Contenido

[1. INTRODUCCIÓN 2](#_Toc57788273)

[2. CONCEPTOS BÁSICOS DE REDES 2](#_Toc57788274)

[3. CLASES JAVA PARA COMUNICACIONES EN RED 5](#_Toc57788275)

[3.1. Clase InetAddress 5](#_Toc57788276)

[3.2. Clase URL 7](#_Toc57788277)

[3.3. Clase URLConnection 10](#_Toc57788278)

[4. SOCKETS 13](#_Toc57788279)

[4.1. Funcionamiento de un socket 13](#_Toc57788280)

[4.2. Tipos de Sockets 14](#_Toc57788281)

[4.3. Sockets TCP u orientados a conexión 14](#_Toc57788282)

[4.3.1. Clases Java 14](#_Toc57788283)

[4.3.2. Gestión de sockets TCP 18](#_Toc57788284)

[4.4. Sockets UDP o no orientados a conexión 24](#_Toc57788285)

[4.4.1. Clases Java 24](#_Toc57788286)

[4.4.2. Gestión de Sockets UDP 27](#_Toc57788287)

[4.3.3. Clase Java MulticastSocket 31](#_Toc57788288)

[4.5. Envío de objetos a través de Sockets 35](#_Toc57788289)

[4.5.1. Objetos en Sockets TCP 35](#_Toc57788290)

[4.5.2. Objetos en sockets UDP 38](#_Toc57788291)

[5. CONEXIONES MÚLTIPLES CLIENTES. HILOS 39](#_Toc57788292)

# INTRODUCCIÓN

Antiguamente la programación de aplicaciones entre diferentes máquinas era difícil, compleja y fuente de muchos errores; el programador tenía que conocer detalles sobre las capas del protocolo de red, incluso sobre el hardware de la máquina. Java ha trabajado en el desarrollo de diferentes librerías con las que ha conseguido que desarrollar aplicaciones en red no sea una tarea tan compleja.

Actualmente Java dispone de clases para establecer conexiones, crear servidores, enviar y recibir datos y para el resto de operaciones utilizadas en las comunicaciones a través de redes de ordenadores. Además, el uso de hilos, que se trataron en la unidad didáctica anterior, nos va a permitir la manipulación simultánea de múltiples conexiones.

# CONCEPTOS BÁSICOS DE REDES

**PROTOCOLOS TCP/IP**

En Internet se utiliza la pila de protocolos TCP/IP para el establecimiento y realización de conexiones, la cual está basada en un conjunto de protocolos organizados en diferentes niveles o capas.

|  |
| --- |
| **Capa de aplicación**  (HTTP, ftp, telnet,…) |
| **Capa de transporte**  (TCP, UDP) |
| **Capa de Red**  (IP) |
| **Capa de Enlace o Interfaz de red**  (Ethernet, ...) |

*Figura 1.1 – Capas del Modelo TCP/IP*

La **capa de aplicación**: en este nivel se encuentran las aplicaciones disponibles para los usuarios. Por ejemplo: FTP, SMTP, Telnet, HTTP, etc.

La **capa de transporte**: suministra a las aplicaciones servicio de comunicaciones de extremo a extremo, utilizando dos tipos de protocolos: TCP (Transmision Control Protocol) y UDP (User Datagram Protocol).

La **capa de Red:** tiene como propósito seleccionar la mejor ruta para enviar los paquetes por la red. El protocolo principal que funciona en esta capa es el Protocolo de Internet (IP).

La **capa de Enlace**: es la interfaz con la red real; recibe los datagramas de la capa de red y los transmite al hardware de red.

Normalmente, cuando se escriben aplicaciones Java que se comunican a través de la Red se está programando en la capa de aplicación. Como veremos en esta unidad didáctica es posible realizar programas a más bajo nivel utilizando el paquete *java.net* de la API de Java, el cual está compuesto por una serie de clases de bajo nivel y otro por clases de alto nivel.

**DIRECCIONES IP Y PUERTOS**

Un ordenador tiene una o varias conexiones físicas a la red. A través de esas conexiones recibe los datos dirigidos a la máquina. Las **direcciones IP** nos permiten identificar el interface de la máquina a la que queremos enviar los datos mientras que los **puertos** nos indican la aplicación a la que queremos enviar los datos. TCP y UDP utilizan los puertos para dirigir los datos a la aplicación correcta de entre todas las que se estén ejecutando en la máquina.

Por lo tanto, los datos transmitidos a través de Internet (o de una red local), contienen información de direccionamiento que identifica a la máquina y al puerto a los que van dirigidos:

* La máquina se identifica a través de una dirección IP de 32 bits.
* Los puertos se identifican por un número de 16 bits.

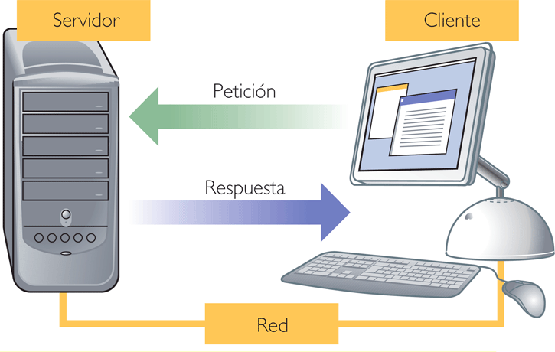
Los puertos utilizan 16 bits (0 a 65535) para identificarse. Estos se utilizan tanto para el protocolo TCP (orientado a conexión) como para el protocolo UDP (no orientado a conexión) y del 0 al 1023 están reservados para aplicaciones concretas.

Información de detalle sobre el uso que se les da a los puertos está disponible en el fichero /etc/services (en sistemas operativos Linux) o en el fichero C:\Windows\System32\drivers\etc (en sistemas operativos Windows).

**CLIENTES Y SERVIDORES**

Muchas de las aplicaciones existentes hoy en día utilizan el modelo Cliente-Servidor, donde:

* El **Cliente**: programa que ejecuta el usuario y que solicita servicio al Servidor.
* El **Servidor**: ofrece servicio a múltiples Clientes.



*Figura 1.2 – Modelo cliente-servidor*

Algunos ejemplos de aplicaciones que utilizan este modelo son: telnet, ftp, web, ssh. Un cliente ejecuta el comando telnet contra un servidor en el que está corriendo el demonio telnetd. O un cliente utiliza un programa cliente (navegador) y accede a un servidor que tiene corriendo un servidor web (Apache).

**CAPA DE TRANSPORTE: TCP vs UDP**

Los equipos conectados a Internet se comunican entre sí utilizando los protocolos TCP o UDP. Cuando se escriben programas Java que se comunican a través de la red, se está programando en la capa de aplicación. Normalmente, no es necesario preocuparse por las capas TCP, para lo que se pueden utilizar las clases del paquete **java.net**. Sin embargo existen diferencias entre una y otra para decidir qué clases usar en los programas:

* **TCP: Transmission Control Protocol**
  + Protocolo orientado a conexión.
  + Provee un flujo de bytes fiable entre dos ordenadores.
  + Llegada en orden, correcta, sin perder nada. Si no fuese así se notificaría un error.
  + Protocolos del nivel de aplicación que usan TCP: telnet, http, ftp.
* **UDP: User Datagram Protocol**
  + No orientado a conexión.
  + Envía paquetes de datos (datagramas) independientes, sin garantías de que vayan a llegar al destino.
  + Permite broadcast.
  + Protocolos del nivel de aplicación que usan UDP: tftp, ping.

