Sumario

Networking

Introducción a Networking

Qué es Networking

El término Networking se refiere a la posibilidad de trabajar con diversas aplicaciones ubicadas físicamente en distintas estaciones de trabajo y permitir que se conecten vía una red, para trabajar de manera cooperativa o simplemente enviar y recibir información.

Qué es un socket

Un Socket es una representación abstracta del extremo (endpoint) en un proceso de comunicación. Para que se dé la comunicación en una Red, el proceso de comunicación requiere un Socket a cada extremo Emisor/Receptor y viceversa.

La comunicación con sockets sigue el modelo Cliente/Servidor/Cliente. En la mayoría de los casos un programa Servidor fundamentalmente envía datos, mientras que un programa Cliente recibe esos datos, aunque es raro que un programa exclusivamente reciba o envíe datos. Una distinción confiable se logra si consideramos Cliente al programa que inicia la comunicación y Servidor al programa que espera a que algún otro inicie comunicación con él.

La comunicación con sockets se hace mediante un protocolo de la familia TCP/IP en la mayoría de los casos, y es el programador quien decide qué protocolo utilizar dependiendo de la necesidades de la aplicación a desarrollar.

Socket designa un concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiarse cualquier flujo de datos, generalmente de manera fiable y ordenada.

Un socket queda definido por una dirección IP, un protocolo y un número de puerto.

Para que dos programas puedan comunicarse entre sí es necesario que se cumplan ciertos requisitos:

- Que un programa sea capaz de localizar al otro.

- Que ambos programas sean capaces de intercambiarse cualquier secuencia de octetos, es decir, datos relevantes a su finalidad.

Para ello son necesarios los tres recursos que originan el concepto de socket:

- Un protocolo de comunicaciones, que permite el intercambio de octetos.
- Una dirección del Protocolo de Red (Dirección IP, si se utiliza el Protocolo TCP/IP), que identifica una computadora.
- Un número de puerto, que identifica a un programa dentro de una computadora.

Los sockets permiten implementar una arquitectura cliente-servidor. La comunicación ha de ser iniciada por uno de los programas que se denomina programa cliente. El segundo programa espera a que otro inicie la comunicación, por este motivo se denomina programa servidor.

Características de un socket

Las propiedades de un socket dependen de las características del protocolo en el que se implementan. El protocolo más utilizado es TCP, aunque también es posible utilizar UDP o IPX. Gracias al protocolo TCP, los sockets tienen las siguientes propiedades:

- Orientado a conexión.
- Se garantiza la transmisión de todos los octetos sin errores ni omisiones.
- Se garantiza que todo octeto llegará a su destino en el mismo orden en que se ha transmitido.

Estas propiedades son muy importantes para garantizar la corrección de los programas que tratan la información.

El protocolo UDP es un protocolo no orientado a la conexión. Sólo se garantiza que si un mensaje llega, llegue bien. En ningún caso se garantiza que llegue o que lleguen todos los mensajes en el mismo orden que se mandaron. Esto lo hace adecuado para el envío de mensajes frecuentes pero no demasiado importantes, como por ejemplo, mensajes para los refrescos (actualizaciones) de un gráfico.

Identificación de procesos

Qué es una dirección IP

Una dirección IP es un número que identifica de manera lógica y jerárquica a una interfaz de un dispositivo (habitualmente una computadora) dentro de una red que utilice el protocolo IP (Internet Protocol), que corresponde al nivel de red del protocolo TCP/IP. Dicho número no se ha de confundir con la dirección MAC que es un número hexadecimal fijo que es asignado a la tarjeta o dispositivo de red por el fabricante, mientras que la dirección IP se puede cambiar.

Es habitual que un usuario que se conecta desde su hogar a Internet utilice una dirección IP. Esta dirección puede cambiar cada vez que se conecta; y a esta forma de asignación de dirección IP se denomina una dirección IP dinámica (normalmente se abrevia como IP dinámica).

Los sitios de Internet que por su naturaleza necesitan estar permanentemente conectados, generalmente tienen una dirección IP fija (se aplica la misma reducción por IP fija o IP estática), es decir, no cambia con el tiempo. Los servidores de correo, DNS, FTP públicos, y servidores de páginas web necesariamente deben contar con una dirección IP fija o estática, ya que de esta forma se permite su localización en la red.

Qué es un puerto

Un puerto es una dirección numérica a través de la cual se procesa un servicio, es decir, no son puertos físicos semejantes al puerto paralelo para conectar la impresora, sino que son direcciones lógicas proporcionadas por el sistema operativo para poder responder.

Las comunicaciones de información relacionada con Web tienen lugar a través del puerto 80 mediante protocolo TCP. Para emular esto en Java, se utiliza la clase Socket.

