente tiene **permisos** para realizar dicha operación. De igual forma que deberemos tener un **registro de clientes válidos** en el sistema, le podemos agregar una serie de datos indicando qué **operaciones pueden realizar**.

Pseudocódigo Pasos a seguir: comprobaciones en petición de cliente

```
// Comprobamos que el cliente es valido
c_valido = comprobarCliente (nombre)
// Comprobamos que la operación solicitada
// es valida
op_valida = comprobarOperacion (operación)
// Comprobamos que el cliente tiene permiso para
// realizar dicha operación
permisos = comprobarPermisos(nombre, operación)
si c_valido = verdad Y op_valida = verdad Y permisos = verdad
        Realizamos la operación solicitada
sino
        Devolvemos un mensaje de error
```

Programación de un cliente Telnet

El protocolo Telnet, cuyo nombre significa TELecomunication NETwork, nos va a permitir acceder a otros equipos conectados a nuestra red, pudiendo realizar así una **administración de forma remota**, como si estuviéramos sentados frente al equipo que estamos administrando remotamente.

Gracias a este protocolo, se hace muy simple la administración de equipos que no tengan pantalla ni teclado, es decir, que sean simplemente un servidor que se arranca y lanza todas sus tareas y servicios automáticamente.

Bases del protocolo Telnet:

- Tiene el **esquema** básico del **protocolo cliente/servidor**.
- El **puerto** que utiliza es el **23**.
- Funciona mediante **comandos** en modo texto.

Aunque sea un protocolo que nos ayudará y simplificará la administración de equipos, **no ofrece mucha seguridad**, y es por esta misma razón por la que apenas se utiliza en las grandes empresas. Esto se debe a que toda la **información se transmite en texto plano**, incluidos los datos y contraseñas de los usuarios, cuando toda la información debería intercambiarse cifrada.

La biblioteca que Apache nos ofrece en su paquete org.apache.commons.net.telnet para poder trabajar con este protocolo es:

Clase **TelnetClient**: Esta es la clase que nos va a permitir **implementar un terminal** para usar el protocolo Telnet. Esta clase **hereda de la clase SocketClient**.

Entre los métodos más útiles de esta clase tenemos:

- SocketClient.connect(), que nos permitirá realizar una **conexión** al servidor.
- TelnetClient.getInputStream() y TelnetClient.getOutputStream(), que nos permitirán obtener los **flujos de comunicación**.
- TelnetClient.disconnect(), que nos permitirá desconectar.

Programación de un cliente SMTP

El protocolo SMTP es el que se utiliza para **enviar y recibir correos electrónicos**.

Para poder implementar un cliente SMTP vamos a utilizar la API javax.mail que nos proporciona el lenguaje de programación Java.

Dentro de esta API tenemos las siguientes **clases** que nos van a permitir realizar una gestión de correos electrónicos de una manera fácil y sencilla:

- Session:

  Esta clase representa una **sesión de usuario** para correo electrónico. Aquí vamos a tener agrupada **toda la configuración por defecto** que utiliza la API javax.mail para la gestión de correos electrónicos. Mediante el método getDefaultInstance() podremos obtener **una sesión por defecto**, con toda su configuración correspondiente.

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://docs.oracle.com/javaee/7/api/javax/jms/Session.html.

  https://docs.oracle.com/javaee/6/api/javax/mail/Session.html.

- Message:

  Esta clase representa un **mensaje de correo electrónico**. Podremos configurar:

  - **hacia quién va dirigido** mediante el método setFrom(), (en realidad es quién lo envía, es un error del temario...).
  - el **asunto** del correo electrónico, mediante el método setSubject(),
  - el **texto** del mismo, mediante el método setText().

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://javaee.github.io/javamail/docs/api/javax/mail/Message.html.

- Transport:

  Esta clase es la que representa el envío de los correos electrónicos. **Hereda de la clase Service**, que es una clase que proporciona las **funcionalidades comunes** a todos los servicios de **mensajería**.

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://docs.oracle.com/javaee/7/api/javax/mail/Transport.html.

Mediante las clases anteriores podemos crear fácilmente una **aplicación que envíe correos electrónicos** mediante nuestra cuenta de Gmail, por ejemplo.

