

## PROGRAMACIÓN DE COMUNICACIONES EN RED

En este capítulo estudiaremos las clases de programación de sockets que nos permitirán crear una aplicación que se comunica con otra a través de la red. Estas aplicaciones se conocen como cliente-servidor. Una aplicación cliente es aquella que solicita información de otra aplicación que se conoce como servidor. La aplicación cliente es la que inicia la conexión y la aplicación servidor es la que responde a la solicitud de información.

En este capítulo estudiaremos las clases de programación de sockets que nos permitirán crear una aplicación que se comunica con otra a través de la red. Una aplicación cliente es aquella que solicita información de otra aplicación que se conoce como servidor. Una aplicación cliente es la que inicia la conexión y la aplicación servidor es la que responde a la solicitud de información.

### CAPÍTULO 3

## PROGRAMACIÓN DE COMUNICACIONES EN RED

### CONTENIDOS

- Clases Java para comunicaciones en red.
- Sockets. Tipos de sockets.
- Servidores y clientes basados en sockets.
- Gestión de sockets.

### OBJETIVOS

- Comunicar en red varias aplicaciones.
- Identificar los roles cliente y servidor.
- Utilizar y gestionar sockets.
- Realizar aplicaciones cliente-servidor con sockets.

### RESUMEN DEL CAPÍTULO

En este capítulo estudiaremos los sockets en Java. Aprenderemos a crear y gestionar aplicaciones cliente-servidor comunicándose a través de sockets.

### 3.1. INTRODUCCION

Antiguamente la programación de aplicaciones que comunican 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. Los diseñadores de las librerías Java han hecho que la programación en red para comunicar distintas máquinas no sea una tarea tan compleja.

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 el capítulo anterior, nos va a permitir la manipulación simultánea de múltiples conexiones.

En este capítulo usaremos Java para programar comunicaciones en red.

### 3.2. CLASES JAVA PARA COMUNICACIONES EN RED

**TCP/IP** es una familia de protocolos desarrollados para permitir la comunicación entre cualquier par de ordenadores de cualquier red o fabricante, respetando los protocolos de cada red individual. Tiene 4 capas o niveles de abstracción, Figura 3.1:

- **Capa de aplicación:** en este nivel se encuentran las aplicaciones disponibles para los usuarios. Por ejemplo FTP, SMTP, Telnet, HTTP, etc.
- **Capa de transporte:** suministra a las aplicaciones servicio de comunicaciones extremo a extremo utilizando dos tipos de protocolos: TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*).
- **Capa de red:** tiene como propósito seleccionar la mejor ruta para enviar paquetes por la red. El protocolo principal que funciona en esta capa es el **Protocolo de Internet (IP)**.
- **Capa de enlace o interfaz de red:** es la interfaz con la red real. Recibe los datagramas de la capa de red y los transmite al hardware de la red.

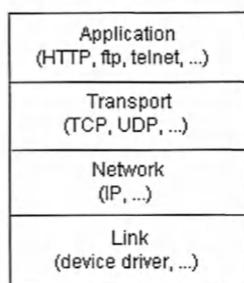


Figura 3.1. Modelo básico de red<sup>1</sup>.

Los equipos conectados a Internet se comunican entre sí utilizando el protocolo 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 y UDP; en su lugar, se pueden utilizar las clases del paquete **java.net**. Sin embargo existen algunas diferencias entre una y otra que conviene saber para decidir qué clases usar en los programas:

<sup>1</sup> Figura obtenida de: <http://docs.oracle.com/javase/tutorial/networking/overview/networking.html>

- **TCP:** protocolo basado en la conexión, garantiza que los datos enviados desde un extremo de la conexión llega al otro extremo y en el mismo orden en que fueron enviados. De lo contrario, se notifica un error.
- **UDP:** no está basado en la conexión como TCP. Envía paquetes de datos independientes, denominados **datagramas**, de una aplicación a otra; el orden de entrega no es importante y no se garantiza la recepción de los paquetes enviados.

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 abstracciones siguientes:

- Las 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:

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

### 3.2.1. La clase InetAddress

La clase **InetAddress** es la abstracción que representa una dirección IP (*Internet Protocol*). Tiene dos subclases: *Inet4Address* para direcciones IPv4 e *Inet6Address* para direcciones IPv6; pero en la mayoría de los casos **InetAddress** aporta la funcionalidad necesaria y no es necesario recurrir a ellas.

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

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

Los 3 primeros métodos pueden lanzar la excepción *UnknownHostException*. En el siguiente ejemplo se define un objeto **InetAddress** de nombre *dir*. En primer lugar lo utilizamos para obtener la dirección IP de la máquina local en la que se ejecuta el programa, en el ejemplo su-

nombre es *NUEVOMJ*. A continuación llamamos al método *pruebaMetodos()* llevando el objeto creado. En dicho método se prueban los métodos de la clase **InetAddress**. Después utilizamos el objeto para obtener la dirección IP de la URL *www.google.es* y volvemos a invocar a *pruebaMetodos()* (para que funcione en este segundo caso necesitamos estar conectados a Internet). Por último utilizamos el método *getAllByName()* para ver todas las direcciones IP asignadas a la máquina representada por *www.google.es*. Se encierra todo en un bloque **try-catch**:

```

} ///pruebaMetodos

}//fin

```

La salida generada es la siguiente:

```
D:\CAPIT3>javac TestInetAddress.java
```

```
D:\CAPIT3>java TestInetAddress
```

```
=====
SALIDA PARA LOCALHOST:
```

```

    Metodo getByName(): NUEVOMJ/192.168.0.193
    Metodo getLocalHost(): NUEVOMJ/192.168.0.193
    Metodo getHostName(): NUEVOMJ
    Metodo getHostAddress(): 192.168.0.193
    Metodo toString(): NUEVOMJ/192.168.0.193
    Metodo getCanonicalHostName(): 192.168.0.193
=====
```

```
SALIDA PARA UNA URL:
```

```

    Metodo getByName(): www.google.es/173.194.41.215
    Metodo getLocalHost(): NUEVOMJ/192.168.0.193
    Metodo getHostName(): www.google.es
    Metodo getHostAddress(): 173.194.41.215
    Metodo toString(): www.google.es/173.194.41.215
    Metodo getCanonicalHostName(): lis01s05-in-f23.1e100.net
    DIRECCIONES IP PARA: www.google.es
        www.google.es/173.194.41.215
        www.google.es/173.194.41.216
        www.google.es/173.194.41.223
=====
```

### ACTIVIDAD 3.1

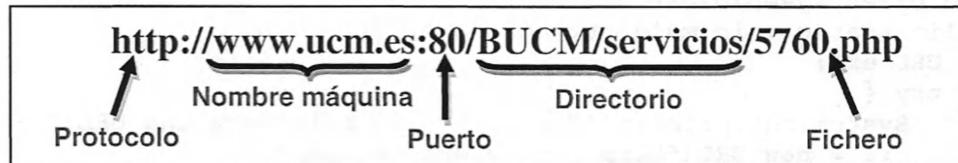
Realiza un programa Java que admita desde la línea de comandos un nombre de máquina o una dirección IP y visualice información sobre ella.

#### 3.2.2. 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 siguiente URL: <http://www.ucm.es/BUCM/servicios/5760.php>, encontramos el protocolo (*http*), el nombre de máquina (*www.ucm.es*) y el fichero (*5760.php*) que está en un directorio dentro del servidor (*/BUCM/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 80, la URL anterior con el puerto sería:



La clase **URL** contiene varios constructores, algunos son:

CONSTRUCTOR	MISIÓN
<b>URL (String url)</b>	Crea un objeto URL a partir del String <i>url</i>
<b>URL(String protocolo, String host, String fichero)</b>	Crea un objeto URL a partir de los parámetros <i>protocolo</i> , <i>host</i> y <i>fichero</i>
<b>URL(String protocolo, String host, int puerto, String fichero)</b>	Crea un objeto URL en el que se especifica el protocolo, host, puerto y fichero representados mediante String
<b>URL(URL context, String url)</b>	Crea un objeto URL a partir de la dirección del host dada por <i>URL context</i> y una URL relativa dada en el String (un directorio)

Estos 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:

MÉTODOS	MISIÓN
<b>String getAuthority()</b>	Obtiene la autoridad del objeto URL
<b>int getDefaultPort()</b>	Devuelve el puerto asociado por defecto al objeto URL
<b>int getPort()</b>	Devuelve el número de puerto de la URL, -1 si no se indica
<b>String getHost()</b>	Devuelve el nombre de la máquina
<b>String getQuery()</b>	Devuelve la cadena que se envía a una página para ser procesada (es lo que sigue al signo ? de una URL)
<b>String getPath()</b>	Devuelve una cadena con la ruta hacia el fichero desde el servidor y el nombre completo del fichero
<b>String getFile()</b>	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
<b>String getUserInfo()</b>	Devuelve la parte con los datos del usuario de la dirección URL o nulo si no existe
<b>InputStream openStream()</b>	Abre una conexión al objeto URL y devuelve un <b>InputStream</b> para la lectura de esa conexión
<b>URLConnection openConnection()</b>	Devuelve un objeto <b>URLConnection</b> 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()* muestra información de la URL usando los métodos de la tabla anterior:

```
import java.net.*;
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://localhost/PFC/gest/cli_gestion.php?S=3");
    Visualizar(url);

    System.out.println("Const. para protocolo +URL +directorío:");
    url = new URL("http", "docs.oracle.com", "/javase/7");
    Visualizar(url);

    System.out.println("Constructor para protocolo + URL + puerto +"
                       "directorío:");
    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) { System.out.println(e); }
// main

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("=====");
}
// Ejemplo1URL

```

La salida generada es la siguiente:

D:\CAPIT3>javac Ejemplo1URL.java

D:\CAPIT3>java Ejemplo1URL  
 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://localhost/PFC/gest/cli_gestion.php?S=3
    getProtocol(): http
    getHost(): localhost
    getPort(): -1
    getFile(): /PFC/gest/cli_gestion.php?S=3
    getUserInfo(): null
    getPath(): /PFC/gest/cli_gestion.php
    getAuthority(): localhost
    getQuery(): S=3
=====
Const. 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 <http://www.elaltozano.es>, 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.net.*;
import java.io.*;

public class Ejemplo2URL {
```

```

public static void main(String[] args) {
    URL url=null;
    try {
        url = new URL("http://www.elaltozano.es");
    } 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(); }
}
//Ejemplo2URL

```

### 3.2.3. 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, entonces tenemos una instancia de un objeto de la clase **URLConnection**:

```

URL url = new URL("http://www.elaltozano.es");
URLConnection urlCon= url.openConnection();

```

La clase **URLConnection** es una clase abstracta que contiene métodos que permiten la comunicación entre la aplicación y una URL. Para conseguir un objeto de este tipo se invoca al método **openConnection()**, con ello obtenemos una conexión al objeto URL referenciado. 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:

MÉTODOS	MISIÓN
<b>InputStream getInputStream()</b>	Devuelve un objeto <b>InputStream</b> para leer datos de esta conexión
<b>OutputStream getOutputStream()</b>	Devuelve un objeto <b>OutputStream</b> para escribir datos en esta conexión
<b>void setDoInput (boolean b)</b>	Permite que el usuario reciba datos desde la URL si el parámetro <i>b</i> es <i>true</i> (por defecto está establecido <i>o true</i> )
<b>void setDoOutput (boolean b)</b>	Permite que el usuario envíe datos si el parámetro <i>b</i> es <i>true</i> (no está establecido al principio)
<b>void connect()</b>	Abre una conexión al recurso remoto si tal conexión no se ha establecido ya
<b>int getContentLength()</b>	Devuelve el valor del campo de cabecera <i>content-length</i> o -1 si no está definido

<code>String getContentType()</code>	Devuelve el valor del campo de cabecera <i>content-type</i> o null si no está definido
<code>long getDate()</code>	Devuelve el valor del campo de cabecera <i>date</i> o 0 si no está definido
<code>long getLastModified()</code>	Devuelve el valor del campo de cabecera <i>last-modified</i>
<code>String getHeaderField(int n)</code>	Devuelve el valor del enésimo campo de cabecera especificado o null si no está definido
<code>Map&lt;String, List&lt;String&gt;&gt; getHeaderFields()</code>	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
<code>URL getURL()</code>	Devuelve la dirección URL

El siguiente ejemplo crea un objeto URL a la dirección *http://www.elaltozano.es*, 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.net.*;
import java.io.*;

public class Ejemplo1urlCon {
    public static void main(String[] args) {
        URL url=null;
        URLConnection urlCon=null;
        try {
            url = new URL("http://www.elaltozano.es");
            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();}
    }
}//Ejemplo1urlCon
```

Gran cantidad de páginas HTML contienen formularios a través de los cuales podemos solicitar información a un servidor rellenando los campos requeridos y pulsando al botón de envío. El servidor recibe la petición, la procesa y envía los datos solicitados al cliente normalmente en formato HTML. Por ejemplo, tenemos una página HTML que contiene un formulario con dos campos de entrada y un botón. En el atributo “action” se indica el tipo de

acción que va a realizar el formulario, en este caso los datos se envían a un script PHP de nombre *vernombre.php*; con *method=post* e indicamos la forma en que se envía el formulario:

```
<html>
<body>
  <form action="vernombre.php" method="post" >
    <p>Escribe tu nombre:
      <input name="nombre" type="text" size="15"></p>
    <p>Escribe tus apellidos:
      <input name="apellidos" type="text" size="15"></p>
      <input type="submit" name="ver" value="Ver">
  </form>
</body>
</html>
```

El script PHP que recibe los datos del formulario es el siguiente:

```
<?php
$nom=$_POST[nombre];
$ape=$_POST[apellidos];
echo "El nombre recibido es: $nom, y ";
echo "los apellidos son: $ape ";
?>
```

Lo único que hace es recibir el valor introducido en los campos *nombre* y *apellidos* del formulario mediante la instrucción *\$\_POST[campo]* y visualizarlos en pantalla mediante la orden *echo*. Desde Java usando la clase **URLConnection** podemos interactuar con scripts del lado del servidor y podemos enviar valores a los campos del script sin necesidad de abrir un formulario HTML, será necesario escribir en la URL para dar los datos al script. Nuestro programa tendrá que hacer lo siguiente:

- Crear el objeto URL al script con el que va a interactuar. Por ejemplo, en nuestra máquina local tenemos instalado un servidor web Apache y dentro de *htdocs* tenemos la carpeta *2014* con el script PHP *vernombre.php*, la URL sería la siguiente: *URL url = new URL("http://localhost/2014/vernombre.php")*.
- Abrir una conexión con la URL, es decir obtener el objeto **URLConnection**: *URLConnection conexion = url.openConnection()*.
- Configurar la conexión para que se puedan enviar datos: *conexion.setDoOutput(true)*.
- Obtener un stream de salida sobre la conexión: *PrintWriter output = new PrintWriter(conexion.getOutputStream())*.
- Escribir en el stream de salida, en este caso mandamos una cadena con los datos que necesita el script: *output.write(cadena)*. La cadena tiene el siguiente formato: *parámetro=valor*, si el script recibe varios parámetros sería: *parámetro1=valor1&parámetro2=valor2&parámetro3=valor3*, y así sucesivamente.
- Cerrar el stream de salida: *output.close()*.

