Práctica 3:
Introducción a los *sockets* en Java

1. Trabajo previo

En esta práctica realizarás una primera toma de contacto con la interfaz de los *sockets* en Java. Como prerrequisito es necesario un conocimiento básico del funcionamiento de los *sockets* en general. Para ello, en PoliformaT tienes disponibles dos vídeos "***Vídeo 1: La interfaz de los sockets***" y "***Vídeo 2: Programación de clientes TCP en java***". Debes estudiarlos antes de la sesión de laboratorio correspondiente a esta práctica. Además, en PoliformaT también tienes disponible dos juegos de transparencias (llamados "***Transparencias Práctica 3"*** y "**Transparencias Tema 3 info adicional**") con información útil que también debes estudiar antes de la sesión de prácticas.

En las páginas siguientes se plantean una serie de ejercicios que, tras haber realizado el trabajo previo, te permitirán adquirir mayor destreza en la utilización de los *sockets* cliente TCP en Java.

2. Entorno de trabajo

Java está disponible para diferentes sistemas operativos y con diversos entornos de programación. La práctica se realizará en la máquina virtual Linux que se proporciona para el trabajo de laboratorio. Para el desarrollo de esta práctica puedes utilizar cualquiera de estos dos entornos de trabajo (el que a ti te resulte más cómodo):

a) El entorno de trabajo ***BlueJ*** (www.bluej.org), de la Universidad de Kent, y disponible tanto en Windows como en Linux y Mac OS.
b) Uso del compilador "**`javac`**" y del lanzador de la máquina virtual de java ("**`java`**"), junto con un editor de textos como "**`gedit`**", por ejemplo, en el que escribir el código de los programas.

Toda la información sobre las clases de Java puede encontrarse en la documentación on-line ofrecida por Oracle a través de su página Web (https://docs.oracle.com/en/java/javase/15/docs/api/). Por otra parte, existen múltiples tutoriales de Java en la red, como por ejemplo https://openlibra.com/es/book/programacion-basica-en-java.

3. Un primer cliente TCP

La clase Java fundamental para realizar operaciones TCP desde el extremo cliente es **`java.net.Socket`**[1]. A partir de aquí se hablará de ella como clase **`Socket`**. Esta clase posee varios constructores que permiten especificar el *host* destino y el número de puerto al que queremos conectarnos. El *host* puede especificarse como un objeto **`InetAddress`** (que corresponde a una dirección IP) o como un **`String`** que indica el nombre del *host*. El número de puerto siempre se indica como un valor **`int`** que puede variar desde 1 hasta 65.535. El constructor más habitual para la clase **`Socket`** es:

```
public Socket (String host, int port)
              throws UnknownHostException, IOException
```
[2]

Ten en cuenta que lo anterior es la declaración de la clase tal como aparece en la página web de Oracle, tu la utilizarás instanciando un objeto de esa clase en la forma habitual.

El constructor **`Socket`** anterior crea un *socket* TCP, asignándole una dirección IP y un número de puerto local entre los disponibles en el sistema, e intenta conectarlo al *host* y puerto remotos especificados como parámetros. Si la instanciación tiene éxito, se obtendrá un objeto **`Socket`** conectado con el destino. Para poder conectar necesitará conocer la dirección IP del destino. Habitualmente, esta dirección se obtiene invocando – de forma transparente al programador – el servicio de resolución de nombres de dominio (DNS), que se estudia en las sesiones de aula. También se admite la utilización de la dirección IP en formato *String*, como por ejemplo **`"158.42.180.62"`**.

Es importante cerrar las conexiones de forma consensuada. Para ello se emplea el método **`close()`** de la clase **`Socket`**. Una vez que un *socket* se ha cerrado, ya no está disponible para volverlo a utilizar (por ejemplo, no puede ser reconectado). En caso necesario, hay que volver a crear un nuevo *socket*. Si este *socket* tenía un flujo de datos asociado (como se verá más adelante) éste se cierra también. La conexión se eliminará definitivamente cuando ambos procesos, cliente y servidor, ejecuten el cierre.

---

[1] Se recomienda al principio del programa importar todo el paquete java (**`import java.net.*;`**).

[2] La clase **`IOException`** y sus clases derivadas como **`UnknownHostException`** requieren importar el paquete io (**`import java.io.*;`**).

