Práctica 2 – Cliente Servidor (2 sesiones, 1.5 puntos)

1.1 Objetivo

El objetivo de esta práctica es introducir al alumno en los conceptos básicos para el desarrollo de aplicaciones cliente-servidor iterativas y concurrentes, que usen *sockets* TCP y *sockets* UDP.

1.2 Información básica para la realización de la práctica

En esta sección se ofrece la información básica y las referencias necesarias para llevar a cabo las tareas que se proponen en la práctica.

1.2.1 Acceso al sistema y elección del sistema operativo

Para la realización de esta práctica no es necesario el uso de ningún sistema operativo en particular, ya que se realizará en Java, que es un lenguaje de programación multiplataforma. No obstante, recomendamos el uso de "Ubuntu 12.04" instalado en el laboratorio.

En la imagen "Ubuntu 12.04" no se disponen de privilegios de administrador, luego no es posible instalar nuevos programas. No obstante, se disponen de las herramientas básicas para el desarrollo de aplicaciones como son Netbeans y el *runtime* Java.

2 Paradigma cliente/servidor

La programación de aplicaciones para Internet se basa en el paradigma cliente/servidor. El funcionamiento general de este tipo de aplicaciones se puede resumir en que un proceso servidor recibe las peticiones de los clientes, devolviéndoles una respuesta con el procesamiento que se le ha solicitado. El tipo de interacción entre el cliente y el servidor puede ser orientada a conexión (que se corresponde con el protocolo TCP) o no orientada a conexión (al que pertenece el protocolo UDP).

Adicionalmente, los servidores pueden clasificarse como servidores *iterativos* o servidores *concurrentes*, atendiendo al número de conexiones simultáneas que pueden servir.

2.1 Servidores iterativos

Este tipo de servidores sólo pueden atender a un cliente a la vez. Se encolarán los clientes que realicen peticiones mientras el servidor iterativo está dando servicio a otro cliente.

2.2 Servidores concurrentes

Este tipo de servidores puede dar servicio a varios clientes a la vez. Para ello, suele lanzarse un proceso o una hebra por cada cliente, para procesar concurrentemente las peticiones.
En Java se pueden utilizar las hebras para dotar de concurrencia a un servidor. La estructura básica de una clase que extienda a una hebra es la siguiente:

```
// Ejemplo de hebra para servidores concurrentes: Hebrita.java
//
// La clase creada debe extender "Thread".
public class Hebrita extends Thread{

  // En este ejemplo almacenamos el número
  // de ejecuciones del bucle principal.
  private int nVueltas;

  // En el constructor podemos pasarle alguna variable que hayamos
  // creado en otra clase. Así podemos compartir algunos datos.
  Hebrita(int nEjecuciones){
    this.nVueltas=nEjecuciones;
  }

  // El contenido de este método se ejecutará tras llamar al
  // método "start()". Se trata del procesamiento de la hebra.
  public void run() {

    for(int i=0;i<nVueltas;i++){
          System.out.println("Voy por la vuelta "+i);

          // Indicamos a la hebra que pause para dejar a otras
          // hebras ejecutarse:
           Thread.yield();
    }
  }
}
```

Para lanzar la anterior hebra desde otra clase, se puede utilizar el siguiente fragmento de código:

```
...
Hebrita h=new Hebrita(22);
h.start();
...
```

 Las hebras pueden acceder concurrentemente a objetos compartidos. Para mantener la coherencia de esos datos es necesario utilizar mecanismos de exclusión mutua, como semáforos (clase *Semaphore*) o métodos sincronizados (indicando *synchronized* antes del método que sólo puede ejecutar una hebra).

3 Sockets

Los sockets son puntos de interconexión entre dos procesos que pueden ejecutarse en máquinas en una red IP, que se identifican mediante una dirección IP y un número de puerto.


El número de puerto está en el rango [0-65535]. En algunos sistemas operativos, como GNU/Linux o Unix, los puertos en el rango [0-1024] están reservados para ciertos servicios (e.g. Telnet, FTP, HTTP). Por este motivo para acceder a un puerto menor a 1024 son necesarios permisos de administrador. Si tiene curiosidad por saber la asociación entre puertos y servicios puede consultar el fichero `/etc/services` disponible en cualquier distribución GNU/Linux.

Los sockets permiten a un programador solicitar los servicios que ofrece la capa de transporte. Así, se puede utilizar sockets TCP y sockets UDP. A continuación se resumen las características y clases de Java relacionadas con cada uno de los tipos de sockets.

3.1 Sockets TCP

TCP ofrece un servicio de transmisión orientado a conexión, en el que se garantiza que los datos llegan en el orden en que fueron enviados, libres de errores, sin repetición, proporcionando de forma transparente control de flujo extremo a extremo y el control de errores.