Normalmente cuando se pasa información a algún script PHP, este realiza alguna acción y después envía la información de vuelta por la misma URL. Por tanto, si queremos ver lo que devuelve será necesario leer desde la URL. Para ello se abre un stream de entrada sobre esa conexión mediante el método **getInputStream()**: *BufferedReader reader = new BufferedReader*

(new InputStreamReader(conexion.getInputStream())); y después se realiza la lectura para obtener los resultados devueltos por el script. El código completo es el siguiente:

```

import java.io.*;
import java.net.*;

public class Ejemplo2urlCon {
    public static void main(String[] args) {
        try {
            URL url = new URL("http://localhost/2014/vernombre.php");
            URLConnection conexion = url.openConnection();
            conexion.setDoOutput(true);

            String cadena = "nombre=Maria Jesus&apellidos=Ramos Martin";

            //ESCRIBIR EN LA URL
            PrintWriter output = new PrintWriter
                (conexion.getOutputStream());
            output.write(cadena);
            output.close(); //cerrar flujo

            //LEER DE LA URL
            BufferedReader reader = new BufferedReader
                (new InputStreamReader(conexion.getInputStream()));
            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println(linea);
            }
            reader.close(); //cerrar flujo

        } catch (MalformedURLException me) {
            System.err.println("MalformedURLException: " + me);
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
    } //main
} //Ejemplo2urlCon

```

La compilación y ejecución muestran la siguiente salida:

D:\CAPIT3>javac Ejemplo2urlCon.java

D:\CAPIT3>java Ejemplo2urlCon

El nombre recibido es: Maria Jesus, y los apellidos son: Ramos Martin

En el siguiente ejemplo se prueban algunos de los métodos de la clase **URLConnection**:

```

import java.net.*;
import java.io.*;
import java.util.*;

public class Ejemplo3urlCon {
    @SuppressWarnings("rawtypes")
    public static void main(String[] args) throws Exception {

```

```

String cadena;
URL url = new URL("http://localhost/2014/vernombre.html");
URLConnection conexion = url.openConnection();

System.out.println("Direccion [getURL()]: " + conexion.getURL());

Date fecha = new Date(conexion.getLastModified());
System.out.println("Fecha ultima modificacion
[getLastModified()]: " + fecha);
System.out.println("Tipo de Contenido [getContentType()]: "
+ conexion.getContentType());

System.out.println("===== ");
System.out.println("TODOS LOS CAMPOS DE CABECERA CON
getHeaderFields(): ");

//USAMOS UNA ESTRUCTURA Map PARA RECUPERAR CABECERAS
Map camposcabecera = conexion.getHeaderFields();
Iterator it = camposcabecera.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry map = (Map.Entry) it.next();
    System.out.println(map.getKey() + " : " + map.getValue());
}

System.out.println("===== ");
System.out.println("CAMPOS 1 Y 4 DE CABECERA:");
System.out.println("getHeaderField(1)=> "+
conexion.getHeaderField(1));
System.out.println("getHeaderField(4)=> "+
conexion.getHeaderField(4));
System.out.println("===== ");

System.out.println("CONTENIDO DE [url.getFile()]: "+url.getFile());
BufferedReader pagina = new BufferedReader
(new InputStreamReader(url.openStream()));

while ((cadena = pagina.readLine()) != null) {
    System.out.println(cadena);
}
}
}
}
//
```

**NOTA:** Para recorrer una estructura **Map** podemos usar una estructura **Iterator**. Para obtener un iterador sobre el map se invoca a los métodos *entrySet()* e *iterator()*. Para mover el iterador utilizaremos el método *next()* y para comprobar si ha llegado al final usamos el método *hasNext()*. De la estructura recuperaremos los valores mediante *getKey()*, para la clave y *getValue()*, para el valor.

La compilación y ejecución muestra la siguiente salida:

D:\CAPIT3>javac Ejemplo3urlCon.java

D:\CAPIT3>java Ejemplo3urlCon

```

Direccion [getURL()]:http://localhost/2014/vernombre.html
Fecha ultima modificacion [getLastModified()]: Thu Jan 24 18:27:22 CET
2013
Tipo de Contenido [getContentType()]: text/html
=====
TODOS LOS CAMPOS DE CABECERA CON getHeaderFields():
null : [HTTP/1.1 200 OK]
ETag : ["1f0000000f63b-138-4d40c1fabceaf"]
Date : [Thu, 31 Jan 2013 15:37:44 GMT]
Content-Length : [312]
Last-Modified : [Thu, 24 Jan 2013 17:27:22 GMT]
Keep-Alive : [timeout=5, max=100]
Content-Type : [text/html]
Connection : [Keep-Alive]
Accept-Ranges : [bytes]
Server : [Apache/2.2.11 (Win32) DAV/2 mod_ssl/2.2.11 OpenSSL/0.9.8i
mod_autoindex_color PHP/5.2.8]
=====
CAMPOS 1 Y 4 DE CABECERA:
getHeaderField(1) => Thu, 31 Jan 2013 15:37:44 GMT
getHeaderField(4) => "1f0000000f63b-138-4d40c1fabceaf"
=====
CONTENIDO DE [url.getFile()]:/2014/vernombre.html
<html>
<body>
<form action="vernombre.php" method="post" >
<p>Escribe tu nombre:</p>
<input name="nombre" type="text" size="15"></p>
<p>Escribe tus apellidos:</p>
<input name="apellidos" type="text" size="15"></p>
<input type="submit" name="ver" value="Ver">
</form>
</body>
</html>

```

### 3.3. QUÉ SON LOS 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 este conector es a lo que llamamos **socket**.

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. La Figura 3.2 muestra un proceso cliente (envía un mensaje) y un proceso servidor (recibe un mensaje) comunicándose mediante sockets. 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.

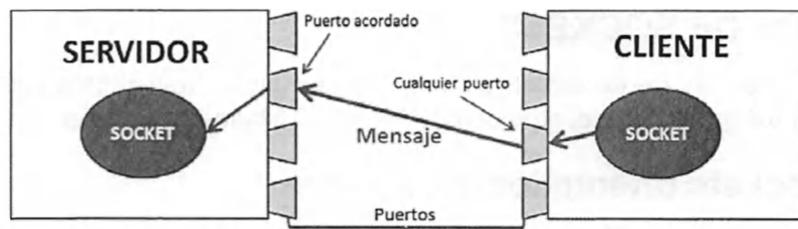


Figura 3.2. Socket y puertos.

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.

### 3.3.1. 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, Figura 3.3; el cliente también debe identificarse ante el servidor por lo que durante la conexión se utilizará un puerto local asignado por el sistema.



Figura 3.3. Petición de conexión del cliente.

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, Figura 3.4.

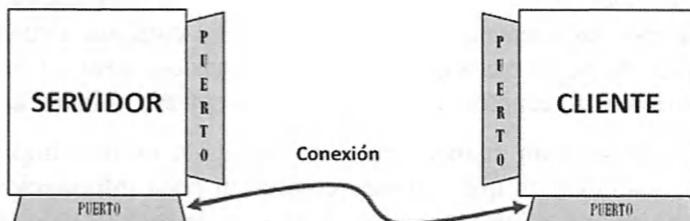


Figura 3.4. Conexión cliente-servidor.

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.

## 3.4. 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.

### 3.4.1. 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: FTP (puerto 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.

### 3.4.2. 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**.

### 3.5. 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.

#### 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*):

CONSTRUCTOR	MISIÓN
<b>ServerSocket()</b>	Crea un socket de servidor sin ningún puerto asociado
<b>ServerSocket(int port)</b>	Crea un socket de servidor, que se enlaza al puerto especificado
<b>ServerSocket(int port, int máximo)</b>	Crea un socket de servidor y lo enlaza con el número de puerto local especificado. El parámetro <i>máximo</i> especifica, el número máximo de peticiones de conexión que se pueden mantener en cola
<b>ServerSocket(int port, int máximo, InetAddress direc)</b>	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:

MÉTODOS	MISIÓN
<b>Socket accept ()</b>	El método <i>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 <b>Socket</b> , a través del cual se establecerá la comunicación con el cliente. Tras esto, el <b>ServerSocket</b> sigue disponible para realizar nuevos <i>accept()</i> . Puede lanzar <i>IOException</i>
<b>close ()</b>	Se encarga de cerrar el <b>ServerSocket</b>
<b>int getLocalPort ()</b>	Devuelve el puerto local al que está enlazado el <b>ServerSocket</b>

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:

```
int Puerto = 6000;// Puerto
ServerSocket Servidor = new ServerSocket(Puerto);
System.out.println("Escuchando en " + Servidor.getLocalPort());
Socket cliente1= Servidor.accept();//esperando a un cliente
//realizar acciones con cliente1
Socket cliente2 = Servidor.accept();//esperando a otro cliente
//realizar acciones con cliente2
Servidor.close(); //cierro socket servidor
```

## Clase Socket

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

CONSTRUCTOR	MISIÓN
<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 <i>UnKnownHostException</i> , <i>IOException</i>

Algunos métodos importantes son:

MÉTODOS	MISIÓN
<code>InputStream getInputStream ()</code>	Devuelve un <b>InputStream</b> 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 <b>OutputStream</b> 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 null
<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). Despues 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):

```

String Host = "localhost";
int Puerto = 6000;//puerto remoto

// ABRIR SOCKET
Socket Cliente = new Socket(Host, Puerto);//conecta

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();// Cierra el socket

```

La salida que se genera es la siguiente:

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

### 3.5.1. Gestión de Sockets TCP

El modelo de sockets más simple se muestra en la Figura 3.5:

- El programa servidor crea un socket de servidor definiendo un puerto, mediante el método `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 la máquina host a través del puerto especificado mediante el método `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**.

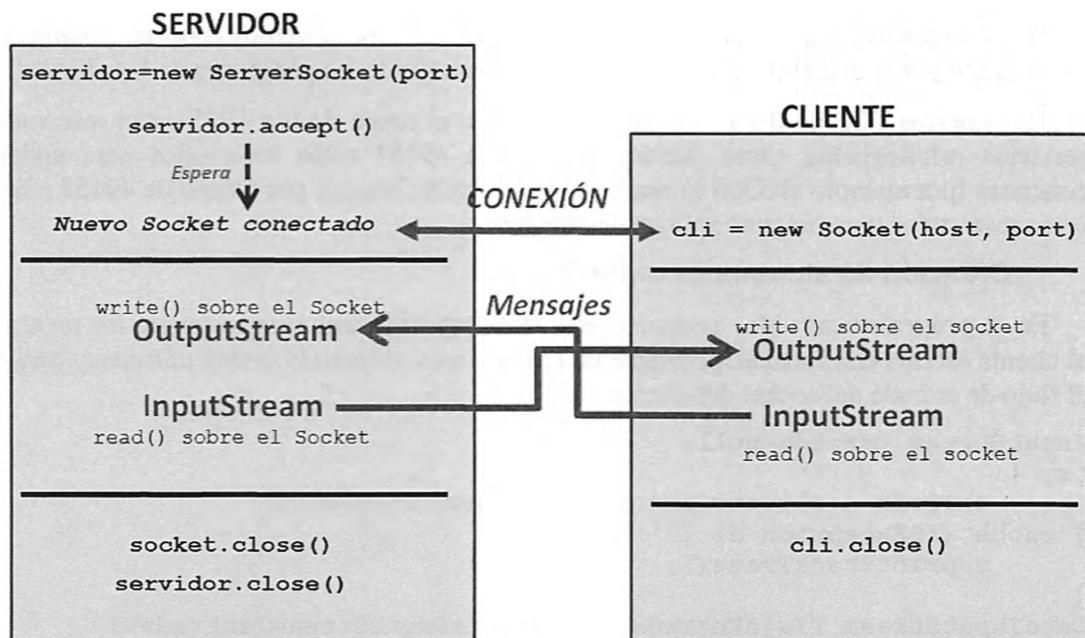


Figura 3.5. Modelo de Socket TCP.

### Apertura de sockets

En el **programa servidor** se crea un objeto **ServerSocket** invocando al método **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=null;
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 io) {
    io.printStackTrace();
}
```

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

```
Socket cliente;
try {
    cliente = new Socket("máquina", numeroPuerto);
} catch (IOException io) {
    io.printStackTrace();
}
```

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

Hay puertos TCP 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 entrada=null;
try {
    entrada = clienteConectado.getInputStream();
} catch (IOException e) {
    e.printStackTrace();
}
DataInputStream flujoEntrada = new DataInputStream(entrada);
```

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

La clase **DataInputStream** permite la lectura de líneas de texto y tipos primitivos Java. Algunos de sus métodos son: *readInt()*, *readDouble()*, *readLine()*, *readUTF()*, etc.

### 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 salida=null;
try {
    salida = clienteConectado.getOutputStream();
} catch (IOException e1) {
    e1.printStackTrace();
}
DataOutputStream flujoSalida = new DataOutputStream(salida);
```

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

La clase **DataOutputStream** dispone de métodos para escribir tipos primitivos Java: *writeInt()*, *writeDouble()*, *writeBytes()*, *writeUTF()*, etc.

### 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:

```
try {
    entrada.close();
    flujoEntrada.close();
    salida.close();
    flujoSalida.close();
    clienteConectado.close();
    servidor.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

A continuación se muestra un ejemplo de un **programa servidor** que recibe un mensaje de un cliente y lo muestra por pantalla; después envía un mensaje al cliente. Se han eliminado los bloques **try-catch** para que el código resulte más legible:

```
import java.io.*;
import java.net.*;

public class ejemplo1Servidor {
    public static void main(String[] arg) throws IOException {
        int numeroPuerto = 6000;// Puerto
        ServerSocket servidor = new ServerSocket(numeroPuerto);
        Socket clienteConectado = null;
        System.out.println("Esperando al cliente.....");
        clienteConectado = servidor.accept();

        // CREO FLUJO DE ENTRADA DEL CLIENTE
```

```

InputStream entrada = null;
entrada = clienteConectado.getInputStream();
DataInputStream flujoEntrada = new DataInputStream(entrada);
// EL CLIENTE ME ENVIA UN MENSAJE
System.out.println("Recibiendo del CLIENTE: \n\t" +
    flujoEntrada.readUTF());

// CREO FLUJO DE SALIDA AL CLIENTE
OutputStream salida = null;
salida = clienteConectado.getOutputStream();
DataOutputStream flujoSalida = new DataOutputStream(salida);

// ENVIO UN SALUDO AL CLIENTE
flujoSalida.writeUTF("Saludos al cliente del servidor");

// CERRAR STREAMS Y SOCKETS
entrada.close();
flujoEntrada.close();
salida.close();
flujoSalida.close();
clienteConectado.close();
servidor.close();
}// main
}// fin

```

El **programa cliente**, en primer lugar envía un mensaje al servidor y después recibe un mensaje del servidor visualizándolo en pantalla, se ha simplificado la obtención de los flujos de entrada y salida:

```

import java.io.*;
import java.net.*;

public class ejemplo1Cliente {
    public static void main(String[] args) throws Exception {
        String Host = "localhost";
        int Puerto = 6000;//puerto remoto

        System.out.println("PROGRAMA CLIENTE INICIADO....");
        Socket Cliente = new Socket(Host, Puerto);

        // CREO FLUJO DE SALIDA AL SERVIDOR
        DataOutputStream flujoSalida = new
            DataOutputStream(Cliente.getOutputStream());

        // ENVIO UN SALUDO AL SERVIDOR
        flujoSalida.writeUTF("Saludos al SERVIDOR DESDE EL CLIENTE");

        // CREO FLUJO DE ENTRADA AL SERVIDOR
        DataInputStream flujoEntrada = new
            DataInputStream(Cliente.getInputStream());

        // EL SERVIDOR ME ENVIA UN MENSAJE

```

```

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

        // CERRAR STREAMS Y SOCKETS
        flujoEntrada.close();
        flujoSalida.close();
        Cliente.close();
    } // main
}

```

La compilación y ejecución se muestra en la Figura 3.6. En una ventana se ejecuta el programa servidor y en otra se ejecuta el programa cliente.

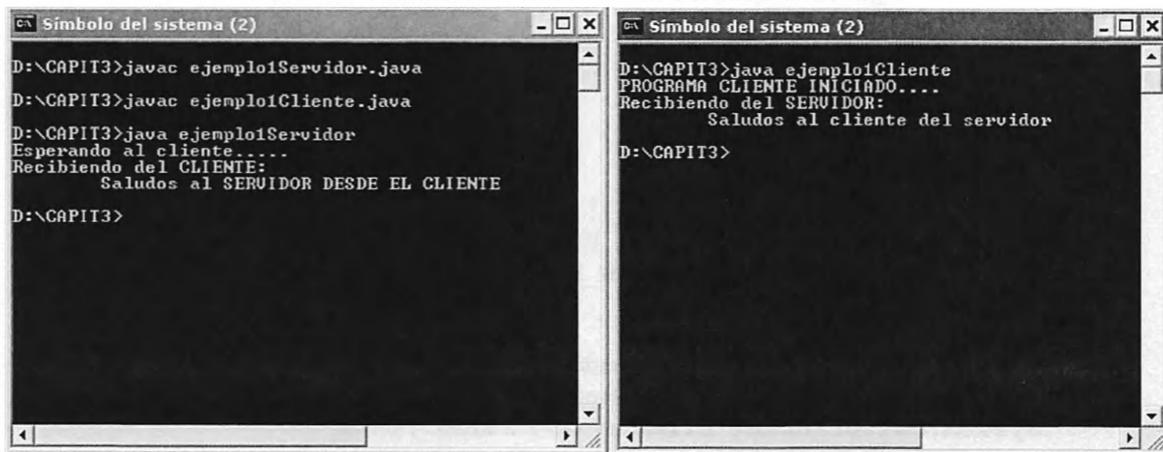


Figura 3.6. Ejecución de un programa cliente y otro servidor con TCP.