Teóricamente hay 65535 puertos disponibles, aunque los puertos del 1 al 1023 están reservados al uso de servicios estándar proporcionados por el sistema, quedando el resto libre para utilización por las aplicaciones de usuario. De no existir los puertos, solamente se podría ofrecer un servicio por máquina. Nótese que el protocolo IP no sabe nada al respecto de los números de puerto.

Se puede decir que IP pone en contacto las máquinas, TCP y UDP (protocolos de transmisión) establecen un canal de comunicación entre determinados procesos que se ejecutan en tales equipos y, los números de puerto se pueden entender como números de oficinas dentro de un gran edificio. El edificio (equipo), tendrá una única dirección IP, pero dentro de él, cada tipo de negocio, en este caso HTTP, FTP, etc., dispone de una oficina individual.

Qué es la URL (Uniform Resource Locator)

Una URL , o dirección, es en realidad un puntero a un determinado recurso de un determinado sitio de Internet. Al especificar una URL, se está indicando:

- El protocolo utilizado para acceder al servidor (http, por ejemplo)
- El nombre del servidor
- El puerto de conexión (opcional)
- El camino, y
- El nombre de un archivo determinado en el servidor (opcional a veces)
- Un punto de referencia dentro del archivo (opcional)

La sintaxis general, para una dirección URL, sería:

protocolo://nombre_servidor[:puerto]/directorio/archivo#referencia

El puerto es opcional y normalmente no es necesario especificarlo si se está accediendo a un servidor que proporcione sus servicios a través de los puertos estándar; tanto el navegador como cualquier otra herramienta que se utilice en la conexión conocen perfectamente los puertos por los

cuales se proporciona cada uno de los servicios e intentan conectarse directamente a ellos por defecto.

Implementación en JAVA (TCP)

La clase URL

Para comenzar con la programación de sockets, resulta necesario comprender las clases que ofrece Java. En primer lugar, la clase URL contiene constructores y métodos para la manipulación de URL (Universal Resource Locator): un objeto o servicio en Internet. El protocolo TCP necesita dos tipos de información: la dirección IP y el número de puerto. Vamos a ver cómo podemos recibir pues la página Web principal de nuestro buscador favorito al teclear:

http://www.yahoo.com

En primer lugar, Yahoo tiene registrado su nombre, permitiendo que se use yahoo.com como su dirección IP, o lo que es lo mismo, cuando indicamos yahoo.com es como si hubiesemos indicado 205.216.146.71, su dirección IP real.

Si queremos obtener la dirección IP real de la red en que estamos corriendo, podemos realizar llamadas a los métodos getLocalHost() y getAddress(). Primero, getLocalHost() nos devuelve un objeto iNetAddress, que si usamos con getAddress() generará un array con los cuatro bytes de la dirección IP, por ejemplo:

InetAddress direccion = InetAddress.getLocalHost(); byte direccionIp[] = direccion.getAddress();

Si la dirección de la máquina en que estamos corriendo es 150.150.112.145, entonces:

direccionIp[0] = 150 direccionIp[1] = 150 direccionIp[2] = 112 direccionIp[3] = 145

Por otro lado, podemos especificar las partes que componen una url y así podríamos construir un objeto de tipo url utilizando los siguientes constructores:

URL("http","www.yahoo.com","80","index.html");

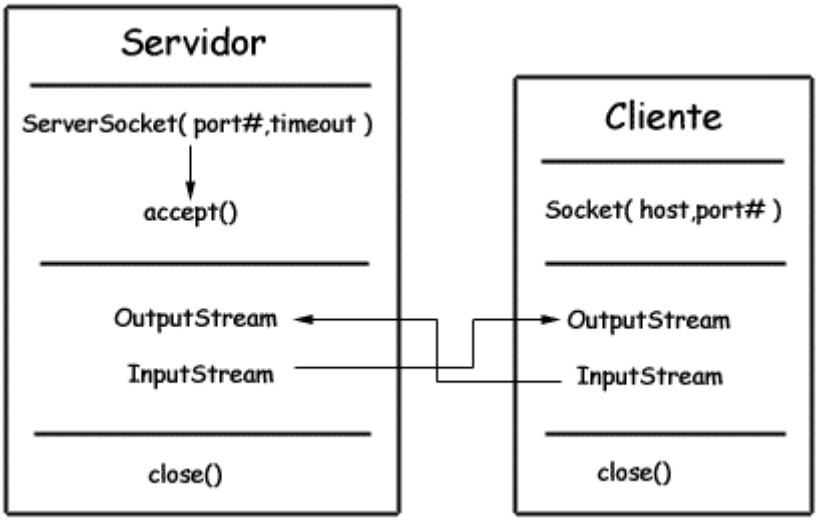
o dejar que los sistemas utilicen todos los valores por defecto que tienen definidos, como en:

URL("http://www.yahoo.com");

y en los dos casos obtendríamos la visualización de la página principal de Yahoo en nuestro navegador.

