Comunicaciones en Java

Para la comunicación en red, Java ofrece dispone de las clases del paquete **java.net**. Podemos dividir las clases end os grupos:

- Una API de bajo nivel, que permite gestionar los identificadores y las direcciones de red (las IP), los **sockets** y las interfaces de red.

- Una API de alto nivel que permite gestionar los identificadores de recursos universales (URI), los localizadores de recursos universales (URL), y gestionar las conexiones entre diferentes URL.

InetAddress

En Java, los dispositivos de red se representan mediante las clases **InetAddress**, o sus diferentes subclases **Inet4Address** o **Inet6Address**. Entre los métodos más importantes de esta clase tenemos:

getAddress	Retorna la IP
getByAddress	Retorna un InetAddress a partir de una IP
getByName	Proporciona la IP de un host a partir del nombre
getHostAddress	Proporciona la IP de un InetAddress
getHostName	Retorna el nombre del host de un InetAddress
getLocalHost	Retorna la IP del host local

Un ejemplo del uso de InetAddress:

```
InetAddress inetAddress1 = InetAddress.getByName("www.google.es");

InetAddress inetAddress2 = InetAddress.getByName("192.168.11.15");
```

SocketAddress

Clase abstracta que representa la dirección de un Socket.

InetSocketAddress

Subclase de **SocketAddress**, representa la dirección de un Socket formada por su IP y su Puerto.

ServerSocket

Esta clase representa un socket de un servidor. Este tipo de socket, cuando son creados, permanecen a la espera de recibir peticiones de los clientes. Cuando reciben una, generan un socket para atender a dichas peticiones de comunicación.

accept	Recibe peticiones y proporciona sockets
bind	Asocia un socket a una petición
close	Cierra el socket
isBound	Devuelve el estado de asociación del socket
isConnected	Determina si el socket está conectado

Socket

Esta clase representa un socket cliente.

bind	Asocia un socket a una petición
close	Cierra el socket
connect	Conecta el socket
getInputStream	Proporciona un Stream de lectura
getOutputStream	Proporciona un Stream de escritura
isBound	Devuelve el estado de asociación del socket
isClosed	Determina si el socket está cerrado
isConnected	Determina si el socket está conectado

DatagramPacket

Representa un Datagrama (UDP).

DatagramSocket

Esta clase representa un socket para el envío y recepción de datagramas (UDP).

bind	Asocia un socket a una petición
close	Cierra el socket
connect	Conecta el socket
diconnect	Desconecta el socket
isBound	Devuelve el estado de asociación del socket
isClosed	Determina si el socket está cerrado
isConnected	Determina si el socket está conectado
receive	Recibe un datagrama
send	Envía un datagrama

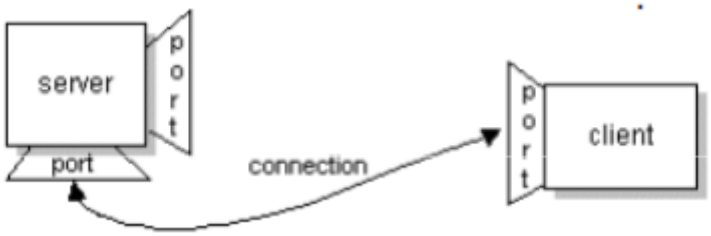
Sockets TCP en Java

Un **Socket TCP** es un punto de comunicación por el que un proceso puede emitir o recibir información. Cada Socket tiene una **dirección IP** y un **puerto** asociado, así que para que un Cliente se comunique con un Servidor necesita conocer el puerto y la IP del Servidor; y viceversa.

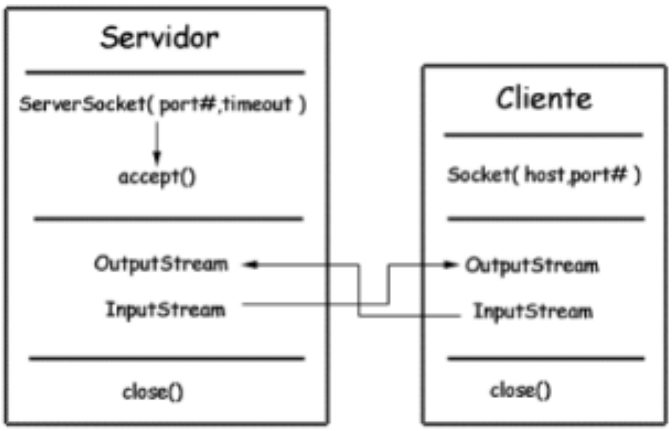
Normalmente, un **Servidor** se ejecuta sobre una máquina y tiene un **ServerSocket** que responde por un puerto específico. El Servidor normalmente se pone en espera, es decir, escucha a través de ese **ServerSocket** a que uno o más **Clientes** le hagan peticiones.

En el lado del **Cliente**, cada uno de ellos debe conocer la IP y puerto de la máquina **Servidor**. Para realizar una petición de conexión, el **Cliente** intenta encontrar el puerto especificado en el **Servidor**. Si todo va bien, el **Servidor** acepta la conexión.

El **Servidor** obtiene un nuevo **Socket** generado por su **ServerSocket**, que utiliza para procesar la petición. Para evitar bloquear al **Servidor**, se utilizan hilos para liberar al **ServerSocket** y que pueda seguir atendiendo las necesidades de los **Clientes**.



Por la parte del **Cliente**, si la conexión es aceptada, se crea un **Socket** y puede seguir usándolo para comunicarse con el Servidor.



Si estamos en el lado **Cliente**, el Socket se crea de la siguiente forma, donde máquina es el nombre de la máquina destino (o la dirección IP) y numeroPuerto es el puerto por el que escucha el Servidor.

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

Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (root). Estos puertos son los que utilizan los servicios estándar del sistema, como email, ftp o http. Para las aplicaciones que se desarrollen, asegurarse deseleccionar un puerto por encima del 1023.

Dado que existen muchos problemas a la hora de enviar un mensaje a través de una red, se tienen que controlar excepciones.

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

En el lado **Servidor**, la apertura del Socket se puede hacer de forma similar; aunque no utilizar ServerSocket nos complica bastante las cosas.

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

Para crear un Socket desde un ServerSocket que **acepte** conexiones, se hace:

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

En la parte del **Cliente**, 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 );
}
```

Hay diferentes tipos de Streams que podemos utilizar para comunicar dos Sockets. EN este caso usamos **DataInputStream**, que permite la lectura de líneas de texto y de tipos primitivos de datos en Java.

En el lado **Servidor** también se usará **DataInputStream**, pero para leer de los **Clientes**.

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

En el lado del **Cliente**, creamos un Stream de salida para enviar información al **Servidor** utilizando las clases **PrintStream** o **DataOutputStream**

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

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.

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.

Cuando trabajamos con **Sockets**, siempre se deben de cerrar los canales de E/S que se hayan abierto, y de forma inversa a su creación. Se deben cerrar primero los Streams relacionados con un Socket antes que el propio socket, ya que de esta forma evitamos posibles errores de escrituras o lecturas sobre descriptores ya cerrados.

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

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