Práctica 7: Programación de *sockets* UDP

En esta práctica vamos a introducir la programación en java de *sockets* UDP. Aunque UDP es un protocolo de transporte mucho menos complejo que TCP, su programación resulta ser, inicialmente, un poco más laboriosa y compleja que la de TCP que hemos realizado en las prácticas anteriores.

Como ya hemos visto en clase, con el protocolo UDP no se establece "conexión" entre el cliente y el servidor. Por este motivo, en la creación del socket UDP no se especifica el ordenador destino (dirección IP y puerto) sino que, al programar el envío de un datagrama UDP, hay que indicar la dirección IP y el puerto del destino cada vez que enviamos datos.

Además, los datos transmitidos pueden llegar fuera de orden o incluso perderse, sin que la aplicación receptora sea informada de estas eventualidades. Por este motivo, la propia aplicación debe comprobar la entrega de los datos si necesita fiabilidad. Como paralelismo con la vida diaria, podríamos decir que UDP tiene un funcionamiento similar al del correo postal. La "carta" (como el datagrama UDP) lleva el remite y la dirección del destinatario. Si el remitente no recibe una respuesta a su mensaje no puede saber si el mensaje se ha recibido o no, ya que puede haberse perdido el envío o la respuesta. Al mismo tiempo, si se envían varias cartas de forma secuencial, el correo postal no garantiza que se vayan a recibir en el mismo orden en que fueron enviadas. Esto mismo ocurre cuando se envían varios datagramas UDP consecutivos.

1. Lectura previa

Se recomienda leer previamente la Sección 2.8 del libro de Kurose, "Redes de Computadores" – 5ª edición, que explica la programación de *sockets* UDP en *java*. En la 7ª edición del mismo libro, la sección 2.7.1 explica la programación de *sockets* UDP en *python*. En la web de acompañamiento del libro se pueden encontrar todos los ejemplos de esa sección en *java*.

2. Objetivos de la práctica

Al acabar la práctica deberías ser capaz de escribir programas cliente y servidor básicos en Java que trabajen con *sockets* UDP y realicen servicios sencillos. En particular deberías saber:

1) Utilizar la clase **`InetAddress`** para almacenar una dirección IP.
2) Utilizar la clase **`DatagramSocket`** para crear *sockets* UDP tanto de clientes (asociados a un puerto elegido por el sistema operativo) como de servidores (asociados a un puerto concreto).
3) Utilizar los métodos **`send()`** y **`receive()`** de la clase **`DatagramSocket`** para enviar y recibir datagramas UDP, respectivamente.
4) Utilizar diferentes constructores de la clase **`DatagramPacket`** para crear datagramas adecuados para envío y recepción.
5) Utilizar los métodos **`getPort()`**, **`getData()`**, **`getLengh()`** y **`getAddress()`** de la clase **`DatagramPacket`** para obtener información almacenada en un objeto de tipo **`DatagramPacket`**.
6) Utilizar los métodos **`setPort()`**, **`setData()`**, **`setLengh()`** y **`setAddress()`** de la clase **`DatagramPacket`** para configurar el envío de información a un destino.
7) Generar una cadena de texto a partir de un vector de bytes para visualizar cómodamente la información recibida.

3. La clase InetAddress

Al trabajar sin conexión, una de las primeras tareas con las que se encuentra el cliente UDP es cómo especificar la dirección del destinatario. Esta dirección se tiene que incluir en cada uno de los datagramas UDP a enviar. Para este fin, en java suele utilizarse un objeto de la clase **`InetAddress`**, perteneciente al paquete *java.net*. Entre los métodos que ofrece esta clase, el método **`getByName()`** acepta como parámetro una cadena de texto que representa el nombre de un host o su dirección IP, y nos devuelve un objeto de tipo **`InetAddress`** que contiene la dirección IP asociada.

Veamos unos ejemplos de uso:

```
InetAddress ipServer = InetAddress.getByName("www.upv.es");
InetAddress ipServer = InetAddress.getByName(args[0]);
InetAddress ipServer = InetAddress.getByName("127.0.0.1");
```

Cuando el parámetro contiene una dirección IP, por ejemplo, "127.0.0.1", solo se comprueba que la dirección proporcionada tenga un formato válido. Por el contrario, si el argumento contiene un nombre de dominio como "*www.upv.es*", se intentará resolver el nombre, generando si es necesario una consulta al servidor DNS local. Si la resolución del nombre falla se generará la excepción **`UnknownHostException`**.