Debido a que TCP está orientado a flujo de bytes, la filosofía de trabajo con los sockets TCP es la de leer y escribir conjuntos de bytes por una tubería (la conexión TCP). Concretamente, para una conexión TCP existe un flujo para lectura y otro para escritura. Pero antes de leer o escribir por los flujos del socket TCP, es necesario abrir e inicializar el socket. La forma de inicializar el servidor y el cliente difieren, como se verá en los siguientes apartados.

3.1.1 Socket TCP en el servidor

Un servidor que utilice un socket TCP debe realizar las siguientes acciones:

1. Abrir un `ServerSocket` en modo pasivo, pasándole el puerto en el que escuchar. De esta manera se inicializa el servidor para que se prepare a recibir peticiones por el puerto indicado. Este socket se abre sólo una vez, y sirve como punto de encuentro para recibir las peticiones de conexión. Ej. de ejecución:

```
ServerSocket socketServidor;
int puerto=888;
try {
  socketServidor = new ServerSocket(puerto);
} catch (IOException e) {
  System.out.println("Error: no se pudo atender en el puerto "+puerto);
}
```

2. Usar el método accept() para que se quede esperando una petición de conexión. Cuando esto suceda, el ServerSocket devuelve un objeto Socket que contiene los datos de la tubería correspondiente a la conexión entre cliente y servidor. A partir de ahora el nuevo socket es el que se utilizará para enviar y recibir bytes, tal y como se detalla en la sección del cliente TCP. Ej.:

```
Socket socketConexion = null;
try {
  socketConexion = socketServidor.accept();
} catch (IOException e) {
  System.out.println("Error: no se pudo aceptar la conexión solicitada");
}
```

3. El servidor puede volver a usar el método accept() para atender más peticiones de conexión.

3.1.2 Socket TCP en el cliente

Un cliente TCP debe realizar las siguientes pasos:

4. Establecer la conexión con el servidor. Para eso se usa la clase Socket. indicándole la dirección IP o nombre del servidor, y el puerto donde el servidor debe estar esperando las peticiones. Ej.:

```
Socket socket;
int puerto=8888;
String anfitrion=”ei150142.ugr.es”

try {
  socket=new Socket (anfitrion,puerto);
} catch (UnknownHostException e) {
  System.err.println(“Error: equipo desconocido”);
} catch (IOException e) {
  System.err.println(“Error: no se pudo establecer la conexión”);
}
```

5. Obtener los flujos de lectura y escritura del socket. Para ello se puede utilizar getInpuStream() y getOutPutStream(). Ej.:

```
InputStream inputStream = socketServicio.getInputStream();
OutputStream outputStream =
                    socketServicio.getOutputStream();
```

6. Para enviar datos, podemos utilizar el método write con el array de bytes que se desean enviar. Ej.:

```
byte [] buferEnvio="Hola caracola".getBytes();
outputStream.write(buferEnvio,0,buferEnvio.length);
```

7. Para recibir datos, podemos utilizar el método read, con el array de bytes en el que queremos que se lean los bytes recibidos. Ej.:

```
// Hay que reservar memoria para almacenar lo leído
byte []buferRecepcion=new byte[256];
// Intenta leer tantos bytes como posiciones tiene el
// array de bytes, aunque es posible que no haya
// tantos datos, y sólo se lean bytesLeidos:
int bytesLeidos = inputStream.read(buferRecepcion);
```

8. Finalmente, terminada la sesión de intercambio de bytes, se procede a cerrar el socket de forma ordenada, para no perder ninguno de los bytes que pudiera seguir en tránsito, mediante le método close().

```
socketServicio.close();
```

3.1.3 Lectura y escritura en modo texto

La JDK ofrece unas clases que permiten recubrir los flujos de bytes de entrada y de salida, interpretándolos y formateándolos como cadenas de caracteres. Es decir, en lugar de enviar arrays de bytes de un tamaño concreto, se pueden enviar y recibir objetos de la clase String, facilitando su uso para desarrollar protocolos en ASCII o de texto plano.

La única diferencia con los casos estudiados anteriormente aparece al leer y escribir, además de la creación de estos objetos de recubrimiento sobre los flujos del socket.

