

## 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

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

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 las 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.

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:

- **TCP:** Protocolo basado en la conexión, garantiza que los datos enviados desde un extremo de la conexión llegan 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** contiene clases e interfaces para la implementación de aplicaciones de red. Éstas incluyen:

- La clase **URL**, *Uniform Resource Locator* (Localizador Uniforme de Recursos). Representa un puntero a un recurso en la Web.
- La clase **URLConnection**, que admite operaciones más complejas en las URL.
- Las clases **ServerSocket** y **Socket**, para dar soporte a sockets TCP. **ServerSocket**: utilizada por el programa servidor para crear un socket en el puerto en el que escucha las peticiones de conexión de los clientes. **Socket**: utilizada tanto por el cliente como por el servidor para comunicarse entre sí leyendo y escribiendo datos usando streams.
- Las clases **DatagramSocket**, **MulticastSocket** y **DatagramPacket** para dar soporte a la comunicación vía datagramas UDP.
- La clase **InetAddress**, que representa las direcciones de Internet.

### 3.2.1. LOS PUERTOS

Los protocolos TCP y UDP usan **puertos** para asignar datos entrantes a un proceso en particular que se ejecuta en un ordenador.

En términos generales, un ordenador tiene una única conexión física a la red. Los datos destinados a este ordenador llegan a través de esa conexión. Sin embargo, los datos pueden estar destinados a diferentes aplicaciones que se ejecutan en el ordenador. Entonces, ¿cómo sabe el ordenador a qué aplicación enviar los datos? Mediante el uso de puertos.

Los datos transmitidos a través de Internet van acompañados de información de direccionamiento que identifica la máquina y el puerto para el que está destinada. La máquina se identifica por su dirección IP de 32 bits, para entregar datos a una máquina concreta necesitaremos conocer su IP. Los puertos se identifican mediante un número de 16 bits, que TCP y UDP utilizan para entregar los datos a la aplicación correcta.

En la comunicación basada en TCP, una aplicación de servidor vincula un socket a un número de puerto específico. Esto tiene el efecto de registrar el servidor en el sistema para recibir todos los datos destinados a ese puerto. Una aplicación cliente puede entonces comunicarse con el servidor enviándole peticiones a través de ese puerto.

En la comunicación basada en datagramas, como UDP, el paquete de datagramas contiene el número de puerto de su destino y UDP enruta el paquete a la aplicación adecuada.

### 3.2.2. 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
InetAddress getLocalHost()	Devuelve un objeto <i>InetAddress</i> que representa la dirección IP de la máquina donde se está ejecutando el programa.
InetAddress getByName(String host)	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.
InetAddress[] getAllByName(String host)	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.
String getHostAddress()	Devuelve la dirección IP de un objeto <i>InetAddress</i> en forma de cadena.
String getHostName()	Devuelve el nombre del host de un objeto <i>InetAddress</i> .
String getCanonicalHostName()	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*. La forma más típica de crear instancias de **InetAddress**, es invocando al método estático *getByName(String)* pasándole el nombre DNS del host como parámetro. Este objeto representará la dirección IP de ese host, y se podrá utilizar para construir sockets.

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 *localhost*. A continuación llamamos al método *pruebaMetodos()* llevando el objeto creado. En dicho método se prueban los métodos de la clase **InetAddress**. Despues 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**:

```
import java.net.*;
public class TestInetAddress {
    public static void main(String[] args) {
        InetAddress dir = null;
        System.out.println("=====");
        System.out.println("SALIDA PARA LOCALHOST: ");
        try {
            //LOCALHOST
            dir = InetAddress.getByName("localhost");
            pruebaMetodos(dir);

            //URL www.google.es
            System.out.println("=====");
            System.out.println("SALIDA PARA UNA URL:");
            dir = InetAddress.getByName("www.google.es");
            pruebaMetodos(dir);

            //Array de tipo InetAddress con todas las direcciones IP
            //asignadas a google.es
            System.out.println("\tDIRECCIONES IP PARA: " + dir.getHostName());
            InetAddress[] direcciones =

```

```

        InetAddress.getAllByName(dir.getHostName());
    for (int i = 0; i < direcciones.length; i++)
        System.out.println("\t\t"+direcciones[i].toString());

    System.out.println("=====");

} catch (UnknownHostException e1) {e1.printStackTrace();}

}// main

private static void pruebaMetodos(InetAddress dir) {
    System.out.println("\tMetodo getByName(): " + dir);
    InetAddress dir2;
    try {
        dir2 = InetAddress.getLocalHost();
        System.out.println("\tMetodo getLocalHost(): " + dir2);
    } catch (UnknownHostException e) {e.printStackTrace();}

//USAMOS MÉTODOS DE LA CLASE
System.out.println("\tMetodo getHostName(): "+dir.getHostName());
System.out.println("\tMetodo getHostAddress(): "+
    dir.getHostAddress());
System.out.println("\tMetodo toString(): " + dir.toString());
System.out.println("\tMetodo getCanonicalHostName(): "+
    dir.getCanonicalHostName());

}//pruebaMetodos

}//Fin

```

La salida generada es la siguiente:

---

#### SALIDA PARA LOCALHOST:

```

Metodo getByName(): localhost/127.0.0.1
Metodo getLocalHost(): PC-ASUS/192.168.56.1
Metodo getHostName(): localhost
Metodo getHostAddress(): 127.0.0.1
Metodo toString(): localhost/127.0.0.1
Metodo getCanonicalHostName(): 127.0.0.1

```

---

#### SALIDA PARA UNA URL:

```

Metodo getByName(): www.google.es/216.58.210.131
Metodo getLocalHost(): PC-ASUS/192.168.56.1
Metodo getHostName(): www.google.es
Metodo getHostAddress(): 216.58.210.131
Metodo toString(): www.google.es/216.58.210.131
Metodo getCanonicalHostName(): mad06s09-in-f3.1e100.net
DIRECCIONES IP PARA: www.google.es
    www.google.es/216.58.210.131

```

---



---

#### 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.

---

### 3.2.3. LA CLASE URL

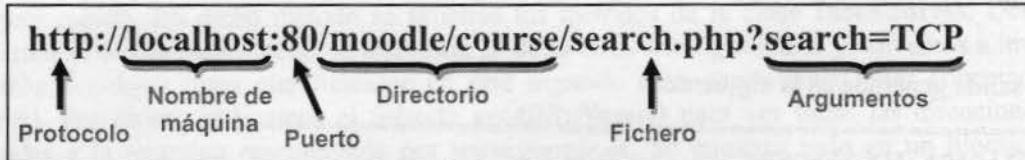
La clase **URL** (*Uniform Resource Locator*) representa un puntero a un recurso en la Web. Un recurso puede ser algo tan simple como un fichero o un directorio, o puede ser una referencia a un objeto más complicado, como una consulta a una base de datos o a un motor de búsqueda.

