# Práctica 6: Programación de *sockets* UDP

En esta práctica vamos a introducir la programación en java de *sockets* UDP. Veremos que debido a las características de UDP, pese a que el protocolo es más sencillo que TCP, su programación inicialmente resultará más compleja.

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

Los datos transmitidos pueden llegar fuera de orden o incluso perderse. Por tanto, la aplicación es responsable de verificar la entrega de los datos si se requiere fiabilidad. Como paralelismo, podría decirse que UDP tiene un funcionamiento similar al del correo postal: la "carta" (el datagrama UDP) lleva el remite y la dirección del destinatario. Si el remitente no recibe una respuesta a su mensaje no es posible garantizar su entrega, pudiendo haberse perdido el envío o la respuesta.

## 1. Lectura previa

Se recomienda leer previamente la Sección 2.8 del libro de Kurose, "Redes de Computadores", que explica la programación de *sockets* UDP 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 poder:

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.

Boletín realizado por Elvira Baydal, Alberto Bonastre y Federico Silla. Cualquier errata o sugerencia, por favor hacédsela llegar a los autores ({elvira, bonastre, fsilla}@disca.upv.es ).

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 almacenar información en un objeto de tipo **`DatagramPacket`**.
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, uno de los primeros problemas que se encuentra el cliente UDP es cómo especificar la dirección del destinatario. En java para este fin 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 proporciona directamente 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, 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 esta clase que puede resultarnos útil es el método **`toString()`** que convierte la dirección IP en 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 deberá ejecutar desde una consola. 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, ofreciéndonos distintas posibilidades. Los dos constructores que vamos a ver pueden generar excepciones del tipo **`SocketException`** (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 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 esta 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`** vacío en el cual se almacenará el mensaje recibido, así como su longitud y la dirección IP 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. Sin embargo, el método receptor **`receive()`** 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 el método **`receive()`**, las mismas 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*. Por otra parte, el sistema operativo descartará los datagramas UDP que lleguen a un ordenador donde no existe un **`DatagramSocket`** vinculado a su puerto destino.

Como se ha comentado, en ocasiones es interesante limitar el tiempo máximo de espera en un **`receive()`** para evitar que el proceso espere indefinidamente en ausencia de datagramas recibidos. Esta limitación puede conseguirse fácilmente mediante el método **`setSoTimeout()`** de la clase **`DatagramSocket`**:

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

Este método establece un intervalo de espera máximo (expresado en ms) para el tiempo que el método **`receive()`** se bloqueará esperando un paquete. Si transcurre el intervalo establecido (argumento **`timeout`**) 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
// Se bloquea como máximo durante 3 s
socket.receive(p);
```

## 5. La clase DatagramPacket

Esta clase representa un paquete UDP a transmitir o recibir. 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 (cuando se envía un paquete la dirección IP es la del destinatario y cuando se ha recibido un paquete la 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 paquete 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 (**`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 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 utiliza la orden **`nc -u -l 7777`**, que establece un servidor UDP que escucha en el puerto 7777 de tu ordenador. Ejecuta el programa varias veces. Curiosamente, observarás que sólo se recibe el nombre la primera vez (justificamos este comportamiento más abajo).

Una vez que compruebes que tu programa funciona correctamente cambia la dirección para que lo reciba el ordenador del profesor. El profesor tendrá que usar la orden **`sock -ul :7777`** para recibir los datos.

**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. La interfaz de los *sockets* permite restringir la comunicación de un socket UDP a un único destino. Con este fin la clase **`DatagramSocket`** proporciona el método **`connect()`**. Tras ejecutar este método, el *socket* sólo recibirá datagramas de la dirección IP y puerto especificados y descartará los que provengan de otro origen. Un intento de enviar un datagrama a un destino distinto del especificado generará una excepción. 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 datagrama con tu nombre hubiese enviado el mismo datagrama tres veces? Piénsalo unos instantes.

## 7. Recepción de paquetes

Para recibir datagramas es suficiente crear un objeto **`DatagramPacket`** con un *buffer* asociado. Utilizaremos el constructor

**`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 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);
```

Se puede obtener el mensaje recibido mediante el método **`getData()`** que devuelve el vector de bytes asociado al campo de datos del **`DatagramPacket`**. Como **`getData()`** nos devuelve el vector completo, para conocer la longitud del datagrama recibido, y por lo tanto, obtener los datos del vector que realmente nos interesan, debemos utilizar el método **`getLength()`**. El uso de este método es importante especialmente cuando se reutiliza un objeto **`DatagramPacket`** para varias recepciones, ya que podría quedar basura de alguna recepción anterior en el *buffer*.

Muchas veces necesitamos 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*:

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

El resto de información relevante del datagrama puede extraerse mediante los métodos **`getPort()`** y **`getAddress()`** de la clase **`DatagramPacket`**, que permiten obtener el puerto y la dirección IP desde los que se ha enviado el mismo.

```
InetAddress getAddress()
int getPort()
```

### Envío de respuestas

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

```
void setData(byte[] buf)
void setLengt(int length)
```

Ejemplo de uso:

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

### ¿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). Los paquetes mayores simplemente se truncan, descartando los datos que no caben en el *buffer* (precaución, 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. 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 de ese mismo puerto, 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`**". Para generar la respuesta, introduce un texto 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`**". El datagrama que envíe el cliente no es relevante, pero habrá que introducir al menos un retorno de carro (Intro) para que éste 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 name = now.toString() + "\n";
```

# 8. Ampliaciones opcionales

## Ejercicio 6:

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

Como segunda ampliación opcional, vamos a trabajar con los DNS. En ocasiones un nombre de dominio tiene varias direcciones IP asociadas. El método **`getAllByName`** nos permite obtenerlas todas en un vector de **`InetAddress`**:

```
static InetAddress[] getAllByName (String host) throws UnknownHostException
```

Como también ocurría con el método **`getByName()`**, si la resolución del nombre falla se lanzará una excepción **`UnknownHostException.`**

Ejemplo de uso:

```
InetAddress[] listaIp = InetAddress.getAllByName("www.hotmail.es");
```

La información obtenida puede imprimirse cómodamente mediante el método **`Arrays.toString(Object[] a)`**, disponible en la clase **`java.util.Arrays`**).

Ejemplo de uso:

```
System.out.println(Arrays.toString
                    (InetAddress.getAllByName("www.hotmail.es")));
```

## Ejercicio 7:

Escribe un programa en java que imprima por pantalla las múltiples direcciones de “www.google.es”.