Arquitectura de comunicaciones

En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete java.net. A continuación mostramos un diagrama de lo que ocurre en el lado del cliente y del servidor:



El modelo de sockets más simple es:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método accept().
- El cliente establece una conexión con la máquina host a través del puerto que se designe en puerto#
- El cliente y el servidor se comunican con manejadores InputStream y OutputStream

La clase Socket

Si estamos programando un cliente, el socket se abre de la forma:

Socket miCliente; miCliente = new Socket("maquina",numeroPuerto);

Dónde máquina es el nombre de la máquina en donde estamos intentando abrir la conexión y numeroPuerto es el puerto (un número) del servidor que está corriendo sobre el cual nos queremos conectar. Para las aplicaciones que se desarrollen, asegurarse de seleccionar un puerto por encima del 1023.

En el ejemplo anterior no se usan excepciones; sin embargo, es una gran idea la captura de excepciones cuando se está trabajando con sockets. El mismo ejemplo quedaría como:

Socket miCliente;

```
try {
      miCliente = new Socket( "maquina",numeroPuerto );
} catch( IOException e ) { System.out.println( e ); }
```

La clase ServerSocket

Si estamos programando un servidor, la forma de apertura del socket es la que muestra el siguiente ejemplo:

Socket miServicio;

```
try {
```

```
      miServicio = new ServerSocket( numeroPuerto );
} catch( IOException e ) { System.out.println( e ); }
```

A la hora de la implementación de un servidor también necesitamos crear un objeto socket desde el ServerSocket para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio = null;
try {
      socketServicio = miServicio.accept();
} catch( IOException e ) {
      System.out.println( e );
}
```

Creación de Streams de Entrada

En la parte cliente de la aplicación, se puede utilizar la clase DataInputStream para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.

DataInputStream entrada;

```
try {
      entrada = new DataInputStream( miCliente.getInputStream() );
} catch( IOException e ) { System.out.println( e ); }
```

La clase DataInputStream permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: read(), readChar(), readInt(), readDouble() y readLine(). Deberemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperamos recibir del servidor.

En el lado del servidor, también usaremos DataInputStream, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

DataInputStream entrada;

```
try {
      entrada = new DataInputStream( socketServicio.getInputStream() );
} catch( IOException e ) { System.out.println( e ); }
```

Creación de Streams de Salida

En el lado del cliente, podemos crear un stream de salida para enviar información al socket del servidor utilizando las clases PrintStream o DataOutputStream:

PrintStream salida;

```
try {
      salida = new PrintStream( miCliente.getOutputStream() );
} catch( IOException e ) { System.out.println( e ); }
```

La clase PrintStream tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos write y println() tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar DataOutputStream:

DataOutputStream salida;

```
try {
      salida = new DataOutputStream( miCliente.getOutputStream() );
} catch( IOException e ) { System.out.println( e ); }
```

La clase DataOutputStream permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea writeBytes().

En el lado del servidor, podemos utilizar la clase PrintStream para enviar información al cliente:

PrintStream salida;

```
try {
      salida = new PrintStream( socketServicio.getOutputStream() );
} catch( IOException e ) { System.out.println( e ); }
```

Pero también podemos utilizar la clase DataOutputStream como en el caso de envío de información desde el cliente.

Cierre de Sockets

Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente:

```
try {
      salida.close();
      entrada.close();
      miCliente.close();
} catch( IOException e ) { System.out.println( e ); }
```

Y en la parte del servidor:

```
try {
      salida.close();
      entrada.close();
      socketServicio.close();
      miServicio.close();
} catch( IOException e ) {
      System.out.println( e );
}
```

Uso de Closeable para el cierre de Sockets

La clase Socket, DataOutputStream, DataInputStream y PrintStream implementan Closeable por lo tanto se puede usar try with resources para el cierre de los sockets y las corrientes.

```
        try (Socket so=new Socket("maquina",3333);
            PrintStream salida = new PrintStream(so.getOutputStream());
        ){
            salida.println("hola");
        } catch( IOException e ) { System.out.println( e ); }
```

Construcción de un Servidor TCP/IP

La clase Servidor TCP/IP

Veamos el código que presentamos en el siguiente ejemplo, donde desarrollamos un mínimo servidor TCP/IP, para el cual desarrollaremos después su contrapartida cliente TCP/IP. La aplicación servidor TCP/IP depende de una clase de comunicaciones proporcionada por Java: ServerSocket. Esta clase realiza la mayor parte del trabajo de crear un servidor.