Sockets e hilos I

Lo ideal es que cada servidor pueda **atender a muchos clientes** al mismo tiempo, haciendo así una **implementación concurrente de los servicios**.

La concurrencia la conseguimos utilizando hebras o hilos, haciendo así que se lance una hebra por cada tarea concurrente que queramos realizar, dando la sensación de que se estaban ejecutando todas al mismo tiempo.

A la hora de programar servicios con sockets también vamos a hacer uso de las hebras, haciendo que **cada servidor pueda atender a varios clientes a la vez**.

Cuando el servidor detecte que un cliente le ha hecho una petición éste deberá aceptarla, procesarla y crear una hebra que sea capaz de atender la petición del cliente.

De esta forma, cada cliente será atendido por una hebra diferente y el servidor podrá volver a escuchar una petición nueva de otro cliente, inmediatamente después de lanzar la hebra que atenderá al primer cliente.

De esto debemos deducir que las **hebras se ejecutarán en la parte del servidor**, ya que es este el que atiende la petición del cliente, de forma que el cliente no será consciente de si lo está atendiendo el propio servidor o una hebra en el mismo.

Boceto de utilización de hilos con sockets

```
// Crear socket
while (Boolean.TRUE) {

      /**
       * 1. Aceptar una solicitud de cliente
       * 2. Código
       * 3. Crear una hebra independiente en el servidor
       *    para procesar la solicitud
       */
}
```

Ejemplo de Monitorización de tiempos de respuesta

Para monitorizar el tiempo de respuesta del servidor:
Se debe tener en cuenta el tiempo de proceso del servidor **y el tiempo de transmisión**.

```
public class Prueba {

	/**
	 * Esta función calcula la diferencia en segundos entre dos fechas
	 *
	 * @param tiempoinicio Tiempo de inicio
	 * @param tiempofin    Tiempo final
	 * @return Diferencia en segundos entre tiemponicio y tiempofinal
	 */
	private static float diferenciaSegundosTiempo(
			Date tiempoinicio, Date tiempofin) {
		// Calculamos la diferencia de las fechas en segundos
		float segundos =
				(float) ((tiempofin.getTime() / 1000)
						- (tiempoinicio.getTime() / 1000));

		if (segundos > 60) {
			segundos = segundos / 60;
		}

		return segundos;
	}

	/**
	 * @param args the command line arguments
	 */
	public static void main(String[] args) {
		try {
			// Obtenemos la fecha inicial
			Date inicio = Calendar.getInstance().getTime();

			// Dormimos el programa 2 segundos
			Thread.sleep(2000);

			// Obtenemos la fecha final
			Date fin = Calendar.getInstance().getTime();

			// Mostramos la diferencia en segundos entre las dos fechas
			System.out.println("La diferencia en segundos es: "
```

```
                + diferenciaSegundosTiempo(inicio, fin));
        } catch (InterruptedException ex) {
                Logger.getLogger(
                        Prueba.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Vamos a ver cómo podemos monitorizar el tiempo de respuesta de una serie de procesos u operaciones a los que queramos medir lo que tardan en ejecutarse:

- En primer lugar, deberemos realizar las siguientes operaciones que son tres:
  1. La primera será obtener la fecha antes de ejecutar las tareas que queremos monitorizar en tiempo. Para ello, podemos utilizar la clase Date que mediante la clase Calendar y el método getInstance().getTime() nos permitirá obtener el momento de tiempo en el que se ejecuta.

```
// Obtenemos la fecha inicial
Date inicio = Calendar.getInstance().getTime();
```

  2. Una vez hecho esto, ejecutaremos todas las instrucciones u procesos que se necesitan para obtener la fecha antes de ejecutar las tareas que queremos monitorizar en tiempo. Una vez hecho esto, ejecutaremos todas las instrucciones u procesos que queramos monitorizar. En este caso, al ser un ejemplo sencillo, lo único que vamos a hacer es hacer que el **programa se duerma durante dos segundos** mediante el método Thread.sleep().

