

UNIDAD 3

Programación de Comunicaciones en Red

Tabla de contenido

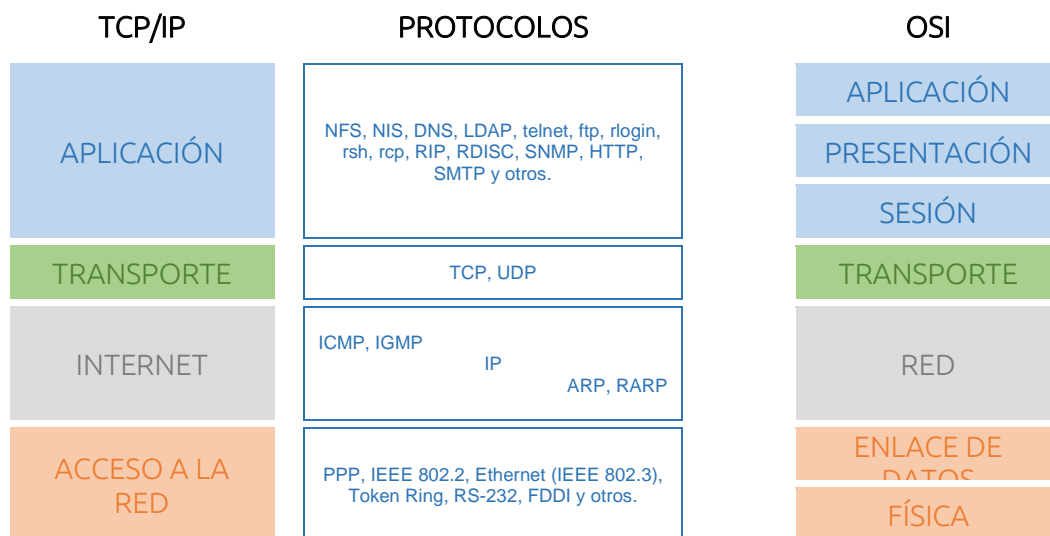
1	ARQUITECTURA TCP/IP	2
1.1	CAPA DE APLICACIÓN.....	2
1.2	CAPA DE TRANSPORTE	3
1.3	CAPA DE INTERNET:.....	3
1.4	CAPA DE ACCESO A RED.....	3
2	INTRODUCCIÓN AL DESARROLLO DE APLICACIONES DE RED CON JAVA	4
2.1	LA CLASE INETADDRESS	4
2.2	LA CLASE URL	6
2.3	LA CLASE URLCONNECTION	9
3	SOCKETS.....	10
3.1	FUNCIONAMIENTO EN GENERAL DE UN SOCKET	10
3.2	TIPOS DE SOCKETS	11
3.2.1	<i>Sockets orientados a conexión</i>	<i>11</i>
3.2.2	<i>Sockets no orientados a conexión</i>	<i>11</i>
4	CLASES PARA SOCKETS TCP	12
4.1	CLASE SERVERSOCKET	12
4.2	CLASE SOCKET	13
4.3	GESTIÓN DE SOCKETS TCP	14
5	CLASES PARA SOCKETS UDP	19
5.1	GESTIÓN DE SOCKETS UDP	22
5.2	MULTICASTSOCKET	25
6	GESTIÓN DE LA CONEXIÓN DE MÚLTIPLES CLIENTES CON HILOS.....	28

1 Arquitectura TCP/IP

Las redes informáticas interconectan ordenadores y otros tipos de dispositivos usando diferentes tipos de medios de transmisión con el propósito de compartir información, así como hardware y software de uso común. Con la fuerte expansión que ha tenido Internet se ha generalizado su uso tanto en empresas como en los hogares. Hoy en día para una empresa es totalmente necesario disponer de una red interna que le permita compartir información, conectarse a Internet o incluso, ofrecer sus servicios en Internet.

En la actualidad se ha adoptado el modelo TCP/IP para la comunicación en todo tipo de redes, desde pequeñas redes de área local (LAN) hasta redes de área extensa (WAN) como Internet. Este modelo describe una arquitectura de red basada en capas y una pila de protocolos, de los cuales los dos más importantes se han usado para darle su nombre: **TCP** o *Transmission Control Protocol* e **IP** o *Internet Protocol*.

La figura siguiente muestra las cuatro capas de la arquitectura TCP/IP y su correspondencia con las capas del modelo de referencia OSI:



1.1 Capa de Aplicación

Proporciona a las aplicaciones la capacidad de acceder a los servicios de las demás capas y define los protocolos que utilizan las aplicaciones para intercambiar datos. Existen muchos protocolos de capa de aplicación y continuamente se están desarrollando más.

En esta arquitectura de protocolos, los de capa de aplicación más ampliamente conocidos son los utilizados para el intercambio de información de los usuarios:

- **Hypertext Transfer Protocol (HTTP):** se utiliza para transferir archivos que componen las páginas Web de la World Wide Web.
- **File Transfer Protocol (FTP):** se utiliza para la transferencia interactiva de archivos.
- **Simple Mail Transfer Protocol (SMTP):** se utiliza para la transferencia de mensajes de correo electrónico y archivos adjuntos.
- **Telnet:** es un protocolo de emulación de terminal, se utiliza para iniciar la sesión de forma remota en máquinas de la red.

Además, dentro de la arquitectura de protocolos TCP/IP, estos otros protocolos de capa de aplicación ayudan a facilitar el uso y la gestión de redes TCP/IP:

- **Domain Name System (DNS):** se utiliza para resolver nombres de host a direcciones IP.
- **Routing Information Protocol (RIP):** es un protocolo de enrutamiento que los enrutadores utilizan para intercambiar información de enrutamiento en una red IP.
- **Simple Network Management Protocol (SNMP):** se utiliza entre una consola de gestión de red y dispositivos de red (routers, bridges, hubs inteligentes) para recoger e intercambiar información de gestión de la red.

1.2 Capa de Transporte

Es la responsable de proporcionar a la capa de aplicación, servicios de sesión y de comunicación de datagramas. Los protocolos básicos de la capa de transporte son:

- **Transmission Control Protocol (TCP):** proporciona un servicio de comunicaciones fiable orientado a la conexión uno a uno. TCP es responsable del establecimiento de una conexión TCP, la secuencia y el acuse de recibo de los paquetes enviados, y la recuperación de paquetes perdidos durante la transmisión.
- **User Datagram Protocol (UDP):** proporciona una conexión, uno a uno o uno a muchos poco fiable. Por eso UDP se utiliza cuando la cantidad de datos a transferir es pequeña y no se desea la sobrecarga que supone establecer una conexión TCP o cuando las aplicaciones o protocolos de capa superior proporcionan una entrega fiable.

1.3 Capa de Internet:

Es responsable de las funciones de direccionamiento, empaquetado y enrutamiento. Los protocolos básicos de la capa de Internet son:

- **Internet Protocol (IP):** es un protocolo enrutable responsable del direccionamiento IP, enrutamiento y fragmentación y reensamblado de paquetes.
- **Address Resolution Protocol (ARP):** es responsable de la resolución de la dirección de la capa de Internet a la dirección de la capa de interfaz de red, tales como una dirección de hardware.
- **Internet Control Message Protocol (ICMP):** es responsable de proporcionar funciones de diagnóstico y notificación de errores debidos a la entrega sin éxito de paquetes IP.
- **Internet Group Management Protocol (IGMP):** es responsable de la gestión de grupos de multidifusión IP.