```
import java.net.*;
import java.io.*;
class minimoServidor {
	public static void main( String args[] ) {
		ServerSocket s = (ServerSocket)null;
		Socket s1;
		String cadena = "Tutorial de Java!";
		int longCad;
		OutputStream s1out;
		// Establece el servidor en el socket 4321 (espera 300 segundos)
		try {
			s = new ServerSocket( 4321,300 );
		} catch( IOException e ) { System.out.println( e ); }
		// Ejecuta un bucle infinito de listen/accept
		while( true ) {
			try {
			// Espera para aceptar una conexión
			s1 = s.accept();
			// Obtiene un controlador de salida asociado con el socket
			s1out = s1.getOutputStream();
			// Enviamos nuestro texto
			longCad = sendString.length();
			for( int i=0; i < longCad; i++ )
				s1out.write( (int)sendString.charAt( i ) );
			// Cierra la conexión, pero no el socket del servidor
			s1.close();
			} catch( IOException e ) { System.out.println( e ); }
		}
	}
}
```

La clase Cliente TCP/IP

El lado cliente de una aplicación TCP/IP descansa en la clase Socket. De nuevo, mucho del trabajo necesario para establecer la conexión lo ha realizado la clase Socket. Vamos a presentar ahora el código de nuestro cliente más simple, que encaja con el servidor presentado antes. El trabajo que realiza este cliente es que todo lo que recibe del servidor lo imprime por la salida estándar del sistema.

```
import java.net.*;
import java.io.*;
class minimoCliente {
	public static void main( String args[] ) throws IOException {
		int c;
		Socket s;
		InputStream sIn;
		// Abrimos una conexión con breogan en el puerto 4321
		try {
			s = new Socket( "breogan",4321 );
```

```
            } catch( IOException e ) { System.out.println( e ); }
            // Obtenemos un controlador de entrada del socket
            sIn = s.getInputStream();
            while( ( c = sIn.read() ) != -1 ) System.out.print( (char)c );
            s.close();
        }
}
```

Protocolo UDP

En los sockets TCP es necesario establecer una conexión. El servidor TCP espera que un cliente TCP se le conecte. Una vez hecha la conexión, se pueden enviar mensajes. El protocolo TCP garantiza que todos los mensajes enviados van a llegar bien y en el orden enviado. Sólo el cliente necesita saber dónde está el servidor y en qué puerto está escuchando.

Por el contrario, en los sockets UDP no se establece conexión. El servidor se pone a la escucha de un puerto de su ordenador. El cliente también se pone a la escucha de un puerto de su ordenador. En cualquier momento, cualquiera de ellos puede enviar un mensaje al otro. Ambos necesitan saber en qué ordenador y en qué puerto está escuchando el otro. Aquí el concepto de cliente y servidor está un poco más difuso que en el caso de TCP. Podemos considerar servidor al que espera un mensaje y responde. Cliente sería el que inicia el envío de mensajes. El servidor debería, además, estar siempre arrancado y a la escucha.

Además, en contra de TCP, el protocolo UDP sólo garantiza que si el mensaje llega, llega bien. No garantiza que llegue ni que lleguen en el mismo orden que se han enviado. Este tipo de sockets es útil para el envío más o menos masivo de información no crucial. Por ejemplo, enviar los datos para el refresco de gráficos en una pantalla.

Otra ventaja de UDP respecto a TCP, es que con UDP se puede enviar un mensaje a varios receptores a la vez, mientras que en TCP, al haber una conexión previa, sólo se puede enviar el mensaje al que está conectado al otro lado. Con UDP se puede, por ejemplo, enviar una sola vez los datos para que varias pantallas refresquen sus gráficos a la vez.

Implementación en JAVA (UDP)

La clase InetAddress

La clase InetAddress proporciona objetos que se pueden utilizar para manipular tanto direcciones IP como nombres de dominio; sin embargo, no se pueden instanciar estos objetos directamente. La clase proporciona varios métodos estáticos que devuelven un objeto de tipo InetAddress.

El método estático getByName() devuelve un objeto InetAddress representando el host que se le pasa como parámetro. Este método se puede utilizar para determinar la dirección IP de un host, conociendo su nombre; entendiendo por nombre del host el nombre de la máquina, como "java.sun.com", o la representación como cadena de su dirección IP, como "206.26.48.100". El método getAllByName() devuelve un array de objetos InetAddress, y se puede utilizar para determinar todas las direcciones IP asignadas a un host. El método getLocalHost() devuelve un objeto InetAddress representando el ordenador local sobre el que se ejecuta la aplicación.

El primer trozo de código interesante es el que obtiene un objeto InetAddress representando un determinado servidor y presenta esta dirección utilizando el método sobrecargado toString() de la clase InetAddress.

```
InetAddress address = InetAddress.getByName( "www.educacionit.com.ar" );
System.out.println( address );
```

El siguiente código es la acción contraria al anterior, en que se proporciona la dirección IP para presentar el nombre del host. Como ya se ha indicado, el método getByName() puede aceptar como parámetro de entrada tanto el nombre del host, como su dirección IP en forma de cadena. El código, utiliza el resultado de la llamada al método anterior para construir una cadena con la parte numérica del resultado, que es pasada al método getByName().