```
// Dormimos el programa 2 segundos
Thread.sleep(2000);
```

  3. Una vez que terminemos de ejecutar todas las operaciones u procesos que queremos monitorizar, deberemos volver a obtener el instante de tiempo que va a ser diferente al instante de tiempo en el que cogimos el inicio. Una vez hecho esto, ya tenemos el instante de tiempo de inicio y el instante de tiempo de fin.

```
// Obtenemos la fecha final
Date fin = Calendar.getInstance().getTime();
```

Hemos creado una función que se llama diferenciaSegundosTiempo que tiene como parámetros los dos tiempos, el de inicio y el del fin, y que nos va a devolver un float con los segundos de diferencia.

```
private static float diferenciaSegundosTiempo(
                  Date tiempoinicio, Date tiempofin) {
            // Calculamos la diferencia de las fechas en segundos
            float segundos =
                        (float) ((tiempofin.getTime() / 1000)
                                    - (tiempoinicio.getTime() / 1000));

            if (segundos > 60) {
                  segundos = segundos / 60;
            }

            return segundos;
      }
```

Teniendo nuestra función diferenciaSegundosTiempo, lo que haremos será después de obtener el tiempo de fin del proceso, obtener la diferencia en segundos y mostrarla por pantalla.

```
            // Mostramos la diferencia en segundos entre las dos fechas
            System.out.println("La diferencia en segundos es: "
            + diferenciaSegundosTiempo(inicio, fin));
```

Vamos a ejecutar este programa para ver cómo funciona.

```
La diferencia en segundos es: 2.0
```

Como podemos ver, la diferencia en segundo es de 2.0 ya que el programa duerme dos segundos.

También podríamos dar esta diferencia en milisegundos.

Ejemplo de servicio concurrente

Vamos a ver la forma de realizar un **servidor TCP** que una vez que acepte la solicitud con un número de DNI de un cliente, lance una **hebra que atienda la solicitud**.

La hebra que lancemos **se ejecutará en el servidor**, y gracias a esto podremos pasar los datos que necesitemos del cliente a la misma, ya que el servidor en sí los tiene.

Para resolver este problema deberemos:

1. crear **una clase que sea una hebra**, pudiendo hacerlo de los dos métodos posibles, heredando de Thread o implementando Runnable.
2. A esta clase le tendremos que implementar un **método** que proporcione el servicio deseado, siendo en este caso una función que **calcule y devuelva la letra del DNI**.
3. Dentro de los parámetros de la clase le vamos a pasar:
    1. el número del DNI del cliente,
    2. el flujo de salida del cliente,

    para que de esta forma podamos enviar la letra al cliente una vez calculada.

Si hemos seguido estos pasos podremos atender a tantos clientes como necesitemos de forma concurrente, aunque siempre **es muy recomendable** establecer un **tope máximo de clientes**, dependiendo esto del servicio que queramos dar y de las prestaciones de nuestro servidor.

Pseudocódigo del servidor concurrente