1.4 Capa de Acceso a Red

Implementa la interfaz con la red física. TCP/IP fue diseñado para ser independiente del método de acceso a la red, el formato y el medio. De esta manera, TCP/IP se puede utilizar para conectar diferentes tipos de red. Estas incluyen tecnologías LAN como las tecnologías Ethernet y Token Ring, y WAN tales como X.25 y Frame Relay. Su independencia de cualquier tecnología de red específica da a TCP/IP la capacidad de adaptarse a nuevas tecnologías tales como el modo de transferencia asíncrono o Asynchronous Transfer Mode (ATM)

2 Introducción al desarrollo de aplicaciones de red con Java

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

Una API de bajo nivel, que se ocupa de las abstracciones siguientes:

- **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 abstracciones siguientes:

- **URIs:** representan identificadores de recursos universales.
- **URLs:** representan los localizadores de recursos universales.
- **Conexiones:** representa las conexiones al recurso apuntado por URL.

2.1 La clase `InetAddress`

La clase `InetAddress` es la abstracción que representa una dirección IP. Tiene dos subclases: `Inet4Address` para direcciones IPv4 e `Inet6Address` para direcciones IPv6.

En la siguiente tabla se muestran algunos métodos importantes de esta clase:

<code>InetAddress getLocalHost()</code>	Devuelve un objeto <code>InetAddress</code> que representa la dirección IP de la máquina donde se está ejecutando el programa.
<code>InetAddress getByName(String host)</code>	Devuelve un objeto <code>InetAddress</code> que representa la dirección IP de la máquina que se especifica en el parámetro <code>host</code> . Este parámetro puede ser el nombre de la máquina, un nombre de dominio o una dirección IP.
<code>InetAddress[] getAllByName(String host)</code>	Devuelve un array de objetos de tipo <code>InetAddress</code> . Este método es útil para averiguar todas las direcciones IP que tenga asignada una máquina en particular.
<code>String getHostAddress()</code>	Devuelve la dirección IP de un objeto <code>InetAddress</code> en forma de cadena.
<code>String getHostName()</code>	Devuelve el nombre del host de un objeto <code>InetAddress</code> .
<code>String getCanonicalHostName()</code>	Obtiene el nombre canónico completo (suele ser la dirección real del host) de un objeto <code>InetAddress</code> .

Los 3 primeros son métodos de clase que retornan objetos `InetAddress` y pueden lanzar la excepción `UnknownHostException`. El resto son métodos de instancia que retornan información relativa a una dirección concreta. En el siguiente ejemplo se ponen a prueba algunos de estos métodos:

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class TestInetAddress {
    public static void main(String[] args) {
        try {
            pruebaMetodos(InetAddress.getLocalHost(), "getLocalHost()");
            pruebaMetodos(InetAddress.getLoopbackAddress(),
                "getLoopbackAddress()");
            String host = "www.google.com";
```

```

        pruebaMetodos(InetAddress.getByName(host), "getByName()");
        obtenerTodas(host);
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}

private static void obtenerTodas(String host) {
    System.out.println("=====");
    System.out.println("Direcciones obtenidas con el método getAllByName()");
    try {
        InetAddress[] direcciones = InetAddress.getAllByName(host);
        for (InetAddress d: direcciones)
            System.out.println("\t" + d.toString());
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
    System.out.println("=====");
}

private static void pruebaMetodos(InetAddress dir, String metodo) {
    System.out.println("=====");
    System.out.println("Dirección obtenida con el método " + metodo);
    System.out.println("\tMétodo getHostName(): " + dir.getHostName());
    System.out.println("\tMétodo getHostAddress(): " + dir.getHostAddress());
    System.out.println("\tMétodo toString(): " + dir.toString());
    System.out.println("\tMétodo getCanonicalHostName(): " +
        dir.getCanonicalHostName());
}
}

```

La ejecución de este ejemplo genera la salida siguiente:

```

=====
Dirección obtenida con el método getLocalHost()
Método getHostName(): DESKTOP-HG6KJ7P
Método getHostAddress(): 127.0.1.1
Método toString(): DESKTOP-HG6KJ7P/127.0.1.1
Método getCanonicalHostName(): DESKTOP-HG6KJ7P
=====
Dirección obtenida con el método getLoopbackAddress()
Método getHostName(): localhost
Método getHostAddress(): 127.0.0.1
Método toString(): localhost/127.0.0.1
Método getCanonicalHostName(): localhost
=====
Dirección obtenida con el método getByName()
Método getHostName(): www.google.com
Método getHostAddress(): 172.217.19.132
Método toString(): www.google.com/172.217.19.132
Método getCanonicalHostName(): mrs08s04-in-f4.1e100.net
=====
Direcciones obtenidas con el método getAllByName()
www.google.com/172.217.19.132
www.google.com/2a00:1450:4006:806:0:0:0:2004
=====

```

PRÁCTICA 1

Realiza un programa Java que admita desde la línea de comandos un nombre de máquina o una dirección IP y visualice información sobre ella.

2.2 La 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 un directorio, o puede ser una referencia a un objeto más complicado, 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 URL

`https://docs.oracle.com/javase/7/docs/api/java/net/URL.html` encontramos el protocolo (`https`), el nombre de máquina (`docs.oracle.com`) y el fichero (`URL.html`) que está dentro del servidor en el directorio especificado (`/javase/7/docs/api/java/net`).

Una URL puede especificar opcionalmente un puerto TCP o UDP, aunque normalmente se omite ya protocolos como HTTP tienen asignado un número de puerto por defecto. En caso de especificarlo se haría de la forma siguiente:



La clase URL contiene varios constructores, algunos son:

<code>URL(String url)</code>	Crea un objeto <i>URL</i> a partir de la cadena <i>url</i> .
<code>URL(String protocolo, String host, String fichero)</code>	Crea un objeto <i>URL</i> a partir de las cadenas <i>protocolo</i> , <i>host</i> y <i>fichero</i> .
<code>URL(String protocolo, String host, int puerto, String fichero)</code>	Crea un objeto <i>URL</i> en especificando <i>protocolo</i> , <i>host</i> , <i>puerto</i> y <i>fichero</i> .
<code>URL(URL context, String url)</code>	Crea un objeto <i>URL</i> a partir de la dirección del host dada por la <i>URL context</i> y una URL relativa especificada en la cadena <i>url</i> (un directorio).

Estos constructores pueden lanzar la excepción *MalformedURLException* si la URL está mal construida, no se hace ninguna verificación de que realmente exista la máquina o el recurso en la red.

Algunos de los métodos de la clase URL son los siguientes:

<code>String getAuthority()</code>	Obtiene la autoridad del objeto <i>URL</i> .
<code>int getDefaultPort()</code>	Devuelve el puerto asociado por defecto al objeto <i>URL</i> .
<code>int getPort()</code>	Devuelve el número de puerto de la <i>URL</i> , <code>-1</code> si no se indica.
<code>String getHost()</code>	Devuelve el nombre de la máquina.
<code>String getQuery()</code>	Devuelve la cadena que se envía a una página para ser procesada (es lo que sigue al signo <code>?</code> de una URL).
<code>String getPath()</code>	Devuelve una cadena con la ruta hacia el fichero desde el servidor y el nombre completo del fichero.
<code>String getFile()</code>	Devuelve lo mismo que <i>getPath</i> , además de la concatenación del valor de <i>getQuery</i> si lo hubiese. Si no hay una porción consulta, este método y <i>getPath</i> devolverán los mismos resultados.
<code>String getUserInfo()</code>	Devuelve la parte con los datos del usuario de la dirección URL o nulo si no existe.
<code>InputStream openStream()</code>	Abre una conexión al objeto URL y devuelve un <i>InputStream</i> para la lectura de esa conexión.

<code>URLConnection openConnection()</code>	Devuelve un objeto <code>URLConnection</code> que representa la conexión a un objeto remoto referenciado por la URL (se ve en el siguiente apartado).
---	---

El siguiente ejemplo muestra el uso de los constructores definidos anteriormente; el método Visualizar 0 muestra información de la URL usando los métodos de la tabla anterior:

```
import java.net.MalformedURLException;
import java.net.URL;

public class Ejemplo1URL {

    public static void main(String[] args) {
        URL url;
        try {
            System.out.println("Constructor simple para una URL:");
            url = new URL("http://docs.oracle.com/");
            Visualizar(url);
            System.out.println("Otro constructor simple para una URL:");
            url = new URL("http://www.example.com/action.php?name=john&age=24");
            Visualizar(url);
            System.out.println("Constructor para protocolo + URL + directorio:");
            url = new URL("http", "docs.oracle.com", "/javase/7");
            Visualizar(url);
            System.out.println("Constructor para protocolo + URL + puerto + directorio:");
            url = new URL("http", "docs.oracle.com", 80, "/javase/7");
            Visualizar(url);
            System.out.println("Constructor para un objeto URL y un directorio:");
            URL urlBase = new URL("http://docs.oracle.com/");
            url = new URL(urlBase, "/javase/7/docs/api/java/net/URL.html");
            Visualizar(url);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    private static void Visualizar(URL url) {
        System.out.println("\tURL completa: " + url.toString());
        System.out.println("\tgetProtocol(): " + url.getProtocol());
        System.out.println("\tgetHost(): " + url.getHost());
        System.out.println("\tgetPort(): " + url.getPort());
        System.out.println("\tgetFile(): " + url.getFile());
        System.out.println("\tgetUserInfo(): " + url.getUserInfo());
        System.out.println("\tgetPath(): " + url.getPath());
        System.out.println("\tgetAuthority(): " + url.getAuthority());
        System.out.println("\tgetQuery(): " + url.getQuery());
        System.out.println("=====");
    }
}
```

La ejecución de este ejemplo genera la salida siguiente:

```
Constructor simple para una URL:
URL completa: http://docs.oracle.com/
getProtocol(): http
getHost(): docs.oracle.com
getPort(): -1
getFile(): /
getUserInfo(): null
getPath(): /
getAuthority(): docs.oracle.com
getQuery(): null
=====
Otro constructor simple para una URL:
URL completa: http://www.example.com/action.php?name=john&age=24
getProtocol(): http
getHost(): www.example.com
getPort(): -1
getFile(): /action.php?name=john&age=24
getUserInfo(): null
```

```

    getPath(): /action.php
    getAuthority(): www.example.com
    getQuery(): name=john&age=24
=====
Constructor para protocolo + URL + directorio:
    URL completa: http://docs.oracle.com/javase/7
    getProtocol(): http
    getHost(): docs.oracle.com
    getPort(): -1
    getFile(): /javase/7
    getUserInfo(): null
    getPath(): /javase/7
    getAuthority(): docs.oracle.com
    getQuery(): null
=====
Constructor para protocolo + URL + puerto + directorio:
    URL completa: http://docs.oracle.com:80/javase/7
    getProtocol(): http
    getHost(): docs.oracle.com
    getPort(): 80
    getFile(): /javase/7
    getUserInfo(): null
    getPath(): /javase/7
    getAuthority(): docs.oracle.com:80
    getQuery(): null
=====
Constructor para un objeto URL y un directorio:
    URL completa: http://docs.oracle.com/javase/7/docs/api/java/net/URL.html
    getProtocol(): http
    getHost(): docs.oracle.com
    getPort(): -1
    getFile(): /javase/7/docs/api/java/net/URL.html
    getUserInfo(): null
    getPath(): /javase/7/docs/api/java/net/URL.html
    getAuthority(): docs.oracle.com
    getQuery(): null
=====

```

El siguiente ejemplo crea un objeto *URL* a la dirección <https://amessagefrom.earth>, abre una conexión con él creando un objeto *InputStream* y lo utiliza como flujo de entrada para leer los datos de la página inicial del sitio; al ejecutar el programa se muestra en pantalla el código HTML de la página inicial del sitio:

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;

public class Ejemplo2URL {

    public static void main(String[] args) {
        URL url = null;
        try {
            url = new URL("https://amessagefrom.earth");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        BufferedReader in;
        try {
            InputStream inputStream = url.openStream();
            in = new BufferedReader(new InputStreamReader(inputStream));
            String inputLine;
            while ((inputLine = in.readLine()) != null)
                System.out.println(inputLine);
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```



```
}
```

2.3 La clase `URLConnection`

Una vez que tenemos un objeto de la clase `URL`, si se invoca al método `openConnection` para realizar la comunicación con el objeto y la conexión se establece satisfactoriamente, éste retornará una instancia de la clase `URLConnection`:

```
URL url = new URL("https://amessagefrom.earth");
URLConnection urlCon= url.openConnection();
```

La clase `URLConnection` es una clase abstracta que contiene métodos que permiten la comunicación entre la aplicación y una URL. Las instancias de esta clase se pueden utilizar tanto para leer como para escribir al recurso referenciado por la URL. Puede lanzar la excepción `IOException`.

Algunos de los métodos de esta clase son:

<code>InputStream getInputStream()</code>	Devuelve un objeto <code>InputStream</code> para leer datos de esta conexión.
<code>OutputStream getOutputStream()</code>	Devuelve un objeto <code>OutputStream</code> para escribir datos en esta conexión.
<code>void setDoInput (boolean b)</code>	Permite que el usuario reciba datos desde la URL si el parámetro <code>b</code> es true (por defecto está establecido a true).
<code>void setDoOutput((boolean b)</code>	Permite que el usuario envíe datos si el parámetro <code>b</code> es true (no está establecido al principio).
<code>void connect()</code>	Abre una conexión al recurso remoto si tal conexión no se ha establecido ya.
<code>int getContentLength()</code>	Devuelve el valor del campo de cabecera <code>content-length</code> o <code>-1</code> si no está definido.
<code>String getContentType()</code>	Devuelve el valor del campo de cabecera <code>content-type</code> o <code>null</code> si no está definido.
<code>Long getDate()</code>	Devuelve el valor del campo de cabecera <code>date</code> o <code>0</code> si no está definido.
<code>Long getLastModified()</code>	Devuelve el valor del campo de cabecera <code>last-modified</code> .
<code>String getHeaderField(int n)</code>	Devuelve el valor del enésimo campo de cabecera especificado o <code>null</code> si no está definido.
<code>Map<String, List<String>> getHeaderFields()</code>	Devuelve una estructura <code>Map</code> 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.
<code>URL getURL()</code>	Devuelve la dirección URL.

El siguiente ejemplo crea un objeto `URL` a la dirección <https://amessagefrom.earth>, se invoca al método `openConnection` del objeto para crear una conexión y se obtiene un `URLConnection`. Después se abre un stream de entrada sobre esa conexión mediante el método `getInputStream`. Al ejecutar el programa se muestra la misma salida que en el ejemplo anterior; sin embargo, este programa crea una conexión con la URL y el anterior abre directamente un stream desde la URL:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
```

```

import java.net.URL;
import java.net.URLConnection;

public class Ejemplo1URLConnection {

    public static void main(String[] args) {
        URL url = null;
        URLConnection urlCon = null;
        try {
            url = new URL("https://amessagefrom.earth");
            urlCon = url.openConnection();
            BufferedReader in;
            InputStream inputStream = urlCon.getInputStream();
            in = new BufferedReader(new InputStreamReader(inputStream));
            String inputLine;
            while ((inputLine = in.readLine()) != null)
                System.out.println(inputLine);
            in.close();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

3 SOCKETS

Los protocolos TCP y UDP utilizan la abstracción de sockets para proporcionar los puntos extremos de la comunicación entre aplicaciones o procesos. La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, a estos conectores es a lo que llamamos sockets.

Para los procesos receptores de mensajes, su conector debe tener asociado dos campos:

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

Así, todos los mensajes enviados a esa dirección IP y a ese puerto concreto llegarán al proceso receptor. Cada socket tiene un puerto asociado, el proceso cliente debe conocer el puerto y la IP del proceso servidor. Los mensajes al servidor le deben llegar al puerto acordado. El proceso cliente podrá enviar el mensaje por el puerto que quiera.

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

3.1 Funcionamiento en general 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 sobre ese puerto.

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

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.

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 al servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo por sus respectivos sockets.

3.2 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.

3.2.1 Sockets 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.

Los procesos que se van a comunicar deben establecer antes una conexión mediante un stream. Un stream es una secuencia ordenada de unidades de información (bytes, caracteres, etc.) que puede fluir en dos direcciones: hacia fuera de un proceso (de salida) o hacia dentro de un proceso (de entrada). Están diseñados para acceder a los datos de manera secuencial.

Una vez establecida la conexión, los procesos leen y escriben en el stream sin tener que preocuparse de las direcciones de Internet ni de los números de puerto. El establecimiento de la conexión implica:

- Una petición de conexión desde el proceso cliente al servidor.
- Una aceptación de la conexión del proceso servidor al cliente.

Los sockets TCP se utilizan en la gran mayoría de las aplicaciones IP. Algunos servicios con sus números de puerto reservados son:

SERVICIO	PUERTO
FTP	21
Telnet	23
HTTP	80
SMTP	25

En Java hay dos tipos de *stream sockets* que tienen asociadas las clases *Socket* para implementar el cliente y *ServerSocket* para el servidor.

3.2.2 Sockets no orientados a conexión

En este tipo de sockets la comunicación entre las aplicaciones se realiza por medio del protocolo UDP. Esta conexión no es fiable y no garantiza que la información enviada llegue a su destino, tampoco se garantiza el orden de llegada de los paquetes que puede llegar en distinto orden al

que se envía. Los datagramas se transmiten desde un proceso emisor a otro receptor sin que se haya establecido previamente una conexión, sin acuse de recibo ni reintentos.

Cualquier proceso que necesite enviar o recibir mensajes debe crear primero un conector asociado a una dirección IP y a un puerto local. El servidor enlazará su conector a un puerto de servidor conocido por los clientes. El cliente enlazará su conector a cualquier puerto local libre. Cuando un receptor recibe un mensaje, se obtiene además del mensaje, la dirección IP y el puerto del emisor, permitiendo al receptor enviar la respuesta correspondiente al emisor.

Los sockets UDP se usan cuando una entrega rápida es más importante que una entrega garantizada, o en los casos en que se desea enviar tan poca información que cabe en un único datagrama. Se usan en aplicaciones para la transmisión de audio y vídeo en tiempo real donde no es posible el reenvío de paquetes retrasados; algunas aplicaciones como NFS (Network File System), DNS (Domain Name Server) o SNMP (Simple Network Management Protocol) usan este protocolo.

Para implementar en Java este tipo de sockets se utilizan las clases *DatagramSocket* y *DatagramPacket*.

4 CLASES PARA SOCKETS TCP

El paquete `java.net` proporciona las clases *ServerSocket* y *Socket* para trabajar con sockets TCP. TCP es un protocolo orientado a conexión, por lo que para establecer una comunicación es necesario especificar una conexión entre un par de sockets. Uno de los sockets, el cliente, solicita una conexión, y el otro socket, el servidor, atiende las peticiones de los clientes. Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

4.1 Clase ServerSocket

La clase *ServerSocket* se utiliza para implementar el extremo de la conexión que corresponde al servidor, donde se crea un conector en el puerto de servidor que escucha las peticiones de conexión de los clientes.

Algunos de los constructores de esta clase son (pueden lanzar la excepción *IOException*):

<i>ServerSocket()</i>	Crea un socket de servidor sin ningún puerto asociado.
<i>ServerSocket(int port)</i>	Crea un socket de servidor, que se enlaza al puerto especificado.
<i>ServerSocket(int port, int máximo)</i>	Crea un socket de servidor y lo enlaza con el número de puerto local especificado. El parámetro máximo especifica el número máximo de peticiones de conexión que se pueden mantener en cola.
<i>ServerSocket(int port, int máximo, InetAddress dirección)</i>	Crea un socket de servidor en el puerto indicado, especificando un máximo de peticiones y conexiones entrantes y la dirección IP local.

Algunos métodos importantes son:

<i>Socket accept ()</i>	Escucha una solicitud de conexión de un cliente y la acepta cuando se recibe. Una vez que se ha establecido la conexión con el cliente, devuelve un objeto de tipo <i>Socket</i> , a través del cual se establecerá la comunicación con el cliente. Tras esto, el <i>ServerSocket</i> sigue disponible para aceptar nuevas conexiones. Puede lanzar <i>IOException</i> .
<i>close ()</i>	Cierra el <i>ServerSocket</i> .
<i>int getLocalPort ()</i>	Devuelve el puerto local al que está enlazado el <i>ServerSocket</i> .

El siguiente ejemplo crea un socket de servidor y lo enlaza al puerto 6000, visualiza el puerto por el que se esperan las conexiones y espera que se conecten 2 clientes:

```
try {
    int puerto = 6000;
    ServerSocket servidor;
    servidor = new ServerSocket(puerto);
    System.out.println("Escuchando en " + servidor.getLocalPort());
    Socket cliente1= servidor.accept();
    // realizar acciones con el primer cliente
    :

    Socket cliente2 = servidor.accept();
    // realizar acciones con el segundo cliente
    :

    servidor.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

4.2 Clase Socket

La clase Socket implementa un extremo de la conexión TCP. Algunos de sus constructores son (pueden lanzar la excepción IOException):

<code>Socket()</code>	Crea un socket sin ningún puerto asociado.
<code>Socket (InetAddress address, int port)</code>	Crea un socket y lo conecta al puerto y dirección IP especificados.
<code>Socket(InetAddress address, int port, InetAddress localAddr, int localPort)</code>	Permite además especificar la dirección IP local y el puerto local a los que se asociará el socket.
<code>Socket (String host, int port)</code>	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar también <i>UnknownHostException</i> .

Algunos métodos importantes son:

<code>InputStream getInputStream()</code>	Devuelve un <i>InputStream</i> que permite leer bytes desde el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <i>IOException</i> .
<code>OutputStream getOutputStream()</code>	Devuelve un <i>OutputStream</i> que permite escribir bytes sobre el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <i>IOException</i> .
<code>close()</code>	Se encarga de cerrar el socket.
<code>InetAddress getInetAddress()</code>	Devuelve la dirección IP y puerto a la que el socket está conectado. Si no lo está devuelve <i>null</i> .
<code>int getLocalPort()</code>	Devuelve el puerto local al que está enlazado el socket, -1 si no está enlazado a ningún puerto.
<code>int getPort()</code>	Devuelve el puerto remoto al que está conectado el socket, 0 si no está conectado a ningún puerto.