En general una URL que localiza recursos empleando el protocolo HTTP se divide en varias partes: *http://host[:puerto][/nombredelpathdelservidor][?argumentos]*, las partes encerradas entre corchetes son opcionales:

- **host:** Es el nombre de la máquina en la que reside el recurso.
- **[:puerto]:** Número de puerto en el que el servidor escucha las peticiones. Este parámetro es opcional y si no se indica se considera el puerto defecto. Para el protocolo HTTP es el 80.
- **[/directoriodelservidor]:** Es el path o directorio donde se encuentra el recurso en el sistema de ficheros del servidor. Si no se indica se proporciona la página por defecto del servidor web.
- **[?argumentos]:** Parámetros que se envían al servidor. Por ejemplo, cuando realizamos una consulta se pueden enviar parámetros a un fichero PHP para procesarla.

Por ejemplo en la siguiente URL:

*http://localhost:80/moodle/course/search.php?search=TCP*, encontramos el protocolo (*http*), el puerto (80), el nombre de máquina (*localhost*), el fichero (*search.php*) que está en un directorio dentro del servidor (*/moodle/course*) y los argumentos (*search=TCP*) que se envían al fichero *search.php* para realizar una búsqueda:



La clase URL contiene varios constructores, algunos son:

CONSTRUCTOR	MISIÓN
<code>URL(String url)</code>	Crea un objeto URL a partir del String indicado en <i>url</i> .
<code>URL(String protocolo, String host, String fichero)</code>	Crea un objeto URL a partir de los parámetros <i>protocolo</i> , <i>host</i> y <i>fichero</i> (o directorio).
<code>URL(String protocolo, String host, int puerto, String fichero)</code>	Crea un objeto URL en el que se especifica el <i>protocolo</i> , <i>host</i> , <i>punto</i> y <i>fichero</i> (o directorio) representados mediante String.
<code>URL(URL contexto, String especificación)</code>	Crea un objeto URL analizando la especificación dada dentro de un contexto específico.

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
<code>String getAuthority()</code>	Obtiene la autoridad del objeto URL.
<code>int getDefaultPort()</code>	Devuelve el puerto asociado por defecto al objeto URL.
<code>int getPort()</code>	Devuelve el número de puerto de la URL, -1 si no se indica.
<code>String getHost()</code>	Devuelve el nombre de la máquina.
<code>String getQuery()</code>	Devuelve la cadena que se envía a una página para ser procesada (es lo que sigue al signo? de una URL).
<code>String getPath()</code>	Devuelve una cadena con la ruta hacia el fichero desde el servidor y el nombre completo del fichero.
<code>String getFile()</code>	Devuelve lo mismo que <code>getPath()</code> , además de la concatenación del valor de <code>getQuery()</code> si lo hubiese. Si no hay una porción consulta, este método y <code>getPath()</code> devolverán los mismos resultados.
<code>String getProtocol()</code>	Devuelve el nombre del protocolo asociado al objeto URL.
<code>String getUserInfo()</code>	Devuelve la parte con los datos del usuario o nulo si no existe.
<code>InputStream openStream()</code>	Devuelve un <b>InputStream</b> del que podremos leer el contenido del recurso que identifica la URL.
<code>URLConnection openConnection()</code>	Devuelve un objeto <b>URLConnection</b> que nos permite abrir una conexión con el recurso y realizar operaciones de lectura y escritura sobre él.

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 + directorio:");
            url = new URL("http", "docs.oracle.com", "/javase/10");
            Visualizar(url);

            System.out.println("Constructor para protocolo + URL + puerto +
                               directorio:");
            url = new URL("http", "localhost", 8084,
                          "/WebApp/Controlador?accion=modificar");
            Visualizar(url);

            System.out
                .println("Constructor para un objeto URL en un contexto:");
            URL urlBase = new URL("https://docs.oracle.com/");
            url = new URL(urlBase, "/javase/10/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("\tgetDefaultPort(): " + url.getDefaultPort());
    System.out
        .println("=====");
} //
} // Ejemplo1URL

```

La salida generada es la siguiente:

Constructor simple para una URL:

```

    URL completa: http://docs.oracle.com/
    getProtocol(): http
    getHost(): docs.oracle.com
    getPort(): -1
    getFile(): /
    getUserInfo(): null
    getPath(): /
    getAuthority(): docs.oracle.com
    getQuery(): null
    getDefaultPort(): 80
=====

```

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
    getDefaultPort(): 80
=====
```

Const. para protocolo +URL + directorio:

```

    URL completa: http://docs.oracle.com/javase/10
    getProtocol(): http
    getHost(): docs.oracle.com
    getPort(): -1
    getFile(): /javase/10
    getUserInfo(): null
    getPath(): /javase/10
    getAuthority(): docs.oracle.com
    getQuery(): null
    getDefaultPort(): 80
=====
```

```
=====
Constructor para protocolo + URL + puerto + directorio:  
    URL completa:  
http://localhost:8084/WebApp/Controlador?accion=modificar  
    getProtocol(): http  
    getHost(): localhost  
    getPort(): 8084  
    getFile(): /WebApp/Controlador?accion=modificar  
    getUserInfo(): null  
    getPath(): /WebApp/Controlador  
    getAuthority(): localhost:8084  
    getQuery(): accion=modificar  
    getDefaultPort(): 80  
=====  
Constructor para un objeto URL en un contexto:  
    URL completa:  
https://docs.oracle.com/javase/10/docs/api/java/net/URL.html  
    getProtocol(): https  
    getHost(): docs.oracle.com  
    getPort(): -1  
    getFile(): /javase/10/docs/api/java/net/URL.html  
    getUserInfo(): null  
    getPath(): /javase/10/docs/api/java/net/URL.html  
    getAuthority(): docs.oracle.com  
    getQuery(): null  
    getDefaultPort(): 443  
=====
```