```
int temp = address.toString().indexOf( '/' );
address = InetAddress.getByName( address.toString().substring(temp+1) );
System.out.println( address );
```

El término localhost se utiliza para describir el ordenador local, la máquina en la que se está ejecutando la aplicación. Cuando se conecta a una red IP, el ordenador local debe tener una dirección IP, que puede conseguir de diferentes formas; no obstante, la siguiente explicación se basa en que el ordenador sobre el cual se ejecuta la aplicación se conecta a través de la línea telefónica con un proveedor de Internet, que es el que abre el acceso a Internet de la máquina local.

El proveedor de Internet tiene reservadas una serie de direcciones IP, que puede compartir entre todos sus clientes. Cuando alguien se conecta al proveedor, automáticamente se le asigna una dirección a esa conexión, válida durante todo el tiempo que dure esa sesión. Si se produce una desconexión y luego se vuelve a conectar, lo más seguro es que la dirección IP no se la mima que se había asignado a la primera conexión.

Aunque esa sea la situación más habitual del lector, en otras ocasiones puede ser diferente. Por ejemplo, si el ordenador se encuentra en la red interna de ordenadores de una empresa, esa empresa puede tener un bloque de direcciones IP reservadas y asignar permanentemente las direcciones a los ordenadores; en cuyo caso, cada vez que se ejecute el programa, la dirección IP será siempre la misma. También es posible que el lector disponga de su propia dirección permanente IP y nombre de dominio.

En cualquier caso, el método getLocalHost() se puede utilizar para obtener un objeto de tipo InetAddress que represente al ordenador en el cual se está ejecutando la aplicación. Eso es justamente lo que muestra el código que aparece a continuación.

```
address = InetAddress.getLocalHost();
System.out.println( address );
```

En este caso, la parte numérica que aparece al ejecutar el programa, corresponde a la dirección que el proveedor de Internet ha asignado a la conexión sobre la cual se estaba corriendo el programa.

Las líneas de código siguientes realizan la operación contraria a las anteriores; se utiliza el método getByName() para determinar el nombre por el cual el servidor de nombres de dominio reconoce a esa dirección numérica.

```
temp = address.toString().indexOf( '/' );
address = InetAddress.getByName( address.toString().substring(temp+1) );
System.out.println( address );
```

Una vez que se ha obtenido el objeto InetAddress, hay otra serie de métodos de la clase InetAddress que se pueden utilizar para invocar a este objeto; por ejemplo, las siguientes líneas de código muestran la invocación del método getHostName(), para obtener el nombre de la máquina.

```
System.out.println( address.getHostName() );
```

De forma semejante, el siguiente fragmento de código utiliza la invocación del método getAddress() para obtener un array de bytes conteniendo la dirección IP de la máquina.