# CLASES JAVA PARA COMUNICACIONES EN RED

El paquete **java.net** proporciona las clases para la implementación de aplicaciones de red. Se pueden dividir en dos secciones:

* Una **API de bajo nivel**, que se ocupa de las siguientes abstracciones:
  + Direcciones: son los identificadores de red, como por ejemplo las direcciones IP.
  + Sockets: son los mecanismos básicos de comunicación bidireccional de datos.
  + Interfaces: describen las interfaces de red.
* Una **API de alto nivel**, que se ocupa de las siguientes abstracciones:
  + URIs: representan identificadores de recursos universales.
  + URLs: representan los localizadores de recursos universales.
  + Conexiones: representan las conexiones al recurso apuntado por la URL.

Mientras que la API de bajo nivel se usa por las aplicaciones cliente/servidor basadas en protocolos, la API de alto nivel es utilizada en el acceso a los recursos de la red.

El paquete java.net proporciona, entre otras, las siguientes clases:

* Clase **InetAddress**: para manipular direcciones IP y nombres de dominio.
* Clases **URL, URLConnection, ServerSocket, Socket**: utilizan el protocolo TCP para comunicaciones de red.
* Clases **DatagramPacket, DatagramSocket, MulticastSocket**: utilizan el protocolo UDP para comunicaciones de red.

## Clase InetAddress

Proporciona objetos que se pueden utilizar para manipular tanto direcciones IP como nombres de dominio. Se encuentra dentro de la clase **java.net.InetAddress**.

Las instancias de esta clase no se crean llamando a un constructor (de hecho no tiene constructores), sino llamando a los siguientes **métodos**, los cuales devuelven un objeto de tipo InetAddress:

* **InetAddress getByName**(String host)
  + Obtiene la dirección IP de la máquina
  + Recibe el nombre de la máquina o su dirección IP como cadena
* **InetAddress[] getAllByName**(String host)
  + Obtiene todas las direcciones IP de una máquina
  + Recibe el nombre de la máquina o su dirección IP como cadena
* **InetAddress getLocalHost**():
  + Obtiene la dirección IP de la máquina en la que se está ejecutando

Los tres métodos anteriores lanzan la excepción **UnknowHostException**, en caso de que el nombre o la dirección de la máquina no se encuentre o no exista en Internet.

Además de estos 3 métodos, existen otros métodos adicionales de esta clase:

* **public String getHostName()**
  + Devuelve el nombre del host correspondiente a un objeto InetAddress
* **public String getHostAddress()**
  + Devuelve la dirección IP correspondiente a un objeto InetAddress
* **public String getCannonicalHostName()**
  + Devuelve el nombre canónico completo de la máquina de un objeto InetAddress. Suele ser la dirección real del host

**Ejemplo 1 – Uso de la clase InetAddress:**

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**class** Ejemplo1 {

**public** **static** **void** main( String[] args ) {

InetAddress address;

**try** {

System.***out***.println( "Direccion IP de una URL, por nombre" );

address = InetAddress.*getByName*( "www.google.com" );

System.***out***.println( address );

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

System.***out***.println( "Direccion IP actual de LocalHost" );

address = InetAddress.*getLocalHost*();

System.***out***.println( address );

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Resultado:**

Direccion IP de una URL, por nombre

www.google.com/216.58.210.228

Direccion IP actual de LocalHost

DM2-PROF/172.20.105.120

## Clase URL

La clase **URL** (Uniform Resource Locator) representa un puntero a un recurso en la Web. Un recurso puede ser algo tan simple como un fichero o directorio, o puede ser una referencia a un objeto más complejo, como una consulta a una base de datos o a un motor de búsqueda.

En general una URL se divide en varias partes. Por ejemplo en la siguiente URL:

<http://www.google.es/DIR/Servicios/listado.php>,

aparece el protocolo (http), el nombre de la máquina ([www.google.es](http://www.google.es)) y el fichero (listado.php) que está en el directorio (DIR/Servicios).

Una URL puede especificar opcionalmente un puerto (punto de destino para la comunicación dentro de una máquina) para realizar la conexión TCP. Por defecto para el protocolo HTTP, es el puerto 80.

**Constructores**

Java permite los siguientes constructorespara la clase URL:

* public **URL(String url)** throws MalformedURLException;

Crea un objeto URL a partir del String url

* public **URL(String protocolo,String host,int port,String file)** throws MalformedURLException;

Crea un objeto URL a partir de los parámetros protocolo, host, puerto y fichero.

* public **URL(String protocol,String host,String file)** throws MalformedURLException;

Crea un objeto URL a partir de los parámetros protocolo, host y fichero.

* Public **URL( URL context,String url)** throws MalformedURLException;

Crea un objeto URL a partir de la dirección del host dada por URL context y una URL relativa dada en el String (un directorio).

**Ejemplos:**

Así que podríamos especificar todos los componentes de la URL como en los siguientes ejemplos:

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

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

**Métodos**

Esta clase URL contiene diferentes **métodos** que permiten obtener el siguiente tipo de información:

* publicString **getPath():**devuelve la URL de la página.
* public String **getQuery()**: devuelve parte de una consulta de la URL.
* public int **getPort()**: devuelve el puerto de la web.
* public int **getDefaultPort()**: devuelve el puerto por defecto para esa web.
* public **getProtocol()**: devuelve el protocolo de la URL.
* public String **getHost():** devuelve el host de la URL.
* public String **getFile():** devuelve el nombre del archivo de la URL.
* public String **getRef():** devuelve la referencia de la URL.
* **InputStream openStream():** abre una conexión stream por la cual se puede leer el código HTML de la URL.

**Ejemplo 2 – Uso de los constructores de la clase URL.** Crear 4 objetos URL a partir de los 4 constructores de la clase URL. Además, mediante el método *openStream()* abrir una conexión stream con la última URL y visualizar el contenido del fichero *gpl.txt.*

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.net.MalformedURLException;

**import** java.net.URL;

**public** **class** Ejemplo2 {

**public** **static** **void** main(String[] args) {

**try** {

// Obtener URL Absoluta

// Base URL = www.gnu.org

URL url1 = **new** URL("http://www.gnu.org");

System.***out***.println("URL1: " + url1.toString());

// Generar URL a partir de una URL base

URL url2 = **new** URL(url1, "licenses/gpl.txt");

System.***out***.println("URL2: " + url2.toString());

// Generar URL a partir del protocolo, host y fichero

URL url3 = **new** URL("http", "www.gnu.org", "/licenses/gpl.txt");

System.***out***.println("URL3: " + url3.toString());

// Generar URL a partir del protocolo, host, puerto y fichero

URL url4 = **new** URL("http", "www.gnu.org", 80, "/licenses/gpl.txt");

System.***out***.println("URL4: " + url4.toString() + "\n");

// Abrir el buffer de lectura a partir de la URL

**try** (BufferedReader in = **new** BufferedReader(**new** InputStreamReader(url4.openStream()))) {

String inputLine;

// Leemos el fichero "gpl.txt"

System.***out***.println("/\*\*\*\*\* Contenido del fichero URL4 \*\*\*\*\*/n");

**while**((inputLine = in.readLine()) != **null**) {

System.***out***.println(inputLine);

}

} **catch** (IOException ioe) {

ioe.printStackTrace(System.***err***);

}

} **catch** (MalformedURLException mue) {

mue.printStackTrace(System.***err***);

}

}

}

**Resultado:**

URL1: http://www.gnu.org

URL2: http://www.gnu.org/licenses/gpl.txt

URL3: http://www.gnu.org/licenses/gpl.txt

URL4: http://www.gnu.org:80/licenses/gpl.txt

/\*\*\*\*\* Contenido del fichero URL4 \*\*\*\*\*

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

…

## Clase URLConnection

Es una **clase abstracta** con métodos que permiten la comunicación entre una aplicación y una URL. Para conseguir un objeto de este tipo, se invoca al **método openConnection()**, con lo que se obtiene una conexión con el objeto URL referenciado. Las instancias de esta clase se pueden utilizar tanto para leer como para escribir en el recurso referenciado por la URL. Puede lanzar la excepción IOException.