Para utilizar estas clases, es necesario seguir los siguientes pasos

9. Crear un objeto PrintWriter para enviar texto. Puede enviar cadenas de texto con los métodos: print o println. Ej.

```
PrintWriter outPrinter = new
  PrintWriter(socketServicio.getOutputStream(),true);
```

10. Crear un objeto BufferedReader para leer texto desde el socket. Se pueden recibir líneas de caracteres completas mediante el método readLine(). Ej.:

```
BufferedReader inReader = new
 BufferedReader(new
InputStreamReader(socketServicio.getInputStream()));
```

El segundo argumento del constructor PrintWriter sirve para indicar si ha de hacerse un vaciado automático del búfer de escritura o no. Esto significa que si no está activada esa opción, los datos se enviarán efectivamente cuando el sistema operativo lo crea conveniente, sin que el programa tenga control sobre cuándo se envían los datos realmente. Si está activada, esperará a un salto de línea o carácter especial en el búfer de salida, y cuando lo encuentra, inmediatamente envía los datos almacenados en el búfer.

3.2 Sockets UDP

Al contrario que TCP, UDP no garantiza ni el orden de llegada ni la ausencia de errores en los datagramas que encapsula. Este servicio no orientado a conexión introduce muy poco *overhead* (datos de las cabeceras con respecto a la cantidad de datos transportados).

La filosofía de los sockets UDP son totalmente distinta a las de TCP. Para UDP, al no existir el concepto de conexión ni flujo, cada datagrama debe contener, además de los datos a enviar, la información del destinatario. Así, un socket UDP simplemente será el punto para enviar o recibir datagramas UDP.

Para cada datagrama (clase DatagramPacket) se ha de reservar la memoria donde se va a recibir los datos, o donde residen los datos que se van a transmitir. En este caso, la diferencia de funcionamiento entre el servidor y cliente es mínimo.

3.2.1 Socket UDP en el servidor

El servidor debe seguir los siguientes pasos:

1. Abrir un DatagramSocket en modo pasivo, indicando el puerto donde quiere recibir los mensajes UDP. Ej.:

```
DatagramSocket socketServidor;
int puerto=888;
try {
  socketServidor = new DatagramSocket(puerto);
} catch (IOException e) {
  System.out.println("Error: no se pudo atender en el puerto "+puerto);
}
```

2. Leer o escribir un datagrama UDP, tal como se detalla en la sección del cliente.
3. Cerrar el socket al final. Ej.:

```
socketServidor.close();
```

3.2.2 Socket UDP en el cliente

El cliente sigue una estructura análoga al servidor, aunque durante la apertura no tiene por qué indicar ningún puerto de escucha:

1. Abrir un DatagramSocket.
2. Para enviar un mensaje, se debe reservar un array de bytes con los datos, buscar la dirección IP del servidor mediante InetAddress.getByName y el nombre del servidor, y crear un DatagramPacket con todos los datos, para enviarlo por el DatagramSocket creado previamente. Ej.:

```
int puerto=8888;
InetAddress direccion;
DatagramPacket paquete;
byte[] bufer = new byte[256];
DatagramSocket socket;

socket = new DatagramSocket();

direccion =
  InetAddress.getByName(“ei150142.ugr.es”);

paquete =
   new DatagramPacket(bufer, bufer.length, direccion,
                                          puerto);
socket.send(paquete);
```

3. Para recibir un mensaje se debe reservar un array de bytes donde se almacenarán los datos recibidos, y crear un DatagramPacket indicando cuántos bytes se van a leer del total recibidos. Para saber a qué dirección enviar la respuesta, es necesario obtener la dirección y el puerto del remitente del mensaje.

```
paquete = new DatagramPacket(bufer, bufer.length);
```

```
socket.receive(paquete);
paquete.getData();
paquete.getAddress();
paquete.getPort();
```

4. Como en el caso del servidor, al final hay que cerrar el socket.

```
socket.close();
```

4 Realización de la práctica

1. Rellene los huecos de los ficheros <servicio>ServidorIterativo.java y <servicio>Cliente.java para que funcionen como cliente/servidor TCP iterativo.
2. Modifique el cliente y servidor anteriores para utilicen las clases PrintWriter y BufferedReader.
3. Modifique el servidor anterior para que funcionen como servidor TCP concurrente.
4. Modifique <servicio>ServidorIterativo.java y <servicio>Cliente.java para que usen datagramas UDP.
5. Cree una aplicación cliente/servidor TCP concurrente que implemente el juego de adivinar números: el servidor inicialmente pensará en un número aleatorio, y el usuario que lanza el cliente también. Por turnos deberán preguntarse si es mayor o menor que un número dado. El primero que acierte el número que ha pensado el contrincante, gana.

5 Referencias básicas

- **"Thinking in JAVA 3nd Edition**", Bruce Eckel, Prentice Hall 2002. (Accesible en formato electrónico en http://mindview.net/Books/DownloadSites/)
- **"Trail: Custom Networking"** del tutorial "The JAVATM Tutorial. A practical guide for programmers.", http://java.sun.com/docs/books/tutorial/networking/index.html.