El siguiente ejemplo crea un socket cliente y lo conecta al host local al puerto 6000 (tiene que haber un *ServerSocket* escuchando en ese puerto). Después visualiza el puerto local al que está

conectado el socket, y el puerto, host y dirección IP de la máquina remota a la que se conecta (en este caso es el host local):

```
try {
    String host = "localhost";
    int puerto = 6000;
    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().toString());
    System.out.println ("IP Host Remoto: "+ i.getHostAddress().toString());
    cliente.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

La salida que se genera es la siguiente:

```
Puerto local: 8784
Puerto remoto: 6000
Host Remoto: localhost
IP Host Remoto: 127.0.0.1
```

4.3 Gestión de Sockets TCP

El modelo de sockets más simple sigue el esquema de funcionamiento siguiente:

- El programa servidor crea un socket de servidor definiendo un puerto, mediante el constructor *ServerSocket(port)*, y espera mediante el método *accept* a que el cliente solicite la conexión.
- Cuando el cliente solicita una conexión, el servidor abrirá la conexión al socket con el método *accept*.
- El cliente establece una conexión con el servidor a través del puerto especificado mediante el constructor *Socket(host, port)*.
- 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 de *InputStream*. Igualmente el servidor escribirá los mensajes al *OutputStream* y el cliente los leerá del *InputStream*.

Apertura de sockets:

En el programa servidor se crea un objeto *ServerSocket* invocando al constructor de *ServerSocket* en el que indicamos el número de puerto por el que el servidor escucha las peticiones de conexión de los clientes (se considera el tratamiento de excepciones):

```
ServerSocket servidor;
try {
    servidor = new ServerSocket(numeroPuerto);
} catch (IOException io) {
    io.printStackTrace();
}
```

Necesitamos también crear un objeto *Socket* desde el *ServerSocket* para aceptar las conexiones, se usa el método *accept*:

```

Socket clienteConectado = null;
try {
    clienteConectado = servidor.accept();
} catch (IOException e) {
    e.printStackTrace();
}

```

En el programa cliente es necesario crear un objeto *Socket*; el socket se abre de la siguiente manera:

```

try {
    Socket servidor = new Socket(host, puerto);
} catch (IOException e) {
    e.printStackTrace();
}

```

Donde *host* es el nombre de la máquina a la que nos queremos conectar *puerto* es el puerto por el que el programa servidor está escuchando las peticiones de los clientes.

Los puertos TCP se numeran de 0 a 65535. Los puertos en el rango de 0 a 1023 están reservados para servicios privilegiados; otros puertos de 1024 a 49151 están reservados para aplicaciones concretas (por ejemplo el 3306 lo usa MySQL, el 1521 Oracle); por último de 49152 a 65535 no están reservados para ninguna aplicación concreta.

Creación de streams de entrada:

En el programa servidor podemos usar *DataInputStream* para recuperar los mensajes que el cliente escriba en el socket, previamente hay que usar el método *getInputStream* para obtener el flujo de entrada del socket del cliente:

```

InputStream is = null;
DataInputStream dis = null;
try {
    is = cliente.getInputStream();
    dis = new DataInputStream(is);
} catch (IOException e) {
    e.printStackTrace();
}

```

En el programa cliente podemos realizar la misma operación para recibir los mensajes procedentes del programa servidor.

Creación de streams de salida:

En el programa servidor podemos usar *DataOutputStream* para escribir los mensajes que queramos que el cliente reciba, previamente hay que usar el método *getOutputStream* para obtener el flujo de salida del socket del cliente:

```

OutputStream os = null;
DataOutputStream dos = null;
try {
    os = cliente.getOutputStream();
    dos = new DataOutputStream(os);
} catch (IOException e) {
    e.printStackTrace();
}

```

En el programa cliente podemos realizar la misma operación para enviar mensajes al programa servidor.

Cierre de sockets:

El orden de cierre de los sockets es relevante, primero se han de cerrar los streams relacionados con un socket antes que el propio socket.

PRÁCTICA 2

Crea un programa servidor que envíe un mensaje a otro programa cliente y un programa cliente que devuelva el mensaje recibido al servidor.

PRÁCTICA 3

Crea un programa servidor que pueda atender hasta 3 clientes. Debe enviar a cada cliente un mensaje indicando el número de cliente que es. Este número será 1, 2 o 3. El cliente mostrará el mensaje recibido por el servidor. Cambia el programa para que lo haga para N clientes, siendo N un parámetro que tendrás que definir en el programa.

En el siguiente ejemplo el programa cliente envía el texto tecleado en su entrada estándar al servidor (en un puerto pactado) escribiendo en el socket, el servidor lee del socket y devuelve de nuevo al cliente el texto recibido escribiendo en el socket; el programa cliente lee del socket lo que le envía el servidor de vuelta y lo muestra en pantalla. El programa servidor finaliza cuando el cliente termine la entrada por teclado o cuando recibe como cadena un asterisco; el cliente finaliza cuando se detiene la entrada de datos mediante las teclas *Ctrl+C* o *Ctrl+Z*.

El programa servidor es el siguiente:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor {

    public static void main(String[] args) {
        ServerSocket servidor = null;
        Socket cliente = null;
        PrintWriter writer = null;
        BufferedReader reader = null;

        try {
            servidor = new ServerSocket(6000);
            System.out.println("Esperando conexión...");
            cliente = servidor.accept();
            System.out.println("Cliente conectado...");

            // CREO FLUJO DE SALIDA AL CLIENTE
            writer = new PrintWriter(cliente.getOutputStream(), true);
```



```

        // CREO FLUJO DE ENTRADA DEL CLIENTE
        reader = new BufferedReader(
            new InputStreamReader(cliente.getInputStream()));
        String cadena;
        while ((cadena = reader.readLine()) != null) {
            writer.println(cadena);
            System.out.println("Recibiendo: " + cadena);
            if(cadena.equals("*"))
                break;
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        // CERRAR STREAMS Y SOCKETS
        System.out.println("Cerrando conexión...");
        if (reader != null)
            try {
                reader.close() ;
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        writer.close();
        if (cliente != null)
            try {
                cliente.close() ;
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        if (servidor != null)
            try {
                servidor.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
    }
}
}

```

En este ejemplo se han usado las clases *PrintWriter* para el flujo de salida al socket y *BufferedReader* para el flujo de entrada. Se han utilizado los métodos *readLine* para leer cada línea de texto y *println* para escribirlas.

El programa cliente es el siguiente:

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class Cliente {

    public static void main(String[] args) {
        String Host = "localhost";
        int Puerto = 6000;
        Socket cliente = null;
        PrintWriter writer = null;
        BufferedReader reader = null;
    }
}

```

```

try {
    cliente = new Socket(Host, Puerto);

    // CREO FLUJO DE SALIDA AL SERVIDOR
    writer = new PrintWriter(cliente.getOutputStream(), true);

    // CREO FLUJO DE ENTRADA AL SERVIDOR
    reader = new BufferedReader(
        new InputStreamReader(cliente.getInputStream()));

    // FLUJO PARA ENTRADA ESTANDAR
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    String cadena, eco;
    System.out.print("Introduce cadena: ");
    cadena = in.readLine();
    while (cadena != null) {
        writer.println(cadena);
        eco = reader.readLine();
        System.out.println(" =>ECO: " + eco);
        System.out.print("Introduce cadena: ");
        cadena = in.readLine();
    }
    in.close();
} catch (UnknownHostException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (writer != null)
        writer.close();
    if (reader != null)
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    System.out.println("Fin del envío... ");
    if (cliente != null)
        try {
            cliente.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
}

```

Como resultado de la ejecución de servidor y cliente, se obtiene la salida siguiente:

```

C:\Windows\system32\cmd.exe
C:\psp\unidad3>java -jar servidor.jar
Esperando conexión...
Cliente conectado...
Recibiendo: psp
Recibiendo: unidad 3
Recibiendo: prueba de sockets
Cerrando conexión...
C:\psp\unidad3>

```