Observa que **`getByName()`** es un método estático, lo que significa que está disponible siempre directamente anteponiendo el nombre de la clase. No necesita que se

haya instanciado un objeto previamente. La particularidad de estos métodos es que aparece la palabra **`static`** en la definición del método (o atributo).

Otro método de la clase **`InetAddress`** que puede resultarnos útil es el método **`toString()`** que traduce la información del objeto **`InetAddress`** a una cadena de texto. La cadena que devuelve es de la forma: "<nombre de host>/<dirección IP>". Si no se tiene información sobre el nombre del host, la parte de nombre de host se muestra como una cadena vacía.

Ejercicio 1:

Escribe un programa en java, denominado **`dnslookup`**, que acepte como argumento de entrada el nombre de un host y visualice en pantalla el nombre del host y su dirección IP o un mensaje de error indicando que no se ha podido traducir el nombre. El programa utilizará el método **`getByName()`** de la clase **`InetAddress`** para traducir el nombre y el método **`toString()`** para visualizarla.

El programa se puede ejecutar tanto desde el entorno *BlueJ*, completando los argumentos del **main** (en este caso la cadena con el nombre del *host* a buscar) al ejecutar el código como desde una consola. En este último caso el uso del programa deberá ser: **`java dnslookup <hostname>`**

Ejecuta tu programa para averiguar las direcciones IP de "www.eltiempo.es" y del servidor web de la UPV, "www.upv.es".

4. La clase DatagramSocket

Esta clase permite crear *sockets* para el envío y la recepción de datagramas UDP. Los dos constructores que vamos a ver pueden generar excepciones del tipo **`SocketException`**, que es una subclase de las excepciones de entrada/salida, **`IOException`**:

- **`DatagramSocket() throws SocketException`**. Crea un *socket* UDP asociado a un puerto elegido por el sistema operativo de entre los que haya disponibles. Utilizaremos normalmente este constructor para crear los *sockets* de los clientes. Se generará una excepción **`SocketException`** si no se puede crear el *socket*, por ejemplo, porque no quedan puertos UDP libres en el sistema.
- **`DatagramSocket(int numeroPuerto) throws SocketException`.** Crea un *socket* asociado al puerto **`numeroPuerto`**. Utilizaremos este formato para los *sockets* de los servidores, ya que habitualmente nos interesa que escuchen en un número de puerto específico. Si se utiliza el argumento **`numeroPuerto=0`**, el comportamiento es equivalente a emplear el constructor **`DatagramSocket()`** sin parámetros.

La clase **`DatagramSocket`** dispone del método **`getLocalPort()`** que devuelve el número de puerto del host local al que el *socket* está ligado.

Ejemplo de uso:

```
DatagramSocket ds = new DatagramSocket();
int p = ds.getLocalPort();
```

Ejercicio 2:

Crea un **cliente** que instancie un **`DatagramSocket`** sin especificar el número de puerto asociado. Muestra por pantalla el número de puerto que le ha asignado el sistema operativo, y comprueba cómo cambia tras cada ejecución.

Una vez hemos creado el *socket* normalmente nos interesa utilizarlo para enviar y recibir datagramas. Para ello podemos utilizar los métodos **`send()`** y **`receive()`** de la clase **`DatagramSocket`**, respectivamente:

- **`send(DatagramPacket p) throws IOException`**
- **`receive(DatagramPacket p) throws IOException`**

Como vemos, ambos métodos utilizan como argumento un objeto de tipo **`DatagramPacket`** (esta clase se verá en el apartado siguiente). En el caso del método **`send()`** habrá que incluir en el datagrama los datos del mensaje, su longitud y la dirección IP y número de puerto del destinatario. El argumento de **`receive()`** es un objeto **`DatagramPacket`** en el cual se almacenará el mensaje recibido, su longitud y la dirección IP y número de puerto del emisor del datagrama. Tanto el método **`send()`** como el método **`receive()`** pueden generar una **`IOException`**.