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
```

### 3.2.4. 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>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.
<b>String getContentType()</b>	Devuelve el valor del campo de cabecera <i>content-type</i> o null si no está definido.
<b>long getDate()</b>	Devuelve el valor del campo de cabecera <i>date</i> o 0 si no está definido.
<b>long getLastModified()</b>	Devuelve el valor del campo de cabecera <i>last-modified</i>
<b>String getHeaderField(int n)</b>	Devuelve el valor del enésimo campo de cabecera especificado o null si no está definido.
<b>Map&lt; String, List&lt;String&gt; &gt; getHeaderFields()</b>	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.
<b>URL getURL()</b>	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 el recurso representado por la URL y el anterior abre directamente un stream desde la URL:

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

public class EjemplolurlCon {
    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();}
    //}
}//EjemplolurlCon
```

Gran cantidad de páginas HTML contienen formularios a través de los cuales podemos solicitar información a un servidor llenando 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* 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 ";
```

?>

En él se reciben los valores introducidos en los campos *nombre* y *apellidos* del formulario, mediante la instrucción `$_POST["nombrecampo"]`, y se visualizan en la pantalla del navegador mediante la orden `echo`. La URL para enviar datos al formulario, suponiendo que los ficheros `.html` y `.php` residen en la carpeta `2018` del servidor web local sería similar a la siguiente:

`http://localhost/2018/vernombre.php?nombre=María&apellidos=Ramos&ver=Ver`

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 `2018` con el script PHP `vernombre.php`, la URL sería la siguiente: `URL url = new URL("http://localhost/2018/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 usando el método `setDoOutput()`: `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, éste 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/2018/vernombre.php");
            URLConnection conexion = url.openConnection();
            conexion.setDoOutput(true);

            String cadena ="nombre=María Jesús&apellidos=Ramos Martín";

            //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/2018/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);
}
}
}//Ejemplo3urlCon

```

**NOTA:** Para recorrer una estructura **Map** podemos usar una la interfaz **Iterator**. Para obtener un iterador sobre el map se invoca a los métodos *entrySet()* e *iterator()*. El método *entrySet()* devuelve un Set de objetos **Map.Entry** que contienen los pares. 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 ejecución muestra la siguiente salida:

```

Direccion [getURL()]:http://localhost/2018/vernombre.html
Fecha ultima modificacion [getLastModified()]: Thu Dec 07 19:26:20 CET
2017
Tipo de Contenido [getContentType()]: text/html
=====
TODOS LOS CAMPOS DE CABECERA CON getHeaderFields():
Keep-Alive : [timeout=5, max=100]
Accept-Ranges : [bytes]
null : [HTTP/1.1 200 OK]
Server : [Apache/2.4.4 (Win32) OpenSSL/0.9.8y PHP/5.4.16]
ETag : ["138-55fc4345d98b5"]
Connection : [Keep-Alive]
Last-Modified : [Thu, 07 Dec 2017 18:26:20 GMT]
Content-Length : [312]
Date : [Thu, 07 Dec 2017 18:32:45 GMT]
Content-Type : [text/html]
=====
CAMPOS 1 Y 4 DE CABECERA:
getHeaderField(1)=> Thu, 07 Dec 2017 18:32:45 GMT
getHeaderField(4)=> "138-55fc4345d98b5"
=====
CONTENIDO DE [url.getFile()]:/2018/vernombre.html
<html>
<body>
<form action="vernombre.php" method="post" >
    <p>Escribe tu nombre:
        <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 el concepto 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.

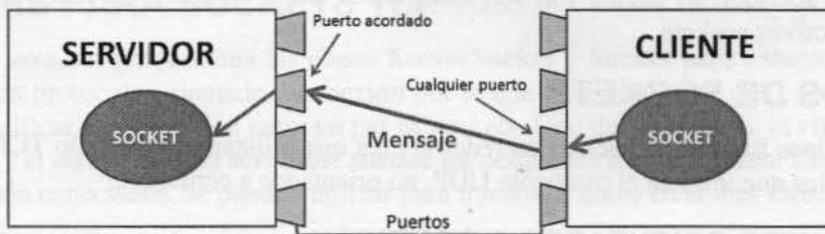


Figura 3.2. Socket y puertos.

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.

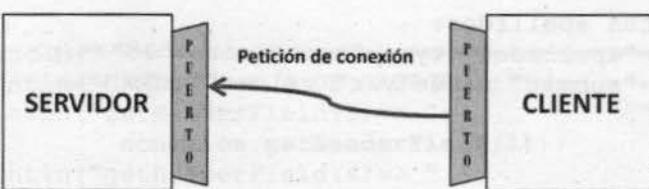


Figura 3.3. Petición de conexión del cliente.

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.

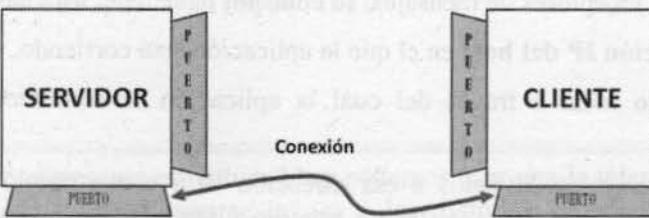


Figura 3.4. Conexión cliente-servidor.

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 se 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
Socket accept ()	El método <b>accept()</b> 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 <b>accept()</b> . Puede lanzar <b>IOException</b> .
close ()	Se encarga de cerrar el <b>ServerSocket</b> .
int getLocalPort ()	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(); //cierra 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
Socket()	Crea un socket sin ningún puerto asociado.
Socket (InetAddress address, int port)	Crea un socket y lo conecta al puerto y dirección IP especificados.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Permite además especificar la dirección IP local y el puerto local a los que se asociará el socket.
Socket (String host, int port)	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar <b>UnKnownHostException</b> , <b>IOException</b>

Algunos métodos importantes son:

MÉTODOS	MISIÓN
InputStream getInputStream ()	Devuelve un <b>InputStream</b> que permite leer bytes desde el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <b>IOException</b> .
OutputStream getOutputStream ()	Devuelve un <b>OutputStream</b> que permite escribir bytes sobre el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar <b>IOException</b> .

MÉTODOS	MISIÓN
<code>close ()</code>	Se encarga de cerrar el socket.
<code>InetAddress getInetAddress ()</code>	Devuelve la dirección IP 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 ("Nombre Host/IP: "+ Cliente.getInetAddress());
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: 63120
Puerto Remoto: 6000
Nombre Host/IP: localhost/127.0.0.1
Host Remoto: localhost
IP Host Remoto: 127.0.0.1
```