### ACTIVIDAD 3.2

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

### ACTIVIDAD 3.3

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. La Figura 3.7 muestra la ejecución.



Figura 3.7. Ejemplo servidor recibe del cliente y envía lo que recibe.

El programa servidor es el siguiente:

```

import java.io.*;
import java.net.*;

public class ejemplo2Servidor {
    public static void main(String[] arg) throws IOException {
        int numeroPuerto = 6000;// Puerto
        ServerSocket servidor = new ServerSocket(numeroPuerto);
        String cad="";

        System.out.println("Esperando conexión...");
        Socket clienteConectado = servidor.accept();
        System.out.println("Cliente conectado...");

        // CREO FLUJO DE SALIDA AL CLIENTE
        PrintWriter fsalida = new PrintWriter
            (clienteConectado.getOutputStream(),true);
        // CREO FLUJO DE ENTRADA DEL CLIENTE
        BufferedReader fentrada = new BufferedReader
            (new InputStreamReader(clienteConectado.getInputStream()));

        while ((cad=fentrada.readLine())!= null)//recibo cad del cliente
        {
            fsalida.println(cad); //envio cadena al cliente
            System.out.println("Recibiendo: " + cad);
            if(cad.equals("*")) break;
        }
        // CERRAR STREAMS Y SOCKETS
        System.out.println("Cerrando conexión...");
        fentrada.close();
        fsalida.close();
        clienteConectado.close();
        servidor.close();
    }
}

```

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

El programa cliente es el siguiente:

```

import java.io.*;
import java.net.*;

public class ejemplo2Cliente {
    public static void main(String[] args) throws IOException {
        String Host = "localhost";
        int Puerto = 6000;// puerto remoto
        Socket Cliente = new Socket(Host, Puerto);

        // CREO FLUJO DE SALIDA AL SERVIDOR
        PrintWriter fsalida = new PrintWriter
            (Cliente.getOutputStream(), true);
        // CREO FLUJO DE ENTRADA AL SERVIDOR
        BufferedReader fentrada = 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(); //lectura por teclado
        while(cadena !=null){
            fsalida.println(cadena); //envio cadena al servidor
            eco=fentrada.readLine(); //recibo cadena del servidor
            System.out.println(" =>ECO: "+eco);
            System.out.print("Introduce cadena: ");
            cadena = in.readLine(); //lectura por teclado
        }
        fsalida.close();
        fentrada.close();
        System.out.println("Fin del envío... ");
        in.close();
        Cliente.close();
    //}
}

```

### 3.6. 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	DIRECCIÓN IP DESTINO	Nº DE PUERTO DESTINO
---	-------------------------	-------------------------	-------------------------

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

### Clase DatagramPacket

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

CONSTRUCTOR	MISIÓN
<b>DatagramPacket(byte[] buf, int length)</b>	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
<b>DatagramPacket(byte[] buf, int offset, int length)</b>	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje, la longitud de la misma y el offset ( <i>offset</i> ) dentro de la cadena
<b>DatagramPacket(byte[] buf, int length, InetAddress addrss, int port)</b>	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>addrss</i>
<b>DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)</b>	Igual que el anterior pero se especifica un offset 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 !!* 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:

<i>mensaje: Enviando Saludos !!,</i>	<i>Longitud: 19</i>	<i>destino: 192.168.21</i> <i>IP del host local</i>	<i>port:12345</i>
--------------------------------------	---------------------	--	-------------------

```

int port = 12345; //puerto al que envío
InetAddress destino = InetAddress.getLocalHost(); //IP a la que envío
byte[] mensaje = new byte[1024]; //matriz de bytes
String Saludo = "Enviando Saludos !!";
mensaje = Saludo.getBytes(); //codificarlo a bytes para enviarlo

//construyo el datagrama a enviar
DatagramPacket envio = new DatagramPacket
(mensaje, mensaje.length, destino, port);

```

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

```

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

```

Algunos métodos importantes son:

MÉTODOS	MISIÓN
<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 bytesRec = recibo.getLength();//obtengo longitud del mensaje
String paquete= new String(recibo.getData())//obtengo mensaje
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:

CONSTRUCTOR	MISIÓN
<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

El siguiente ejemplo construye un socket para datagrama y no lo conecta a ningún puerto, el sistema elige el puerto:

```
DatagramSocket socket = new DatagramSocket();
```

Para enlazar el socket a un puerto específico, por ejemplo al puerto 34567, escribimos:

```
DatagramSocket socket = new DatagramSocket(34567);
```

Algunos métodos importantes son:

MÉTODOS	MISIÓN
<b>receive (DatagramPacket paquete)</b>	Recibe un <b>DatagramPacket</b> del socket, y llena <i>paquete</i> con los datos que recibe (mensaje, longitud y origen). Puede lanzar <i>IOException</i>
<b>send (DatagramPacket paquete)</b>	Envía un <b>DatagramPacket</b> a través del socket. El argumento <i>paquete</i> contiene el mensaje y su destino. Puede lanzar <i>IOException</i>
<b>close ()</b>	Se encarga de cerrar el socket
<b>int getLocalPort ()</b>	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
<b>int getPort()</b>	Devuelve el número de puerto al que está conectado el socket, -1 si no está conectado
<b>connect(InetAddress address, int port)</b>	Conecta el socket a un puerto remoto y una dirección IP concretos, el socket solo podrá enviar y recibir mensajes desde esa dirección
<b>setSoTimeout(int timeout)</b>	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>InterruptedException</i>

Siguiendo con el ejemplo inicial, una vez construido el datagrama lo enviamos usando un **DatagramSocket**, en el ejemplo se enlaza al puerto 34567. Mediante el método *send()* se envía el datagrama:

```
//construyo datagrama a enviar indicando el host destino y puerto
DatagramPacket envio = new DatagramPacket
    (mensaje, mensaje.length, destino, port);
DatagramSocket socket = new DatagramSocket(34567);
socket.send(envio); //envio datagrama a destino y port
```

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 = new DatagramSocket(12345);
//construyo datagrama a recibir
DatagramPacket recipro = new DatagramPacket(bufer, bufer.length);
socket.receive(recipro); //recipro datagrama

int bytesRec = recipro.getLength(); //obtengo numero de bytes
String paquete= new String(recipro.getData()); //obtengo String

System.out.println("Número de Bytes recibidos: " + bytesRec);
System.out.println("Contenido del Paquete: " + paquete.trim());
System.out.println("Puerto origen del mensaje: " + recipro.getPort());
System.out.println("IP de origen: " + recipro.getAddress().getHostAddress());
System.out.println("Puerto destino del mensaje: " + socket.getLocalPort());
socket.close(); //cierro el socket
```

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

```

### 3.6.1. 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.

La Figura 3.8 muestra el flujo de la comunicación entre cliente y servidor usando 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.

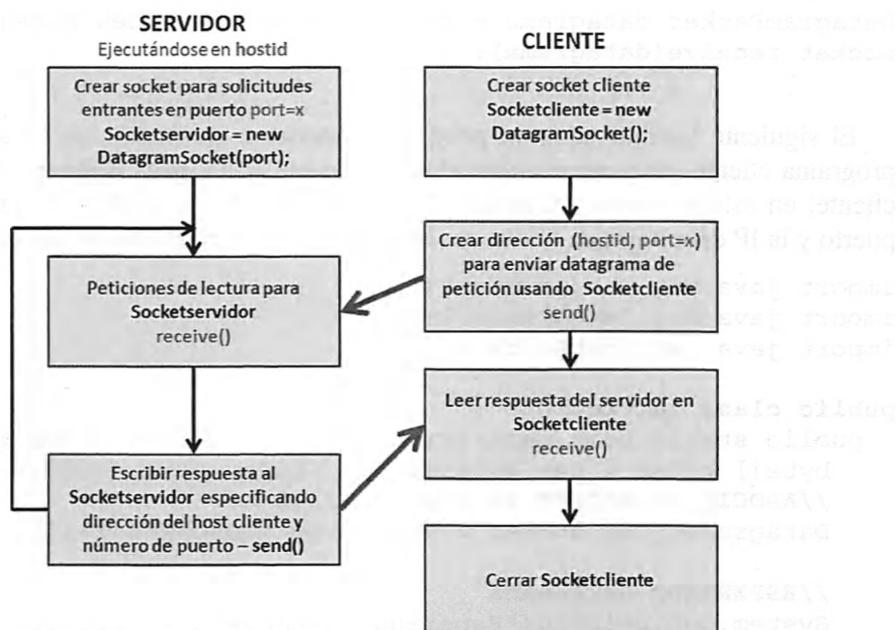


Figura 3.8. Envío y recepción de un datagrama.

## 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, vimos anteriormente algunos ejemplos de cómo se usa. Para escuchar peticiones en un puerto UDP concreto pasamos al constructor el número de puerto. El siguiente ejemplo crea un socket datagrama, le pasamos 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 *close()*: *socket.close()*.

## Envío y recepción de datagramas

Para enviar y recibir datagramas usamos la clase **DatagramPacket**.

Para enviar usamos el método *send()* de **DatagramSocket** pasando como parámetro el **DatagramPacket** que acabamos de crear:

```
DatagramPacket datagrama = new DatagramPacket(
    mensajeEnBytes,           // el array de bytes
    mensajeEnBytes.length,    // su longitud
    InetAddress.getByName("localhost"), // máquina destino
    PuertoDelServidor);      // puerto del destinatario
socket.send(datagrama);
```

Para recibir usamos el método *receive()* de **DatagramSocket** pasando como parámetro el **DatagramPacket** que acabamos de crear. Este método se bloquea hasta que se recibe un datagrama, a menos que se establezca un tiempo límite (timeout) sobre el socket.

```
DatagramPacket datagrama = new DatagramPacket(new byte[1024], 1024);
socket.receive(datagrama);
```

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.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class servidorUDP {
    public static void main(String[] argv) throws Exception {
        byte[] bufer = new byte[1024]; //bufer para recibir el datagrama
        //ASOCIO EL SOCKET AL PUERTO 12345
        DatagramSocket socket = new DatagramSocket(12345);

        //ESPERANDO DATAGRAMA
        System.out.println("Esperando Datagrama ..... ");
        DatagramPacket recibo = new DatagramPacket(bufer, bufer.length);
        socket.receive(recibo); //recibo datagrama
        int bytesRec = recibo.getLength(); //obtengo numero de bytes
```

```

String paquete= new String(recibo.getData());//obtengo String

//VISUALIZO INFORMACIÓN
System.out.println("Número de Bytes recibidos: "+ bytesRec);
System.out.println("Contenido del Paquete : "+ paquete.trim());
System.out.println("Puerto origen del mensaje: "+ recibo.getPort());
System.out.println("IP de origen : "+ 
                    recibo.getAddress().getHostAddress());
System.out.println("Puerto destino del mensaje: " +
                    socket.getLocalPort());
socket.close(); //cierro el socket
}
}

```

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;

public class clienteUDP {
    public static void main(String[] argv) throws Exception {
        InetAddress destino = InetAddress.getLocalHost();
        int port = 12345; //puerto al que envio el datagrama
        byte[] mensaje = new byte[1024];

        String Saludo="Enviando Saludos !!";
        mensaje = Saludo.getBytes(); //codifico String a bytes

        //CONSTRUYO EL DATAGRAMA A ENVIAR
        DatagramPacket envio = new DatagramPacket
            (mensaje, mensaje.length, destino, port);
        DatagramSocket socket = new DatagramSocket(34567); //Puerto local
        System.out.println("Enviando 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());

        //ENVIO DATAGRAMA
        socket.send(envio);
        socket.close(); //cierro el socket
    }
}

```

La ejecución de los programas cliente y servidor se muestra en la Figura 3.9.

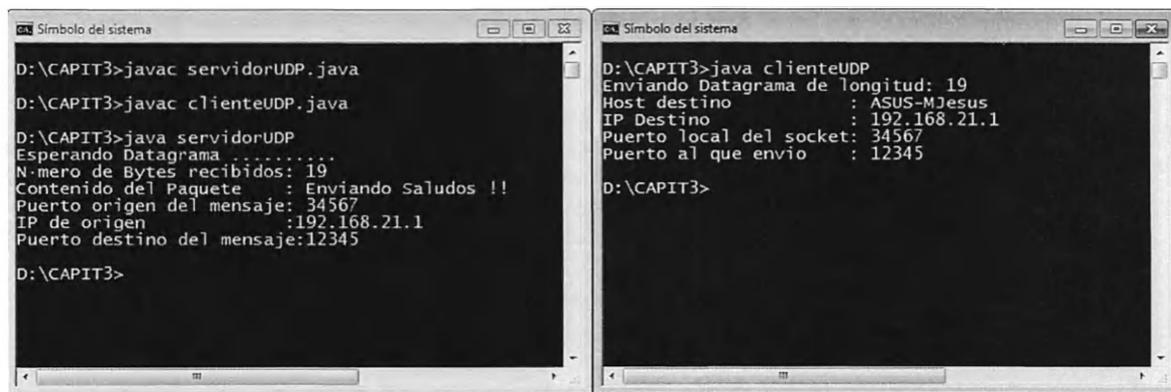


Figura 3.9. El servidor recibe un datagrama del cliente.

En primer lugar, desde una consola ejecutamos el programa servidor, y una vez iniciado abrimos otra consola y ejecutamos el programa cliente.

En el siguiente ejemplo el programa cliente envía un texto tecleado en su entrada estándar al servidor (en un puerto pactado), el servidor lee el datagrama y devuelve al cliente el texto en mayúscula. El programa cliente recibe un datagrama del servidor y muestra información del mismo en pantalla (IP, puerto del servidor y el texto en mayúscula). El programa servidor finaliza cuando recibe como cadena un asterisco. Para comenzar la ejecución primero ejecutamos el programa servidor (desde la consola) que permanecerá a la espera, y después (desde otra consola) ejecutamos el programa cliente varias veces. La Figura 3.10 muestra la ejecución.

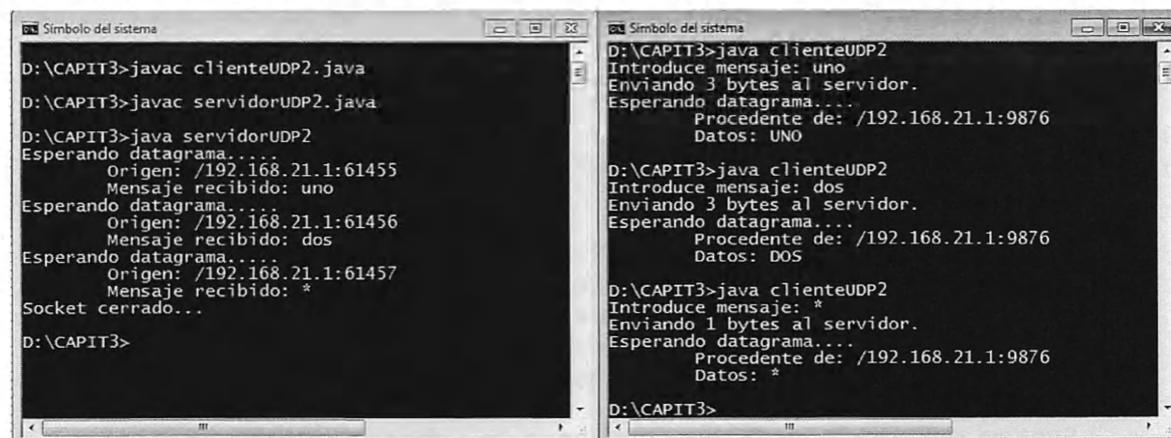


Figura 3.10. Intercambio de datagramas entre cliente y servidor.