El método **`send()`** retorna en cuanto pasa el datagrama al nivel inferior, responsable de trasmitirlo a su destino. Además, los datagramas que llegan a un socket son encolados en él hasta que se invoque su método **`receive()`.** Si se invoca el método **`receive()`** de un socket cuya cola de entrada se encuentra vacía, el proceso se bloquea indefinidamente hasta que se recibe un datagrama, a menos que se establezca un tiempo límite de escucha sobre el *socket*. Por lo tanto, para ejecutar otras tareas mientras se espera un datagrama mediante **`receive()`,** esas tareas deberían planificarse en un flujo de ejecución (*thread*) separado. La idea es similar a la que vimos en la práctica de los servidores concurrentes TCP en el ejercicio del *chat*.

En case de querer limitar el tiempo máximo de espera de un **`receive()`** para evitar que el proceso espere indefinidamente, se puede recurrir al método **`setSoTimeout`** de la clase **`DatagramSocket()`**:

```
setSoTimeout(int timeout) throws SocketTimeoutException
```

Este método establece el intervalo de espera máximo (expresado en milisegundos) que el método **`receive()`** se bloqueará esperando un paquete. Si transcurre el intervalo establecido (el argumento **`timeout`** de la declaración anterior) y no se ha recibido nada, se generará una excepción del tipo **`SocketTimeoutException`**. Como es lógico, es necesario establecer el tiempo antes de invocar el método **`receive()`**.

Ejemplo de uso:

```
DatagramSocket socket = new DatagramSocket(4444);
DatagramPacket p = new DatagramPacket(new byte[1024], 1024);
socket.setSoTimeout(3000); //tiempo de espera 3 s
// La ejecución se bloquea como máximo durante 3 s
socket.receive(p);
```

Por último, hay que tener en cuenta que el sistema operativo descartará los datagramas UDP que se reciban destinados a un puerto destino para el que no exista un *socket* UDP activo. En el caso particular de java, esto implica que debe haber un objeto **`DatagramSocket`** vinculado a ese puerto para que el datagrama sea encolado en lugar de descartado.

5. La clase DatagramPacket

Esta clase representa un datagrama UDP para transmitir o recibir. Como hemos indicado previamente, los objetos **`DatagramPacket`** contienen un campo de datos, una dirección IP y un número de puerto. Estos dos últimos valores corresponden al origen en los datagramas recibidos, y al destino en los que transmitimos. Es decir, cuando se envía un datagrama la dirección IP es la del destinatario y cuando se recibe un datagrama es la dirección del emisor.

Mensaje (Vector de bytes)	Longitud del mensaje	Dirección IP	Número de puerto

6. Envío de paquetes

A la hora de enviar un mensaje mediante un datagrama UDP hay que especificar, además de los datos a enviar y su longitud, la dirección IP y el número de puerto del destinatario. Estos valores pueden indicarse directamente al crear el datagrama mediante el constructor:

```
DatagramPacket (byte buf[ ], int longitud, InetAddress dirIP,
int puerto) throws SocketException
```

Una vez creado, podemos enviarlo pasando el **`DatagramPacket`** creado como argumento al método **`send()`** de un **`DatagramSocket`**.

El siguiente código genera un datagrama con el contenido "hola" destinado al puerto 7777 del ordenador donde se ejecute el programa (ya que el destino elegido es **`localhost`**):

```
String ms = new String("hola\n");
DatagramPacket dp = new DatagramPacket(ms.getBytes(),
ms.getBytes().length, InetAddress.getByName("localhost"),7777);
```

Ejercicio 3:

Escribe un programa **cliente** en java que envíe un datagrama UDP con tu nombre y apellidos al puerto 7777 de tu ordenador y termine. Recuerda que puedes identificar tu ordenador como "localhost". Para comprobar el funcionamiento del programa, abre un terminal y ejecuta la orden **`nc -k -u -l 7777`**, que establece un servidor UDP que escucha en el puerto 7777 de tu ordenador. Ejecuta tu cliente varias veces. Ahora haz lo mismo pero usando el comando **`nc -u -l 7777`**. Curiosamente, observarás que sólo se recibe el nombre la primera vez (justificamos este comportamiento más abajo).