```
// Aceptamos el cliente
cliente = esperarCliente()
// Obtenemos el flujo de entrada y el número
entrada = cliente.obtenerFlujoEntrada()
numerodni = entrada.obtenerMensaje()
// Obtenemos el flujo de salida del cliente
salida = cliente.obtenerFlujoSalida()
// Creamos el objeto de la hebra y la lanzamos
hebraservidor = Hebra(numerodni, salida)
hebraservidor.lanzar()
```

Sockets e hilos

Vamos a ver cómo podemos crear un programa **cliente/servidor concurrente** con **TCP** mediante hilos.

Para esto deberemos crear las siguientes **clases**:

• Clase Servidor: Esta clase será la que va a **representar el servidor** de nuestra aplicación.

• Clase ServidorHebra: Esta clase será la que va a representar una **hebra que se lanzará en el servidor** y que será la encargada realmente de **dar servicio al cliente** conectado, haciendo posible la **concurrencia**.

• Clase Cliente: Esta clase será la que va a **representar un cliente** de nuestra aplicación. Se podrán ejecutar **tantos clientes como se quiera**.

➔ Dentro de la clase servidor deberemos crear un **método main**, en el que:

- crearemos un **ServerSocket** con el que esperaremos que **se conecte un cliente**.
- Una vez **conectado el cliente** a nuestro servidor, deberemos crear un **objeto de la clase ServidorHebra**, al que le pasaremos como mínimo, los **flujos de entrada y salida** del cliente, para poder así poder establecer una **comunicación con él**. Después de crear y lanzar la hebra, nuestro servidor **volverá a escuchar** para que se conecte otro cliente al que dar servicio.

➔ Dentro de la clase cliente deberemos crear un **método main**, en el que:

- creamos un **Socket** que conectaremos al servidor.
- Una vez hecho, nos **comunicamos con el servidor normalmente**.

➔ Dentro de la clase ServidorHebra deberemos implementar el **método run** y el método que ejecutará la hebra, implementando en dicho método toda la **funcionalidad que debe dar el servidor normalmente**, es decir, realizando una **comunicación normal con el cliente**, tanto para **recibir** como para **enviarle** información.

- Será dentro de la clase ServidorHebra donde implementemos **todas las funciones** que necesitemos para poder dar los **servicios** requeridos a los clientes.

Ejemplo de sockets con hilos paso a paso

Vamos a ver cómo podemos crear una **aplicación cliente-servidor** utilizando **Sockets TCP** pero **concurrente**, es decir, que un servidor va a poder dar servicio a más de un cliente simultáneamente. Para ello necesitamos crear mínimo tres clases:

- la clase Cliente,
- la clase Servidor,
- la clase ServidorHebra.

La **clase cliente** será la que va a representar un cliente de nuestra aplicación y tendrá un **método main** para poder ejecutarse que hará lo siguiente:

1. Lo primero será configurar nuestro cliente. Para ello, nos declaramos:
   a) la **dirección IP** que será localhost ya que el ejercicio se ejecutará en nuestra propia máquina,
   b) el **puerto** al que nos vamos a conectar.
   c) Obtenemos la **IP real**,
   d) creamos un **socket** mediante la IP y el puerto.

   Aquí nuestro cliente se conectará a nuestro servidor y daremos el mensaje de que la conexión se ha establecido.

```
String direccionIP = "localhost";
int puerto = 5056;
// Optengo la IP real
InetAddress ip = InetAddress.getByName(direccionIP);
System.out.println("CLIENTE: Conectando con "
+ direccionIP + ":" + puerto + "...");
// Establezco la conexión con la IP y el puerto
Socket socket = new Socket(ip, puerto);
System.out.println("CLIENTE: Conexión establecida.");
```

2. Ahora obtenemos los flujos tanto de **entrada** como de **salida** para poder realizar una conversación con nuestro servidor:

```
// Obtengo los flujos de entrada y salida
DataInputStream entrada =
new DataInputStream(socket.getInputStream());
DataOutputStream salida =
new DataOutputStream(socket.getOutputStream());
```

y realizamos un bucle que nos va a permitir realizar dos tareas:

a) Si escribimos un 1, el servidor nos enviará un número aleatorio,

b) si escribimos un 2, saldremos.

```
// Este es el bucle que va a permitir la comunicación entre el
// cliente y el cliente hebra
boolean salir = false;
Scanner teclado_String = new Scanner(System.in);
while (!salir)
{
    // Imprimo el mensaje del cliente
    System.out.println(entrada.readUTF());
    // Leo la respuesta y la envío
    // (Si quiero número aleatorio o salir)
    String textoenviar = teclado_String.nextLine();
    salida.writeUTF(textoenviar);

    // Según lo que le haya enviado al servidor...
    switch(textoenviar)
    {
        case "1": // Opción mostrar número aleatorio
            // Imprimo el mensaje del número aleatorio
            // del cliente
            String mensajerecibido = entrada.readUTF();
            System.out.println(mensajerecibido);
            break;
        case "2": // Opción salir
            System.out.println(""
                    + "CLIENTE: Cerrando la conexión...");
            socket.close();
            System.out.println(""
                    + "CLIENTE: Conexión cerrada.");
            salir = true;
            break;
        default:
            System.out.println(""
                    + "CLIENTE: Opción incorrecta.");
            break;
    }
}
```

Antes de seguir con la clase cliente, vamos a ver la **clase servidor** que será la que represente un servidor de nuestra aplicación:

En ella tendremos un **método main** que hará lo siguiente:

Nos creamos un ServerSocket para crear nuestro servidor en el mismo puerto al que se conectan los clientes.