El **programa cliente** es el siguiente:

```
import java.io.*;  

import java.net.*;  

public class clienteUDP2 {  

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

        // FLUJO PARA ENTRADA ESTANDAR  

        BufferedReader in = new BufferedReader  

            (new InputStreamReader(System.in));  

        DatagramSocket clientSocket = new DatagramSocket(); //socket cliente
```

```

byte[] enviados = new byte[1024];
byte[] recibidos = new byte[1024];

// DATOS DEL SERVIDOR al que enviar mensaje
InetAddress IPServidor = InetAddress.getLocalHost(); // localhost
int puerto = 9876; // puerto por el que escucha

//INTRODUCIR DATOS POR TECLADO
System.out.print("Introduce mensaje: ");
String cadena = in.readLine();
enviados = cadena.getBytes();

// ENVIANDO DATAGRAMA AL SERVIDOR
System.out.println("Enviando " + enviados.length + " bytes al
servidor.");
DatagramPacket envio = new DatagramPacket
    (enviados, enviados.length, IPServidor, puerto);
clientSocket.send(envio);

// RECIBIENDO DATAGRAMA DEL SERVIDOR
DatagramPacket recibo = new DatagramPacket
    (recibidos, recibidos.length);
System.out.println("Esperando datagrama....");
clientSocket.receive(recibo);
String mayuscula = new String(recibo.getData());

//OBTENIDENDO INFORMACIÓN DEL DATAGRAMA
InetAddress IPOrigen = recibo.getAddress();
int puertoOrigen = recibo.getPort();
System.out.println("\tProcedente de: " +
    IPOrigen + ":" + puertoOrigen);
System.out.println("\tDatos: " + mayuscula.trim());

clientSocket.close(); //cerrar socket
}
}

El programa servidor es el siguiente:

```

```

import java.io.*;
import java.net.*;

public class servidorUDP2 {
    public static void main(String args[]) throws Exception {
        //Puerto por el que escucha el servidor: 9876
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] recibidos = new byte[1024];
        byte[] enviados = new byte[1024];
        String cadena;

        while(true) {
            System.out.println ("Esperando datagrama....");
            //RECIBO DATAGRAMA
            recibidos = new byte[1024];
            DatagramPacket paqRecibido = new DatagramPacket
                (recibidos, recibidos.length);

```

```

serverSocket.receive(paqRecibido);
cadena = new String(paqRecibido.getData());

//DIRECCION ORIGEN
InetAddress IPOrigen = paqRecibido.getAddress();
int puerto = paqRecibido.getPort();
System.out.println ("\tOrigen: " + IPOrigen + ":" + puerto);
System.out.println ("\tMensaje recibido: " + cadena.trim());

//CONVERTIR CADENA A MAYÚSCULA
String mayuscula = cadena.trim().toUpperCase();
enviados = mayuscula.getBytes();

//ENVIO DATAGRAMA AL CLIENTE
DatagramPacket paqEnviado = new DatagramPacket
    (enviados, enviados.length, IPOrigen, puerto);
serverSocket.send(paqEnviado);

//Para terminar
if(cadena.trim().equals("*")) break;
}
serverSocket.close();
System.out.println ("Socket cerrado...");
}
}

```

#### ACTIVIDAD 3.4

Partiendo del programa cliente *clienteUPD2* crea otro programa cliente (llámalo *clienteUDP3.java*) en el que la entrada por teclado se repita hasta que se introduzca un asterisco. Establece un tiempo de espera de 5000 milisegundos con el método *setSoTimeout(5000)* para hacer que el método *receive()* se bloquee. Pasado ese tiempo controla si no se reciben datos lanzando la excepción *InterruptedException*, en cuyo caso visualiza un mensaje indicando que el paquete se ha perdido. Para probarlo ejecuta primero el programa *servidorUDP2*, luego *clienteUDP3* y después ejecuta *clienteUDP2* varias veces, la última vez introduce el asterisco; observa lo que pasa cada vez que mandas un paquete al servidor desde *clienteUDP3*.

### 3.6.2. 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*):

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

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

MÉTODO	MISIÓN
<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:

```
//Se crea el socket multicast. No hace falta especificar puerto:
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");
String msg = "Bienvenidos!!";
//Se crea el datagrama:
DatagramPacket paquete = new DatagramPacket
    (msg.getBytes(), msg.length(), grupo, Puerto);
//Se envía el paquete al grupo:
ms.send (paquete);
//Se cierra el socket:
ms.close ();
```

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:

```
//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");
//Se une al grupo
ms.joinGroup (grupo);
//Recibe el paquete del servidor multicast:
byte[] buf = new byte[1000];
DatagramPacket recibido = new DatagramPacket(buf, buf.length);
ms.receive(recibido);
//Salimos del grupo multicast
```

```
ms.leaveGroup (grupo);
//Se cierra el socket:
ms.close ();
```

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 un asterisco:

```
import java.io.*;
import java.net.*;
public class servidorMC1 {
    public static void main(String args[]) throws Exception {
        // FLUJO PARA ENTRADA ESTANDAR
        BufferedReader in = new
            BufferedReader(new InputStreamReader(System.in));
        //Se crea el socket multicast.
        MulticastSocket ms = new MulticastSocket();
        int Puerto = 12345;//Puerto multicast
        InetAddress grupo = InetAddress.getByName("225.0.0.1");//Grupo
        String cadena="";
        //
        while(!cadena.trim().equals("*")) {
            System.out.print("Datos a enviar al grupo: ");
            cadena = in.readLine();
            // ENVIANDO AL GRUPO
            DatagramPacket paquete = new DatagramPacket
                (cadena.getBytes(), cadena.length(), grupo, Puerto);
            ms.send (paquete);
        }
        ms.close ()//cierro socket
        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.*;
import java.net.*;
public class clienteMC1 {
    public static void main(String args[]) throws Exception {
        //Se crea el socket multicast
        int Puerto = 12345;//Puerto multicast
        MulticastSocket ms = new MulticastSocket(Puerto);
        InetAddress grupo = InetAddress.getByName("225.0.0.1");//Grupo
        //Nos unimos al grupo
        ms.joinGroup (grupo);
        String msg="";
        byte[] buf = new byte[1000];
        //
        while(!msg.trim().equals("*")) {
            //Recibe el paquete del servidor multicast
            DatagramPacket paquete = new DatagramPacket(buf, buf.length);
            ms.receive(paquete);
            msg = new String(paquete.getData());
            System.out.println ("Recibo: " + msg.trim());
        }
    }
}
```

```

        }
        ms.leaveGroup (grupo); //abandonamos grupo
        ms.close (); //cierra socket
        System.out.println ("Socket cerrado..."); 
    }
}

```

Para probarlo ejecutamos el programa servidor en una consola y a continuación ejecutamos diferentes instancias del programa cliente, véase Figura 3.11.

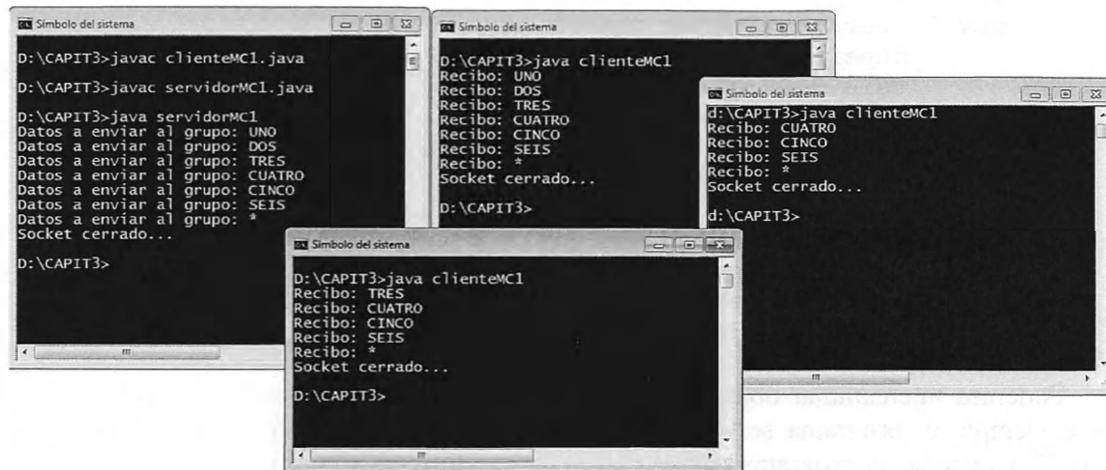


Figura 3.11. Servidor multicast enviando mensajes a clientes.

## 3.7 ENVÍO DE OBJETOS A TRAVÉS DE SOCKETS

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

En este apartado veremos cómo se pueden intercambiar objetos entre programas emisor y receptor o entre programas cliente y servidor usando sockets.

### 3.7.1. Objetos en Sockets TCP

Las clases **ObjectInputStream** y **ObjectOutputStream** nos permiten enviar objetos a través de sockets TCP. Utilizaremos los métodos *readObject()* para leer el objeto del stream y *writeObject()* para escribir el objeto al stream. Usaremos el constructor que admite un **InputStream** y un **OutputStream**. Para preparar el flujo de salida para escribir objetos escribimos:

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

Para preparar el flujo de entrada para leer objetos escribimos:

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

Las clases a las que pertenecen estos objetos deben implementar la interfaz **Serializable**. Por ejemplo, sea la clase *Persona* con 2 atributos, nombre y edad, 2 constructores y los métodos get y set correspondientes:

```
import java.io.Serializable;
@SuppressWarnings("serial")
public class Persona implements Serializable {
    String nombre;
    int edad;
    public Persona(String nombre, int edad) {
        super();
        this.nombre = nombre;
        this.edad = edad;
    }
    public Persona() {super();}

    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;}
}
```

Podemos intercambiar objetos *Persona* entre un cliente y un servidor usando sockets TCP. Por ejemplo el programa servidor crea un objeto *Persona*, dándole valores y se lo envía al programa cliente, el programa cliente realiza los cambios oportunos en el objeto y se lo devuelve modificado al servidor. El **programa servidor** es el siguiente:

```
import java.io.*;
import java.net.*;
public class Servidor1Objeto {
    public static void main(String[] arg) throws IOException,
        ClassNotFoundException {
        int numeroPuerto = 6000;// Puerto
        ServerSocket servidor = new ServerSocket(numeroPuerto);
        System.out.println("Esperando al cliente.....");
        Socket cliente = servidor.accept();

        // Se prepara un flujo de salida para objetos
        ObjectOutputStream outObjeto = new ObjectOutputStream(
            cliente.getOutputStream());
        // Se prepara un objeto y se envía
        Persona per = new Persona("Juan", 20);
        outObjeto.writeObject(per); //enviando objeto
        System.out.println("Envio: " + per.getNombre() + "*" + per.getEdad());

        // Se obtiene un stream para leer objetos
        ObjectInputStream inObjeto = new ObjectInputStream(
            cliente.getInputStream());
        Persona dato = (Persona) inObjeto.readObject();
        System.out.println("Recibo: "+dato.getNombre()+"*"+dato.getEdad());

        // CERRAR STREAMS Y SOCKETS
        outObjeto.close();
        inObjeto.close();
        cliente.close();
    }
}
```

```

        servidor.close();
    }
}//...

```

El **programa cliente** es el siguiente:

```

import java.io.*;
import java.net.*;
public class Cliente1Objeto {
    public static void main(String[] arg) throws IOException,
        ClassNotFoundException {
        String Host = "localhost";
        int Puerto = 6000;//puerto remoto
        System.out.println("PROGRAMA CLIENTE INICIADO....");
        Socket cliente = new Socket(Host, Puerto);

        //Flujo de entrada para objetos
        ObjectInputStream perEnt = new ObjectInputStream(
            cliente.getInputStream());
        //Se recibe un objeto
        Persona dato = (Persona) perEnt.readObject();//recibo objeto
        System.out.println("Recibo: "+dato.getNombre()+"*"+dato.getEdad());

        //Modifico el objeto
        dato.setNombre("Juan Ramos");
        dato.setEdad(22);

        //FLUJO DE SALIDA PARA OBJETOS
        ObjectOutputStream perSal = new ObjectOutputStream(
            cliente.getOutputStream());
        // Se envía el objeto
        perSal.writeObject(dato);
        System.out.println("Envio: "+dato.getNombre()+"*"+dato.getEdad());

        // CERRAR STREAMS Y SOCKETS
        perEnt.close();
        perSal.close();
        cliente.close();
    }
}//...

```

La compilación y ejecución se muestra en la Figura 3.12.

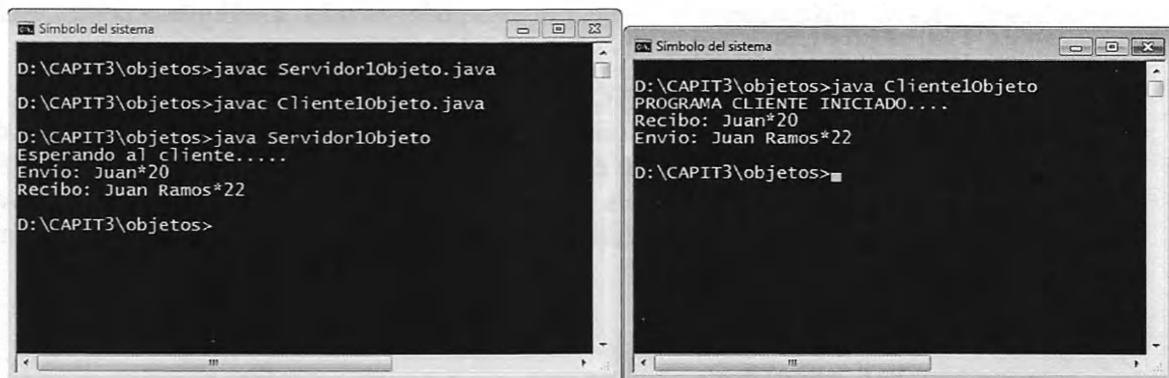


Figura 3.12. Servidor y cliente TCP intercambiando objetos.

### 3.7.2. Objetos en Sockets UDP

Para intercambiar objetos en sockets UDP utilizaremos las clases **ByteArrayOutputStream** y **ByteArrayInputStream**. Se necesita convertir el objeto a un array de bytes. Por ejemplo, para convertir un objeto *Persona* a un array de bytes escribimos las siguientes líneas:

```
Persona persona = new Persona("Maria", 22);
//CONVERTIMOS OBJETO A BYTES
ByteArrayOutputStream bs= new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream (bs);
out.writeObject(persona); //escribir objeto Persona en el stream
out.close(); //cerrar stream
byte[] bytes = bs.toByteArray(); // objeto en bytes
```

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

```
// RECIBO DATAGRAMA
byte[] recibidos = new byte[1024];
DatagramPacket paqRecibido = new
    DatagramPacket(recibidos, recibidos.length);
socket.receive(paqRecibido); //recibo el datagrama
// CONVERTIMOS BYTES A OBJETO
ByteArrayInputStream bais = new ByteArrayInputStream(recibidos);
ObjectInputStream in = new ObjectInputStream(bais);
Persona persona = (Persona) in.readObject(); //obtengo objeto
in.close();
```

---

#### ACTIVIDAD 3.5

Realiza un programa servidor que espere un datagrama de un cliente. El cliente le envía un objeto *Persona* que previamente había inicializado. El servidor modifica los datos del objeto *Persona* y se lo envía de vuelta al cliente. Visualiza los datos del objeto *Persona* tanto en el programa cliente cuando los envía y los recibe como en el programa servidor cuando los recibe y los envía modificados.

---

### 3.8. CONEXIÓN DE MÚLTIPLES CLIENTES. 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 método *accept()* devuelve un objeto **Socket**, éste se usará para crear un 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 infinito:

```
public class Servidor {
    public static void main(String args[]) throws IOException {
        ServerSocket servidor;
        servidor = new ServerSocket(6000);
```

```

        System.out.println("Servidor iniciado...");  

        while (true) {  

            Socket cliente = new Socket();  

            cliente=servidor.accept(); //esperando cliente  

            HiloServidor hilo = new HiloServidor(cliente);  

            hilo.start(); //se atiende al cliente  

        }  

    }  

}//..

