Tema 7

Programacion en red.

Departament d'Informàtica. Universitat de València

Índice

Vamos a ver algunas de las clases que proporciona Java para realizar programas en los que la información se transmita a través de una red.

Veremos 4 grupos de clases:

- Las que sirven para representar una dirección IP.
- Clases que sirven para enviar información utilizando el protocolo UDP (User Datagram Protocol)
- Clases que sirven para enviar información utilizando el protocolo TCP (Transmission Control Protocol)
- Clases que sirven para trabajar con el protocolo HTTP (HyperText Transfer Protocol)

1. Direcciones IP

Una dirección IP es un número sin signo de 32 bits (IPv4) o de 128 bits (IPv6).

Una instancia de la clase InetAddress sirve para representar una dirección IP.

Algunos de los métodos que define esta clase son:

```
public static InetAddress getByName(String host) throws UnknownHostException
```

Devuelve un objeto del tipo InetAddress a partir del nombre.

```
public byte[] getAddress()
```

Devuelve la dirección IP en un vector de byte.

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

Dado un nombre de servidor, devuelve un vector con todas sus direcciones IP. El nombre puede ser un nombre de máquina, como por ejemplo `sweb.uv.e` o una representación textual de su dirección IP como por ejemplo `147.156.16.46`

```
public static InetAddress getLocalHost()
```

Devuelve la dirección de la máquina local.

El siguiente código muestra un ejemplo de aplicación de esta clase:

```
import java.net.*;

public class DireccionesIP {

   public static void main(String[] args) {
      byte[] b;
      try{
         // Si se pasa un argumento obtenemos la direccion IP
         if (args.length>0){
            String host = args[0];
            InetAddress[] direcciones = InetAddress.getAllByName(host);

            for (int i=0;i<direcciones.length;i++){
               System.out.println(direcciones[i]);
            }
         }
         // Si no obtenemos la direccion IP de la maquina local
         else{
            InetAddress direccion = InetAddress.getLocalHost();
            System.out.println(direccion);
         }


      }catch(Exception e){
         e.printStackTrace();
      }

   }
}
```

2. UDP

Es un protocolo de la capa de transporte que las aplicaciones pueden utilizar para enviar paquetes de datos.

UDP soporta la especificación de un número de puerto por lo que puede ser utilizado para enviar datagramas a aplicaciones o servicios especificos.

- **Inconvenientes**: no garantiza la entrega de los paquetes ni que éstos lleguen en el orden en el que han sido enviados.
- **Ventajas**: Es eficiente, causa poca sobrecarga, es conveniente para aplicaciones en tiempo real y un socket UDP puede recibir datos de diferentes máquinas.

Java proporciona dos clases para el trabajo con el protocolo UDP:

- java.net.DatagramPacket
- java.net.DatagramSocket

La clase `DatagramPacket` representa un paquete de datos que va a ser enviado o recibido utilizando el protocolo UDP. Los paquetes contienen datos, una dirección IP y un número de puerto.

Hay dos situaciones en las que se debe crear un objeto del tipo DatagramPacket:

- para enviar un paquete a una máquina remota utilizando UDP (la dirección IP y el puerto se refieren a la máquina destino).
- para recibir un paquete enviado por una máquina remota utilizando UDP (la dirección IP y el número de puerto se refieren a la máquina que ha enviado el paquete).

Esta clase dispone de varios constructores (solo se muestran algunos):

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

Crea un paquete UDP para **recibir** paquetes de longitud long, almacenando los datos en buf.

```
DatagramPacket(byte[] buf, int long, InetAddress direcc, int puerto)
```

Construye un paquete UDP para **enviar** la información contenida en buf cuya longitud es long a la máquina y puertos especificados

Algunos de los métodos que ofrece esta clase son:

```
public InetAddress getAddress()
```

Devuelve la dirección IP asociada al paquete UDP.

```
public byte[] getData()
```

Devuelve el buffer con los datos.

```
public int getLength()
```

Devuelve el número de bytes almacenado en el datagrama. Este número puede ser menor que el tamaño del buffer.

```
public void setAddress(InetAddress dir)
```

Sirve para asignar una dirección IP al paquete UDP.

```
public setData(byte[] buffer)
```

Asigna nuevos datos al paquete UDP.

```
public setLength(int long)
```

Sirve para asignar la longitud real de los datos que contiene el paquete UDP. Este número deber ser menor o igual que el tamaño del buffer.

La clase DatagramSocket representa un socket UDP para permitir el envío y recepción de paquetes UDP.

Una misma instancia de la clase DatagramSocket puede ser utilizada tnto para recibir paquetes como para enviarlos.