```
ServerSocket socketservidor = new ServerSocket(5056);
System.out.println(""
+ "SERVIDOR: Escuchando en localhost:5056...");
```

Con un bucle infinito realizamos la funcionalidad de nuestro servidor que será la siguiente:

Esperaremos que se nos conecte un cliente y una vez conectado y aceptado obtendremos sus flujos de entrada y salida para poder establecer una conversación con él.

```
// Espero a que se conecte un nuevo cliente
socketcliente = socketservidor.accept();
System.out.println(""
        + "SERVIDOR: Cliente nuevo conectado: "
        + socketcliente);
// Obtengo los flujos de entrada y salida del socket cliente
DataInputStream entrada =
        new DataInputStream(
                socketcliente.getInputStream());
DataOutputStream salida =
        new DataOutputStream(
                socketcliente.getOutputStream());
```

Una vez hecho esto, nos deberemos crear una hebra que atenderá a ese cliente y lanzarla. Para ello tenemos la clase ServidorHebra.

Aquí podemos ver que nos hemos creado un **objeto ClienteHebra del tipo thread** que es del tipo servidor hebra al que le pasamos el nombre de nuestro servidor. Y una vez lanzado nuestro servidor, se despreocupa y vuelve a escuchar el siguiente cliente, siendo la hebra quien atiende al cliente.

```
System.out.println(""
        + "SERVIDOR: Creando una hebra nueva "
        + "para el cliente nuevo...");
```

```
            // Creo una hebra para el cliente que se ha conectado
            Thread clientehebra = new ServidorHebra(
                      socketcliente, entrada, salida);
              // Lanzo la hebra del cliente
              clientehebra.start();
              System.out.println(""
                      + "SERVIDOR: Hebra del cliente nuevo "
                      + "creada (yo ya me despreocupo).");
```

La clase **servidor hebra** hereda de Thread y tiene como datos el socket del cliente y los flujos de entrada y salida. Aquí podemos ver el **constructor** de nuestra clase:

```
public class ServidorHebra extends Thread {

private DataInputStream entrada;
private DataOutputStream salida;
private Socket socketcliente;

public ServidorHebra(
        Socket socketcliente,
        DataInputStream entrada,
        DataOutputStream salida) {
    this.socketcliente = socketcliente;
    this.entrada = entrada;
    this.salida = salida;
}
```

Y el **método run** que será el que haga la funcionalidad.

Este método será el que **hable verdaderamente con el cliente** y hace lo siguiente:

Lo primero que hará será mandarle un mensaje a nuestro cliente preguntado qué quiere hacer: si generar un número aleatorio o salir. En este momento se ejecutará esta orden de la clase Cliente:

```
salida.writeUTF(textoenviar);
```

y se mostrará ese mensaje pidiendo al cliente una opción: si uno o dos y enviándola a el servidor. En este caso realmente se envía a la hebra.

Una vez que ha llegado, se ejecuta esta instrucción de la clase ServidorHebra:

```
// Recibo la respuesta del cliente
mensajerecibido = entrada.readUTF();
```

y se hará una distinción, en el caso de que sea uno o de que sea dos.

En el caso de que la instrucción **sea uno**, generamos un número aleatorio y lo enviamos a el cliente.

```
// Según el mensaje recibido...
switch (mensajerecibido) {
    case "1":
        int aleatorio = generador.nextInt(500);
        salida.writeUTF(""
              + "SERVIDOR: El número aleatorio generado es "
              + aleatorio);
        break;
    case "2":
        System.out.println(""
                  + "SERVIDOR: El cliente "
                  + this.socketcliente + " envía salir...");
        System.out.println(""
                  + "SERVIDOR: Cerrando la conexión...");
        this.socketcliente.close();
        System.out.println(""
                  + "SERVIDOR: Conexión cerrada.");
        salir = false;
        break;
      }
```

En este caso, se ejecutará este trozo de aquí, de la clase Cliente, recibiendo el mensaje e imprimiéndolo por pantalla y volviéndonos a preguntar qué queremos hacer.