**Ejemplo de creación de objetos de la clase:**

URL url = new URL(“<http://www.google.com>”);

URLConnection urlCon = url.openConnection();

Una vez que se dispone del objeto de la **clase URL**, si se invoca al método openConnection para realizar la comunicación con el objeto y la conexión se establece satisfactoriamente, entonces se dispone de una instancia de un objeto de la clase **URLConnection**.

Con el **comando telnet** se puede establecer la conexión de forma similar a como lo hace esta clase abstracta.

**Métodos**

* **InputStream getInputStream():** devuelve un objeto InputStream para leer datos de la conexión.
* **OutputStream getOutputStream():** devuelve un objeto OutputStream para escribir datos de la conexión.
* **void setDoInput (boolean b):** permite que el usuario reciba datos desde la URL si el parámetro b es true (por defecto está establecido a true).
* **void setDoOutput (boolean b):** permite que el usuario envíe datos desde la URL si el parámetro b es true (al principio no está establecido).
* **void connect():** abre una conexión al recurso remoto si tal conexión no se ha establecido ya.
* **int getContentLength():** devuelve el valor del campo de cabecera content-lenght o -1 si no está definido.
* **String getContentType():** devuelve el valor del campo de cabecera content-type o null si no está definido.
* **long getDate():** devuelve el valor del campo de cabecera date ó 0 si no está definido.
* **long getLastModified():** devuelve el valor del campo de cabecera last-modified.
* **String getHeaderField(int n):** devuelve el valor enésimo campo de cabecera especificado o null si no está definido.
* **Map<String,List<String>> getHeaderFields():** devuelve una estructura MAP (estructura de Java que nos permite almacenar pares clave/valor) con los campos de cabecera. Las claves son cadenas que representan los nombres de los campos de cabecera y los valores son cadenas que representan los valores de los campos correspondientes.
* **URL getURL():** devuelve la dirección URL.

**Ejemplo 3 – Conexión con la URL www.euskadi.net**

**import** java.io.\*;

**import** java.net.\*;

**public** **class** Ejemplo3 {

**public** **static** **void** main(String argv[]) {

URL url=**null**;

**try** {

url = **new** URL ("http://www.euskadi.net");

URLConnection conex = url.openConnection();

System.***out***.println("Cargando "+url);

BufferedReader in;

DataInputStream datos = **new** DataInputStream(conex.getInputStream());

in = **new** BufferedReader (**new** InputStreamReader(datos));

String inputLine;

**while** ((inputLine = in.readLine()) != **null**)

System.***out***.println(inputLine);

in.close();

}

**catch** (MalformedURLException e) {

e.printStackTrace();

}

**catch** (UnknownHostException e) {

System.***out***.println("El host no existe o no responde");

}

**catch** (Exception e) {

e.printStackTrace();

}

}

}

**Resultado:**

Cargando http://www.euskadi.net

<!DOCTYPE html>

<html lang="eu">

<head>

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<!-- Directiva para la previsualizacización de la plantilla, se borrar, de las páginas -->

….

</body>

</html>

# SOCKETS

Un socket es un conector que permite la comunicación bidireccional entre dos procesos que se encuentran distribuidos. Por lo tanto, los sockets permiten una comunicación **bidireccional** entre dos programas que se comunican por la red.

Los procesos receptores de mensajes (los Servidores) deben de tener:

* La **dirección IP** del host en el que la aplicación está corriendo.
* Un **puerto local** a través del cual la aplicación se comunica y que identifica al proceso.

De esta forma todas las conexiones realizadas con esa dirección IP y ese puerto llegarán al proceso receptor.

**Cliente Servidor**

Cualquier puerto –---------------------> Puerto acordado

Los procesos pueden utilizar un mismo conector (socket) tanto para enviar como para recibir mensajes. Cada conector se asocia a un protocolo concreto que puede ser TCP o UDP.

## Funcionamiento de un socket

Un puerto es un punto de destino que identifica hacia qué aplicación o proceso deben dirigirse los datos. Normalmente en una aplicación cliente-servidor, el programa servidor se ejecuta en una máquina específica y tiene un socket que está unido a un número de puerto específico. El servidor queda a la espera, “escuchando” las solicitudes de conexión de los clientes por ese puerto.

El programa cliente conoce el nombre de la máquina en la que se ejecuta el servidor así como el número de puerto por el que escucha las peticiones. Para realizar una solicitud de conexión, el cliente realiza la petición a la máquina servidor a través del puerto acordado. El cliente también debe identificarse ante el servidor, por lo que durante la conexión utilizará un puerto local asignado por el sistema:

**Cliente Servidor**

Petición de conexión PCA –----------------------------> PSA

Si todo va bien, el servidor acepta la conexión. Una vez aceptada, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que por un lado debe seguir atendiendo las peticiones de conexión mediante el socket original y, por otro, debe atender las necesidades del cliente que se conectó:

**Cliente Servidor**

PCB <---------------------------> PSB

En el lado del cliente, si se acepta la conexión, se crea un socket y el cliente puede utilizarlo para comunicarse con el servidor. Este socket utiliza un número de puerto diferente al usado para conectarse con el servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo respectivamente por sus respectivos sockets.

## Tipos de Sockets

Hay dos tipos básicos de sockets en redes IP: los que utilizan el protocolo TCP (orientados a conexión) y los que utilizan el protocolo UDP (no orientados a conexión).

## Sockets TCP u orientados a conexión

La comunicación entre las aplicaciones se realiza por medio del protocolo TCP. Por tanto es una conexión fiable en la que se **garantiza la entrega** de los paquetes de datos y el orden en que fueron enviados. TCP utiliza un esquema de acuse de recibo de los mensajes de tal forma que si el emisor no recibe dicho acuse dentro de un tiempo determinado, vuelve a transmitir el mensaje.

La gran mayoría de aplicaciones IP utilizan sockets TCP. Ejemplos de algunos servicios son: FTP (puerto 21), Telnet (puerto 23), HTTP (puerto 80), SMTP (puerto 25).

### Clases Java

Las clases que se van a analizar para trabajar con sockets TCP son las siguientes:

* **java.net.ServerSocket:** implementa un socket que los servidores pueden utilizar para escuchar y aceptar peticiones de clientes.
* **java.net.Socket:** implementa un extremo de una conexión bidireccional. Se utiliza para implementar un cliente.