```

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 (por claridad se han eliminado los bloques **try-catch**). El proceso de tratamiento de la cadena se realiza en un hilo, en este caso se llama *HiloServidor*:

```

import java.io.*;  

import java.net.*;  

public class HiloServidor extends Thread {  

    BufferedReader fentrada;  

    PrintWriter fsalida;  

    Socket socket = null;  

    public HiloServidor(Socket s) { //CONSTRUCTOR  

        socket =s;  

        //se crean flujos de entrada y salida  

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

        fentrada = new BufferedReader(new InputStreamReader(  

                                         socket.getInputStream()));  

    }  

    public void run() { //tarea a realizar con el cliente  

        String cadena="";  

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

            System.out.println("COMUNICO CON: "+ socket.toString());  

            cadena = fentrada.readLine(); //obtener cadena  

            fsalida.println(cadena.trim().toUpperCase()); //enviar mayúscula  

        } // fin while  

        System.out.println("FIN CON: "+ socket.toString());  

        fsalida.close();  

        fentrada.close();  

        socket.close();  

    }  

}//..

```

Como programa cliente podemos ejecutar el programa *ejemplo2Cliente.java* que se creó en epígrafes anteriores. El programa se conectará con el servidor en el puerto 6000 y le enviará cadenas introducidas por teclado; cuando le envíe un asterisco el servidor finalizará la comunicación con el cliente. La Figura 3.13 muestra un momento de la ejecución, primero se ejecuta el programa servidor y a continuación el programa cliente; se puede observar cómo los 3 clientes conectados están siendo atendidos por el servidor de forma simultánea.

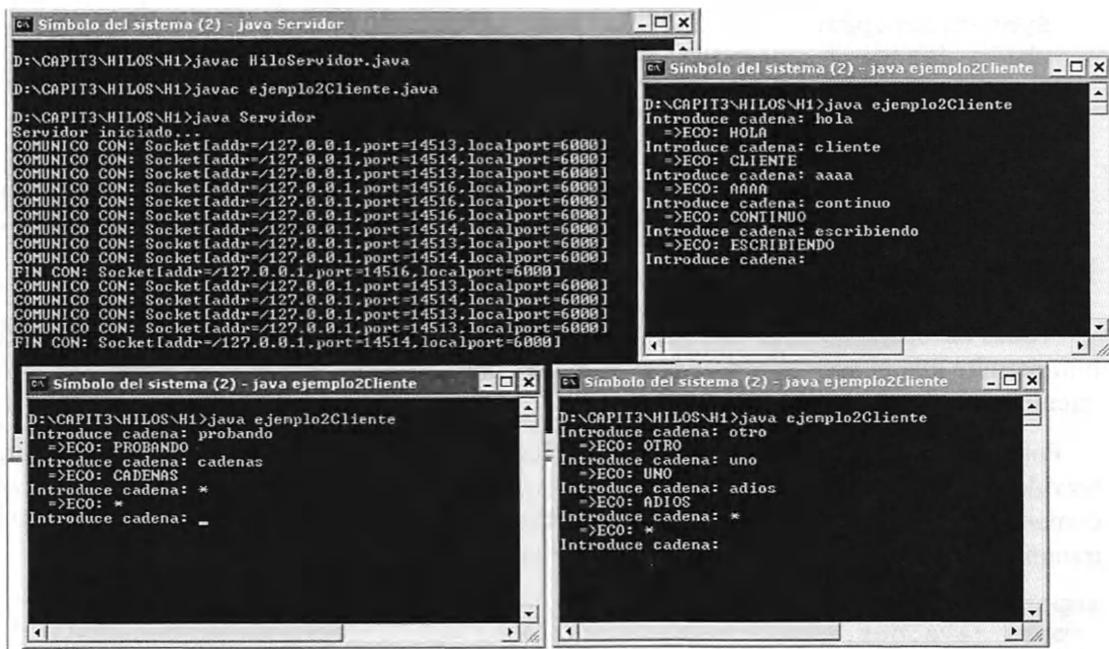


Figura 3.13. Servidor atendiendo a múltiples clientes.

### 3.8.1. Creación de un chat con TCP

Una situación típica de un servidor que atiende a múltiples clientes es un servidor de chat. Vamos a construir uno sencillo que pueda atender a varios clientes a la vez, cada cliente será atendido en un hilo de ejecución; en ese hilo se recibirán sus mensajes y se enviarán al resto. La idea básica en el servidor es la siguiente:

- Al iniciar el servidor se muestra una pantalla donde se visualiza el número de clientes que actualmente están conectados al chat y la conversación mantenida entre ellos. La conversación de chat se va visualizando en un textarea. El botón *Salir* finaliza el servidor de chat, Figura 3.14.
- El servidor se mantiene a la escucha (en un puerto pactado) de cualquier petición de un cliente para conectarse.
- El servidor acepta al cliente, guarda en un array de sockets el que se acaba de crear. Este array se usará en el hilo de ejecución para enviar la conversación del chat a todos los clientes conectados.
- Cuando se conecta un cliente se incrementa en un contador el número de conexiones actuales (ACTUALES), si se desconecta se decrementa. Otro contador (CONEXIONES) se usará para contar las conexiones de clientes, el máximo de conexiones viene dado en la variable MAXIMO.
- Lanza un hilo de comunicación con el cliente (programa **HiloServidor**). Por el hilo se reciben y envían los mensajes de los clientes. Si el cliente cierra la comunicación, el hilo se rompe y se corta la comunicación con ese cliente.

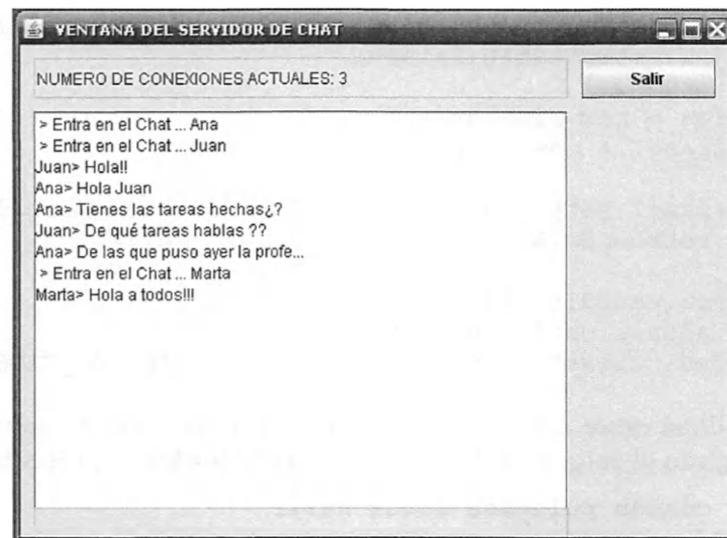


Figura 3.14. Servidor de chat.

- Se admite hasta un máximo de conexiones, en el ejemplo 10.

El **programa servidor**, **ServidorChat**, es el siguiente. En primer lugar se definen las variables y campos de la pantalla:

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ServidorChat extends JFrame implements ActionListener {
    private static final long serialVersionUID = 1L;
    static ServerSocket servidor;
    static final int PUERTO = 44444; // puerto por el que escucha
    static int CONEXIONES = 0; // cuenta las conexiones
    static int ACTUALES = 0; // nº de conexiones actuales activas
    static int MAXIMO=10; //máximo de conexiones permitidas

    static JTextField mensaje=new JTextField("");
    static JTextField mensaje2=new JTextField("");
    private JScrollPane scrollpanel1;
    static JTextArea textarea;
    JButton salir = new JButton("Salir");
    static Socket tabla[] = new Socket[10]; //almacena sockets de clientes
```

Desde el constructor se prepara la pantalla:

```
// Constructor
public ServidorChat() {
    super(" VENTANA DEL SERVIDOR DE CHAT ");
    setLayout(null);
    mensaje.setBounds(10, 10, 400, 30);
    mensaje.setEditable(false);
    add(mensaje);
```

```

mensaje2.setBounds(10, 348, 400, 30);add(mensaje2);
mensaje2.setEditable(false);

textarea = new JTextArea();
scrollpanel = new JScrollPane(textarea);

scrollpanel.setBounds(10, 50, 400, 300);add(scrollpanel);
salir.setBounds(420, 10, 100, 30);add(salir);

textarea.setEditable(false);
salir.addActionListener(this);
setDefaultCloseOperation(JFrame.DO NOTHING_ON_CLOSE);
}

```

Se ha anulado el cierre de la ventana para que la finalización del servidor se haga desde el botón *Salir*. Cuando se pulsa el botón se cierra el **ServerSocket** y finaliza la ejecución:

```

// Acción cuando pulsamos botón Salir
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == salir) { // SE PULSA SALIR
        try {
            servidor.close(); // cierro
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        System.exit(0); //fin
    }
}

```

Desde *main()* se inicia el servidor y las variables y se prepara la pantalla:

```

public static void main(String args[]) throws IOException {
    servidor = new ServerSocket(PUERTO);
    System.out.println("Servidor iniciado...");
    ServidorChat pantalla = new ServidorChat();
    pantalla.setBounds(0, 0, 540, 400);
    pantalla.setVisible(true);
    mensaje.setText("NUMERO DE CONEXIONES ACTUALES: " + 0);
}

```

Se hace un bucle para controlar el número de conexiones. Dentro del bucle el servidor espera la conexión del cliente y cuando se conecta se crea un socket:

```

// SE ADMITEN HASTA 10 CONEXIONES
while (CONEXIONES < MAXIMO) {
    Socket s = new Socket();
    try {
        s = servidor.accept(); // esperando cliente
    } catch (SocketException ns) {
        //sale por aqui si pulsamos botón Salir y
        //no se ejecuta todo el bucle
        break; //salir del bucle
    }
}

```

El socket creado satisfactoriamente se almacena en la tabla, se cuenta el número de conexiones, se incrementan las conexiones actuales y se lanza el hilo para gestionar los mensajes del cliente que se acaba de conectar:

```
tabla[CONEXIONES] = s; //almacenar socket
```

```

CONEXIONES++;
ACTUALES++;
HiloServidor hilo = new HiloServidor(s);
hilo.start(); //lanzar hilo
}//fin while

```

Se sale del bucle anterior si ha habido 10 conexiones o si se pulsa el botón *Salir*. Al pulsar el botón salir se cierra el **ServerSocket**, esto causa que la sentencia `s = servidor.accept()` lance la excepción *SocketException* (ya que el servidor está cerrado) desde donde se hace *break* para salir del bucle.

Al salir del bucle se comprueba si el servidor está cerrado, si no lo está es que se han establecido las 10 conexiones, se visualiza un mensaje y se cierra el servidor:

```

//Cuando finaliza bucle cerrar servidor si no se ha cerrado antes
if (!servidor.isClosed())
    try {
        // sale cuando se llega al máximo de conexiones
        mensaje2.setForeground(Color.red);
        mensaje2.setText("MÁXIMO N° DE CONEXIONES
ESTABLECIDAS: " + CONEXIONES);
        servidor.close();
    } catch (IOException e1) {
        e1.printStackTrace();
    }
    System.out.println("Servidor finalizado...");
}//main
}// .. Fin servidorChat

```

El hilo **HiloServidor** se encarga de recibir y enviar los mensajes a los clientes de chat. En el constructor, se recibe el socket creado y se crea el flujo de entrada desde el que se leen los mensajes que el cliente de chat envía:

```

import java.io.*;
import java.net.*;
public class HiloServidor extends Thread {
    DataInputStream fentrada;
    Socket socket = null;
    public HiloServidor(Socket s) {
        socket = s;
        try {
            // CREO FLUJO DE ENTRADA
            fentrada = new DataInputStream(socket.getInputStream());
        } catch (IOException e) {
            System.out.println("ERROR DE E/S");
            e.printStackTrace();
        }
    }
}

```

En el método *run()*, lo primero que hacemos es enviar los mensajes que hay actualmente en el chat al programa cliente para que los visualice en la pantalla. Esto se hace en el método

*EnviarMensajes()*. Los mensajes que se envían son los que están en el textarea del servidor de chat:

```
public void run() {
    ServidorChat.mensaje.setText("NUMERO DE CONEXIONES ACTUALES: "
        + ServidorChat.ACTUALES);
    // NADA MAS CONECTARSE EL CLIENTE LE ENVIO TODOS LOS MENSAJES
    String texto = ServidorChat.textarea.getText();
    EnviarMensajes(texto);
```

A continuación se hace un bucle while en el que se recibe lo que el cliente escribe en el chat. Cuando un cliente finaliza (pulsa el botón *Salir* de su pantalla) envía un asterisco al servidor de chat, entonces se sale del bucle while, ya que termina el proceso del cliente, de esta manera se controlan las conexiones actuales:

```
while (true) {
    String cadena = "";
    try {
        cadena = fentrada.readUTF(); //lee lo que el cliente escribe
        //cuando un cliente finaliza envia un *
        if (cadena.trim().equals("*")) {
            ServidorChat.ACTUALES--;
            ServidorChat.mensaje.setText("NUMERO DE CONEXIONES
                ACTUALES: " + ServidorChat.ACTUALES);
            break; //salir del while
        }
    }
```

El texto que el cliente escribe en su chat, se añade al textarea del servidor y el servidor enviará a todos los clientes el texto que hay en su textarea llamando de nuevo a *EnviarMensajes()*, así todos ven la conversación:

```
ServidorChat.textarea.append(cadena + "\n");
texto = ServidorChat.textarea.getText();
EnviarMensajes(texto); //envio texto a todos los clientes
} catch (Exception e) {
    e.printStackTrace();
    break;
}
}// fin while
}//run
```

El método *EnviarMensajes()* envía el texto del textarea a todos los sockets que están en la tabla de sockets, de esta manera todos ven la conversación. Será necesario abrir un stream de escritura a cada socket y escribir el texto:

```
// ENVIA LOS MENSAJES DEL TEXTAREA A LOS CLIENTES DEL CHAT
private void EnviarMensajes(String texto) {
    int i;
    //recorremos tabla de sockets para enviarles los mensajes
    for (i = 0; i < ServidorChat.CONEXIONES; i++) {
        Socket s1 = ServidorChat.tabla[i]; //obtener socket
        try {
            DataOutputStream fsalida = new DataOutputStream(
                s1.getOutputStream());
            fsalida.writeUTF(texto); //escribir en el socket el texto
        }
```

```

        } catch (SocketException se) {
            // esta excepción ocurre cuando escribimos en un socket
            // de un cliente que ha finalizado
        } catch (IOException e) {
            e.printStackTrace();
        }
    } // for
}// EnviarMensajes
}// .. Fin HiloServidor

```

Desde el **programa cliente** se realizan las siguientes funciones:

- En primer lugar se pide el nombre que el usuario utilizará en el chat, Figura 3.15.

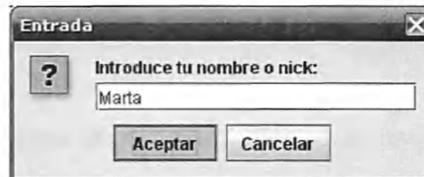


Figura 3.15. Identificación del cliente de chat.

- Se crea un socket al servidor de chat en el puerto pactado. Si todo va bien, el servidor asignará un hilo al cliente y se mostrará en la pantalla de chat del cliente la conversación que hay hasta el momento, Figura 3.16. Si no se puede establecer la conexión, se visualiza un mensaje de error.

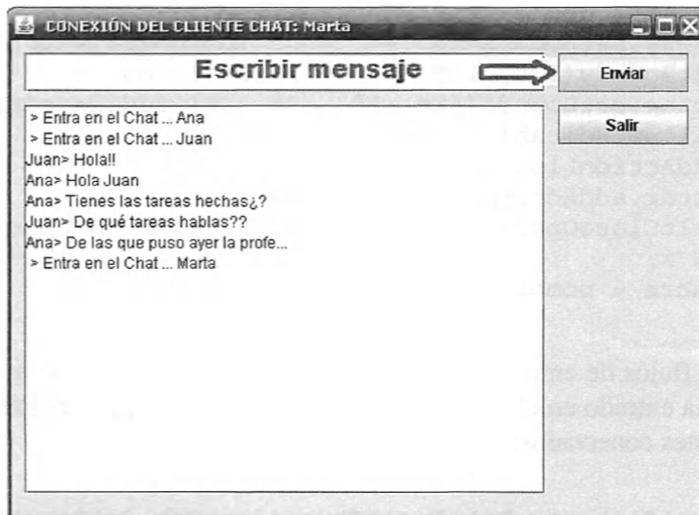


Figura 3.16. Entrada del cliente al chat.

- El cliente puede escribir sus mensajes y pulsar el botón *Enviar*, automáticamente su mensaje será enviado a todos los clientes de chat.
- El botón *Salir* finaliza la conexión del cliente de chat.

El código de la clase **ClienteChat** es el siguiente. En primer lugar se definen variables, campos de la pantalla y los streams de entrada y de salida:

```

import java.awt.event.*;
import java.io.*;

```

```

import java.net.*;
import javax.swing.*;

public class ClienteChat extends JFrame implements ActionListener {
    private static final long serialVersionUID = 1L;
    Socket socket = null;
    // streams
    DataInputStream fentrada; //para leer mensajes de todos
    DataOutputStream fsalida; //para escribir sus mensajes
    String nombre;
    static JTextField mensaje = new JTextField();
    private JScrollPane scrollpanel;
    static JTextArea textareal;
    JButton boton = new JButton("Enviar");
    JButton desconectar = new JButton("Salir");
    boolean repetir = true;

```

En el constructor se prepara la pantalla. Se recibe el socket creado y el nombre del cliente de chat:

```

// constructor
public ClienteChat(Socket s, String nombre) {
    super(" CONEXIÓN DEL CLIENTE CHAT: " + nombre);
    setLayout(null);
    mensaje.setBounds(10, 10, 400, 30);add(mensaje);
    textareal = new JTextArea();
    scrollpanel = new JScrollPane(textareal);
    scrollpanel.setBounds(10, 50, 400, 300); add(scrollpanel);
    boton.setBounds(420, 10, 100, 30);add(boton);
    desconectar.setBounds(420, 50, 100, 30);add(desconectar);
    textareal.setEditable(false);
    boton.addActionListener(this);
    desconectar.addActionListener(this);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    socket = s;
    this.nombre = nombre;

```

Se crean los flujos de entrada y salida. Se escribe en el flujo de salida un mensaje indicando que el usuario ha entrado en el chat. Este mensaje lo recibe el hilo (**HiloServidor**) y se lo manda a todos los clientes conectados:

```

try {
    fentrada = new DataInputStream(socket.getInputStream());
    fsalida = new DataOutputStream(socket.getOutputStream());
    String texto = "> Entra en el Chat ... " + nombre;
    fsalida.writeUTF(texto); //escribe mensaje de entrada
} catch (IOException e) {
    System.out.println("ERROR DE E/S");
    e.printStackTrace();
    System.exit(0);
}
}// fin constructor

```

Cuando se pulsa el botón *Enviar* se envía al flujo de salida el mensaje que el cliente ha escrito:

```
// acción cuando pulsamos botones
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == boton) { // SE PULSA botón ENVIAR
        String texto = nombre + "> " + mensaje.getText();
        try {
            mensaje.setText("");
            fsalida.writeUTF(texto);
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Si se pulsa el botón *Salir* se envía primero un mensaje indicando que el usuario abandona el chat y a continuación un asterisco indicando que el usuario va a salir del chat:

```
if (e.getSource() == desconectar) { // SE PULSA botón SALIR
    String texto = "> Abandona el Chat ... " + nombre;
    try {
        fsalida.writeUTF(texto);
        fsalida.writeUTF("*");
        repetir = false; //para salir del bucle
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
} //
```

Dentro del método *ejecutar()*, el cliente lee lo que el hilo le manda (los mensajes de chat) para mostrarlo en el textarea. Esto se realiza en un proceso repetitivo que termina cuando el usuario pulsa el botón *Salir*, que cambiará el valor de la variable *repetir* a *false* para que finalice el bucle:

```
public void ejecutar() {
    String texto = "";
    while (repetir) {
        try {
            texto = fentrada.readUTF(); //leer mensajes
            textareal.setText(texto); //visualizarlos
        } catch (IOException e) {
            // este error sale cuando el servidor se cierra
            JOptionPane.showMessageDialog(
                null, "IMPOSIBLE CONECTAR CON EL SERVIDOR\n" +
                e.getMessage(), "<<MENSAJE DE ERROR:2>>", 
                JOptionPane.ERROR_MESSAGE);
            repetir = false; //salir del bucle
        }
    } //while
    try {
        socket.close(); //cerrar socket
        System.exit(0);
    } catch (IOException e) {
        e.printStackTrace();
    }
} // ejecutar
```

En la función *main()* se pide el nombre de usuario, se realiza la conexión al servidor, se crea un objeto **ClienteChat**, se muestra la pantalla y se ejecuta el método *ejecutar()*:

```

public static void main(String args[]) {
    int puerto = 4444;
    String nombre = JOptionPane.showInputDialog
        ("Introduce tu nombre o nick:");
    Socket s = null;
    try {
        //cliente y servidor se ejecutan en la máquina local
        s = new Socket("localhost", puerto);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
            "IMPOSIBLE CONECTAR CON EL SERVIDOR\n" + e.getMessage(),
            "<<MENSAJE DE ERROR:1>>", JOptionPane.ERROR_MESSAGE);
        System.exit(0);
    }
    if (!nombre.trim().equals("")) {//hay que escribir algo
        ClienteChat cliente = new ClienteChat(s, nombre);
        cliente.setBounds(0, 0, 540, 400);
        cliente.setVisible(true);
        cliente.ejecutar();
    } else {
        System.out.println("El nombre está vacío....");
    }
}
//main
} // ..ClienteChat

```

Aunque no se ha utilizado un hilo para implementar esta clase, lo más típico es usarlo; se implementaría **Runnable** para definir la clase:

```

public class ClienteChat extends JFrame
    implements ActionListener, Runnable {

```

Se cambiaría el método *ejecutar* por *run()* y se lanzaría el hilo cliente de la siguiente manera:

```

ClienteChat cliente = new ClienteChat (s, nombre);
cliente.setBounds(0, 0, 540, 400);
cliente.setVisible(true);
new Thread(cliente).start();

```

Para ejecutar el servidor de chat se necesita que las clases java **ServidorChat** e **HiloServidor** estén en la misma carpeta. El programa cliente **ClienteChat** puede estar en cualquier otra carpeta. Primero se ejecuta el programa servidor:

```

D:\CAPIT3\HILOS\H2>java ServidorChat
Servidor iniciado...

```

Y Luego el cliente desde la carpeta donde esté:

```

D:\>java ClienteChat

```

En el código expuesto el programa cliente y el servidor se ejecutan en la misma máquina. Pero lo normal es que el servidor esté en una máquina y el cliente en otra. En este caso es

necesario especificar en el programa cliente, en la creación del socket, la dirección IP donde está el servidor de chat, por ejemplo si el servidor se ejecuta en la máquina con IP 192.168.0.194, creo así el socket en el programa cliente:

```
//servidor se ejecuta en la máquina con IP 192.168.0.194
s = new Socket("192.168.0.194", puerto);
```

### ACTIVIDAD 3.6

Prueba los programas cliente y servidor de chat desde diferentes máquinas. El programa *ServidorChat* e *HiloServidor* tienen que estar en la misma máquina y el programa *ClienteChat* puedes instalarlo en la que quieras que participe en el chat.

### 3.8.2. Creación de un chat con UDP

A continuación vamos a crear un chat más sencillo utilizando **MulticastSocket**. Crearemos una única clase, *MultiChatUDP* que extiende **Runnable**, en la que se define una pantalla similar a la del cliente chat TCP. Tenemos 2 botones, uno para enviar el mensaje tecleado, otro para finalizar y un textarea donde se muestran los mensajes (Figura 3.17):

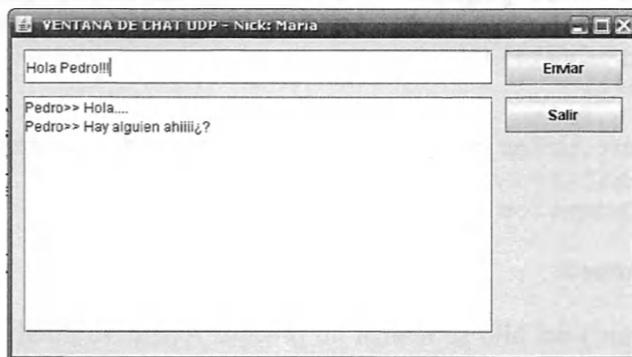


Figura 3.17 Chat UDP.

En el método *main()* se pide un nombre al usuario (nick), se crea un socket multicast en un puerto determinado, se configura la IP del grupo al que nos conectaremos, nos unimos al grupo para enviar y recibir mensajes, se comprueba si se ha escrito algo en el nombre, se muestra la pantalla y por último se lanza el hilo multichat:

```
public static void main(String args[]) throws IOException {
    String nombre = JOptionPane.showInputDialog
        ("Introduce tu nombre o nick:");
    // Se crea el socket multicast
    ms = new MulticastSocket(Puerto);
    grupo = InetAddress.getByName("225.0.0.1");// Grupo multicast
    // Nos unimos al grupo
    ms.joinGroup(grupo);
    if (!nombre.trim().equals("")) {
        MultichatUDP server = new MultichatUDP(nombre);
        server.setBounds(0, 0, 540, 400);
        server.setVisible(true);
        new Thread(server).start(); // lanzar hilo
    } else {
        System.out.println("El nombre está vacío....");
    }
}
```

```
// main
Cada vez que se pulse el botón Enviar se envían los mensajes al grupo de multicast:
```

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == boton) { // SE PULSA ENVIAR
        String texto = nombre + ">> " + mensaje.getText();
        try {
            // ENVIANDO mensaje al grupo
            DatagramPacket paquete = new DatagramPacket(texto.getBytes(),
                texto.length(), grupo, Puerto);
            ms.send(paquete);
        } catch (IOException e1) { e1.printStackTrace(); }
    } //fin enviar
```

```
El botón Salir envia el mensaje de despedida al grupo y cierra el socket:
```

```
if (e.getSource() == desconectar) { // SE PULSA SALIR
    String texto = "**** Abandona el chat: " + nombre + " ****";
    try {
        // ENVIANDO DESPEDIDA AL GRUPO
        DatagramPacket paquete = new DatagramPacket(texto.getBytes(),
            texto.length(), grupo, Puerto);
        ms.send(paquete);
        ms.close();
        repetir = false;
        System.out.println("Abandona el chat: " + nombre);
        System.exit(0);
    } catch (IOException e1) { e1.printStackTrace(); }
} //actionPerformed
```

En el método *run()* del hilo se realiza un proceso repetitivo donde se muestran los mensajes que se reciben del grupo multicast en el textarea:

```
public void run() {
    while (repetir) {
        try {
            DatagramPacket p = new DatagramPacket(buf, buf.length);
            ms.receive(p); //recibo mensajes
            String texto = new String(p.getData(), 0, p.getLength());
            textareal.append(texto + "\n");
        } catch (SocketException s) { System.out.println(s.getMessage()); }
        catch (IOException e) { e.printStackTrace(); }
    } // run
```

Por último se muestra la definición de las variables puerto, multicast y grupo, y la cabecera de la clase:

```
public class MultiChatUDP extends JFrame implements ActionListener,
    Runnable {
    static MulticastSocket ms = null;
    static byte[] buf = new byte[1000];
    static InetAddress grupo = null;
    static int Puerto = 12345; // Puerto multicast
```

El resto de variables de pantalla son similares al ejemplo con TCP. Para probarlo se ejecuta el programa *MultiChatUDP* en las máquinas que quieran participar en el chat.

### 3.8.3. Consulta de base de datos

Una aplicación cliente-servidor típica es la consulta a bases de datos donde el cliente solicita una información y el servidor se la envía. En el siguiente ejemplo partimos de una base de datos **Db4o** que almacena objetos *Empleados* y *Departamentos*. Desde el programa cliente se introduce por medio de su pantalla el número de departamento que se desea consultar y se pulsa el botón *Enviar* para enviar la solicitud al servidor. El servidor le devolverá los datos solicitados para que los muestre en pantalla, véase Figura 3.18.

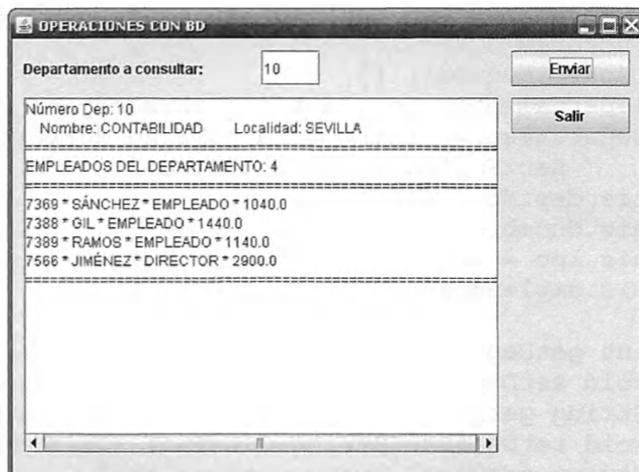


Figura 3.18. Cliente de BD.

El servidor, es muy similar al servidor de chat TCP creado anteriormente, espera las conexiones de los clientes y cuando le llega uno crea un hilo para satisfacer sus necesidades. El hilo devolverá al cliente la información solicitada. En este caso le devolverá un objeto *Departamentos*, ya que el cliente lo que solicita son los datos de un departamento.

El servidor en su pantalla muestra todas las conexiones que se van realizando, Figura 3.19. A cada cliente se le asignará un número de conexión. Cuando un cliente abandona la conexión también se mostrará un mensaje en la pantalla, además se muestra la IP, la hora y la fecha de la conexión. Esto es útil para llevar un registro de la actividad del servidor.



Figura 3.19. Servidor de BD.

Para manejar los datos de la base de datos disponemos de las clases **Departamentos** y **Empleados**. La clase **Departamentos** contiene los atributos *deptNo*, *dnombre*, *loc* y *empleadoses*, este último representa la colección de empleados del departamento. También se definen varios constructores y los métodos getter y setter correspondientes:

```
import java.util.HashSet;
import java.util.Set;
public class Departamentos implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private int deptNo;
    private String dnombre;
    private String loc;
    private Set<Empleados> empleadoses = new HashSet<Empleados>(0);
    //constructores
    public Departamentos() {}
    public Departamentos(int deptNo) {this.deptNo = deptNo;}
    public Departamentos(int deptNo, String dnombre, String loc,
        Set<Empleados> empleadoses) {
        this.deptNo = deptNo;
        this.dnombre = dnombre;
        this.loc = loc;
        this.empleadoses = empleadoses;
    }
    public int getDeptNo() {return this.deptNo;}
    public void setDeptNo(int i) {this.deptNo = i;}
    public String getDnombre() {return this.dnombre;}
    public void setDnombre(String dnombre) {this.dnombre = dnombre;}
    public String getLoc() {return this.loc;}
    public void setLoc(String loc) {this.loc = loc;}
    public Set<Empleados> getEmpleadoses() {return this.empleadoses;}
    public void setEmpleadoses(Set<Empleados> empleadoses) {
        this.empleadoses = empleadoses;
    }
}///..Departamentos
```

La clase **Empleados** define los atributos *empNo*, *apellido*, *oficio*, *dir*, *fechaAlt*, *salario*, *comision* y *departamentos*, este último es un objeto **Departamentos**. También se definen varios constructores y los métodos getter y setter correspondientes:

```
import java.util.Date;
public class Empleados implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private int empNo;
    private Departamentos departamentos;
    private String apellido;
    private String oficio;
    private int dir;
    private Date fechaAlt;
    private Float salario;
    private Float comision;
    //constructores
    public Empleados() {}
    public Empleados(int empNo, Departamentos departamentos) {
        this.empNo = empNo;
```

```

        this.departamentos = departamentos;
    }
    public Empleados(int empNo, Departamentos departamentos,
                     String apellido, String oficio, int dir, Date fechaAlt,
                     Float salario, Float comision) {
        this.empNo = empNo;
        this.departamentos = departamentos;
        this.apellido = apellido;
        this.oficio = oficio;
        this.dir = dir;
        this.fechaAlt = fechaAlt;
        this.salario = salario;
        this.comision = comision;
    }
    public int getEmpNo() { return this.empNo; }
    public void setEmpNo(int empNo) { this.empNo = empNo; }
    public Departamentos getDepartamentos()
        { return this.departamentos; }
    public void setDepartamentos(Departamentos departamentos)
        { this.departamentos = departamentos; }
    public String getApellido() { return this.apellido; }
    public void setApellido(String apellido)
        { this.apellido = apellido; }
    public String getOficio() { return this.oficio; }
    public void setOficio(String oficio) { this.oficio = oficio; }
    public int getDir() { return this.dir; }
    public void setDir(int dir) { this.dir = dir; }
    public Date getFechaAlt() { return this.fechaAlt; }
    public void setFechaAlt(Date fechaAlt)
        { this.fechaAlt = fechaAlt; }
    public Float getSalario() { return this.salario; }
    public void setSalario(Float salario) { this.salario = salario; }
    public Float getComision() { return this.comision; }
    public void setComision(Float comision)
        { this.comision = comision; }
}
//..

```

Utilizaremos la clase **Conexion** para obtener un objeto de base de datos (patrón *singleton*). El método *getDBConexion()* devuelve el objeto creado. La base de datos se llama **EMPLEDEP.YAP**.