```

C:\Windows\system32\cmd.exe
C:\psp\unidad3>java -jar cliente.jar
Introduce cadena: psp
=>ECO: psp
Introduce cadena: unidad 3
=>ECO: unidad 3
Introduce cadena: prueba de sockets
=>ECO: prueba de sockets
Introduce cadena: ^Z
Fin del envío...
C:\psp\unidad3>

```

5 CLASES PARA SOCKETS UDP

Los sockets UDP 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, por ello cada vez que se envíen datagramas el emisor debe indicar explícitamente la dirección IP y el puerto del destino para cada paquete, y el receptor debe extraer la dirección IP y el puerto del emisor del paquete.

El paquete del datagrama está formado por los siguientes campos:

CADENA DE BYTES CONTENIENDO EL MENSAJE	LONGITUD DEL MENSAJE	DIRECCION IP DE DESTINO	Nº DE PUERTO DE DESTINO
--	-------------------------	----------------------------	----------------------------

El paquete *java.net* proporciona las clases *DatagramPacket* y *DatagramSocket* para implementar sockets UDP.

Clase *DatagramPacket*.

Proporciona constructores para crear instancias a partir de los datagramas recibidos y para crear instancias de datagramas que van a ser enviados:

<i>DatagramPacket(byte[] buf, int length)</i>	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje (<i>buf</i>) y la longitud (<i>length</i>) de la misma.
<i>DatagramPacket(byte[] buf, int offset, int length)</i>	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje, la longitud de la misma y el <i>offset</i> dentro de la cadena.
<i>DatagramPacket(byte[] buf, int length, InetAddress address, int port)</i>	Constructor para el envío de datagramas. Se especifica la cadena de bytes a enviar (<i>buf</i>), la longitud (<i>length</i>), el número de puerto de destino (<i>port</i>) y el host especificado en la dirección <i>address</i> .
<i>DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)</i>	Igual que el anterior pero se especifica un <i>offset</i> dentro de la cadena de bytes.

El siguiente ejemplo utiliza el tercer constructor para enviar un datagrama. El datagrama será enviado por el puerto 12345. El mensaje está formado por la cadena Enviando Saludos 11 que es necesario codificar en una secuencia de bytes y almacenar el resultado en una matriz de bytes.

Después será necesario calcular la longitud del mensaje a enviar. Con `InetAddress.getLocalHost()` obtengo la dirección IP del host al que enviaré el mensaje, en este caso el host local:

Mensaje: Enviando Saludos !!,	Longitud: 19	Destino: 192.168.2.1 IP del host local	Puerto: 12345
-------------------------------	--------------	---	---------------

```
try {
    InetAddress destino = InetAddress.getLocalHost();
    byte[] msg = "Enviando Saludos !!".getBytes();
    DatagramPacket envio = new DatagramPacket(msg, msg.length, destino, 12345);
} catch (UnknownHostException e) {
    e.printStackTrace();
}
```

El siguiente ejemplo utiliza el primer constructor para recibir el mensaje de un datagrama, el mensaje se aloja en `buffer`, luego se verá cómo se recupera la información del mensaje:

```
byte[] buffer = new byte[1024];
DatagramPacket recibo = new DatagramPacket(buffer, buffer.length);
```

Algunos métodos importantes son:

<code>InetAddress getAddress()</code>	Devuelve la dirección IP del host al cual se le envía el datagrama o del que el datagrama se recibió.
<code>byte[] getData()</code>	Devuelve el mensaje contenido en el datagrama tanto recibido como enviado.
<code>int getLength()</code>	Devuelve la longitud de los datos a enviar o a recibir.
<code>int getPort()</code>	Devuelve el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió el datagrama.
<code>setAddress(InetAddress addr)</code>	Establece la dirección IP de la máquina a la que se envía el datagrama.
<code>setData(byte[] buf)</code>	Establece el búfer de datos para este paquete.
<code>setLength(int length)</code>	Ajusta la longitud de este paquete.
<code>setPort(int port)</code>	Establece el número de puerto del host remoto al que este datagrama se envía.

El siguiente ejemplo obtiene la longitud y el mensaje del datagrama recibido, el mensaje se convierte a `String`. A continuación visualiza el número de puerto de la máquina que envía el mensaje y su dirección IP:

```
int bytesRecibidos = recibo.getLength();
String paquete = new String(recibo.getData());
System.out.println("Puerto origen del mensaje: " + recibo.getPort());
System.out.println("IP de origen : " + recibo.getAddress().getHostAddress());
```

Clase DatagramSocket

Da soporte a sockets para el envío y recepción de datagramas UDP. Algunos de los constructores de esta clase, que pueden lanzar la excepción `SocketException`, son:

<code>DatagramSocket()</code>	Construye un socket para datagramas, el sistema elige un puerto de los que están libres.
-------------------------------	--

<code>DatagramSocket(int port)</code>	Construye un socket para datagramas y lo conecta al puerto local especificado.
<code>DatagramSocket(int port, InetAddress ip)</code>	Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket.

Algunos métodos importantes son:

<code>receive(DatagramPacket paquete)</code>	Recibe un <i>DatagramPacket</i> del socket, y llena paquete con los datos que recibe (mensaje, longitud y origen). Puede lanzar <i>IOException</i> .
<code>send(DatagramPacket paquete)</code>	Envía un <i>DatagramPacket</i> a través del socket. El argumento paquete contiene el mensaje y su destino. Puede lanzar <i>IOException</i> .
<code>close()</code>	Se encarga de cerrar el socket.
<code>int getLocalPort()</code>	Devuelve el número de puerto en el host local al que está enlazado el socket, -1 si el socket está cerrado y 0 si no está enlazado a ningún puerto.
<code>int getPort()</code>	Devuelve el número de puerto al que está conectado el socket, -1 si no está conectado.
<code>connect(InetAddress address, int port)</code>	Conecta el socket a un puerto remoto y una dirección IP concretos, el socket solo podrá enviar y recibir mensajes desde esa dirección.
<code>setSoTimeout(int timeout)</code>	Permite establecer un tiempo de espera límite. Entonces el método <i>receive</i> se bloquea durante el tiempo fijado. Si no se reciben datos en el tiempo fijado se lanza la excepción <i>InterruptedIOException</i> .

En el siguiente ejemplo se construye un datagrama y lo enviamos usando un *DatagramSocket*, en el ejemplo se enlaza al puerto 34567. Mediante el método *send* se envía el datagrama:

```
try {
    InetAddress destino = InetAddress.getByName("192.168.1.10");
    int puerto = 12345;
    byte[] mensaje = "Enviando saludos !!".getBytes();
    DatagramPacket datagrama = new DatagramPacket(mensaje, mensaje.length,
        destino, puerto);
    DatagramSocket socket;
    try {
        socket = new DatagramSocket(34567);
        try {
            socket.send(datagrama);
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (SocketException e) {
        e.printStackTrace();
    }
} catch (UnknownHostException e) {
    e.printStackTrace();
}
```

En el otro extremo, para recibir el datagrama usamos también un *DatagramSocket*. En primer lugar habrá que enlazar el socket al puerto por el que se va a recibir el mensaje, en este caso el 12345. Después se construye el datagrama para recibir y mediante el método *receive* obtenemos los datos. Luego obtenemos la longitud, la cadena y visualizamos los puertos origen y destino del mensaje:

```

DatagramSocket socket;
try {
    socket = new DatagramSocket(12345);
    byte[] buffer = new byte[1024];
    DatagramPacket datagrama = new DatagramPacket(buffer, buffer.length);
    try {
        socket.receive(datagrama);
        int bytesRec = datagrama.getLength();
        String paquete= new String(datagrama.getData());
        System.out.println("Número de bytes recibidos: " + bytesRec);
        System.out.println("Contenido del Paquete: " + paquete.trim());
        System.out.println("Puerto origen del mensaje: " + datagrama.getPort());
        System.out.println("IP de origen:" +
            datagrama.getAddress().getHostAddress());
        System.out.println("Puerto destino del mensaje:" +
            socket.getLocalPort());
    } catch (IOException e) {
        e.printStackTrace();
    }
    socket.close();
} catch (SocketException e) {
    e.printStackTrace();
}

```

La salida muestra la siguiente información:

```

Número de bytes recibidos: 19
Contenido del Paquete: Enviando Saludos !!
Puerto origen del mensaje: 34567
IP de origen: 192.168.21.1
Puerto destino del mensaje:12345

```

5.1 Gestión de sockets UDP

En los sockets UDP no se establece conexión. Los roles cliente-servidor están un poco más difusos que en el caso de TCP. Podemos 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 necesitan saber en qué ordenador y en qué puerto está escuchando el otro.

En el flujo de la comunicación entre cliente y servidor que usan UDP, ambos necesitan crear un socket *DatagramSocket*:

- El servidor crea un socket asociado a un puerto local para escuchar peticiones de clientes. 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 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.
- El servidor permanece a la espera de recibir más peticiones.

Apertura y cierre de sockets:

Para construir un socket datagrama es necesario instanciar la clase *DatagramSocket* tanto en el programa cliente como en el servidor. Para escuchar peticiones en un puerto UDP concreto, creamos el *DatagramSocket* pasándole al constructor el número de puerto 34567 por el que escucha las peticiones y la dirección *InetAddress* en la que se está ejecutando el programa, que normalmente es *InetAddress.getLocalHost()*:

```
DatagramSocket socket = new DatagramSocket(34567,
    InetAddress.getByName("localhost"));
```

Para cerrar el socket usamos el método *socket.close()*.

Envío y recepción de datagramas:

El siguiente ejemplo crea un programa servidor que recibe un datagrama enviado por un programa cliente. El programa servidor permanece a la espera hasta que le llega un paquete del cliente; en este momento visualiza: el número de bytes recibidos, el contenido del paquete, el puerto y la IP del programa cliente y el puerto local por el que recibe las peticiones:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

public class ServidorUDP {

    public static void main(String[] args) {
        byte[] bufer = new byte[1024];
        DatagramSocket socket;
        try {
            socket = new DatagramSocket(12345);
            System.out.println("Esperando Datagrama ...");
            DatagramPacket datagrama = new DatagramPacket(bufer, bufer.length);
            try {
                socket.receive(datagrama);
                int bytesRecibidos = datagrama.getLength();
                String paquete= new String(datagrama.getData());
                System.out.println("Número de Bytes recibidos: " +
                    bytesRecibidos);
                System.out.println("Contenido del Paquete : " + paquete.trim());
                System.out.println("Puerto de origen: " +
                    datagrama.getPort());
                System.out.println("IP de origen : " +
                    datagrama.getAddress().getHostAddress());
                System.out.println("Puerto local:" + socket.getLocalPort());
            } catch (IOException e) {
                e.printStackTrace();
            }
            socket.close();
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }
}
```

El programa cliente envía un mensaje al servidor (máquina destino, en este caso es la máquina local, *localhost*) al puerto 12345 por el que espera peticiones. Visualiza el nombre del host de

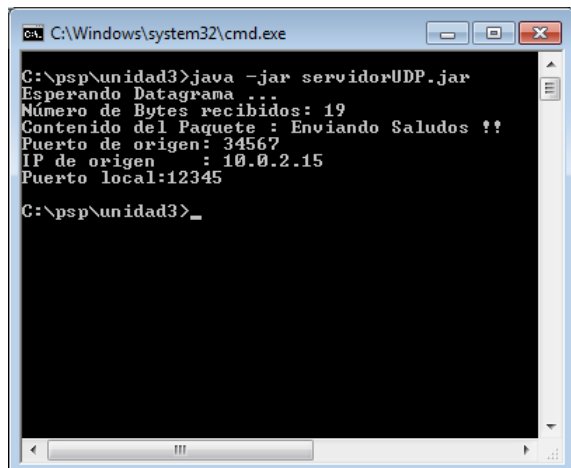
destino y la dirección IP. También visualiza el puerto local del socket y el puerto al que envía el mensaje:

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;

public class ClienteUDP {

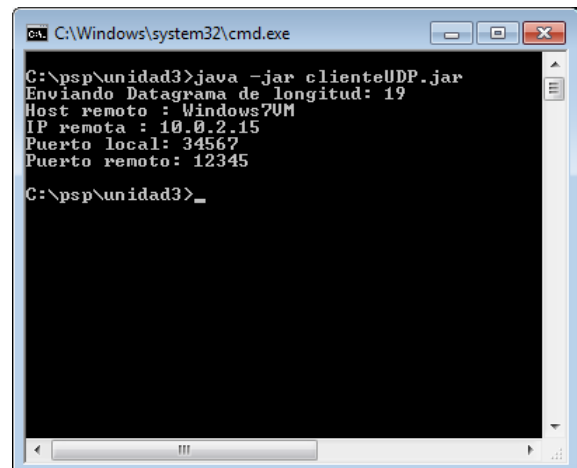
    public static void main(String[] args) {
        try {
            InetAddress destino = InetAddress.getLocalHost();
            int puerto = 12345;
            byte[] mensaje = "Enviando Saludos !!".getBytes();
            DatagramPacket datagrama = new DatagramPacket(mensaje, mensaje.length,
                destino, puerto);
            DatagramSocket socket;
            try {
                socket = new DatagramSocket(34567);
                try {
                    socket.send(datagrama);
                    System.out.println("Enviando Datagrama de longitud: " +
                        mensaje.length);
                    System.out.println("Host remoto : " + destino.getHostName());
                    System.out.println("IP remota : " + destino.getHostAddress());
                    System.out.println("Puerto local: " + socket.getLocalPort());
                    System.out.println("Puerto remoto: " + datagrama.getPort());
                } catch (IOException e) {
                    e.printStackTrace();
                }
                socket.close();
            } catch (SocketException e) {
                e.printStackTrace();
            }
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

En primer lugar, desde una consola ejecutamos el programa servidor, y una vez iniciado abrimos otra consola y ejecutamos el programa cliente. Como resultado se obtiene la salida siguiente:



```
ca. C:\Windows\system32\cmd.exe

C:\psp\unidad3>java -jar servidorUDP.jar
Esperando Datagrama ...
Número de Bytes recibidos: 19
Contenido del Paquete : Enviando Saludos !!
Puerto de origen: 34567
IP de origen : 10.0.2.15
Puerto local:12345
C:\psp\unidad3>_
```



```
ca. C:\Windows\system32\cmd.exe

C:\psp\unidad3>java -jar clienteUDP.jar
Enviando Datagrama de longitud: 19
Host remoto : Windows7UM
IP remota : 10.0.2.15
Puerto local: 34567
Puerto remoto: 12345
C:\psp\unidad3>_
```


PRÁCTICA 4

A partir del ejemplo anterior, modifica el cliente para que envíe al servidor líneas de texto obtenidas a través de la entrada estándar. Modifica el servidor para que responda devolviendo el texto en mayúsculas o finalice si recibe una línea que contenga únicamente un asterisco.

PRÁCTICA 5

Partiendo de la práctica 4, modifica el servidor para establecer con el método `setSoTimeout` de la clase `DatagramSocket` un tiempo de espera de 5 segundos para el método `receive`. Si transcurre el tiempo establecido y no se reciben datos, captura la excepción `InterruptedIOException` para visualizar un mensaje indicando que el paquete se ha perdido.

5.2 MulticastSocket

La clase `MulticastSocket` es útil para enviar paquetes a múltiples destinos simultáneamente. 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 de multicast, todos los que pertenezcan a ese grupo recibirán el mensaje; la pertenencia al grupo es transparente al 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 número de puerto UDP estándar. Las direcciones desde la 224.0.0.0 a la 239.255.255.255 están destinadas para ser direcciones de multicast. La dirección 224.0.0.0 está reservada y no debe ser utilizada.