1. **Clase ServerSocket**

Implementa los sockets de servidor. Un socket de servidor espera la llegada de peticiones de una red enviadas por los clientes de dicha red. Realiza algún tipo de operación basada en la petición y luego posiblemente le devuelve un resultado al cliente solicitante de la petición.

**Constructores**

* **ServerSocket ss=new ServerSocket():** crea un ServerSocketsin estar unido a nada.
* **ServerSocket ss=new ServerSocket(int puerto):** crea un ServerSocketa la escucha del puerto local indicado.
* **ServerSocket ss=new ServerSocket(int puerto, int máximo):** crea un ServerSocket a la escucha del puerto local indicado. El parámetro máximo indica el número máximo de peticiones de conexión.
* **ServerSocket ss=new ServerSocket(int puerto, int máximo, InetAddress direc):** crea un ServerSocket a la escucha del puerto local indicado, con un número máximo de peticiones entrantes y una dirección IP local.

**Métodos**

Algunos de los métodos de esta clase sonlos siguientes:

* **accept():** acepta las conexiones entrantes en el servidor montado por ServerSocket.
* **close():** se encarga de cerrar el ServerSocket.
* **getLocalPort():** devuelve el puerto local al que se enlaza el ServerSocket.

**Ejemplo 4 - Crear un socket de servidor, enlazándolo con el puerto 6000. Visualizar el puerto por el que se esperan las conexiones y esperar a que se conecten 2 clientes**

**import** java.io.IOException;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** Ejemplo4 {

**public** **static** **void** main (String[]args){

**int** Puerto = 6000;

**try** {

ServerSocket Servidor = **new** ServerSocket (Puerto);

System.***out***.println("Hay un servidor escuchando en el puerto " + Servidor.getLocalPort());

// Realiza la conexión con cliente 1

Socket cliente1=Servidor.accept();

// Realiza la conexión con cliente 2

Socket cliente2=Servidor.accept();

// Cierra la conexión

Servidor.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

1. **Clase Socket**

Implementa sockets de cliente (también llamados simplemente "sockets"). Un socket es un extremo de la comunicación entre dos máquinas.

**Constructores**

Algunos de los constructores de esta clase sonlos siguientes:

* **Socket s=new Socket():** crea un Socket por defecto sin ningún puerto asociado.
* **Socket s=new Socket(InetAdress direccion, int puerto):** crea y conecta un Socket a la IP especificada y el puerto especificado.
* **Socket s=new Socket(InetAdress direccion, int puerto, InetAdress direccionlocal, int puertolocal ):** crea y conecta un Socket a la IP especificada y puerto especificados, además de indicar la dirección local y puerto a los que está asociado.
* **Socket s=new Socket(String host, int puerto):** crea y conecta un Socket al host y al puerto especificados. Puede lanzar un UnKnownHostException e IOException.

**Métodos**

Algunos de los métodos más usados de esta clase sonlos siguientes:

* **void close():** cierra el Socket.
* **InetAddres getInetAddress():** retorna la IP a la cual está conectado el socket, si no lo está devuelve NULL.
* **InputStream getInputStream():** devuelve un flujo de datos de entrada para el socket.
* **OutputStream getOutputStream():** devuelve un flujo de datos de salida para el socket.
* **int getPort():** devuelve el puerto remoto al que está conectado el socket; (0 si no está enlazado a ningún puerto)
* **int getLocalPort():** devuelve el puerto local al que está conectado el socket; (-1 si no está enlazado a ningún puerto)
* **boolean isClosed():** devuelve true si el socket está cerrado y false si no lo está.
* **boolean isConnected():** devuelve true si el socket está conectado y false si no lo está.
* **void shutdownInput():** cierra el flujo de entrada del socket.
* **void shutdownoutput():** cierra el flujo de salida del socket.
* **boolean isInputShutdown():** devuelve true si el flujo de entrada está cerrado y false si no lo está.
* **boolean isOutputShutdown():** devuelve true si el flujo de salida está cerrado y false si no lo está.

**Ejemplo 5 - Crear un socket cliente conectándolo al host local por el puerto 6000 (tiene que haber un ServerSocket escuchando en ese puerto). Visualizar el puerto local, el host al que está conectado el socket, la dirección remota y el puerto remoto.**

**import** java.io.IOException;

**import** java.net.InetAddress;

**import** java.net.Socket;

**import** java.net.UnknownHostException;

**public** **class** Ejemplo5 {

**public** **static** **void** main (String[]args){

String Host="localhost";

**int** Puerto = 6000;

**try** {

// Abrimos Socket

Socket Cliente = **new** Socket (Host, Puerto);

InetAddress i = Cliente.getInetAddress();

System.***out***.println("Puerto local: "+ Cliente.getLocalPort());

System.***out***.println("Puerto Remoto: "+ Cliente.getPort());

System.***out***.println("Host Remoto: "+ i.getHostName());

System.***out***.println("IP Host Remoto: "+ i.getHostAddress().toString());

Cliente.close();

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Resultado:**

Puerto local: 51499

Puerto Remoto: 6000

Host Remoto: localhost

IP Host Remoto: 127.0.0.1

### Gestión de sockets TCP

El **modelo de socket** **TCP** más simple es el siguiente:



*Figura 4.1 – Modelo básico de socket TCP*

1.- El **servidor** se ejecuta en una máquina y crea un socket orientado a conexión ligado a un número de puerto específico mediante el método constructor **ServerSocket(port)**. El servidor espera mediante el método **accept()**, escuchando por ese puerto, a que los clientes hagan peticiones de conexión.

2.- El **cliente** envía una petición de conexión al servidor, usando esa máquina y número puerto mediante el método constructor **Socket(host, port)**. Además, el cliente tiene que identificarse ante el servidor; por tanto, al cliente se le asigna un puerto en su máquina, que será utilizado a lo largo de la conexión. Normalmente esta asignación la realiza el sistema.

3.- El **cliente y el servidor** se comunican con manejadores **InputStream** y **OutputStream**. El cliente escribe los mensajes en el OutputStream asociado al socket y el servidor leerá los mensajes del cliente en el InputStream. Igualmente el servidor escribirá los mensajes en el OutputStream y el cliente los leerá del InputStream.

4.- Cuando finaliza la conexión entre el cliente y servidor se **cierra la conexión**.

1. **Apertura de Sockets**

Si se está programando un **CLIENTE**, el socket se abre de la siguiente forma:

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

Donde maquina es el nombre de la máquina en donde se está intentando abrir la conexión y numeroPuerto es el puerto (un número) del servidor que está corriendo con el que se quiere establecer la conexión. 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 (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. 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 );}

Si se está programando un **SERVIDOR**, el socket se abre de la siguiente forma:

ServerSocket miServicio;

try {

miServicio = new ServerSocket( numeroPuerto );

} catch( IOException e ) { System.out.println( e ); }

A la hora de implementar un servidor también es necesario crear un objeto socket desde el ServerSocket, para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar dichas conexiones:

Socket socketServicio = null;

try {

socketServicio= miServicio.accept();

} catch( IOException e ) { System.out.println( e ); }

1. **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(). Será preciso seleccionar el método adecuado en función del tipo de dato que se espere recibir del servidor.