```

import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
public class Conexion {
    final static String BDPer = "EMPLEDEP.YAP";
    static ObjectContainer db;
    static {
        db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);
    }
    public static ObjectContainer getDBConexion() {
        return db;
    }
} // Fin Conexion

```

El **programa servidor** es similar al servidor TCP de chat. Primero se definen las variables y campos de pantalla:

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Servidor extends JFrame {
    private static final long serialVersionUID = 1L;
    static Integer PUERTO = 44441;
    public static Integer conexiones = 0;
    static ServerSocket servidor;
    static java.util.Date hora;
    // campos de la pantalla
    static public JTextField numconex = new JTextField();
    static JLabel numconexLabel = new JLabel();
    static JTextField puerto = new JTextField();
    static JLabel puertoLabel = new JLabel();
    static public JTextArea area = new JTextArea();
    static JScrollPane scroll = new JScrollPane(area);
```

Desde el constructor se prepara la pantalla:

```
// Constructor
public Servidor() {
    super("SERVIDOR - CONTROL DE CONEXIONES A BD");
    Container c = getContentPane();
    numconexLabel.setText("Nº de conexiones actuales:");
    puertoLabel.setText("Número de puerto:");
    numconexLabel.setBounds(new Rectangle(10, 10, 160, 25));
    numconex.setBounds(new Rectangle(175, 10, 45, 25));
    puertoLabel.setBounds(new Rectangle(235, 10, 200, 25));
    puerto.setBounds(new Rectangle(350, 10, 50, 25));

    area.setBounds(new Rectangle(10, 60, 390, 200));
    scroll.setBounds(new Rectangle(10, 60, 400, 200));
    area.setEditable(false);c.add(scroll, null);
    c.add(numconexLabel);c.add(numconex);
    numconex.setEditable(false);c.add(puertoLabel);
    c.add(puerto);puerto.setEditable(false);
    c.setLayout(null);
    area.setForeground(Color.blue);area.setText("");
    setSize(450, 300); //COLOCACIÓN DE LA PANTALLA
    Dimension dim = getToolkit().getScreenSize();
    setLocation(dim.width / 2 - getWidth() / 2 + 200,
               (dim.height / 2 - getHeight() / 2) + 200);
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

También se define la acción a realizar cuando se cierra la ventana (en este ejemplo no hay un botón para cerrar el servidor). Al cerrar la ventana se cierra el servidor y la conexión a la base de datos:

```
//CERRAMOS VENTANA
```

```

addWindowListener(new WindowListener() {
    public void windowClosing(WindowEvent we) {
        try { // CERRAMOS EL SERVERSOCKET
            servidor.close();
            System.out.println("Servidor cerrado ....");
            Conexion.db.close(); // cerrar BD
            System.exit(0);
        } catch (IOException e) {
            System.err.println("NO SE PUEDE CERRAR servidor."
                + e.getMessage());
            System.exit(0);
        }
    }
    //resto de métodos no se implementan
    public void windowOpened(WindowEvent we) {; }
    public void windowClosed(WindowEvent we) {; }
    public void windowIconified(WindowEvent we) {; }
    public void windowDeiconified(WindowEvent we) {; }
    public void windowActivated(WindowEvent we) {; }
    public void windowDeactivated(WindowEvent we) {; }
});
}//.. fin del constructor

```

Desde *main()* se inicia el servidor y las variables, se visualiza el número de puerto por el que escucha y el número de conexiones actuales, y se prepara la pantalla (Figura 3.20):

```

//MAIN
public static void main(String[] args) throws IOException {
    int idCliente = 0; //cada cliente tendra un id
    servidor = new ServerSocket(PUERTO);
    System.out.println("Servidor Iniciado ....");
    Servidor pantalla = new Servidor();
    pantalla.setVisible(true);
    puerto.setText(PUERTO.toString());
    numconex.setText(conexiones.toString());
}

```

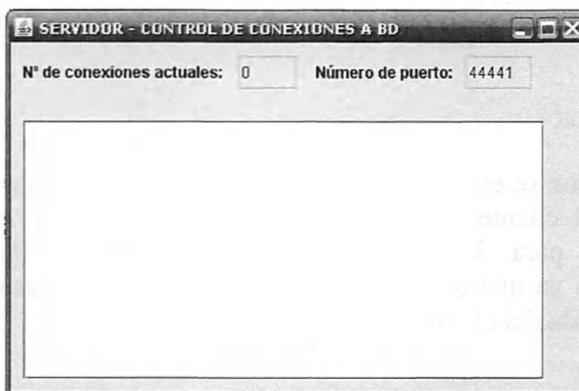


Figura 3.20. Pantalla inicial del servidor.

Se hace un bucle para controlar las conexiones de los clientes. Dentro del bucle el servidor espera la conexión del cliente, cuando uno se conecta se calcula la hora, se incrementa el número de conexiones, se calcula el identificador para el cliente (empieza en 0 y se va incrementando de

1 en 1), se averigua su dirección IP, se muestra toda esta información en pantalla, véase Figura 3.21, y se crea un hilo de ejecución para ese cliente:

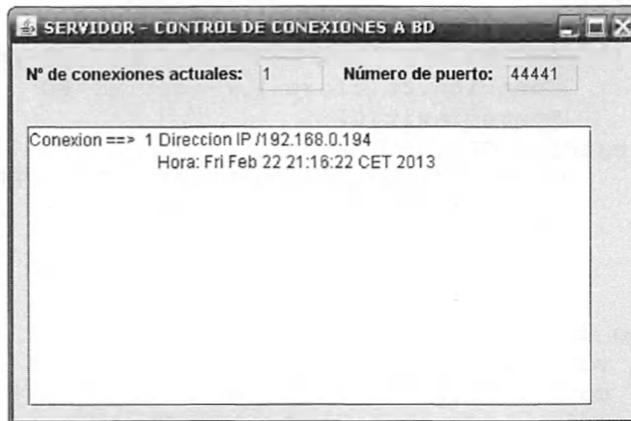


Figura 3.21. Pantalla del servidor cuando se conecta un cliente.

```

while (true) {
    try {
        Socket cliente=servidor.accept(); //esperando al cliente
        hora = new java.util.Date(System.currentTimeMillis());
        conexiones++;
        idCliente++;
        numconex.setText(conexiones.toString());
        area.append("Conexion ==> " + idCliente);
        InetAddress direccion = cliente.getInetAddress();
        area.append(" Direccion IP " + direccion.toString()
                   + "\n\t Hora: " + hora + "\n");
        HiloServidor hilo = new HiloServidor(cliente, idCliente);//
        hilo.start(); // Ejecutamos el hilo
    } catch (IOException e) {
        //ocurre cuando cerramos la ventana
        //porque el servidor esta cerrado
        System.out.println(e.getMessage());
        System.exit(0);
    }
} //while
} //main
} //..fin SERVIDOR

```

El hilo **HiloServidor** se encarga de recibir el número de departamento que el cliente quiere consultar, y enviar al cliente un objeto **Departamentos** con los datos solicitados. Necesita declarar un stream para la entrada **DataInputStream** y para la salida creará un **ObjectOutputStream** ya que devuelve un objeto al cliente. Desde el constructor se crean los flujos de entrada y salida del cliente:

```

import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;

class HiloServidor extends Thread {
    Socket socket;

```

```

int identificador;
static JLabel texto = new JLabel();
ObjectOutputStream outObjeto;
DataInputStream entrada;
//Constructor
public HiloServidor(Socket s, int idCliente) throws IOException {
    socket = s;
    identificador=idCliente;
    entrada = new DataInputStream(socket.getInputStream());
    outObjeto = new ObjectOutputStream(socket.getOutputStream());
}
//
```

En el método *run()* se lee del stream el departamento que el cliente desea consultar y se envía un objeto **Departamentos** con los datos solicitados. Para hacer esto se crea un objeto de la clase **AccesoDatos**. Con este objeto llamamos al método *procesarCadena()* que recibe el departamento que escribió el cliente en su pantalla y devuelve un objeto **Departamentos** con los datos del departamento solicitado:

```

public void run() {
    try {
        AccesoDatos adat = new AccesoDatos();
        while (true) {
            String depar = entrada.readUTF(); //leer stream
            Departamentos dep = adat.procesarCadena(depar.trim());
            // Se envía el objeto al cliente
            outObjeto.writeObject(dep);
        }
    } catch (IOException e) {
        // cuando un cliente Cierra la conexión
        Servidor.conexiones--;
        Servidor.numconex.setText(Servidor.conexiones.toString());
        texto.setText("<<LIBERADA LA CONEXIÓN: " +
                     identificador+" >>\n");
        texto.setForeground(Color.red);
        Servidor.area.append(texto.getText());
        try {
            entrada.close();
            outObjeto.close();
            socket.close();
        } catch (IOException ee) {
            ee.printStackTrace();
        }
    } ///catch
} // de run()
}//..fin HiloServidor
```

Cuando el cliente cierra la conexión (pulsa el botón *Salir* de su pantalla) se produce *IOException*, entonces se decrementa el número de conexiones y se muestra en la pantalla del servidor un mensaje indicando que se ha liberado la conexión del cliente. Se cierran los flujos de entrada y salida y el socket.

La clase **AccesoDatos** es la que comunica con la base de datos. En el constructor se crea un objeto a la base de datos:

```
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

//ACCESO A BD db4o
public class AccesoDatos {
    static ObjectContainer db;
    // Constructor
    public AccesoDatos() {
        db = Conexion.getDBConexion();
    }
}
```

Dispone del método `procesarCadena()` que recibe en un String el departamento a consultar. Lo pasa a entero. Usa el método `queryByExample()` para obtener el departamento deseado, si lo encuentra devolverá un objeto **Departamentos** y si no existe devolverá `null`:

```

// Se procesa la cadena que manda el hilo con el dep a localizar
synchronized Departamentos procesarCadena(String str) {
    int i;
    Departamentos d = null;
    try {
        i = Integer.parseInt(str);
    } catch (NumberFormatException n) {
        System.out.println("<<DEPARTAMENTO: " + str + " "
                           INCORRECTO>> ");
        return d;
    }
    Departamentos dep = new Departamentos(i, null, null, null);
    ObjectSet<Departamentos> result = db.queryByExample(dep);
    if (result.size() == 0)
        System.out.println("<<DEPARTAMENTO: " + i + " NO EXISTE>> ");
    else {
        d = result.next();
    }
    return d;// devuelve un objeto Departamentos
}//procarCadena
}///.fin AccesoDatos

```

El método *procesarCadena()* se define **synchronized** para evitar que dos hilos accedan simultáneamente al método. Solo uno lo podrá utilizar, el otro tendrá que esperar a que el primero termine.

Por último nos queda ver el código del **programa cliente**. En este caso la clase **ClienteBD** implementa **Runnable**. Se definen las variables, campos de la pantalla y los streams de entrada y de salida, en este caso el stream de entrada es un **ObjectInputStream** porque el cliente recibe un objeto; y el stream de salida es un **DataOutputStream** por el que el cliente envía un String:

```

private static final long serialVersionUID = 1L;
static JTextField depconsultar = new JTextField(2);
static JLabel etiqueta = new JLabel("Departamento a consultar:");
private JScrollPane scrollpanel;
static JTextArea textareal;
JButton boton = new JButton("Enviar");
JButton desconectar = new JButton("Salir");
boolean repetir = true;
static Socket socket;

//streams
ObjectInputStream inObjeto;
DataOutputStream salida;

```

En el constructor se recibe el socket creado, se crean los flujos de entrada y de salida y se prepara la pantalla:

```

// constructor
public ClienteBD(Socket s) {
    super("OPERACIONES CON BD");
    socket = s;
    try {
        // flujo de salida - para enviar cadena
        salida = new DataOutputStream(socket.getOutputStream());
        // flujo de entrada - para recibir objeto
        inObjeto = new ObjectInputStream(socket.getInputStream());
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
    }
    setLayout(null);
    etiqueta.setBounds(10, 10, 200, 30);add(etiqueta);
    depconsultar.setBounds(210, 10, 50, 30);add(depconsultar);

    textareal = new JTextArea();
    scrollpanel = new JScrollPane(textareal);
    scrollpanel.setBounds(10, 50, 400, 300); add(scrollpanel);
    boton.setBounds(420, 10, 100, 30); add(boton);
    desconectar.setBounds(420, 50, 100, 30); add(desconectar);

    textareal.setEditable(false);
    boton.addActionListener(this);
    desconectar.addActionListener(this);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
} // constructor

```

Cuando se pulsa el botón *Enviar* se envía el departamento tecleado por el stream de salida al hilo servidor:

```

// acción cuando pulsamos botones
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == boton) { // ENVIAR DEP
        try {
            salida.writeUTF(depconsultar.getText());
        } catch (IOException e1) {

```

```

        e1.printStackTrace();
    }
}//if

```

Cuando se pulsa el botón *Salir* se cierra el socket y finaliza la ejecución del cliente

```

if (e.getSource() == desconectar) { // SALIR
    try {
        socket.close();
    } catch (IOException e1) {
        e1.printStackTrace();
    }
    System.exit(0);
}//if
}// actionPerformed

```

En el método *run()* se realiza un proceso repetitivo donde el cliente recibe del hilo el objeto **Departamentos** con los datos del departamento solicitado. Si es *null* se visualiza un mensaje indicando que no existe el departamento, véase Figura 3.22, si no lo es se visualizarán los datos. Primero se pintan los datos del departamento (número, nombre y localidad) y después se llama al método *PintarEmpleados()* para visualizar los datos de los empleados del departamento:

```

// proceso repetitivo
public void run() {
    String texto = "";
    while (repetir) {
        try {
            Departamentos d = null;
            d = (Departamentos) inObjeto.readObject(); // recibo un objeto
            textareal.setText("");
            textareal.setForeground(Color.blue);
            if (d == null) {
                textareal.setForeground(Color.red);
                PintaMensaje("<<EL DEPARTAMENTO NO EXISTE>>");
            } else {
                // datos del departamento
                texto = "Número Dep: " + d.getDeptNo() + "\n"
                    + " Nombre: " + d.getDnombre() + "\tLocalidad: "
                    + d.getLoc();
                textareal.append(texto);
                PintarEmpleados(d); // visualizar empleados
            }
        } catch (IOException s) {
            repetir=false; // se produce al cerrar socket en botón salir
        } catch (SocketException e) {
            e.printStackTrace();
            repetir = false;
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            repetir = false;
        }
    }//fin while
    try {
        socket.close(); // CERRAR SOCKET
        System.exit(0);
    }
}

```