```
byte[] bytes = address.getAddress();
// Convierte los bytes de la dirección IP a valores sin
// signo y los presenta separados por espacios
for( int cnt=0; cnt < bytes.length; cnt++ ) {
      int uByte = bytes[cnt] < 0 ? bytes[cnt]+256 : bytes[cnt];
      System.out.print( uByte+" " );
}
```

Como no existe nada parecido a un byte sin signo en Java, la conversión del array de bytes en algo que se pueda presentar en pantalla requiere una cierta manipulación de los bytes, tal como se hace en el bucle for del código anterior.

Arquitectura de comunicaciones

La clase DatagramPacket, junto con la clase DatagramSocket, son las que se utilizan para la implementación del protocolo UDP (User Datagram Protocol).

En este protocolo, a diferencia de lo que ocurría en el protocolo TCP, en el cual si un paquete se dañaba durante la transmisión se re enviaba ese paquete, para asegurar una comunicación segura entre cliente y servidor; con UDP no hay garantía alguna de que los paquetes lleguen en el orden correcto a su destino y, ni tan siquiera hay seguridad de que lleguen todos los paquetes que se hayan enviado. Sin embargo, los paquetes que consiguen llegar, lo hacen mucho más rápidamente que con TCP y, en algunos casos, la velocidad de transmisión es mucho más importante que el que lleguen todos los paquetes; por ejemplo, si lo que se están transmitiendo son señales de sensores en tiempo real para la presentación en pantalla, la velocidad es más importante que la integridad, ya que si un paquete no llega o no puede recomponerse, en el instante siguiente llegará otro.

La programación para uso del protocolo UDP se diferencia de la programación para utilizar el protocolo TCP en que no existe el concepto de ServerSocket para los datagramas y que es el programador el que debe construirse sus propios paquetes a enviar por UDP.

Para enviar datos a través de UDP, hay que construir un objeto de tipo DatagramPacket y enviarlo a través de un objeto DatagramSocket, y al revés para recibirlos, es decir, a través de un objeto DatagramSocket se recoge el objeto DatagramPacket. Toda la información respecto a la dirección, puerto y datos, está contenida en el paquete.

Para enviar un paquete, primero se construye ese paquete con la información que se desea transmitir, luego se almacena en un objeto DatagramSocket y, finalmente se invoca el método

send() sobre ese objeto. Para recibir un paquete, primero se construye un paquete vacío, luego se le presenta a un objeto DatagramSocket para que almacene allí el resultado de la ejecución del método receive() sobre ese objeto.

Hay que tener en cuenta que la tarea encargada de todo esto estará bloqueada en el método receive() hasta que un paquete físico de datos se reciba a través de la red; este paquete físico será el que se utilice para rellenar el paquete vacío que se había creado.

También hay que tener cuidado cuando se pone a escuchar a un objeto DatagramSocket en un puerto determinado, ya que va a recibir los datagramas enviados por cualquier cliente. Es decir, que si los mensajes enviados por los clientes están formados por múltiples paquetes; en la recepción pueden llegar paquetes entremezclados de varios clientes y es responsabilidad de la aplicación el ordenarlos.

La clase DatagramPacket

Para la clase DatagramPacket se dispone de dos constructores, uno utilizado cuando se quieran enviar paquetes y el otro se usa cuando se quieren recibir paquetes. Ambos requieren que se les proporcione un array de bytes y la longitud que tiene. En el caso de la recepción de datos, no es necesario nada más, los datos que se reciban se depositarán en el array; aunque, en el caso de que se reciban más datos físicos de los que soporta el array, el exceso de información se perderá y se lanzará una excepción de tipo IllegalArgumentException, que a pesar de que no sea necesaria su captura, siempre es bueno recogerla.

Cuando se construye el paquete a enviar, es necesario colocar los datos en el array antes de llamar al método send(); además de eso, hay que incluir la longitud de ese array y, también se debe proporcionar un objeto de tipo InetAddress indicando la dirección de destino del paquete y el número del puerto de ese destino en cual estará escuchando el receptor del mensaje. Es decir, que la dirección de destino y el puerto de escucha deben ir en el paquete, al contrario de lo que pasaba en el caso de TCP que se indican en el momento de construir el objeto Socket.

El tamaño físico máximo de un datagrama es 65,535 bytes, y teniendo en cuenta que hay que incluir datos de cabecera, esa longitud nunca está disponible para datos de usuario, sino que siempre es algo menor.

La clase DatagramPacket proporciona varios métodos para poder extraer los datos que llegan en el paquete recibido. La información que se obtiene con cada método coincide con el propio nombre del método, aunque hay algunos casos en que es necesario saber interpretar la información que proporciona ese mismo método.

```
// Para envío
DatagramPacket dato = new DatagramPacket(data, // El array de bytes
data.length, // Su longitud
InetAddress.getByName(HOST_DESTINO), // Destinatario
PUERTO_DESTINO);  // Puerto del destinatario
//Para recepción
DatagramPacket dato = new DatagramPacket(new byte[100], 100);
```

El método getAddress() devuelve un objeto de tipo InetAddress que contiene la dirección del host remoto. El saber cuál es el ordenador de origen del envío depende de la forma en que se haya

obtenido el datagrama. Si ese datagrama ha sido recibido a través de Internet, la dirección representará al ordenador que ha enviado el datagrama (el origen del datagrama); pero si el datagrama se ha construido localmente, la dirección representará al ordenador al cual se intenta enviar el datagrama (el destino del datagrama).

De igual modo, el método **getPort()** devuelve el puerto desde el cual ha sido enviado el datagrama, o el puerto a través del cual se enviara, dependiendo de la forma en que se haya obtenido el datagrama.

El método **getData()** devuelve un array de bytes que contiene la parte de datos del datagrama, ya eliminada la cabecera con la información de encaminamiento de ese datagrama. La forma de interpretar ese array depende del tipo de datos que contenga. Los ejemplos que se ven en este Tutorial utilizan exclusivamente datos de tipo String, pero esto no es un requerimiento, y se pueden utilizar datagramas para intercambiar cualquier tipo de datos, siempre que se puedan colocar en un array de bytes en un ordenador y extraerlos de ese array en la parte contraria. Es decir, que la responsabilidad del sistema se limita al desplazamiento del array de bytes de un ordenador a otro, y es responsabilidad del programador el asignar significado a esos bytes.

El método **getLength()** devuelve el número de bytes que contiene la parte de datos del datagrama, y el método **getOffset()** devuelve la posición en la cual empieza el array de bytes dentro del datagrama completo.

La clase DatagramSocket

Un objeto de la clase DatagramSocket puede utilizarse tanto para enviar como para recibir un datagrama.

La clase tiene tres constructores. Uno de ellos se conecta al primer puerto libre de la máquina local; el otro permite especificar el puerto a través del cual operará el socket; y el tercero permite especificar un puerto y una dirección para identificar a una máquina concreta.

Independientemente del constructor que se utilice, el puerto desde el cual se envíe el datagrama, siempre se incluirá en la cabecera del paquete. Normalmente, la parte del servidor utilizará el constructor que permite indicar el puerto concreto a usar, ya que sino, la parte cliente no tendría forma de conocer el puerto por el cual le van a llegar los datagramas.

La parte cliente puede utilizar cualquier constructor, pero por flexibilidad, lo mejor es utilizar el constructor que deja que el sistema seleccione uno de los puertos disponibles. El servidor debería entonces comprobar cuál es el puerto que se está utilizando para el envío de datagramas y enviar la respuesta por ese puerto.

A diferencia de esta posibilidad de especificar el puerto, o no hacerlo, no hay ninguna otra diferencia entre los sockets datagrama utilizados por cliente y servidor. Si el lector se encuentra un poco perdido, no desespere, porque en los ejemplos todo esto que es muy difícil explicar con palabras, se ve mucho más claramente al intuir el funcionamiento físico de la comunicación entre cliente y servidor.