Una curiosidad

¿Por qué no ha aparecido varias veces en la pantalla tu nombre al ejecutar el programa del ejercicio anterior? En realidad, el datagrama UDP que contenía el nombre se ha recibido, pero el programa **`nc`** lo ha descartado porque utiliza el método **connect()** de la clase **DatragramSocket**. Este método restringe la comunicación de un socket UDP a un único destino, lo que quiere decir que, tras ejecutar **connect()**, el *socket* sólo recibirá datagramas de la dirección IP y puerto especificados y descartará los que provengan de otro origen. Ten en cuenta que el método **`connect()`** es estrictamente una operación local, es decir, a diferencia de lo que ocurre en TCP, aquí la ejecución de **`connect()`** no produce ningún intercambio de paquetes con el otro extremo.

Teniendo en cuenta lo anterior, ¿qué hubiese sucedido si tu cliente UDP en lugar de enviar un único datagrama hubiese tenido un bucle para enviar ese mismo datagrama tres veces? Piénsalo unos instantes.

7. Recepción de paquetes

Para recibir datagramas UDP es suficiente crear un objeto **`DatagramPacket`** con un *buffer* asociado y después invocar el método **`receive()`** utilizando ese **`DatagramPacket`** como contenedor de los datos recibidos. Para crear un **`DatagramPacket`** podemos utilizar el constructor siguiente

`DatagramPacket(byte buf[ ], int longitud`)

Donde el argumento **`longitud`** debe ser menor o igual que **`buf.length`**. Este constructor crea un **`DatagramPacket`** para recibir datagramas como máximo de la longitud indicada. Cuando se recibe un mensaje se almacena en el **`DatagramPacket`** junto con su longitud, la dirección IP y el número de puerto del *socket* desde los que se ha enviado el datagrama.

Ejemplo de uso de **`DatagramPacket`** para recepción:

```
byte[] buffer = new byte[1000];
DatagramPacket p = new DatagramPacket(buffer, 1000);
DatagramaSocket ds = new DatagramSocket(7777);
ds.receive(p);
```

Extracción de información de un *DatagramPacket*

Los métodos **getPort()** y **getAddress()** permiten obtener el puerto y la dirección IP desde la que se ha enviado un datagrama:

- **`InetAddress getAddress()`**
- **`int getPort()`**

El método **`getData()`** devuelve el vector de *bytes* asociado al campo de datos (es decir, el mensaje). Esto incluye todos los bytes del *buffer*. Como el mismo **`DatagramPacket`** se puede reutilizar para la recepción de diversos datagramas, nos interesará obtener únicamente los bytes de datos del último datagrama recibido (es decir, sin los datos restantes que pudiera tener el buffer de recepciones previas). Para ello utilizaremos el método **`getLength()`**, que nos devolverá la longitud del datos del datagrama recibido. Además, muchas veces necesitaremos convertir el vector de bytes recibido en una cadena para poder imprimirla o compararla con otras. Existen muchas

maneras de realizar esa conversión. Una posibilidad sencilla pasa por el uso de uno de los constructores **`String`** de la siguiente manera:

```
String s = new String(p.getData(), 0, p.getLength());
```

Envío de respuestas

La respuesta a un datagrama recibido se simplifica notablemente si se reutiliza el mismo **`DatagramPacket`** que se ha recibido, ya que contiene la dirección IP y puerto del destinatario. Para ello, será suficiente reescribir los datos de respuesta mediante el método **`setData()`**. La longitud de los datos puede establecerse con el método **`setLength()`** de la misma clase.

- **`void setData(byte[] buf)`**
- **`void setLength(int length)`**

Ejemplo de uso:

```
ds.receive(dp);    //recibimos el DatagramPacket
String name = "hola\n";
dp.setData(name.getBytes());
dp.setLength(name.length());
ds.send(dp); //enviamos el DatagramPacket
```

Si se deseara reutilizar el **`DatagramPacket`**, por ejemplo, para reenviar los datos recibidos a un nuevo destino, es necesario indicar o actualizar esa nueva dirección destino en el **`DatagramPacket`**. Para ello se podría emplear el método **`setAddress()`**:

- **`void setAddress(InetAddress.getByName("nombre_destino"));`**

¿Qué tamaño de buffer elegir?

A pesar de que teóricamente el tamaño máximo de un datagrama UDP casi alcanza los 64KB (65.531 bytes), muchas implementaciones no soportan *buffers* de más de 8 KB (8.192 bytes). En caso de superar el tamaño máximo en una implementación concreta, los datos que no caben en el *buffer* se descartan (hay que tener precaución con esto, puesto que *Java* no avisa a la aplicación). Muchos protocolos como DNS o TFTP usan paquetes de 512 bytes o menos. El tamaño más grande en usos comunes es de 8.192 bytes para NFS (sistema de ficheros de red: *Network File System*). Para nuestros ejercicios 512 bytes de *buffer* suelen resultar suficientes.