En el lado del **SERVIDOR**, también se usará la clase 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 ); }

1. **Creación de Streams de Salida**

En el lado del **CLIENTE**, se puede 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 se puede utilizar la clase 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**, se puede usar 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 se puede utilizar la clase DataOutputStream como en el caso de envío de información desde el cliente.

1. **Cierre de Sockets**

Siempre será preciso 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 ); }

Es importante destacar que el orden de cierre es relevante. Es decir, se deben cerrar primero los streams relacionados con un socket antes que el propio socket, ya que de esta forma se evitan posibles errores de escrituras o lecturas sobre descriptores ya cerrados.

**Ejemplo 6 – Un servidor recibe un mensaje de un cliente, lo muestra por pantalla y envía otro mensaje al cliente**

**//Servidor.java:**

**import** java.io.DataInputStream;

**import** java.io.DataOutputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.OutputStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** Servidor {

**public** **static** **void** main (String[] args) **throws** IOException {

**int** Puerto = 6000;

ServerSocket servidor = **new** ServerSocket (Puerto);

Socket clienteConectado = **null**;

System.***out***.println("Esperando al cliente.....");

clienteConectado = servidor.accept();

//// Creación del flujo de entrada del cliente

InputStream entrada = **null**;

entrada = clienteConectado.getInputStream();

DataInputStream flujoEntrada = **new** DataInputStream (entrada);

////// El cliente envía un mensaje al servidor

System.***out***.println("Recibiendo mensaje del cliente: \n\t" + flujoEntrada.readUTF());

/////// Flujo de salida hacia el cliente

OutputStream salida=**null**;

salida = clienteConectado.getOutputStream();

DataOutputStream flujoSalida = **new** DataOutputStream (salida);

////// Envio un saludo al Cliente

flujoSalida.writeUTF("Saludos al cliente desde el servidor");

////// Cerrar streams y sockets

entrada.close();

flujoEntrada.close();

salida.close();

flujoSalida.close();

clienteConectado.close();

servidor.close();

}

}

**//Cliente.java:**

**import** java.io.DataInputStream;

**import** java.io.DataOutputStream;

**import** java.io.IOException;

**import** java.net.Socket;

**import** java.net.UnknownHostException;

**public** **class** Cliente {

**public** **static** **void** main (String[] args) {

String Host="localhost";

**int** Puerto = 6000;

System.***out***.println("PROGRAMA CLIENTE INICIANDO");

**try** {

Socket Cliente = **new** Socket (Host, Puerto);

/////// Flujo de salida al servidor

DataOutputStream flujoSalida = **new** DataOutputStream (Cliente.getOutputStream());

////// Envío de un saludo al servidor

flujoSalida.writeUTF("Saludos al servidor desde el cliente");

//// Creación del flujo de entrada al servidor

DataInputStream flujoEntrada = **new** DataInputStream (Cliente.getInputStream());

////// El servidor envía un mensaje al cliente

System.***out***.println("Recibiendo mensaje del servidor: \n\t" + flujoEntrada.readUTF());

////// Cerrar streams y sockets

flujoEntrada.close();

flujoSalida.close();

Cliente.close();

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

## Sockets UDP o no orientados a conexión

Los sockets UPD son más simples y eficientes que los TCP, pero no está garantizada la entrega de paquetes. No es necesario establecer una “conexión” entre cliente y servidor, como en el caso de TCP. Para ello, cada vez que se envíen datagramas, el emisor debe indicar explícitamente la dirección IP y el puerto del destino de cada paquete, y el receptor debe extraer la dirección IP y el puerto del emisor del paquete.

Este tipo de conexión no es fiable, no se garantiza que un mensaje que se ha enviado llegue a su destino. No existe acuse de recibo y el orden de llegada puede ser diferente al de envío.

Un **Datagrama** es un mensaje independiente, enviado a través de una red cuya llegada, tiempo de llegada y contenido no están garantizados. El paquete datagrama está formado por los siguientes campos:

* Cadena de bytes conteniendo el mensaje.
* Longitud del mensaje.
* Dirección IP destino
* Número de puerto destino.

Los sockets UDP se usan cuando es más importante una **entrega rápida** que una entrega garantizada, o en los casos en que se desee enviar tan poca información que cabe en un único datagrama. Se usan en:

* Las transmisiones de audio y vídeo en tiempo real donde no es posible el reenvío de paquetes retrasados.
* La aplicación NFS (Network File System).
* Los protocolos DNS (Domain Name Server) o SNMP (Simple Network Management Protocol).
* El comando RCP que permite copiar archivos o directorios entre un sistema local y un sistema remoto, o entre dos sistemas remotos.
* Los protocolos de Información de Enrutamiento (RIP) y de configuración dinámica de host (DHCP).

### Clases Java

1. **Clase DatagramPacket**

La clase **DatagramPacket** representa un paquete de datos destinado a la transmisión mediante el uso del protocolo UDP. Los paquetes son contenedores de una pequeña secuencia de bytes e incluyen información de direccionamiento como una dirección IP y un puerto:

Mensaje: Enviado Saludos!!!

Longitud: 18

Destino: 192.168.0.1

Puerto: 6300

El significado de los datos almacenados en un DatagramPacket está **determinado por su contexto**:

* Cuando un DatagramPacket se ha leído desde un socket UDP, la dirección IP del paquete representa la dirección del remitente (lo mismo con el número de puerto)
* Cuando un DatagramPacket es usado para enviar un paquete UDP, la dirección IP almacenada en DatagramPacket representa la dirección del destinatario (lo mismo con el número de puerto). Esta inversión de sentido es importante para no enviar un paquete de respuesta a uno mismo

Los objetos del tipo DatagramPacket se pueden transmitir entre procesos cuando un proceso los envía y otro proceso los recibe. Por tanto, las razones para crear un nuevo DatagramPacket son las siguientes:

* Enviar datos a una máquina remota usando UDP
* Recibir los datos enviados por una máquina remota usando UDP

**Constructores**

**Algunos de los constructores de esta clase son:**

* **DatagramPacket(byte[] buffer, int length):** crea un DatagramPacket para la recepción de paquetes UDP entrantes.
* **DatagramPacket(byte[] buffer, int length, InetAddress dest\_addr, int dest\_port):** crea un DatagramPacket para enviar un DatagramPacket a una máquina remota.

**Métodos**

**La clase DatagramPacket proporciona algunos métodos que permiten obtener la dirección remota, puerto remoto, los datos del mensaje (como un array de bytes) y la longitud del paquete.**

* **InetAddress getAddress():** devuelve la dirección IP desde la que un DatagramPacket fue enviado, o si el paquete va a ser enviado a una máquina remota, la dirección IP de destino.
* **byte [] getData ():** devuelve el contenido del DatagramPacket, representado como una matriz de bytes.
* **int getLength():** devuelve la longitud de los datos almacenados en un DatagramPacket. Esto puede ser menor que el tamaño real del búfer de datos.
* **int getPort ():** devuelve el número de puerto desde donde se envió un DatagramPacket, o si el paquete va a ser enviado a una máquina remota, el número de puerto de destino.