```
// Para recepción
DatagramSocket socket = new DatagramSocket(
      PUERTO_RECEPCION,
```

```
      InetAddress.getByName("localhost"));
// Para envío
DatagramSocket socket = new DatagramSocket(
      PUERTO_DESTINO,
      InetAddress.getByName(HOST_DESTINO));
```

Para enviar un datagrama hay que invocar al método **send()** sobre un socket datagrama existente, pasándole el objeto paquete como parámetro. Cuando el paquete es enviado, la dirección y número de puerto del ordenador origen se coloca automáticamente en la porción de cabecera del paquete, de forma que esa información pueda ser recuperada en el ordenador destino del paquete.

```
socket.send(packet);
```

Para recibir datagramas, hay que instanciar un objeto de tipo DatagramSocket, conectarse a un puerto determinado e invocar al método **receive()** sobre ese socket. Este método bloquea la tarea hasta que se recibe un datagrama, por lo que si es necesario hacer alguna cosa durante la espera, hay que invocar al método **receive()** en su propia tarea.

```
socket.receive(packet);
```

Si se trata de un servidor, hay que conectarse con un puerto específico. Si se trata de un cliente que está esperando respuestas de un servidor, hay que escuchar en el mismo puerto que fue utilizado para enviar el datagrama inicial. Si se envía un datagrama a un puerto anónimo, se puede mantener el socket abierto que fue utilizado en el envío del primer datagrama y utilizar ese mismo socket para esperar la respuesta. También se puede invocar al método getLocalPort() sobre el socket antes de cerrarlo, de forma que se pueda saber y guardar el número del puerto que se ha empleado; de este modo se puede cerrar el socket original y abrir otro socket en el mismo puerto en el momento en que se necesite.

Para responder a un datagrama, hay que obtener la dirección del origen y el número de puerto a través del cual fue enviado el datagrama, de la cabecera del paquete y luego, colocar esta información en el nuevo paquete que se construya con la información a enviar como respuesta. Una vez pasada esta información a la parte de datos del paquete, se invoca al método send() sobre el objeto DatagramSocket existente, pasándole el objeto paquete como parámetro.

Es importante tener en en cuenta que los números de puerto TCP y UDP no están relacionados. Se puede utilizar el mismo número de puerto en dos procesos si uno se comunica a través de protocolo TCP y el otro lo hace a través de protocolo UDP. Es muy común que los servidores utilicen el mismo puerto para proporcionar servicios similares a través de los dos protocolos en algunos servicios estándar, como puede ser el echo.

Construcción de un Servidor UDP

La clase Servidor UDP

Veamos el código que presentamos en el siguiente ejemplo, donde desarrollamos un mínimo servidor UDP, para el cual desarrollaremos después su contrapartida cliente UDP.

```
public class UDPServer {
      public static void main(String[] args) throws InterruptedException {
            try {
```

```
                  DatagramSocket socket = new DatagramSocket(5000,
                        InetAddress.getByName("localhost"));
                  DatagramPacket dato = new DatagramPacket(
                        "hola".getBytes(), // El array de bytes
                        "hola".getBytes().length, // Su longitud
                        InetAddress.getByName("localhost"),  // Destinatario
                              4000);   // Puerto del destinatario
                  while (true) {
                        try {
                              socket.send(dato);
                              Thread.currentThread().sleep(2000);
                        } catch (Exception ex2) { System.out.println(ex2); }
                  }
            } catch (Exception ex) { System.out.println(ex); }
      }
}
```

La clase Cliente UDP

El lado cliente de una aplicación UDP descansa en las mismas clases que el servidor.