```

    } catch (IOException e) {
        e.printStackTrace();
    }
}// fin run

```

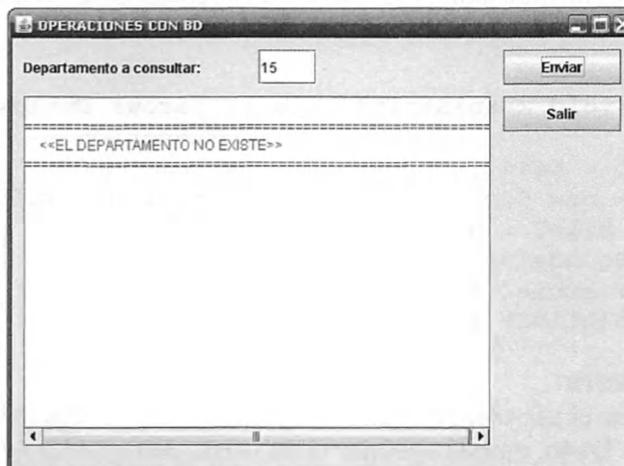


Figura 3.22. Pantalla del cliente, no existe el departamento.

El proceso repetitivo finaliza cuando la variable *repetir* sea *false*, que puede ser cuando ocurra alguna excepción. Al final se cierra el socket.

El método *PintarEmpleados()* pinta los empleados del objeto **Departamentos** que recibe. Primero recupera la colección de empleados en un objeto Set y después los recorre mediante un iterador. Según va recuperando empleados visualiza su datos en el textarea:

```

// PINTA LOS EMPLEADOS EN EL AREA
private void PintarEmpleados(Departamentos d) {
    Set<Empleados> listaemple = d.getEmpleados(); // obtenemos
                                                    // empleados
    textareal.setForeground(Color.blue);
    if (listaemple == null) {
        PintaMensaje("EL DEPARTAMENTO NO TIENE EMPLEADOS");
    } else {
        PintaMensaje("EMPLEADOS DEL DEPARTAMENTO: " +
                     listaemple.size());
        Iterator<Empleados> it = listaemple.iterator();
        while (it.hasNext()) {
            Empleados emple = new Empleados();
            emple = it.next();
            textareal.append("\n" + emple.getEmpNo() + " * "
                           + emple.getApellido() + " * " + emple.getOficio()
                           + " * " + emple.getSalario());
        }
    }
}
//PintarEmpleados
El método PintaMensaje() visualiza en el textarea el mensaje que recibe:
// PINTA CABECERAS
void PintaMensaje(String mensaje) {
    textareal.append("\n=====");

```

```

        textareal.append("\n" + mensaje);
        textareal.append("\n=====");
    }//PintaMensaje

```

Por último desde el método *main()* se realiza la conexión al servidor, se prepara la pantalla y se lanza el hilo cliente:

```

// MAIN
public static void main(String args[]) throws UnknownHostException,
    IOException {
    int puerto = 44441;
    Socket s = new Socket("localhost", puerto); //máquina local
    ClienteBD hiloC = new ClienteBD(s);
    hiloC.setBounds(0, 0, 540, 400);
    hiloC.setVisible(true);
    new Thread(hiloC).start();
} // fin main
}// Fin del CLIENTE

```

Para poder ejecutar el servidor necesitamos incluir en el CLASSPATH la librería para poder usar la base de datos **Db4o**, en este ejemplo se ha usado *db4o-8.0.224.15975-core-java5.jar*. En la máquina donde instalemos el servidor necesitamos las clases: *Servidor*, *HiloServidor*, *Empleados*, *Departamentos*, *Conexion* y *AccesoDatos*; además del fichero JAR. En la máquina donde instalemos el cliente necesitamos las clases *ClienteBD*, *Empleados* y *Departamentos*.

Igual que en el ejemplo del chat TCP, cuando los programas cliente y el servidor estén en máquinas diferentes, al crear el socket en el programa cliente indicaremos la dirección IP donde está la máquina servidora, ejemplo:

```
Socket s = new Socket("192.168.0.194", puerto);
```

### ACTIVIDAD 3.7

Prueba los programas cliente y servidor desde diferentes máquinas. Las clases: *Servidor*, *HiloServidor*, *Empleados*, *Departamentos*, *Conexion*, *AccesoDatos* y el fichero JAR tienen que estar en la máquina servidora. Las clases *ClienteBD*, *Empleados* y *Departamentos* en las máquinas cliente.

## COMPRUEBA TU APRENDIZAJE

1º) Realiza un programa servidor con interfaz gráfica que escuche en el puerto 44444. La pantalla inicial es similar a la vista en el servidor TCP de chat, véase Figura 3.23. Cada vez que se conecte un cliente se creará un nuevo hilo para atenderlo. Se mostrará en la pantalla la IP del cliente que se conecta y cuando el cliente se desconecte se debe mostrar un mensaje indicando que se ha desconectado:

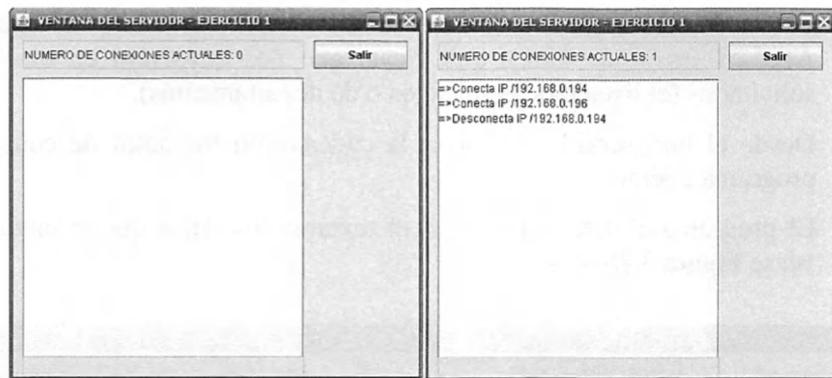


Figura 3.23. Pantalla del servidor Ejercicio 1.

En el hilo, se recibe una cadena de caracteres del cliente, si es distinta de “\*” se enviará de nuevo al cliente convertida a mayúsculas.

En el programa cliente se muestra una pantalla donde el cliente escribe una cadena y al pulsar en el botón *Enviar* se muestra debajo la cadena en mayúsculas, Figura 3.24. El botón *Limpiar* limpia los dos campos y el botón *Salir* envía un \* al servidor y finaliza la ejecución. Si la cadena que envía el cliente es un \* también finaliza la ejecución.

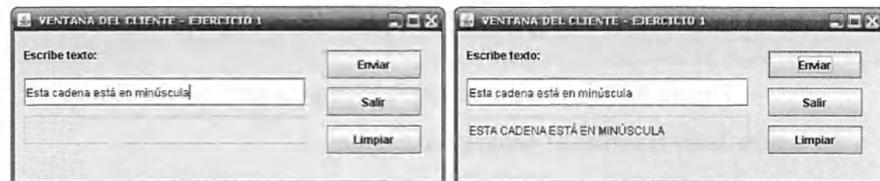


Figura 3.24. Pantalla del cliente Ejercicio 1.

2º) Partimos del ejercicio que utiliza la base de datos **Db4o** EMPLEDEP.YAP. Realiza los cambios necesarios en el programa cliente, en el hilo servidor y en la clase de acceso a la base de datos. La pantalla inicial del programa cliente se muestra en la Figura 3.25.

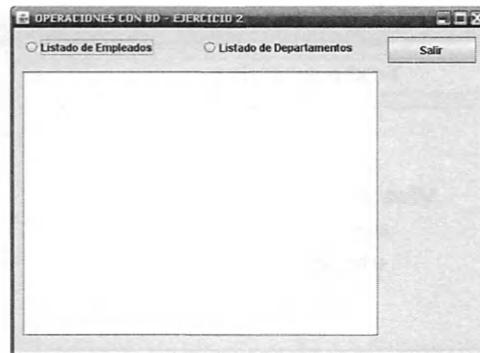


Figura 3.25. Pantalla inicial del cliente Ejercicio 2.

- Desde esta pantalla el cliente puede seleccionar un listado de empleados o un listado de departamentos. Al pulsar en la primera opción (*Listado de Empleados*) se debe enviar al hilo servidor una cadena que indique que se desea un listado de empleados (por ejemplo la cadena EMP), si el listado seleccionado es el de departamentos la cadena será diferente (por ejemplo DEP).

- La cadena recibida en el hilo servidor será enviada a un método de la clase **AccesoDatos** (que tienes que crear) que nos devolverá en otra cadena los datos solicitados (el listado de empleados o de departamentos).
- Desde el hilo servidor se envía la cadena con los datos de consulta solicitados al programa cliente.
- El programa cliente muestra en el textarea los datos que le envía el hilo servidor, véase Figura 3.26.

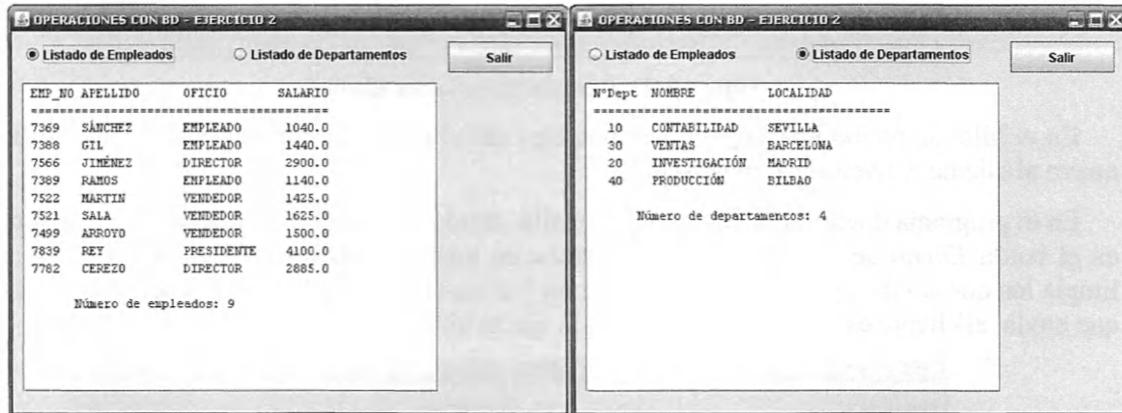


Figura 3.26. Consulta de empleados y departamentos Ejercicio 2.

- El botón *Salir* finaliza el programa cliente.

3º) Realiza un servidor multicast usando sockets UDP. El servidor debe mostrar una pantalla inicial como la mostrada en la Figura 3.27, donde tenemos un campo de texto para escribir el mensaje que se enviará a todos los clientes y un textarea donde se van mostrando los mensajes que se van enviando. El botón *Enviar* envía el mensaje escrito a todos los clientes conectados y el botón *Salir* finaliza la ejecución del servidor.

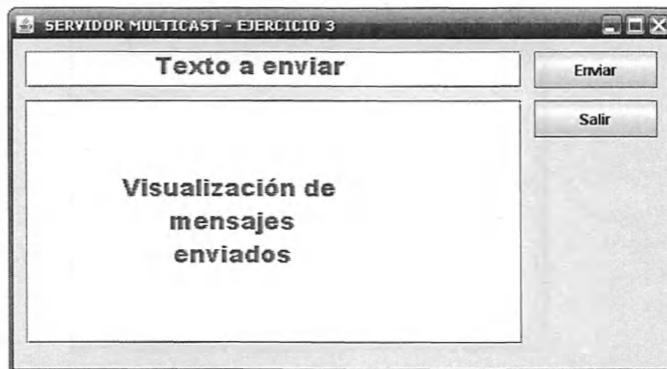


Figura 3.27. Pantalla inicial del servidor multicast.

El programa cliente pide el nombre al usuario y a continuación muestra un textarea donde se irán visualizando los mensajes que envía el servidor. El botón *Salir* finaliza la ejecución.

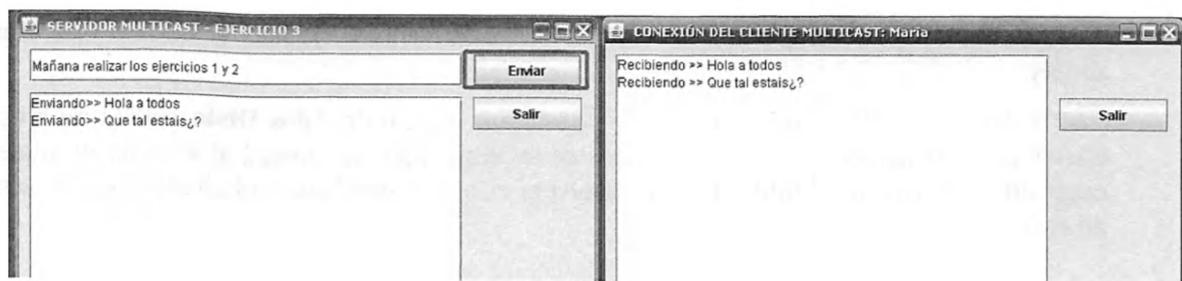


Figura 3.28. Ejecución del Ejercicio 3.

## ACTIVIDADES DE AMPLIACIÓN

1º) Partiendo del ejemplo resuelto de consulta a la base de datos **Db4o** realiza un programa cliente para dar de alta departamentos. Al ejecutar el cliente se debe mostrar la pantalla de alta, Figura 3.29.

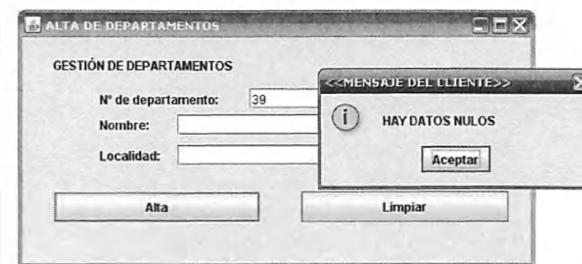
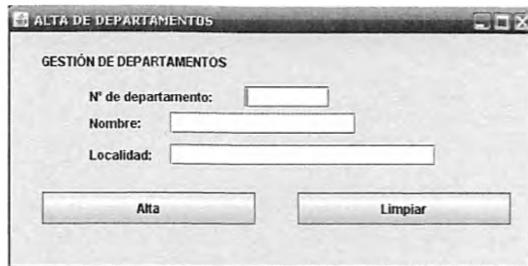


Figura 3.29. Pantalla de entrada de departamentos. Figura 3.30. Pantalla de error en la entrada de datos.

Cuando se pulse el botón de *Alta* se debe comprobar si el departamento es numérico y si los campos nombre y localidad tienen datos. Si ocurre algún error en la comprobación se debe visualizar una ventanita indicando el error, Figura 3.30.

Si todo está correcto el cliente envía un objeto **Departamentos** con los datos tecleados al hilo servidor para que realice la inserción del departamento en la base de datos. El hilo servidor debe mandar como respuesta al cliente los mensajes *Departamento insertado*, si se ha insertado el departamento o *Departamento existente*, si ya existe. El cliente recibe el mensaje y lo visualiza en una ventanita, Figura 3.31.

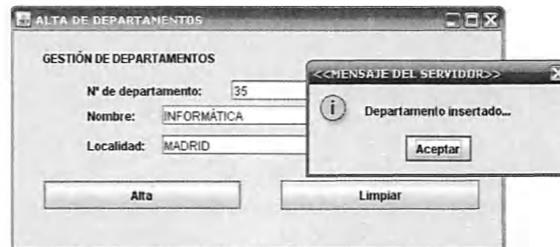


Figura 3.31. Pantalla de entrada de departamentos.

El botón *Limpiar*, limpia la pantalla. Al cerrar la ventana finaliza el cliente y se cierra el socket.

2º Partiendo del ejemplo resuelto de consulta a la base de datos **Db4o** realiza un programa cliente que nos permita consultar los datos de un empleado. Se enviará al servidor el número de empleado a consultar, el hilo servidor devolverá el objeto empleado solicitado si existe o null si no existe.

## VENTANA DE CONSULTA

Este es el resultado de la ejecución de la aplicación. Se muestra la interfaz de usuario que permite introducir el número de empleado a consultar.

Al pulsar el botón *Consultar* se muestra el resultado de la ejecución del hilo.

Al pulsar el botón *Limpiar* se limpia la pantalla y se cierra la aplicación.

Al pulsar el botón *Salir* se cierra la aplicación.

Al pulsar el botón *Salir* se cierra la aplicación.