El esqueleto de un cliente TCP muy simple podría ser el siguiente:

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

public class ClienteTCP {
   public static void main(String args[]) throws
   UnknownHostException, IOException {
      int puerto = 7;      // puerto donde escucha el servidor
      String servidor = "zoltar.redes.upv.es";
      Socket s=new Socket(servidor, puerto);
      // a continuación vendría el diálogo C/S
      s.close();     // cerramos el socket, finaliza la conexión
     }
 }
```

Ejercicio 1:

Escribe un programa en java, llamado "**ClienteTCP1**", que sea capaz de establecer una conexión TCP con el puerto **80** del servidor www.upv.es. Si lo consigue, deberá imprimir por pantalla el mensaje "¡Conectado!". A continuación, cerrará el *socket* y terminará el programa.

En el código anterior no se ha tenido en cuenta la posibilidad de que se produzcan errores que impidan el establecimiento de la conexión. Esto ocurrirá si el nombre del servidor o el número de puerto indicados son incorrectos.

Puede comprobarse sustituyendo en la creación del *socket* el puerto destino por **81**, que no ofrece ningún servicio en www.upv.es, y verificar que el programa aborta por errores. El mismo efecto se observa si el nombre de host no es correcto; por ejemplo, empleando como puerto destino el **80** y cambiando el nombre de host a wwww.upv.es.

Ejercicio 2:

Modifica el programa que has escrito en el ejercicio 1 para comprobar los errores que se indican a continuación. Puedes llamarle "**ClienteTCP2**".

Cambia el nombre del servidor a "wwww.upv.es" y ejecuta el nuevo programa. Observa el resultado obtenido.

Modifica ahora el nombre del servidor a "www.uv.es" (este nombre es correcto) y el número de puerto destino a 81. Compara el resultado con el del apartado anterior. ¿Se ha generado el mismo tipo de error?

Comparando ambos errores, puede apreciarse que la excepción generada no es la misma en ambos casos. Al invocar al constructor de **`Socket`** con un nombre de *host* como parámetro, si el nombre no puede resolverse, por ejemplo, porque el servidor DNS no está en funcionamiento, el constructor generará una excepción del tipo **`UnknownHostException`**. En el segundo caso hemos intentado conectar con un puerto, el 81, en el que el servidor no está ofreciendo ningún servicio. Además, otras razones que impidan la conexión del *socket* pueden lanzar también una excepción del tipo **`ConnectException`**. Entre las causas que pueden producir esta última excepción se encuentran que el puerto destino en el servidor no esté abierto, que existan problemas de encaminamiento en la red para alcanzar el destino o simplemente que el servidor especificado esté apagado. Las clases de las dos excepciones mencionadas descienden de la clase **`IOException`**.

En las aplicaciones reales es necesario contemplar estas posibilidades para evitar comportamientos indeseados. La forma habitual de gestionar estas excepciones consiste en la utilización de cláusulas **`try`** /**`catch`** que capturan las las excepciones que se producen y permiten indicar las medidas adecuadas.

Ejercicio 3:

Escribe un programa en java, llamado **"`ClienteTCP3`"**, que añada al programa del ejercicio 2 las cláusulas "**`try`** / **`catch`**" necesarias para detectar los errores anteriores. El programa debe visualizar un mensaje por pantalla indicando si la conexión se ha establecido o no. En caso de éxito debe mostrar el mensaje "¡Conectado de nuevo!", y en caso de fallo indicará el motivo mostrando el mensaje correspondiente: "Nombre de servidor desconocido" o "No es posible realizar conexión", dependiendo el tipo de excepción detectada.

Para verificar el correcto funcionamiento, repite con este programa las pruebas realizadas con el anterior, empleando en un caso el servidor "www.uv.es" y el puerto destino el **81** e intentando en el otro caso el acceso al servidor wwww.upv.es.

Otro constructor de la clase **`Socket`** emplea como parámetro la dirección IP mediante un objeto **`InetAddress`**, en lugar del nombre de dominio del servidor.

La clase **`InetAddress`** posee varios métodos estáticos que permiten crear objetos **`InetAddress`**, inicializándolos. Por ejemplo, el constructor

```
public static InetAddress InetAddress.getByName(String HostName)
```

permite invocar la resolución DNS sobre un nombre de dominio y almacenar la dirección IP obtenida. Este método se utiliza de la forma siguiente:

```
InetAddress dirIP = InetAddress.getByName("www.upv.es");
```

Tras la ejecución de esta inicialización, si todo ha ido bien, en **`dirIP`** estará almacenada la dirección IP de www.upv.es: **158.42.4.23**. Si la resolución del nombre falla se generará una excepción **`UnknownHostException`**.

En muchos casos los parámetros de entrada de un cliente son el servidor y número de puerto al cual conectarse. Tanto si se leen de la línea de órdenes mediante **`args[]`** como si son introducidos por el usuario por teclado, los parámetros de entrada en Java son, por defecto, de tipo **`String`**[3]. Dado que el puerto en el constructor **`Socket`** es un **`int`**, es necesario realizar una conversión de tipo. De este modo, si **`args[1]`** es el parámetro correspondiente al puerto, la conversión puede hacerse, entre otros métodos, mediante la instrucción:

```
int puerto = Integer.parseInt(args[1]);
```

4. Recepción de datos

El funcionamiento básico de los *sockets* emula a los ficheros en disco, por lo que la recepción consistirá en realizar operaciones de "lectura" sobre el *socket* y, de forma recíproca, la transmisión en operaciones de "escritura". Para leer los datos que se van recibiendo a través del *socket* utilizaremos un objeto del tipo **`InputStream`** (flujo de octetos de entrada) asociado al *socket*, lo que se ajusta bien a la filosofía TCP de transmisión orientada a flujo continuo de datos. Para obtener el **`InputStream`** correspondiente a un *socket* invocaremos el método **`getInputStream`** de la clase **`Socket`**.

`InputStream` permite operaciones de lectura de bytes aislados o vectores de bytes mediante el método **`read()`**, pero no resulta cómodo para los protocolos basados en texto, haciendo complicado leer los mensajes del servidor. Por ello, resulta más conveniente la utilización de algún método que permita leer un flujo de caracteres y, a ser posible, líneas de texto completas.

[3] Este comportamiento por defecto se puede cambiar, por ejemplo, asociando un **`Scanner`** al **`System.in`** y usando el método **`nextInt[]`**, tal y como se explica más adelante.

La clase **`Scanner`** del paquete **`util`** nos proporciona dicha funcionalidad, ofreciendo, entre otros, métodos como **`next()`** para leer el siguiente elemento de tipo *String*; la siguiente línea (*String*) mediante **`nextLine()`**, y los métodos **`nextInt()`** y **`nextFloat()`** para obtener, respectivamente, el siguiente entero o el siguiente número en coma flotante en un flujo de entrada. Recuerda que debes incluir al comienzo del programa la línea "**`import java.util.Scanner;`**" para poder utilizar la clase **`Scanner`**.

En nuestro caso, empleando el método **`nextLine()`** de la clase **`Scanner`** sobre un flujo de octetos de entrada (**`InputStream`**), podremos leer las respuestas del servidor como líneas de texto (**`String`**). Por ejemplo, para imprimir en pantalla una línea de texto recibida a través de **`"s"`**, un objeto de la clase **`Socket`** definido previamente, se emplearía el siguiente código:

```
Scanner lee=new Scanner(s.getInputStream());
System.out.println(lee.nextLine());
```

Ejercicio 4:

Escribe un programa en java, llamado **`"ClienteDayTime"`**, que conecte al puerto **13** del servidor "**zoltar.redes.upv.es**", lea la primera línea de texto que devuelve el servidor y la imprima por pantalla.

NOTA: Recuerda que para poder acceder a zoltar deberás conectarte a través de la VPN si estás fuera de la UPV.

Las funciones de recepción sobre **`Scanner`** son bloqueantes, es decir, tras invocarlas se detiene la ejecución del programa hasta que se recibe el dato correspondiente. Podrán solicitarse lecturas desde el **`Scanner`** mientras la comunicación siga establecida. Para facilitar esta comprobación, el método **`hasNext()`** de **`Scanner`** devuelve **`true`** cuando existe algo pendiente de recoger, **`false`** cuando no se esperan más datos – por ejemplo, porque el *socket*[4] asociado se ha cerrado en el otro extremo – o se bloquea esperando nuevos datos en otro caso. Gracias a este método es posible implementar bucles de recepción de datos que reciben hasta que el otro extremo cierra la conexión. Por ejemplo, suponiendo que **`lee`** es de clase **`Scanner`**, y que el otro extremo cerrará su socket cuando finalice la transmisión de datos, es posible mostrar todas las líneas recibidas hasta el fin de la conexión mediante el código siguiente:

```
while(lee.hasNext())
    System.out.println(lee.nextLine());
```

[4] El cierre del *socket* implica el cierre de la conexión TCP asociada.

5. Transmisión de datos

Siguiendo el modelo anterior, para transmitir información hay que "escribir" sobre el *socket*. Para ello, a semejanza de la recepción, el *socket* tiene asociado un flujo de la clase **`OutputStream`** que permite la transmisión de bytes mediante el método **`write()`**. Al igual que en el caso anterior, en las numerosas ocasiones en que se desean transmitir líneas de texto (por ejemplo, en la mayoría de los protocolos estándar de Internet) es más eficiente utilizar una clase que proporcione la posibilidad de transmitir texto y, de paso, cierta capacidad de almacenamiento (*buffering*). Por este motivo es habitual emplear un objeto de la clase **`java.io.PrintWriter`** conectado al flujo de salida del *socket*. Este objeto permite manejar adecuadamente conjuntos de caracteres, además de proporcionar la capacidad de almacenamiento deseada. Sobre objetos de la clase **`PrintWriter`** es posible emplear los métodos **`print`**, que envía el texto correspondiente, **`println`** que añade al final de la línea de texto un retorno de línea (según el formato predefinido), y **`printf`** que permite especificar como *String* una expresión de formato.

Un constructor de la clase **`PrintWriter`** muy utilizado es:

```
public PrintWriter(OutputStream out, boolean autoFlush)
```

Además, este constructor posee una ventaja importante como comprobaremos a continuación. Para ello el segundo parámetro del constructor debe invocarse con el valor **`true`**.

6. Vaciado del buffer TCP (*flush*)

Aunque las ventajas de emplear para la escritura una clase con almacenamiento son claras, también puede plantear algunos inconvenientes si uno no es cuidadoso, como vamos a ver en el ejercicio siguiente.

Ejercicio 5:

Escribe un programa en java, llamado **`"ClienteEco"`**, que conecte al puerto **7** (servicio de eco) del servidor "**zoltar.redes.upv.es**", transmita la línea de texto "¡¡Hola, Mundo!!", lea la primera línea de texto que devuelve el servidor y la imprima por pantalla, de forma similar al ejercicio 4.

Para la transmisión y recepción de líneas de texto emplea los métodos **`println`** y **`nextLine`** de las clases **`PrintWriter`** y **`Scanner`**, respectivamente. Con el objetivo de observar el cambio de comportamiento de la comunicación en función del

parámetro **`autoFlush`**, a la hora de crear el **`PrintWriter`** dejaremos este parámetro a **`false`.**

Aunque no funcione el programa, sigue leyendo…

Este cliente de **eco** debería enviar un mensaje correcto al servidor **eco** de **zoltar** y recibir su respuesta, pero no recibe nada. ¿Por qué? Porque él tampoco le envía nada al servidor. Para mejorar la eficiencia, el **`PrintWriter`** intenta llenar su *buffer* tanto como sea posible antes de enviar los datos, pero como el cliente no tiene más datos que enviar (de momento) su petición no llega a enviarse nunca.

La solución a este problema la da el método **`flush()`** de la clase **`PrintWriter`**. Este método fuerza a que se envíen los datos, aunque el *buffer* no esté aún lleno. En caso de duda acerca de si resulta o no necesario utilizarlo, es mejor invocarlo, ya que realizar un **`flush`** innecesario consume pocos recursos, pero no utilizarlo cuando se necesita puede provocar bloqueos en el programa.

Ejercicio 6:

Modifica el cliente **eco** anterior para que utilice el método **`flush()`** y comprueba que ahora funciona correctamente.

Podemos realizar el vaciado del *buffer* de forma automática al escribir en éste (sin tener que invocar el método **`flush`** explícitamente). Para ello necesitamos dos cosas:

- El constructor de la clase **`PrintWriter`** debe emplearse tal y como se ha mostrado antes, con un segundo parámetro adicional (**`autoFlush`**) a **`true`**.
- En la escritura, el final de la línea debe indicarse expresamente mediante el uso del método **`println()`**. Adicionalmente, el método **`println`** añade los caracteres de final de línea, por lo que no es necesario transmitirlos expresamente.
- Otra posibilidad para forzar el vaciado del buffer automáticamente es utilizar el método **`printf()`** pasándole como parámetro una línea terminada con los caracteres **`\r\n`**. ¡¡Ojo!! Esta posibilidad **no** funciona con el método **`print().`**

7. Delimitadores de línea

Con respecto a los delimitadores de los finales de línea, hay que tener en cuenta que cuando usamos el método **`println`**, se emplean como delimitadores los predefinidos del sistema. En Linux (y sistemas tipo Unix como Mac OS) corresponde con el carácter de nueva línea (**`\n`**, código ASCII 10). Sin embargo, la mayoría de estándares de aplicaciones de Internet cuyos protocolos utilizan mensajes de texto para el diálogo, especifican que los delimitadores de fin de línea deben ser los caracteres **`\r\n`** (**`\r`** es el retorno de carro, código ASCII 13). No obstante, como se ha comprobado, al usar **`println`**, el programa funciona correctamente. Muchos servidores aceptan generosamente peticiones cuyos finales de línea no se ajustan completamente al estándar. Sin embargo, un cliente que descuide este factor puede encontrarse con problemas de funcionamiento al interactuar con servidores que sigan escrupulosamente el estándar.

Comprobemos ahora los finales de línea que están enviando nuestros programas.

Ejercicio 7:

Escribe un programa en java, llamado **`"ClienteSMTP"`**, que conecte al puerto **25** del servidor "**smtp.upv.es**", lea la primera línea de texto que devuelve el servidor y la imprima por pantalla, de forma similar al ejercicio 4. Comprueba su funcionamiento, mostrando la primera línea de bienvenida del servidor SMTP.

A continuación añade el código necesario para enviar al servidor mediante **`println`** la petición **`"HELO upv.es"`**. Añade también el código necesario para recibir y mostrar la siguiente línea del servidor (si no recibes respuesta, considera si has tenido en cuenta el vaciado de buffers). Tras ello cierra la conexión.

Arranca el analizador wireshark y utiliza un filtro de captura "port 25" (protocolo SMTP). Ejecuta el programa en java que has escrito y analiza los paquetes capturados.

¿Qué caracteres de final de línea se transmiten por la red?

En nuestro código, es posible ajustarnos fácilmente al estándar definiendo los finales de línea como **`\r\n`** con la sentencia:

```
System.setProperty ("line.separator","\r\n");
```

Si utilizas el entorno de programación BlueJ, una vez modificado el separador permanece cambiado hasta que salgas del entorno (aunque comentes la instrucción del programa) o lo modifiques otra vez mediante la instrucción correspondiente.

Ejercicio 8:

Añade al comienzo del programa java escrito en el ejercicio anterior la sentencia **`System.setProperty`** que acabamos de mostrar.

Ejecuta de nuevo el programa y repite la captura con el analizador wireshark y el filtro de captura "port 25" (protocolo SMTP). Comprueba qué caracteres de final de línea se transmiten ahora por la red. ¿Son los mismos que antes?

8. Cómo obtener información sobre la conexión establecida

La clase **`Socket`** dispone de varios métodos que permiten obtener información sobre la conexión establecida entre el cliente y el servidor. En el caso de nuestros programas la información remota, puerto y dirección IP, será la del servidor al que nos hemos conectado.Los métodos de la clase **`Socket`** más habituales para obtener información de la conexión son los siguientes:

- **`public int getPort()`**: devuelve el puerto remoto al que el *socket* está conectado. Coincide con el puerto destino proporcionado al construir el *socket*.
- **`public InetAddress getInetAddress()`**: devuelve la dirección IP remota a la que el *socket* está conectado. Corresponde al servidor proporcionado (por nombre o en forma de **`InetAddress`**) como parámetro en el constructor.
- **`public int getLocalPort()`**: devuelve el puerto local al que el socket está ligado. En los clientes, habitualmente es seleccionado por el sistema operativo. Aunque existen constructores de la clase **`Socket`** que permiten fijar el puerto local, raramente se utilizan.
- **`public InetAddress getLocalAddress()`**: Devuelve la dirección IP local a la que el *socket* está ligado. Un *host* dispone al menos de una dirección IP, pero puede disponer de varias si dispone de varios adaptadores de red (tarjetas reales, adaptadores USB y WiFi, o incluso adaptadores virtuales) de los cuales el sistema operativo generalmente seleccionará uno. También existen constructores que permiten especificarlo.

Ejercicio 9:

Modifica el cliente **`"ClienteTCP3"`** del ejercicio 3 para que muestre información en la pantalla del cliente sobre la conexión que establece (direcciones IP y números de puerto locales y remotos). Ejecútalo cuatro veces seguidas,

conectándote con el servidor www.upv.es en el puerto **80** y comprueba qué valores se modifican. Interpreta el resultado obtenido.

9. Otro ejemplo real

Como síntesis de todos estos conceptos, realizaremos un cliente HTTP 1.0 básico:

Ejercicio 10:

Escribe un programa en java, llamado, "**`ClienteHTTP`**", que conecte al puerto **80** del servidor www.upv.es, envíe la petición "**`GET / HTTP/1.0\r\n\r\n`**" y reciba todas las líneas devueltas por el servidor hasta el cierre de la conexión.