La clase DatagramSocket proporciona entre otros los siguientes constructores:

```
DatagramSocket()
```

Contruye un socket para el trabajo con paquetes UDP y le asigna un puerto libre en la máquina local.

```
DatagramSocket(int puerto)
```

Contruye un socket para el trabajo con paquetes UDP y le asigna el puerto especificado.

Algunos de los métodos que ofrece esta clase son:

```
public void send(DatagramPacket datagrama) throws IOException
```

Envía el datagrama UDP.

```
public void receive(DatagramPacket datagrama) throws IOException
```

Espera a que llegue un paquete UDP y lo almacena en el DatagramPacket pasado como argumento. Una vez recibido, los datos sobre la dirección y puerto serán los correspondientes a la máquina origen. Si no se ha establecido el tiempo de espera, este método bloquea la ejecución del programa hasta que se reciba el paquete UDP.

```
public void setReceiveBufferSize(int long) throws SocketException
```

Establece el tamaño del buffer para los paquetes UDP de entrada. Si este valor es pequeño y llegan muchos paquetes UDP en un corto espacio de tiempo es posible que alguno de ellos se pierda.

```
public void setSendBufferSize(int long) throws SocketException
```

Establece el tamaño del buffer para los datagramas UDP de salida.

```
public void setSoTimeout(int duracion)
```

Establece el tiempo de espera del socket. El valor es el número de milisegundos que el método receive() se esperará antes de lanzar una excepción del tipo SocketTimeoutException.

```
public void close()
```

Cierra el socket y libera el puerto.

La estructura básica que debe tener una aplicación para la creación y el envío de paquetes UDP se muestra en el siguiente código.

```
DatagramSocket socket = new DatagramSocket();
DatagramPacket paquete = new DatagramPacket(new byte[1024],1024);

paquete.setAddress(InetAddress.getByName(maquina));
paquete.setPort(2000);

boolean finalizar = false;

while(!finalizar){
   // Asignar los datos al buffer del paquete UDP
   ...
   socket.send(paquete);
```

```
    ...
}

socket.close();
```

La estructura básica que debe tener una aplicación para la recepción de paquetes UDP a través del puerto 2000 se muestra en el siguiente código.

```
DatagramSocket socket = new DatagramSocket(2000);
DatagramPacket paquete = new DatagramPacket(new byte[1024],1024);

boolean finalizar = false;

while(!finalizar){
   socket.receive(paquete);

   // Extraer los datos del paquete UDP
   ...

}

socket.close();
```

Al procesar el paquete la aplicación debe trabajar con el array de bytes que contiene los datos. Sin embargo, a partir de los datos es posible construir un ByteArrayInputStream y utilizar éste para obtener un flujo de entrada filtrado.

2.1. Ejemplo

Supongamos que para monitorizar durante 40 segundos la trayectoria seguida por un cohete durante su lanzamiento, se obtienen datos enviados desde el cohete y datos de diversas estaciones de radar. Cada estación de radar proporciona datos cada 100 milisegundos y los datos del cohete están disponibles cada 20 milisegundos. Todos estos datos deben ser enviados a una máquina encargada de procesarlos y almacenarlos.

Vamos a suponer que los datos que envía el cohete son: velocidad horizontal y vertical, altitud y temperatura. Cada estación de radar proporciona a su vez el dato de la distancia a la que se encuentra el cohete respecto de la estación de radar.

En el directorio `Ejemplos/UDP/Cohete` hay una serie de clases que permiten realizar una simulación utilizando 20 estaciones de radar.

Los datos que proporciona esta simulación no son realistas ya que la parte relevante es la construcción, envío y obtención de los paquetes UDP.

3. TCP

Es un protocolo de la capa de transporte que las aplicaciones pueden utilizar para enviar paquetes de datos y que garantiza la entrega en el mismo orden en el que se han enviado.

Desde el punto de vista de la aplicación, TCP transfiere un flujo continuo de bytes (por lo que se pueden utilizar los flujos de Entrada/Salida) a través de Internet. La aplicación no ha de preocuparse de dividir los datos en bloques.

El coste de la garantía de entrega y de la ordenación de los datos es la reducción del rendimiento en la red.

Java proporciona dos clases para el trabajo con el protocolo TCP:

- java.net.Socket
- java.net.ServerSocket

La clase Socket se utiliza para crear el cliente que se conecta a un determinado servicio.

La clase ServerSocket se utiliza para crear el servidor asociado a un puerto.

La clase `Socket` representa un canal de comunicación entre dos puertos de comunicación TCP que pertenecen a una o dos máquinas. Un socket TCP no puede comunicar más de dos máquinas.