### ACTIVIDAD 3.2

Realiza un programa servidor TCP que acepte dos clientes. Muestra por cada cliente conectado sus puertos local y remoto.

Crea también el programa cliente que se conecte a ese servidor. Muestra los puertos locales y remotos a los que está conectado su socket, y la dirección IP de la máquina remota a la que se conecta.

#### 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 del **InputStream**. Igualmente, el servidor escribirá los mensajes al **OutputStream** y el cliente los leerá del **InputStream**.

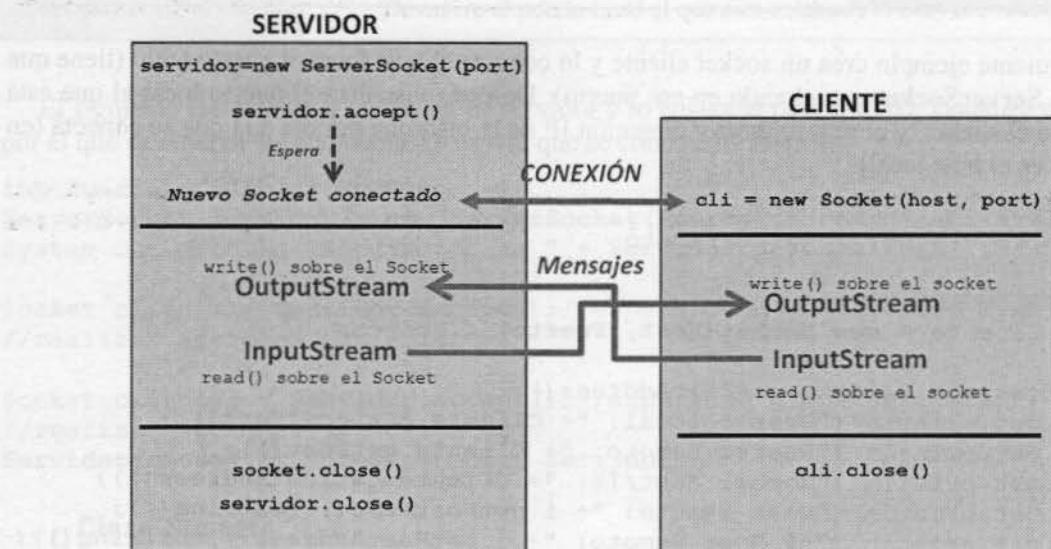


Figura 3.5. Modelo de Socket TCP.

### 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 TCPEjemploServidor {
    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();
    }
}
```

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 TCPEjemplo1Cliente {
    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.

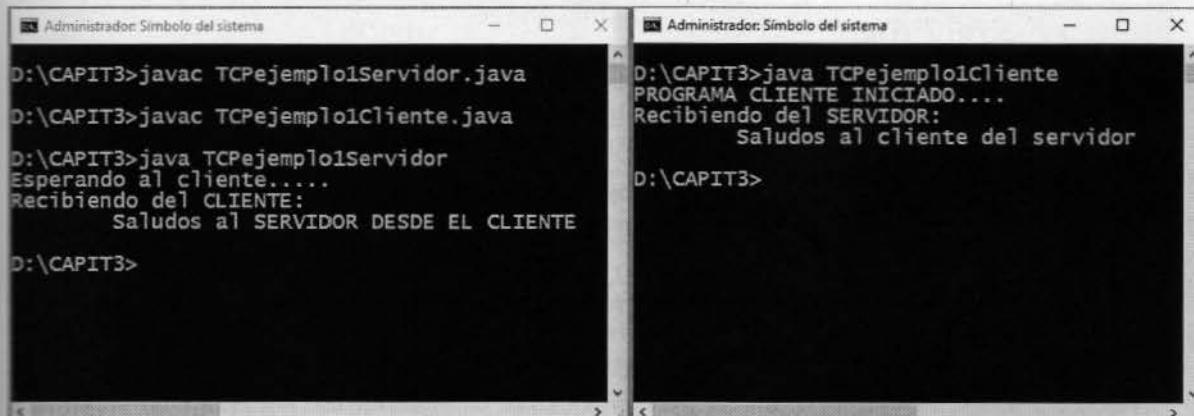


Figura 3.6. Ejecución de un programa cliente y otro servidor con TCP.

**ACTIVIDAD 3.3**

Crea un programa servidor que envíe un mensaje a otro programa cliente y el programa cliente que le devuelva el mensaje en minúscula.

**ACTIVIDAD 3.4**

Crea un programa cliente que introduzca por teclado un número entero y se lo envíe al servidor. El servidor le devolverá el cuadrado del número.

**ACTIVIDAD 3.5**

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. Cambia el programa para que lo haga para N clientes, siendo N un parámetro que tendrás que definir en el programa.

Realiza el Ejercicio 1.

### 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 de los datagramas que se van a recibir y de los datagramas que van a ser enviados:

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

El siguiente ejemplo utiliza el tercer constructor para construir un datagrama de envío. El datagrama se enviará a la dirección IP de la máquina local, que lo estará esperando 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>port: 12345</i>
--------------------------------------	---------------------	----------------------------	--------------------

```
int port = 12345; //puerto por el que escucha
InetAddress destino = InetAddress.getLocalHost(); //IP host local