La clase `MulticastSocket` tiene varios constructores (pueden lanzar la excepción `IOException`):

<code>MulticastSocket ()</code>	Construye un socket multicast dejando al sistema que elija un puerto de los que están libres.
<code>MulticastSocket (int port)</code>	Construye un socket multicast y lo conecta al puerto local especificado.

Algunos métodos importantes son (pueden lanzar la excepción `IOException`):

<code>joinGroup(InetAddress mcastaddr)</code>	Permite al socket multicast unirse al grupo de multicast.
<code>leaveGroup(InetAddress mcastaddr)</code>	El socket multicast abandona el grupo de multicast.
<code>send(DatagramPacket p)</code>	Envía el datagrama a todos los miembros del grupo multicast.
<code>receive(DatagramPacket p)</code>	Recibe el datagrama de un grupo multicast.

El esquema general para un servidor multicast que envía paquetes a todos los miembros del grupo es el siguiente:

```
try {
    /* se crea el socket multicast */
    MulticastSocket ms = new MulticastSocket();

    /* se define el Puerto multicast */
    int puerto = 12345;
```

```

/* se crea el grupo multicast */
InetAddress grupo = InetAddress.getByName("225.0.0.1");

/* se crea el datagrama: */
String msg = "Bienvenidos!!";
DatagramPacket paquete = new DatagramPacket(msg.getBytes(), msg.length(),
      grupo, puerto);

/* se envia el paquete al grupo */
ms.send(paquete);

/* se cierra el socket */
ms.close();
} catch (IOException e) {
    e.printStackTrace();
}

```

Para que un cliente se una al grupo multicast primero crea un *MulticastSocket* con el puerto deseado y luego invoca al método *joinGroup*. El cliente multicast que recibe los paquetes que le envía el servidor tiene la siguiente estructura:

```

try {
    /* se crea un socket multicast en el puerto 12345 */
    MulticastSocket ms = new MulticastSocket(12345);

    /* se configura la IP del grupo al que nos conectaremos */
    InetAddress grupo = InetAddress.getByName("225.0.0.1");

    /* nos unimos al grupo multicast */
    ms.joinGroup (grupo);

    /* recibe el paquete del servidor multicast */
    byte[] buf = new byte[1000];
    DatagramPacket datagrama = new DatagramPacket(buf, buf.length);
    ms .receive(datagrama) ;

    /* salimos del grupo multicast */
    ms.leaveGroup(grupo);

    /* se cierra el socket */
    ms.close();
} catch (IOException e) {
    e.printStackTrace();
}

```

En el siguiente ejemplo tenemos un servidor multicast que lee datos por teclado y los envía a todos los clientes que pertenezcan al grupo multicast, el proceso terminará cuando se introduzca una línea que contenga únicamente un carácter asterisco:

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class ServidorMC {

    public static void main(String[] args) {
        MulticastSocket ms = null;
        try {

```

```

        ms = new MulticastSocket();
        int puerto = 12345;
        InetAddress grupo = InetAddress.getByName("225.0.0.1");
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        String linea;
        do {
            System.out.print("Datos a enviar al grupo: ");
            linea = br.readLine();
            DatagramPacket datagrama = new DatagramPacket(
                linea.getBytes(), linea.length(), grupo, puerto);
            ms.send (datagrama);
        } while (!linea.trim().equals("*"));
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (ms != null) {
            ms.close();
            System.out.println("Socket cerrado...");
        }
    }
}

```

El programa cliente visualiza el paquete que recibe del servidor, su proceso finaliza cuando recibe un asterisco:

```

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class ClienteMC {

    public static void main(String[] args) {
        try {
            int Puerto = 12345;
            MulticastSocket ms = new MulticastSocket(Puerto);
            InetAddress grupo = InetAddress.getByName("225.0.0.1");
            ms.joinGroup(grupo);
            String msg;
            byte[] buf = new byte[1000];
            DatagramPacket datagrama = new DatagramPacket(buf, buf.length);
            do {
                ms.receive(datagrama);
                msg = new String(datagrama.getData(), 0, datagrama.getLength());
                System.out.println ("Recibo: " + msg.trim());
            } while (!msg.trim().equals("*"));
            ms.leaveGroup(grupo);
            ms.close();
            System.out.println ("Socket cerrado...");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Para probarlo ejecutamos el programa servidor en una consola y a continuación ejecutamos diferentes instancias del programa cliente. Como resultado se obtiene la salida siguiente:

```
C:\Windows\system32\cmd.exe
C:\psp\unidad3>java -jar servidorMC.jar
Datos a enviar al grupo: uno
Datos a enviar al grupo: dos
Datos a enviar al grupo: tres
Datos a enviar al grupo: cuatro
Datos a enviar al grupo: cinco
Datos a enviar al grupo: seis
Datos a enviar al grupo: *
Socket cerrado...
C:\psp\unidad3>
```

```
C:\Windows\system32\cmd.exe
C:\psp\unidad3>java -jar clienteMC.jar
Recibo: uno
Recibo: dos
Recibo: tres
Recibo: cuatro
Recibo: cinco
Recibo: seis
Recibo: *
Socket cerrado...
C:\psp\unidad3>
```

```
C:\Windows\system32\cmd.exe
C:\psp\unidad3>java -jar clienteMC.jar
Recibo: tres
Recibo: cuatro
Recibo: cinco
Recibo: seis
Recibo: *
Socket cerrado...
C:\psp\unidad3>
```

```
C:\Windows\system32\cmd.exe
C:\psp\unidad3>java -jar clienteMC.jar
Recibo: cinco
Recibo: seis
Recibo: *
Socket cerrado...
C:\psp\unidad3>
```

6 GESTIÓN DE LA CONEXIÓN DE MÚLTIPLES CLIENTES CON HILOS

Hasta ahora los programas servidores que hemos creado solo son capaces de atender a un cliente en cada momento, pero lo más típico es que un programa servidor pueda atender a muchos clientes simultáneamente. La solución para poder atender a múltiples clientes está en el multihilo, cada cliente será atendido en un hilo.

El esquema básico en sockets TCP sería construir un único servidor con la clase `ServerSocket` e invocar al método `accept` para esperar las peticiones de conexión de los clientes. Cuando un cliente se conecta, el objeto `Socket` devuelto por el método `accept` se usará en un nuevo hilo cuya misión es atender a este cliente. Después se vuelve a invocar a `accept` para esperar a un nuevo cliente; habitualmente la espera de conexiones se hace dentro de un bucle:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor {

    public static void main(String[] args) {
        try {
            ServerSocket servidor = new ServerSocket(6000);
            while (true) {
                Socket cliente = servidor.accept();
                HiloServidor hilo = new HiloServidor(cliente);
                hilo.start();
            }
        } catch (IOException e) {
```

```

        e.printStackTrace();
    }
    System.out.println("Servidor iniciado...");
}
}

```

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

Por ejemplo, supongamos que el cliente envía una cadena de caracteres al servidor y el servidor se la devuelve en mayúsculas, hasta que recibe un asterisco que finalizará la comunicación con el cliente. El proceso de tratamiento de la cadena se realiza en un hilo:

```

private static class HiloServidor extends Thread {

    BufferedReader entrada;
    PrintWriter salida;
    Socket socket;

    public HiloServidor(Socket socket) throws IOException {
        this.socket = socket;
        entrada = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        salida = new PrintWriter(socket.getOutputStream(), true);
    }

    @Override
    public void run() {
        System.out.println("Conectado con: " + socket.toString());
        String cadena = "";
        do {
            try {
                cadena = entrada.readLine();
                salida.println(cadena.trim().toUpperCase());
            } catch (IOException e) {
                e.printStackTrace();
            }
        } while (!cadena.trim().equals("*"));
        System.out.println("Fin conexión con: " + socket.toString());
        salida.close();
        try {
            entrada.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

```