Ejercicio 4:

Crea un programa **cliente** UDP que envíe un datagrama al puerto 7777 de tu ordenador y luego espere una respuesta, imprimiendo por pantalla el contenido del datagrama recibido. Para probarlo lanza un servidor UDP en tu ordenador que escuche en el puerto 7777, mediante la orden "**`nc -u -l 7777`**" (recuerda que debes ejecutar la orden desde un terminal). Una vez hayas recibido el datagrama del cliente, para generar la respuesta del servidor, desde la ventana del terminal teclea una línea de texto cualquiera y pulsa `Intro`.

Ejercicio 5:

Crea un **servidor** UDP de *daytime* que escuche en el puerto 7777, y devuelva un datagrama con la fecha y hora como respuesta a la recepción de cualquier datagrama, sin importar su contenido. Pruébalo utilizando el programa **`nc`** como cliente, utilizando la orden "**`nc -u localhost 7777`**". Los datos del datagrama que envíe el cliente no son relevantes, pero habrá que introducir al menos un retorno de carro (Intro) para que el datagrama se transmita y el servidor pueda contestar.

En java puedes obtener la fecha y la hora con las siguientes instrucciones:

```
Date now = new Date();
String now_string = now.toString() + "\r\n";
```

NOTA: la clase **`Date`** pertenece al paquete java.util. Necesitarás importarla en tu programa.

8. Ampliación opcional

Ejercicio 6:

Modifica el programa del ejercicio 5 para que espere el datagrama del cliente durante un tiempo máximo de 5 segundos. Comprueba que el programa termina transcurridos los 5 segundos, aunque el cliente no haya enviado nada.

Anexo: Resumen de clases y métodos java relacionados con UDP

Clase InetAddress

Tipo devuelto	Método	Descripción
`InetAddress`	**`getByName(String s)`**	Devuelve la dirección IP asociada a s. Es un método estático que no requiere instanciar un objeto.
`String`	**`toString()`**	Devuelve una cadena con el contenido del objeto **`InetAddress`**.

Clase DatagramSocket

Constructor	
`DatagramSocket()`	Socket para cliente UDP.
`DatagramSocket(int num)`	Socket para servidor UDP.

Tipo devuelto	Método	Descripción
`int`	**`getLocalPort()`**	Devuelve el número de puerto (local) asociado al *socket.*
`InetAddress`	**`getLocalAddress()`**	Devuelve la dirección IP local asociada al socket.
`void`	**`send(DatagramPacket p)`**	Envía un datagrama UDP a través de este *socket*. El **`DatagramPacket`** incluye la información del destinatario y los datos a enviar.
`void`	**`receive(DatagramPacket p)`**	Queda a la espera de que se reciba un datagrama UDP por este *socket*. Es un método bloqueante, cuando retorna los datos recibidos están almacenados en **`p`**.

Clase DatagramPacket

Constructor	
`DatagramPacket (byte buf[], int longitud, InetAddress dirIP, int Puerto);`	Datagrama típico para enviar.
`DatagramPacket (byte buf[], int longitud)`	Datagrama típico para recibir. *Con frecuencia `longitud` se especifica como `buf.length`.*

Tipo devuelto	**Método**	**Descripción**
`InetAddress`	**`getAddress()`**	Devuelve la dirección IP desde la que se ha enviado el datagrama o a la que va destinado.
`byte[]`	**`getData()`**	Devuelve el buffer de datos.
`int`	**`getLength()`**	Devuelve la longitud de los datos a ser enviados o la de los datos recibidos.
`int`	**`getPort()`**	Devuelve el número de puerto del *socket* desde el que se envió el datagrama o al que va a ser enviado.
`void`	**`setAddress (InetAddress.getByName ("nombremáquina"))`**	Almacena la dirección IP especificada como destino en el **DatagramPacket**.
`void`	**`setData(byte[] buf)`**	Almacena los datos de **`buf`** en el **`DatagramPacket`**.
`void`	**`setLength(int length)`**	Establece la longitud de los datos del **`DatagramPacket`**.