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);
```

Para definir el destino de un host con una IP concreta, por ejemplo 80.120.54.1, escribo lo siguiente:

```
InetAddress destino = InetAddress.getByName("80.120.54.1");
```

Algunos métodos importantes de la clase **DatagramPacket** son:

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

El siguiente ejemplo utiliza el primer constructor de **DatagramPacket** para construir un datagrama de recepción. El mensaje se aloja en la variable *buf*, se obtiene la longitud y el mensaje contenido en el 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:

```

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

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
DatagramSocket ()	Construye un socket para datagramas, el sistema elige un puerto de los que están libres.
DatagramSocket (int port)	Construye un socket para datagramas y lo conecta al puerto local especificado.
DatagramSocket (int port, InetAddress ip)	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 12345, escribimos:

```
DatagramSocket socket = new DatagramSocket(12345);
```

Algunos métodos importantes son:

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

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. Despues se construye el datagrama para recepción 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); //recibo 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: Enviendo Saludos !!
Puerto origen del mensaje: 34567
IP de origen: 169.254.30.179
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 3.7 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.

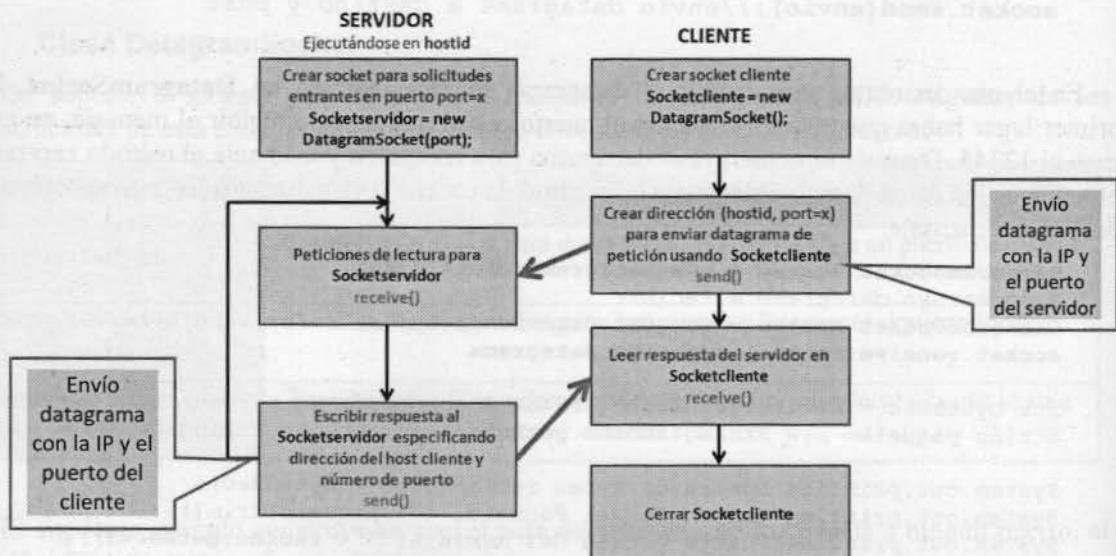


Figura 3.7. Envío y recepción de un datagrama.

### 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 12345 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
    (12345, InetAddress.getByName("localhost"));
```

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

### Envío y recepción de datagramas.

Para crear los datagramas de envío y de recepción 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 UDPservidor {
    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 número 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 UDPcliente {
    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 3.8.

Figura 3.8. 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 número de apariciones de la letra 'a' en el texto. El programa cliente recibe el datagrama del servidor y muestra el número de repeticiones de la letra 'a'. Para comenzar la ejecución primero ejecutamos el programa servidor que permanecerá a la espera, y después (desde otra consola) ejecutamos el programa cliente. La Figura 3.9 muestra la ejecución.

The figure consists of two side-by-side terminal windows. The left window shows the compilation of two Java files: 'javac UDPclienteEjemplo2.java' and 'javac UDPServidorEjemplo2.java'. It also shows the execution of the server code, which prints messages like 'Servidor Esperando Datagrama.....', 'Servidor Recibe: esto es una prueba', and 'Enviendo número de apariciones de la letra a=> 2'. The right window shows the execution of the client code, which prints 'Introduce mensaje: esto es una prueba', 'Esperando datagrama...', 'Recibo N° de caracteres que son a=> 2', and then closes.

Figura 3.9. Intercambio de datagramas entre cliente y servidor.

El programa cliente es el siguiente:

```

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

public class UDPclienteEjemplo2 {
    private static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        DatagramSocket clientSocket = new DatagramSocket();

        //DATOS DEL SERVIDOR AL QUE ENVIAR MENSAJE
        InetAddress IPServidor = InetAddress.getLocalHost();
        int puerto = 12345; // puerto por el que escucha

        //INTRODUCIR DATOS POR TECLADO
        System.out.print("Introduce mensaje: ");
        String cadena = sc.nextLine();

        byte[] enviados = new byte[1024];
        enviados = cadena.getBytes(); //convertir cadena a bytes

        //ENVIANDO DATAGRAMA AL SERVIDOR
        DatagramPacket envio = new DatagramPacket(enviados,
                                                    enviados.length, IPServidor, puerto);
        clientSocket.send(envio);

        //RECIBIENDO DATAGRAMA DEL SERVIDOR
        byte[] recibidos = new byte[2];
        DatagramPacket recipro =
            new DatagramPacket(recibidos, recibidos.length);
        System.out.println("Esperando datagrama....");
        clientSocket.receive(recipro);

        //Obtener el número de caracteres
        byte[] hh = recipro.getData();
        int numero = hh[0];

        System.out.println("Recibo N° de caracteres que son a=> " +
                           numero);
    }
}

```

```

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

```

El programa servidor es el siguiente:

```

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

public class UDPservidorEjemplo2 {

    public static void main(String[] args) throws IOException {
        //ASOCIO EL SOCKET AL PUERTO 12345
        DatagramSocket socket = new DatagramSocket(12345);

        //ESPERANDO DATAGRAMA
        System.out.println("Servidor Esperando Datagrama ..... ");
        DatagramPacket recibo;

        byte[] bufer = new byte[1024];
        recibo = new DatagramPacket(bufer, bufer.length);
        socket.receive(recibo); // RECIBO DATAGRAMA