Si se requiere esta funcionalidad el cliente debe establecer múltiples conexiones, una por cada máquina.

Esta clase ofrece diversos constructores entre los que cabe destacar:

```
Socket(InetAddress dir, int puerto)
```

Crea un socket conectado a la IP y puerto especificados.

```
Socket(InetAddress dir, int puerto,InetAddress localdir, int puertoLocal)
```

Crea un socket conectado la IP y puerto especificados y es asociado a la IP local y al puerto proporcionados.

```
Socket(String maquina, int puerto)
```

Crea un socket conectado la máquina y puerto especificados.

```
Socket(String maquina, int puerto,InetAddress local, int puertoLocal)
```

Crea un socket conectado la maquina y puerto especificados y es asociado a la IP local y al puerto proporcionados.

Una vez que se ha creado el socket las operaciones habituales que se pueden realizar con el objeto son: leer información, enviar datos, establecer opciones y cerrar la conexión.

Vamos a ver cuales son algunos de los métodos que ofrece esta clase.

```
public OutputStream getOutputStream() throws IOException
```

Devuelve un flujo de salida que se puede utilizar para escribir datos que se envían a la aplicación a la que está conectado este socket.

```
public InputStream getInputStream() throws IOException
```

Devuelve un flujo de entrada que se puede utilizar para leer datos de se reciben de la aplicación a la que está conectado este socket.

```
public void setReceiveBufferSize(int tam) throws SocketException
```

Asigna a la opción SO_RCVBUF el valor especificado. Esta opción es utilizada para establecer el tamaño del buffer de red para la entrada.
Para conocer el valor por defecto se puede utilizar getReceiveBufferSize().
La llamada a este método se debe realizar realizar antes de conectar el socket.

```
public void setSendBufferSize(int tam) throws SocketException
```

Asigna a la opción SO_SNDBUF el valor especificado. Esta opción es utilizada para establecer el tamaño del buffer de red para la salida.
Para conocer el valor por defecto se puede utilizar getSendBufferSize().

```
public void setSoTimeout(int duracion) throws SocketException
```

Establece el valor de la opción SO_TIMEOUT que controla el tiempo en milisegundos que bloquearán las operaciones de lectura. Un valor de cero bloquea de forma indefinida, en otro caso al transcurrir el tiempo especificado y no recibir datos, se lanza una excepción del tipo SocketTimeoutException.

```
public void setTcpNoDelay(boolean flag) throws SocketException
```

Establece el valor de la opción TCP_NODELAY que determina si se debe utilizar el algoritmo de Nagle. [1]

```
public void shutdownInput() throws IOException
```

Cierra el flujo de entrada de el socket, de forma que las operaciones de lectura devolverán el marcador de final de fichero.

```
public void shutdownOutput() throws IOException
```

Cierra el flujo de salida de el socket, de forma que las operaciones de escritura lanzarán una excepción del tipo IOException.

```
public void close() throws IOException
```

Cierra el socket. Una vez cerrado, no se puede volver a conectar. Se necesitaría crear un nuevo socket.

Para el resto de métodos consultad el API.

En cuanto a la clase ServerSocket el constructor más sencillo es:

```
ServerSocket(int puerto) throws IOException
```

Crea un socket servidor asociado al puerto que se pasa como argumento de forma que los clientes puedan localizar el servicio TCP que ofrece. Si el puerto ya está asociado se lanza una excepción.

Una vez que se tiene un objeto de este tipo, se puede obtener un Socket para modelar la conexión entre el cliente y el servidor mediante el método:

```
public Socket accept() throws IOException
```

Espera a que el socket cliente realice una petición de conexión y devuelve el socket correspondiente. Es una operación que bloquea hasta que no se produzca una conexión (a no ser que se haya establecido la opción SO_TIMEOUT

Para el resto de métodos consultad el API.

En el directorio `Ejemplos/TCP` hay un ejemplo de uso de estas clases.

4. HTTP

4.1. La clase URL

Esta clase sirve para representar uno de los tipos más usados de dirección en Internet: el *Uniform Resource Locator*.

Las direcciones URL pueden referirse a ficheros, sitios Web, sitios FTP, direcciones de correo electrónico, etc.