**También proporciona métodos para modificar los datos de los paquetes. Esto significa que los paquetes recibidos pueden ser modificados. Por ejemplo, el contenido de un paquete puede ser reemplazado y enviado de vuelta al remitente. Esto ahorra el tener que reiniciar la información de direccionamiento, la dirección y el puerto del paquete que ya está establecido en el remitente.**

* **void setAddress (InetAddress addr):** asigna una nueva dirección de destino a un DatagramPacket.
* **void setData (byte [] buffer):** asigna un buffer de datos nuevos al DatagramPacket.
* **void setLength (int length):** asigna una nueva longitud del DatagramPacket.
* **void setPort (int port):** asigna un puerto de destino a un nuevo DatagramPacket.

1. **Clase DatagramSocket**

Proporciona acceso a un socket UDP, lo que permite que los paquetes UDP **puedan ser enviados y recibidos**. Un DatagramPacket se utiliza para representar un paquete UDP y debe ser creado antes de recibir los paquetes.

DatagramSocket puede ser utilizada tanto para enviar como para recibir paquetes. Cada DatagramSocket se une a un puerto en la máquina local, el cual es usado para dirigir a los paquetes. El número de puerto necesita no ser el mismo al número de la máquina remota, pero si la aplicación es un servidor UDP, se suele elegir un número de puerto específico. Si el DatagramSocket está destinado a ser un cliente, y no necesita un número de puerto específico, puede ser utilizado un constructor sin parámetro.

**Constructores**

**Algunos de los constructores de esta clase son:**

* **DatagramSocket () throws java.net.SocketException:** construye un socket para datagramas, el sistema elige un puerto de los que están libres.
* **DatagramSocket (int puerto) throws java.net.SocketException:** construye un socket para datagramas y lo conecta al puerto local especificado.
* **DatagramSocket (int puerto, InetAddress ip) throws java.net.SocketException:** construye un socket para datagramas y lo conecta al puerto local y dirección local especificados.

**Métodos**

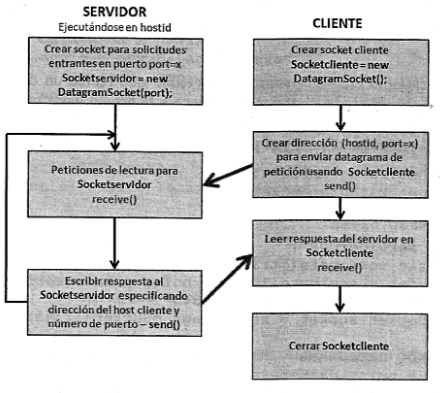
**Algunos de los métodos de esta clase son:**

* **void receive (DatagramPacket packet) throws java.io.IOException:** recibe un datagrama del socket.
* **void send (DatagramPacket packet) throws java.io.IOException:** envía un datagrama del socket.
* **void close ():** cierra un socket y se desliga del puerto local.
* **int getLocalPort ():** devuelve el puerto local al que está conectado el socket: -1 si el socket está cerrado y 0 si no está enlazado a ningún puerto
* **int getPort ():** devuelve el puerto remoto al que está conectado el socket o -1 si no existe dicha conexión.
* **void** [**connect**](https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html#connect(java.net.InetAddress,%20int)) **(**[**InetAddress**](https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html)**address, int port):** conecta el socket a un puerto remoto y una dirección IP concretos; el socket sólo podrá enviar y recibir mensajes desde esa dirección.
* **setSoTimeout (int timeout):** permite establecer un tiempo de espera límite. Entonces el método receive() se bloquea durante el tiempo fijado.

### Gestión de Sockets UDP

En los sockets UPD no se establece conexión. Los roles cliente-servidor están un poco más difusos que en el caso de los sockets TCP. Se puede considerar al servidor como el que espera un mensaje y responde, y al cliente como el que inicia la comunicación. Tanto uno como otro, si desean ponerse en contacto entre ellos, necesitan saber en qué ordenador y en qué puerto está escuchando el otro.

En la siguiente figura se muestra el flujo de la comunicación entre cliente y servidor utilizando el protocolo UDP. Ambos necesitan crear un socket DatagramSocket:



*Figura 4.2 – Modelo básico de socket UDP*

* El **servidor** crea un socket asociado a un puerto local para escuchar peticiones de cliente. Permanece a la espera de recibir peticiones.
* El **cliente** creará un socket para comunicarse con el servidor. Para enviar datagramas necesita conocer su IP y el puerto por el que se escucha. Utilizará el método *send()* del socket para enviar la petición en forma de datagrama.
* El **servidor** recibe las peticiones mediante el método *receive()* del socket. En el datagrama va incluido además del mensaje, el puerto y la IP del cliente emisor de la petición; lo que le permite al servidor conocer la dirección del emisor del datagrama. Utilizando el método *send()* del socket puede enviar la respuesta al cliente emisor.
* El **cliente** recibe la respuesta del servidor mediante el método *receive()* del socket y cierra el socket de cliente.
* El **servidor** permanece a la espera de recibir más peticiones de los clientes.

**Ejemplo 7 – Un cliente UDP envía un datagrama (un mensaje de saludo) a un servidor UDP y muestra la información sobre el envío. Cuando el servidor UDP recibe el datagrama desde el cliente, lo muestra en la consola**

**Servidor.java:**

**import** java.net.DatagramPacket;

**import** java.net.DatagramSocket;

**public** **class** Servidor {

**public** **static** **void** main (String[] args) **throws** Exception {

**int** puerto = 12346;

// Buffer para recibir los datagramas

**byte**[] bufer = **new** **byte**[1024];

// Asociar el socket al puerto 12346 del servidor

DatagramSocket socket = **new** DatagramSocket ( puerto);

// Esperar la llegada del datagrama al servidor

System.***out***.println("Esperando datagrama.......");

DatagramPacket recibo = **new** DatagramPacket (bufer, bufer.length);

// Recibir el datagrama

socket.receive (recibo);

// Obtener el número de bytes del datagrama

**int** bytesRec = recibo.getLength();

// Obtener el datagrama en String

String paquete = **new** String (recibo.getData());

// Visualizar información del datagrama recibido

System.***out***.println("Número de bytes recibidos: " + bytesRec);

System.***out***.println("Contenido del paquete: " + paquete.trim());

System.***out***.println("Puerto de orgen del mensaje: " + recibo.getPort());

System.***out***.println("IP de origen: " + recibo.getAddress().getHostAddress());

System.***out***.println("Puerto destino del mensaje: " + socket.getLocalPort());

//Cerrar el socket de servidor

socket.close();

}

}

**Cliente.java:**

**import** java.net.DatagramPacket;

**import** java.net.DatagramSocket;

**import** java.net.InetAddress;

**public** **class** Cliente {

**public** **static** **void** main (String[] args) **throws** Exception{

**int** puerto = 12346;

InetAddress destino = InetAddress.*getLocalHost*();

// Matriz de bytes

**byte**[] mensaje= **new** **byte**[1024];

//Mensaje a enviar al servidor

String saludo ="Enviando Saludos!!";

// Codificar el string en bytes

mensaje = saludo.getBytes();

// Construir el datagrama a enviar al servidor

DatagramPacket envio = **new** DatagramPacket (mensaje, mensaje.length, destino, puerto);

DatagramSocket socket = **new** DatagramSocket(34567); // Puerto local

// Visualizar información del datagrama enviado

System.***out***.println("Enviando el datagrama de longitud: " + mensaje.length);

System.***out***.println("Host Destino: " + destino.getHostName());

System.***out***.println("IP destino: " + destino.getHostAddress());

System.***out***.println("Puerto local del socket: " + socket.getLocalPort());

System.***out***.println("Puerto al que envio: " + envio.getPort());

// Enviar el datagrama al servidor

socket.send(envio);

// Cerrar el socket de cliente

socket.close();

}

}

**Resultado del Servidor:**

Esperando datagrama.......