```
case "1": // Opción mostrar número aleatorio
          // Imprimo el mensaje del número aleatorio
          // del cliente
          String mensajerecibido = entrada.readUTF();
          System.out.println(mensajerecibido);
          break;
```

En el caso de que introduzcamos **un dos**, se enviará ese dos a nuestro servidor aquí (en la clase ServidorHebra):

```
// Recibo la respuesta del cliente
mensajerecibido = entrada.readUTF();
```

y se ejecutará el trozo correspondiente a salir, que lo que hace es cerrar todos los flujos (como vemos en el código de más arriba en case 2...).

En el **cliente**, se ejecutará la opción salir y también se cerrarán todos los flujos.

```
case "2": // Opción salir
    System.out.println(""
            + "CLIENTE: Cerrando la conexión...");
    socket.close();
    System.out.println(""
            + "CLIENTE: Conexión cerrada.");
    salir = true;
    break;
```

Vamos a ver cómo funciona nuestra aplicación:

En primer lugar, deberemos ejecutar el servidor:

```
SERVIDOR: Escuchando en localhost:5056…
```

El cual nos dice que está escuchando en el puerto 5056.

Ahora deberemos ejecutar un cliente.

```
SERVIDOR: Cliente nuevo conectado:
Socket[addr=/127.0.0.1,port=52364,localport=5056]
SERVIDOR: Creando una hebra nueva para el cliente nuevo...
SERVIDOR: Hebra del cliente nuevo creada (yo ya me despreocupo).
```

Aquí en el servidor se nos indica de que se ha creado un cliente nuevo y se ha lanzado la hebra.

Si volvemos a ejecutar otro cliente:

```
CLIENTE: Conectando con localhost:5056...
CLIENTE: Conexión establecida.
SERVIDOR: ¿Qué quieres hacer?
      1.- Generar número aleatorio.
      2.- Salir.
      (Esperando petición del cliente...)
```

los dos clientes se están ejecutando perfectamente y en el servidor se indica que se han lanzado dos hebras.

```
SERVIDOR: Escuchando en localhost:5056...
SERVIDOR: Cliente nuevo conectado:
Socket[addr=/127.0.0.1,port=52364,localport=5056]
SERVIDOR: Creando una hebra nueva para el cliente nuevo...
SERVIDOR: Hebra del cliente nuevo creada (yo ya me despreocupo).
SERVIDOR: Cliente nuevo conectado:
Socket[addr=/127.0.0.1,port=47260,localport=5056]
SERVIDOR: Creando una hebra nueva para el cliente nuevo...
SERVIDOR: Hebra del cliente nuevo creada (yo ya me despreocupo).
```

Si en el cliente 1 le damos la orden de generar un número aleatorio:

```
CLIENTE: Conexión establecida.
SERVIDOR: ¿Qué quieres hacer?
      1.- Generar número aleatorio.
      2.- Salir.
      (Esperando petición del cliente...)
1
SERVIDOR: El número aleatorio generado es 466
SERVIDOR: ¿Qué quieres hacer?
      1.- Generar número aleatorio.
      2.- Salir.
      (Esperando petición del cliente...)
```

nos dirá que el número es 377 y volverá a preguntar que queremos hacer.

Si le decimos que queremos salir:

```
1
SERVIDOR: El cliente
Socket[addr=/127.0.0.1,port=52364,localport=5056] envía salir...
SERVIDOR: Cerrando la conexión...
SERVIDOR: Conexión cerrada.
```

se cerrará nuestro cliente. El servidor nos dirá que se ha cerrado un cliente.

Pero el cliente lo seguirá ejecutando hasta que introduzcamos la instrucción de salir, que cerrará el cliente, se destruirá la hebra,

```
SERVIDOR: El cliente Socket[addr=/127.0.0.1,port=47260,localport=5056] envía salir...
```

```
SERVIDOR: Cerrando la conexión...
SERVIDOR: Conexión cerrada.
```

pero el servidor sigue funcionando a la espera de más clientes.

https://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html