La estructura de estas direcciones es:

protocolo	: //	*hostname*	(: *puerto*)	*ruta*	(#*referencia*)

La clase URL dispone de varios constructores entre los que cabe destacar:

```
URL(String cadena) throws MalformedException
```

Crea un objeto URL a partir de la dirección pasada en forma de String.

```
URL(String protocolo, String host, String file) throws MalformedException
```

Crea un objeto URL a partir del protocolo, la máquina y el recurso pasados en forma de String.

```
URL(String protocolo, String host, int puerto, String file) throws MalformedException
```

Crea un objeto URL a partir del protocolo, la máquina y el recurso pasados en forma de String, y el puerto pasado como un entero.

```
URL(URL contexto, String relativo) throws MalformedException
```

Crea un objeto URL a partir una ruta relativa a la URL pasada como primer argumento.

```
URL(String protocol, String host, int port, String file, URLStreamHandler handler) throws
    MalformedException
```

Crea un objeto URL donde se especifica un objeto para manejar el protocolo que se utilizará durante la conexión. Por ejemplo un manejador de protocolo HTTP conoce cómo comunicarse con un servidor HTTP y un manejador de protocolo FTP conoce cómo comunicarse con un servidor FTP. Si se va a utilizar un protocolo que no está implementado, se debe proporcional el URLStreamHandler

Algunos de los **métodos** de los que dispone esta clase son:

```
String getProtocol()
```

Devuelve el protocolo asociado al URL

```
String getHost()
```

Devuelve la máquina asociada al URL

```
int getPort()
```

Devuelve el puerto asociado al URL

```
URLConnection openConnection() throws IOException
```

Devuelve un objet URLConnection que representa una conexión con el recurso remoto al que hace referencia el objeto URL.

```
InputStream openStream() throws IOException
```

Abre una conexión con el URL y devuelve un objeto del tipo InputStream para leer a través de esa conexión.

Si se necesita controlar la forma en la que realiza la petición del recurso se debe utilizar el método openConnection(). Si lo único que se desea hacer es leer el recurso se puede utilizarn openStream().

4.2. La clase URLConnection

La clase URLConnection es la superclase abstracta de todas las clases que representan la comunicación entre la aplicación y un URL.

Un objeto de este tipo se puede utilizar para leer desde y para escribir hacia el el recurso al que hace referencia el objeto URL.

En general la creación de una conexión implica los siguientes pasos:

1. Se obtiene un objeto URLConnection utilizando el método openConnection() de un objeto URL.
2. Se establecen los parámetros y las propiedades de la petición.
3. Se establece la conexión utilizando el método connect()
4. Se puede obtener información sobre la cabecera o/y obtener el recurso remoto.

Algunos de los métodos de esta clase son:

```
void connect() throws IOException
```

Establece una conexión entre la aplicación (el cliente) y el recurso (servidor). Antes de invocar a este método la aplicación puede especificar las propiedades de la cabecera de la petición.

```
int getContentLength()
```

Devuelve el valor del campo de cabecera *content-length* o -1 si no está definido.

```
String getContentType()
```

Devuelve el valor del campo de cabecera *content-type* o null si no está definido.

```
String getHeaderField(String campo)
```

Devuelve el valor del campo de cabecera especificado o null si no está definido.

```
OutputStream getOutputStream() throws IOException
```

Devuelve un objeto OutputStream para escribir datos en esta conexión.

```
InputStream getInputStream() throws IOException
```

Devuelve un objeto InputStream para leer datos de esta conexión.

```
void setRequestProperty(String key,String value)
```

Establece una propiedad de la petición. Por ejemplo:
setRequestProperty (Üser-agent", .ªgente");

4.3. La clase HttpURLConnection

Es una subclase de URLConnection que soporta características propias del protocolo HTTP.

Esta clase no tiene constructor, la forma de obtener una referencia a un objeto de este tipo es:

```
URL direccion = new URL( dir );
URLConnection conexion = url.openConnection();

// Comprobamos su tipo
if (conexion instanceof HttpURLConnection ){

   // Se realiza un cast
   HttpURLConnection conexionHTTP = (HttpURLConnection) conexion;

   // Resto del codigo
}
```

Algunos de los métodos que define esta clase (además de los de URLConnection) son:

```
void disconnect()
```

Si la conexión con el servidor Web está activa se cierra

```
void setRequestMethod(String metodo) throws ProtocolException
```

Establece el método de petición que se utilizará en esta conexión (POST, GET, HEAD,...)
Debe ser utilizado antes de llamar a `connect()`.

```
boolean usingProxy()
```

Muestra si se está utilizando un servidor *proxy* para la conexión.

```
int getResponseCode()
```

Obtiene el código de estado del mensaje de respuesta HTTP. Por ejemplo para las siguientes líneas de estado:

HTTP/1.0 200 OK
HTTP/1.0 401 Unauthorized

Se obtendría 200 y 401 respectivamente.
Devuelve -1 si no se puede obtener el código a partir de la respuesta.