Número de bytes recibidos: 18

Contenido del paquete: Enviando Saludos!!

Puerto de orgen del mensaje: 34567

IP de origen: 172.20.105.120

Puerto destino del mensaje: 12346

**Resultado del Cliente:**

Enviando el datagrama de longitud: 18

Host Destino: DM2-PROF

IP destino: 172.20.105.120

Puerto local del socket: 34567

Puerto al que envio: 12346

### Clase Java MulticastSocket

La clase MulticastSocket es útil para enviar paquetes simultáneamente a múltiples destinos. Para poder recibir estos paquetes es necesario establecer un **grupo multicast**, que es un grupo de direcciones IP que comparten el mismo número de puerto. Cuando se envía un mensaje a un grupo multicast, todos los que pertenezcan a ese grupo recibirán el mensaje; la pertenencia al grupo es transparente para el emisor, es decir, el emisor no conoce el número de miembros del grupo ni sus direcciones IP.

Un grupo multicast se especifica mediante una dirección IP de clase D y un puerto UDP estándar. Las direcciones desde la 224.0.0.0 hasta la 239.255.255.255 están destinadas para ser direcciones de multicast, aunque la dirección 224.0.0.1 está reservada y no debe ser utilizada.

El multicast no está activado por defecto en los sistemas operativos Linux y MacOs.

Como la configuración de la red del aula tiene desactivado el multicast, esta funcionalidad no funciona entre los equipos físicos de la red local del aula, por lo que para probarla será preciso identificar otro tipo de escenarios (máquinas virtuales, etc).

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.

**Constructores**

* **MulticastSocket():** crea el socket en cualquiera de los puertos locales libres.
* **MulticastSocket(int port):** crea el socket en el puerto local indicado.

**Métodos**

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

* **joinGroup(InetAddress mcastaddr):** permite al socket multicast unirse al grupo de multicast
* **leaveGroup(InetAddress mcastaddr):** permite al socket multicast abandonar el grupo de multicast
* **send(DatagramPacket p):** para enviar datagramas a todos los miembros de un grupo multicast
* **receive(DatagramPacket p):** para recibir datagramas de un grupo multicast

**Aunque para recibir mensajes de un grupo multicast es necesario pertenecer a dicho grupo, para enviar mensajes al grupo multicast no es necesario pertenecer al mismo.**

**Ejemplo 8 – Un servidor multicast lee datos por el teclado y los envía a todos los clientes que pertenezcan al grupo multicast. El proceso del servidor terminará cuando se introduzca un asterisco. El cliente multicast visualiza el mensaje recibido del servidor y su proceso finaliza cuando recibe un asterisco.**

**Para las pruebas de este ejemplo es preciso tener activada y operativa la funcionalidad de multicast en la red en la que se ejecute el programa. El orden de ejecución sería ejecutar en primer lugar el ServidorMulticast y a continuación varias instancias del ClienteMulticast.**

**ServidorMulticast.java:**

**import** java.net.\*;

**import** java.io.\*;

**public** **class** ServidorMulticast {

**public** **static** **void** main(String args[]) **throws** IOException {

BufferedReader in = **new** BufferedReader(**new** InputStreamReader (System.***in***));

// Creación del socket Multicast

**int** Puerto=12345;

MulticastSocket multisocket = **new** MulticastSocket(Puerto);

// Creación del grupo Multicast

InetAddress grupo = InetAddress.*getByName*("224.0.0.1");

String cadena="";

// Bucle de control de datos leídos por teclado

**while** (!cadena.trim().equals("\*")) {

System.***out***.println ("Datos a enviar al grupo multicast:");

cadena= in.readLine();

// Envío del mensaje al grupo multicast

DatagramPacket paquete=**new** DatagramPacket (cadena.getBytes(), cadena.length(), grupo, Puerto);

multisocket.send(paquete);

}

//Cierre del socket multicast

multisocket.close();

System.***out***.println ("Socket multicast cerrado");

}

}

**ClienteMulticast.java:**

**import** java.net.DatagramPacket;

**import** java.net.InetAddress;

**import** java.net.MulticastSocket;

**public** **class** ClienteMulticast {

**public** **static** **void** main (String args[]) **throws** Exception {

// Creación del socket Multicast

**int** Puerto=12345;

MulticastSocket multisocket = **new** MulticastSocket(Puerto);

// Unión al grupo multicast

InetAddress grupo = InetAddress.*getByName*("224.0.0.1");

multisocket.joinGroup (grupo);

String msg="";

**byte**[] buffer= **new** **byte**[1024];

// Bucle de control de recepción mensaje desde servidor Multicast

**while** (!msg.trim().equals("\*")) {

DatagramPacket paquete=**new** DatagramPacket (buffer, buffer.length);

multisocket.receive(paquete);

msg=**new** String(paquete.getData());

System.***out***.println("Mensaje recibido: "+msg.trim());

}

// Abandono y cierre del grupo multicast

multisocket.leaveGroup(grupo);

System.***out***.println ("Grupo multicast abandonado");

multisocket.close();

System.***out***.println ("Socket multicast cerrado");

}

}

## Envío de objetos a través de Sockets

Hasta ahora se han estado intercambiando cadenas de caracteres entre los programas cliente y servidor. Pero los stream soportan diversos tipos de datos como bytes, los tipos de datos primitivos, caracteres localizados y objetos.

### Objetos en Sockets TCP

Las clases **ObjectInputStream y ObjectOutputStream** permiten enviar objetos a través de sockets TCP. Los métodos **readObject()** y **writeObject()** sirven para leer el objeto del stream y para escribir en el mismo, respectivamente. Se usará el constructor de la clase que admita un InputStream y un OutputStream.

Para preparar el flujo de salida para la escritura de objetos se hace lo siguiente:

ObjectOutputStream outObjeto = **new** ObjectOutputStream (socket.getOutputStream());

Para preparar el flujo de entrada para la lectura de objetos se hace lo siguiente:

ObjectInputStream inObjeto = **new** ObjectInputStream (socket.getInputStream());

Las clases a las que pertenecen estos objetos deben implementar la interface *Serializable*.