        String mensaje = new String(recibo.getData()).trim();
        System.out.println("Servidor Recibe:" + mensaje);

        //CONTAR EL NÚMERO DE LETRAS a
        int contador = 0;
        for (int i = 0; i < mensaje.length(); i++)
            if (mensaje.charAt(i) == 'a')
                contador++;

        //DIRECCIÓN ORIGEN DEL MENSAJE
        InetAddress IPOrigen = recibo.getAddress();
        int puerto = recibo.getPort();

        //ENVIANDO DATAGRAMA AL CLIENTE
        System.out.println("Enviando número de apariciones
                           de la letra a=> " + contador);
        byte b = (byte) contador; // paso entero a byte
        byte[] enviados = new byte[1024];
        enviados[0] = b;

        DatagramPacket envio = new DatagramPacket(enviados,
                                                enviados.length, IPOrigen, puerto);
        socket.send(envio);

        // CERRAR SOCKET
        System.out.println("Cerrando conexión...");
        socket.close();
    }
}

```

### ACTIVIDAD 3.6

Crea un programa cliente usando sockets UDP que envíe el texto escrito desde la entrada estándar al servidor. El servidor le devolverá la cadena en mayúsculas. El proceso de entrada de datos finalizará cuando el cliente introduzca un asterisco. Crea un programa servidor que reciba cadenas de caracteres, las muestre en pantalla y se las envíe al emisor en mayúscula. El proceso servidor finalizará cuando reciba un asterisco.

Establece un tiempo de espera de 5000 milisegundos con el método `setSoTimeout(4000)` para hacer que el método `receive()` del programa cliente se bloquee. Pasado ese tiempo controlar 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 el programa cliente sin ejecutar el servidor. Puedes ejecutar varios programas cliente a la vez.

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

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

MÉTODO	MISIÓN
<code>void joinGroup(InetAddress mcastaddr)</code>	Permite al socket multicast unirse al grupo de multicast.
<code>void leaveGroup(InetAddress mcastaddr)</code>	El socket multicast abandona el grupo de multicast.
<code>void send(DatagramPacket p)</code>	Envía el datagrama a todos los miembros del grupo multicast.
<code>void 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 a grupo!!";

//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** asociado al mismo puerto que el servidor 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);

//Abandona el 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 MCservidor {
    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 MCcliente {
    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="";
        //
        while(!msg.trim().equals("*")) {
            byte[] buf = new byte[1000];
            //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 en distintas consolas, véase Figura 3.10.

```
mj@ubuntu:~$ javac MCservidor.java
mj@ubuntu:~$ java MCservidor
Datos a enviar al grupo: uno
Datos a enviar al grupo: dos
Datos a enviar al grupo: tres
Datos a enviar al grupo: cuatro
Datos a enviar al grupo: cinco
Datos a enviar al grupo: seis
Datos a enviar al grupo: siete
Datos a enviar al grupo: *
Socket cerrado...
mj@ubuntu:~$ 

mj@ubuntu:~$ java MCcliente
Recibo: uno
Recibo: dos
Recibo: tres
Recibo: cuatro
Recibo: cinco
Recibo: seis
Recibo: siete
Recibo: *
Socket cerrado...
mj@ubuntu:~$ 

mj@ubuntu:~$ java MCcliente
Recibo: tres
Recibo: cuatro
Recibo: cinco
Recibo: seis
Recibo: siete
Recibo: *
Socket cerrado...
mj@ubuntu:~$ 
```

Figura 3.10. Servidor multicast en Linux enviando mensajes a clientes.

Realiza el Ejercicio 2.

## 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 como 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 la 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 TCPObjetoServidor1 {
    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 TCPObjetoCliente1 {
    public static void main(String[] args) 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();
        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 3.11.

```

Administrator: Símbolo del sistema - D:\CAPIT3>javac Persona.java
Administrator: Símbolo del sistema - D:\CAPIT3>javac TCPObjetoCliente1.java
Administrator: Símbolo del sistema - D:\CAPIT3>javac TCPObjetoServidor1.java
Administrator: Símbolo del sistema - D:\CAPIT3>java TCPObjetoServidor1
Esperando al cliente.....
Envío: Juan*20
Recibo: Juan Ramos*22
D:\CAPIT3>

Administrator: Símbolo del sistema - D:\CAPIT3>java TCPObjetoCliente1
PROGRAMA CLIENTE INICIADO.....
Recibo: Juan*20
Envio: Juan Ramos*22
D:\CAPIT3>

```

Figura 3.11. Servidor y cliente TCP intercambiando objetos.

Cuando usamos un bucle para enviar objetos se recomienda utilizar el método **reset()** antes de enviar el objeto por el stream, de esta manera se ignorará el estado de cualquier objeto ya escrito en el stream. Ejemplo:

```
outObjeto.reset();
outObjeto.writeObject(per);
```

### ACTIVIDAD 3.7

Crea una clase Java llamada *Numeros* que defina 3 atributos, uno de ellos entero, y los otros dos de tipo *long*:

```
int numero;
long cuadrado;
long cubo;
```

Define un constructor con parámetros y otro sin parámetros. Define los métodos *get* y *set* de los atributos. Crea un programa cliente que introduzca por teclado un número e inicialice un objeto *Numeros*, el atributo *numero* debe contener el número introducido por teclado. Debe enviar ese objeto al programa servidor. El proceso se repetirá mientras el número introducido por teclado sea mayor que 0.

Crea un programa servidor que reciba un objeto *Numeros*. Debe calcular el cuadrado y el cubo del atributo *numero* y debe enviar el objeto al cliente con los cálculos realizados, el cuadrado y el cubo en sus atributos respectivos. El cliente recibirá el objeto y visualizará el cuadrado y el cubo del número introducido por teclado. El programa servidor finalizará cuando el número recibido en el objeto *Numeros* sea menor o igual que 0.

Controlar posibles errores, por ejemplo si ejecutamos el cliente y el servidor no está iniciado, o si estando el servidor ejecutándose ocurre algún error en el cliente, o este finaliza inesperadamente, etc.

### 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 3.8

Usando sockets UDP. 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.

Realiza la Actividad 3.7 con sockets UDP.

Realiza el Ejercicio 3.

## 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:

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

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 CON EL CLIENTE
        fsalida = new PrintWriter(socket.getOutputStream(), true);
        fentrada = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
    }

    public void run() { //tarea a realizar con el cliente
        String cadena = "";
        System.out.println("COMUNICO CON: " + socket.toString());

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

    } //fin run
} //..