```
public class UDPClient {
      public static void main(String[] args) throws Exception {
            try {
                  DatagramSocket socket = new DatagramSocket(4000,
                        InetAddress.getByName("localhost"));
                  DatagramPacket dato = new DatagramPacket(new byte[100], 100);
                  while ( true) {
                        try {
                              socket.receive(dato);
                              System.out.println(new String(dato.getData()));
                              Thread.currentThread().sleep(2000);
                        } catch (Exception ex2) { System.out.println(ex2); }
                  }
            } catch (Exception ex) { System.out.println(ex2); }
      }
}
```

Multicast

El Multicast es un método de direccionamiento IP. Una dirección multicast está asociada con un grupo de receptores interesados. De acuerdo al RFC 3171 las direcciones desde la 224.0.0.0 a la 239.255.255.255 están destinadas para ser direcciones de multicast. Este rango se llama formalmente "Clase D".

El emisor envía un único datagrama (desde la dirección unicast del emisor) a la dirección multicast y el router se encargará de hacer copias y enviarlas a todos los receptores que hayan informado de su interés por los datos de ese emisor.

La clase MulticastSocket

Java proporciona una interfaz de datagramas para multicast IP a través de la clase MulticastSocket, que es una subclase de DatagramSocket, con la capacidad adicional de ser capaz de pertenecer a grupos multicast.

La clase MulticastSocket proporciona dos constructores alternativos:

- MulticastSocket(): que crea el socket en cualquiera de los puertos locales libres.

- MulticastSocket(int port): que crea el socket en el puerto local indicado.

Un proceso puede pertenecer a un grupo multicast invocando el método joinGroup(InetAddress mcastaddr) de su socket multicast. Así, el socket pertenecerá a un grupo de multidifusión en un puerto dado y recibirá los datagramas enviados por los procesos en otros computadores a ese grupo en ese puerto. Un proceso puede dejar un grupo dado invocando el método leaveGroup(InetAddress mcastaddr) de su socket multicast.

Para enviar datos a un grupo multicast se utiliza el método send(DatagramPacket p), este método es muy similar al de la clase DatagramSocket, la diferencia es que este datagrama será enviado a todos los miembros del grupo multicast.

Para recibir datos de un grupo multicast se utiliza el método receive(DatagramPacket p) de la clase DatagramSocket super clase de MulticastSocket.

Es necesario pertenecer a un grupo para recibir mensajes multicast enviados a ese grupo, pero no es necesario para enviar mensajes.

La clase Servidor Multicast UDP

```
public class MulticastServer {
	public static void main(String[] args) throws IOException {
		try {
			byte[] b = new byte[100];
			DatagramPacket dgram = new DatagramPacket(b, b.length);
			MulticastSocket socket = new MulticastSocket(4000);
			// must bind receive side
			socket.joinGroup(InetAddress.getByName("235.1.1.1"));
			while(true) {
				socket.receive(dgram);
				// blocks until a datagram is received
				System.err.println("Received " + dgram.getLength()
					+ " bytes from " + dgram.getAddress()
					+ new String(dgram.getData()));
				dgram.setLength(b.length); // must reset length field!
			}
		} catch (Exception ex) {  }
	}
}
```

La clase Cliente Multicast UDP

```
public class MulticastCliente {
	public static void main(String[] args) throws Exception {
		try {
			DatagramSocket socket = new DatagramSocket();
			byte[] b = "Hola".getBytes();
			DatagramPacket dgram;
			dgram = new DatagramPacket(b,
				b.length, InetAddress.getByName("235.1.1.1"), 4000);
			System.err.println("Sending " + b.length + " bytes to " +
				dgram.getAddress() + ':' + dgram.getPort());
			while(true) {
				System.err.print(".");
				socket.send(dgram);
				Thread.sleep(1000);
			}
		} catch (Exception ex) { }
```

```
    }
}
```

Cliente Http2 JDK 9

JDK 9 incorpora un api de cliente del protocolo http2. El cual viene incluido en el paquete jdk.incubator para la versión de jdk11 dicho paquere fue reemplazado por java.net.http

Ejemplo de uso

```
        //Cuidado en la seccion module se indica en jdk 9 requires
        //jdk.incubator.httpclient;
        //Pero a partir de JDK 11 requires java.net.http;
        //modifique levemente el código para que compile con jdk11

        HttpClient httpClient = HttpClient.newHttpClient();
        //Create a HttpClient
        System.out.println(httpClient.version());
        HttpRequest httpRequest = HttpRequest
                .newBuilder()
                .uri(new URI("https://www.google.com/"))
                //URL a golpear por el cliente
                .GET()
                .build(); //Create a GET request for the given URI

        Map < String, List<String>> headers = httpRequest.headers().map();

        headers.forEach((k, v) -> System.out.println(k + "-" + v));

        HttpResponse <String> httpResponse = httpClient
                .send(httpRequest, HttpResponse.BodyHandlers.ofString());

        System.out.println(httpResponse.headers().toString());
        System.out.println(httpResponse.body());


module TestHttp {
    //requires jdk.incubator.httpclient;
    requires java.net.http;
}
```