**Ejemplo 9 – La clase Persona tiene 2 atributos (nombre y edad), 2 constructores y los correspondientes métodos get y set para la gestión de dichos atributos. Un Servidor crea un objeto Persona dándole valores (Juan para el atributo nombre y 20 para el atributo edad) y se lo envía al programa Cliente. El cliente modifica los datos oportunos en el objeto Persona (Juan Ramos para el atributo nombre y 44 para el atributo edad) y se los devuelve modificados al Servidor**

**Persona.java:**

**import** java.io.Serializable;

**public** **class** Persona **implements** Serializable {

String nombre;

**int** edad;

**public** String getNombre() {

**return** nombre;

}

**public** **void** setNombre(String nombre) {

**this**.nombre = nombre;

}

**public** **int** getEdad() {

**return** edad;

}

**public** **void** setEdad(**int** edad) {

**this**.edad = edad;

}

**public** Persona(String nombre, **int** edad) {

**super**();

**this**.nombre = nombre;

**this**.edad = edad;

}

}

**Servidor.java:**

**import** java.io.DataInputStream;

**import** java.io.DataOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.io.OutputStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.net.UnknownHostException;

**public** **class** Servidor {

**public** **static** **void** main (String[] args) **throws** ClassNotFoundException {

**int** Puerto = 6000;

ServerSocket servidor;

**try** {

servidor = **new** ServerSocket (Puerto);

System.***out***.println("ESPERANDO AL CLIENTE... ");

Socket cliente = **null**;

cliente = servidor.accept();

// Flujo de salida hacia el cliente

ObjectOutputStream outObjeto = **new** ObjectOutputStream (cliente.getOutputStream());

// Creación del objeto y envío al cliente

Persona per = **new** Persona ("Juan",20);

outObjeto.writeObject(per);

System.***out***.println("Envío: "+per.getNombre()+"\*"+per.getEdad());

// Flujo de entrada para leer el objeto

ObjectInputStream inObjeto = **new** ObjectInputStream (cliente.getInputStream());

Persona dato =(Persona) inObjeto.readObject();

System.***out***.println("Recibo: "+dato.getNombre() + "\*" + dato.getEdad());

// Cierre de streams y sockets

outObjeto.close();

inObjeto.close();

servidor.close();

} **catch** (UnknownHostException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**Cliente.java:**

**import** java.io.DataInputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.net.Socket;

**public** **class** Cliente {

**public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException {

String Host="localhost";

**int** Puerto = 6000;

System.***out***.println("PROGRAMA CLIENTE INICIANDO");

Socket cliente = **new** Socket (Host, Puerto);

// Obtener flujo de entrada para leer objetos

ObjectInputStream inObjeto = **new** ObjectInputStream (cliente.getInputStream());

Persona dato =(Persona) inObjeto.readObject();

System.***out***.println("Recibo: "+dato.getNombre() + "\*" + dato.getEdad());

// Modificación atributos del objeto

dato.setNombre("Juan Ramos");

dato.setEdad(44);

// Envío del objeto mediante flujo de salida

ObjectOutputStream perSal = **new** ObjectOutputStream(cliente.getOutputStream());

perSal.writeObject(dato);

System.***out***.println("Envío: "+dato.getNombre()+"\*"+dato.getEdad());

// Cierre de los streams y sockets

inObjeto.close();

perSal.close();

cliente.close();

}

}

**Resultado de la ejecución del programa Servidor (1º en ejecutar):**

ESPERANDO AL CLIENTE...

Envío: Juan\*20

Recibo: Juan Ramos\*44

**Resultado de la ejecución del programa Cliente (2º en ejecutar):**

PROGRAMA CLIENTE INICIANDO

Recibo: Juan\*20

Envio: Juan Ramos\*44

### Objetos en sockets UDP

Para intercambiar objetos en sockets UDP utilizaremos las clases **ByteArrayOutputStream y ByteArrayInputStream** en combinación con las clases ObjectOutputStream y ObjectInputStream. Se necesita convertir el objeto a un array de bytes. Por ejemplo, para convertir a un array de bytes un objeto de la clase Persona:

Persona persona = new Persona("Maria",20);

// Convertir el objeto en bytes

**ByteArrayOutputStream** bs=new ByteArrayOutputStream();

**ObjectOutputStream** out = new ObjectOutputStream (bs);

out.writeObject(persona); //Escribir el objeto persona en el Stream

out.close(); // Cierre del stream

byte[] bytes = bs.toByteArray(); // Objeto en bytes

Para convertir los bytes recibidos en el datagrama en un objeto Persona:

// Recibir el datagrama

byte[] recibidos = new byte[1024];

DatagramPacket paqRecibido = new DatagramPacket (recibidos,recibidos.length);

socket.receive(paqRecibido); // Recepción del datagrama

// Conversión de los bytes al objeto

**ByteArrayInputStream** bais = new ByteArrayInputStream (recibidos);

**ObjectInputStream** in = new ObjectInputStream(bais);

Persona persona = (Persona) in.readObject(); // Obtención del objeto

in.close();// Cierre del stream

# CONEXIONES MÚLTIPLES CLIENTES. HILOS

Los programas servidores vistos hasta ahora solo pueden atender a un cliente en cada momento, pero lo más habitual es que un servidor atienda a muchos clientes de forma simultánea. La solución para poder atender a múltiples clientes está en el uso del concepto del **multihilo**.

El esquema básico de funcionamiento con sockets TCP es el siguiente:

* Construir un único servidor con la clase **ServerSocket** e invocar a su método *accept()* para esperar las peticiones de conexión de los clientes.
* Cuando un cliente se conecta, el método *accept()* devuelve el objeto **Socket**, el cual se usará para crear un hilo cuya misión sea la de atender a ese cliente.
* A continuación se invoca al método *accept()* para esperar a un nuevo cliente.

La **clase Servidor** espera las conexiones de los clientes dentro de un bucle infinito.

**Servidor.java:**

**public** **class** Servidor {

**public** **static** **void** main (String[] args) **throws** IOException {

**int** Puerto = 6000;

ServerSocket servidor = **new** ServerSocket (Puerto);

**while** (**true**) {

Socket cliente = **new** Socket ();

Socket clienteConectado = servidor.accept();

System.***out***.println("Cliente Conectado.....");

HiloServidor hilo = **new** HiloServidor (cliente);

hilo.start();

}

}

}

Todas las operaciones que sirven a un cliente en particular quedan dentro de la **clase HiloServidor**. El hilo permite que el servidor se mantenga a la escucha de peticiones y no interrumpa su proceso mientras los clientes son atendidos:

**HiloServidor.java:**

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**public** **class** HiloServidor **extends** Thread {

BufferedReader fentrada;

PrintWriter fsalida;

**static** Socket *socket* = **null**;

// Constructor

**public** HiloServidor(Socket s) **throws** IOException {

*socket*=s;

fsalida= **new** PrintWriter (*socket*.getOutputStream(),**true**);

fentrada= **new** BufferedReader (**new** InputStreamReader (*socket*.getInputStream()));

}

**public** **void** run() {

String cadena="";

**while** (!cadena.trim().equals("\*")) {

System.***out***.println("Comunico con: "+ *socket*.toString());

**try** {

cadena = fentrada.readLine();

} **catch** (IOException e) {

e.printStackTrace();

}

fsalida.println(cadena.trim().toUpperCase()); // Enviar mayúsculas

}// Fin while

System.***out***.println("Fin Con:" + *socket*.toString());

fsalida.close();

**try** {

fentrada.close();

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

*socket*.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

La clase **Cliente** se conectará con el servidor en un puerto acordado.

El orden de ejecución de este tipo de programas suele ser es el siguiente:

* En primer lugar, el programa Servidor.
* A continuación, el programa Cliente varias veces, observando cómo los diversos clientes conectados son atendidos por el sevidor de forma simultánea.

La conexión de múltiples clientes a un servidor mediante diferentes hilos, suele aplicarse en el desarrollo de los **chats,** los cuales pueden funcionar tanto bajo el protocolo TCP como UDP.