```

Como programa cliente podemos ejecutar el programa *Cliente.java* que 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:

```

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

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

        //SE CREAN LOS FLUJOS DE ENTRADA Y SALIDA
        PrintWriter fsalida = new PrintWriter
            (Cliente.getOutputStream(), true);
        BufferedReader fentrada = new BufferedReader

```

```

        (new InputStreamReader(Cliente.getInputStream())));
    }

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

    String cadena, eco = "";
    do{
        System.out.print("Introduce cadena: ");
        cadena = in.readLine();
        fsalida.println(cadena); //envio cadena al servidor
        eco = fentrada.readLine(); //recibo cadena del servidor
        System.out.println("=>ECO: " + eco);
    } while(!cadena.trim().equals("*"));

    fsalida.close();
    fentrada.close();
    System.out.println("Fin del envío... ");
    in.close();
    Cliente.close();

    }
}
}

```

La Figura 3.12 muestra un momento de la ejecución, primero se ejecuta el programa servidor y a continuación el programa cliente; se puede observar como los 3 clientes conectados están siendo atendidos por el servidor de forma simultánea.

```

Administrator: Símbolo del sistema - java Servidor
D:\CAPIT3>javac Cliente.java
D:\CAPIT3>javac HiloServidor.java
D:\CAPIT3>javac Servidor.java
D:\CAPIT3>java Servidor
Servidor iniciado...
COMUNICO CON: Socket[addr=/127.0.0.1,port=56213,localport=6000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=56238,localport=6000]
COMUNICO CON: Socket[addr=/127.0.0.1,port=56254,localport=6000]
FIN CON: Socket[addr=/127.0.0.1,port=56254,localport=6000]
FIN CON: Socket[addr=/127.0.0.1,port=56238,localport=6000]

Administrator: Símbolo del sistema - java Cliente
Introduce cadena: uno
=>ECO: UNO
Introduce cadena: dos
=>ECO: DOS
Introduce cadena:

Administrator: Símbolo del sistema - java Cliente
Introduce cadena: esto
=>ECO: ESTO
Introduce cadena: es
=>ECO: ES
Introduce cadena: una prueba
=>ECO: UNA PRUEBA
Introduce cadena: *
=>ECO: *
Fin del envío...

D:\CAPIT3>

```

Figura 3.12. Servidor atendiendo a múltiples clientes.

### ACTIVIDAD 3.9

Realiza el Ejercicio 5.

### 3.8.1. HILOS COMPARTIENDO OBJETOS

En el siguiente ejemplo se desarrolla un servidor que define un número que los clientes que se conecten tendrán que adivinar. Los clientes tendrán hasta 5 intentos para adivinar el número. Cada cliente se tratará en un hilo y todos compartirán un objeto que contendrá información sobre el estado del juego: el número a adivinar, si el juego ha terminado o no, quien es el jugador ganador o el método que comprueba la jugada, éste deberá definirse como **synchronized**, para que la comprobación de la jugada no interfiera entre los jugadores y se haga de forma unívoca.

El servidor enviará a cada cliente que se conecte un identificador y el objeto compartido. El cliente enviará al servidor números leídos por teclado, el servidor le devolverá información sobre si ha acertado el número o no, si el número es mayor o menor que el que hay que adivinar, si el juego ha finalizado, los intentos que le quedan, etc. Todo el control de juego del jugador se hará en el hilo. El código del **programa servidor** es el siguiente:

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

public class ServidorAdivina {
    public static void main(String args[]) throws IOException {
        ServerSocket servidor = new ServerSocket(6001);
        System.out.println("Servidor iniciado...");

        // Número a adivinar entre 1 y 25
        int numero = (int) (1 + 25 * Math.random());
        System.out.println("NUMERO A ADIVINAR=> " + numero);

        //Todos los hilos comparten el objeto
        ObjetoCompartido objeto = new ObjetoCompartido(numero);
        int id = 0;
        while (true) {
            Socket cliente = new Socket();
            cliente = servidor.accept(); //esperando cliente
            id++; //identificador para el cliente
            HiloServidorAdivina hilo = new
                HiloServidorAdivina(cliente, objeto, id);
            hilo.start();
        }
    }
}///..
```

El código del objeto compartido por todos los clientes se muestra a continuación, a destacar el método **nuevaJugada()** que recibe el identificador del jugador y el número a adivinar, compara el número del jugador con el número a adivinar y devuelve una cadena indicando lo que ha pasado en la comparación. Cuando un jugador adivina el número se cambia el estado del juego mediante los atributos *acabo* y *ganador*, de esta manera sabremos cuando el juego ha finalizado:

```
public class ObjetoCompartido {
    private int numero; // número a adivinar
    private boolean acabo; // true cuando se haya terminado el juego
    private int ganador; // jugador ganador

    public ObjetoCompartido(int numero) {
        this.numero = numero; // NUMERO A ADIVINAR
        this.acabo = false;
```

```

}

public boolean seAcabo() { return acabo; }
public int getGanador() { return ganador; }

public synchronized String nuevaJugada(int jugador, int suNumero) {
    String cad = "";
    if (!seAcabo()) {
        if (suNumero > numero) { // demasiado grande
            cad = "Número demasiado grande";
        }
        if (suNumero < numero) { // demasiado pequeño
            cad = "Número demasiado bajo";
        }
        if (suNumero == numero) { // ha acertado
            cad = "Jugador " + jugador +
                  " gana, adivinó el número: " + numero;
            acabo = true;
            ganador = jugador;
        }
    } else
        cad = "Jugador " + ganador + " adivinó el número: " + numero;

    return cad;
}

// nuevaJugada

} // ObjetoCompartido

```

La comunicación entre el cliente y el servidor se realiza mediante objetos de tipo *Datos*. En este se definen una serie de atributos, constructores y los métodos get y set. El atributo *cadena* se utilizará para intercambiar mensajes entre el servidor y cliente y para que el cliente envíe el número a adivinar al servidor:

```

import java.io.Serializable;

public class Datos implements Serializable {
    String cadena; //cadena que se intercambia con el servidor
    int intentos; //intentos que lleva el jugador, hasta 5
    int identificador; //id del jugador
    boolean gana; //true si el jugador adivina el número
    boolean juega; //true si el jugador juega, false juego finalizado

    public Datos(String cadena, int intentos, int identificador) {
        this.cadena = cadena;
        this.intentos = intentos;
        this.identificador = identificador;
        this.gana = false;
        this.juega = true;
    }
    public Datos() { super(); }

    //Métodos get y set de los atributos
}

```

**En el hilo**, cuando el jugador (o cliente) se conecta, se le envía un objeto de este tipo con el identificador que le corresponde, un mensaje, el número de intentos, inicialmente 0, y el estado

de juego. Habrá un proceso repetitivo en el que se recibe el número del jugador y se comprueba en el método *nuevaJugada()* del objeto compartido. Después de la comprobación se enviará de nuevo al cliente un objeto *Datos* con la información de lo que ha ocurrido y el estado del juego. El proceso repetitivo del cliente finaliza cuando se acaba el juego o cuando el número de intentos de adivinar el número es 5, entonces el servidor cerrará la conexión con el cliente. El código del hilo controlando los errores que se puedan producir es el siguiente:

```

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

public class HiloServidorAdivina extends Thread {
    ObjectInputStream fentrada;
    ObjectOutputStream fsalida;
    Socket socket = null;
    ObjetoCompartido objeto;
    int identificador;
    int intentos = 0;

    public HiloServidorAdivina(Socket s, ObjetoCompartido objeto, int id) {
        this.socket = s;
        this.objeto = objeto;
        this.identificador = id;
        try {
            fsalida = new ObjectOutputStream(socket.getOutputStream());
            fentrada = new ObjectInputStream(socket.getInputStream());
        } catch (IOException e) {
            System.out.println("ERROR DE E/S en HiloServidor");
            e.printStackTrace();
        }
    }
    // Fin Constructor

    public void run() {
        System.out.println("=>Cliente conectado: " + identificador);

        //PREPARAR PRIMER ENVIO AL CLIENTE
        Datos datos = new Datos("Adivina un NÚMERO ENTRE 1 Y 25",
                                 intentos, identificador);
        if (objeto.seAcabo()) {
            datos.setCadena("LO SENTIMOS, EL JUEGO HA TERMINADO, HAN
                            ADIVINADO EL N°");
            datos.setJuega(false); // YA NO TIENE QUE JUGAR
        }
        try {
            fsalida.reset();
            fsalida.writeObject(datos);
        } catch (IOException e1) {
            System.out.println("Error en el Hilo al realizar " +
                               "el primer envio al jugador: " + identificador);
            return;
        }

        while (!objeto.seAcabo() || !(datos.getIntentos() == 5)) {
            int numecli = 0;
            try {
                //RECIBIR DATOS DEL CLIENTE
                Datos d = (Datos) fentrada.readObject();
                numecli = Integer.parseInt(d.getCadena());
            }

```

```

        } catch (IOException e) {
            System.out.println("Error en el Hilo al leer del
                               jugador: " + identificador);
            break;
        } catch (NumberFormatException n) {
            System.out.println("El jugador:" + identificador + "
                               se ha desconectado");
            break;
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            break;
        }
    // JUGAR EL NÚMERO
    String cad = objeto.nuevaJugada(identificador, numecli);
    intentos++;

    datos = new Datos(cad, intentos, identificador);

    if (objeto.seAcabo()) {
        datos.setJuega(false); // no tiene que seguir jugando
        if (identificador == objeto.getGanador())
            datos.setGana(true);
    }

    try {
        //ENVIAR DATOS AL CLIENTE
        fsalida.reset();
        fsalida.writeObject(datos);
    } catch (IOException nl) {
        System.out.println("Error escribiendo en flujo de
                           salida del jugador: " + identificador + " * " +
                           nl.getMessage());
        break;
    } catch (NullPointerException n) {
        System.out.println("El jugador " + identificador +
                           " ha desconectado ");
        break;
    }
}

} // fin while

if (objeto.seAcabo()) {
    System.out.println("EL JUEGO SE HA ACABADO.....");
    System.out.println("\t==>Desconecta: " + identificador);
}
try {
    fsalida.close();
    fentrada.close();
    socket.close();
} catch (IOException e) {
    System.out.println("Error en Hilo al cerrar cliente: " +
                       identificador);
    e.printStackTrace();
}

}// fin método run

```

```
// Fin HiloServidorAdivina..
```

En el ejemplo se han considerado los posibles errores de comunicación entre cliente y servidor, visualizando en cada mensaje el identificador del cliente que produce la situación de error. En el **programa cliente** se introducen números por teclado y se envían al servidor. Se recibe del servidor el estado de juego y se muestra en pantalla lo que ha ocurrido. El proceso de entrada finalizará cuando se lea un \*. El código es el siguiente:

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

public class JugadorAdivina {
    public static void main(String[] args) throws IOException,
        ClassNotFoundException {
        String Host = "localhost";
        int Puerto = 6001;// puerto remoto
        Socket Cliente = new Socket(Host, Puerto);

        ObjectOutputStream fsalida = new
            ObjectOutputStream(Cliente.getOutputStream());
        ObjectInputStream fentrada = new
            ObjectInputStream(Cliente.getInputStream());

        // FLUJO PARA ENTRADA ESTANDAR
        Scanner sc = new Scanner(System.in);
        String cadena= "";

        //OBTENER PRIMER OBJETO ENVIADO POR EL SERVIDOR
        Datos datos = (Datos) fentrada.readObject();
        int identificador = datos.getIdentificador();
        System.out.println("Id jugador: " + identificador);
        System.out.println(datos.getCadena());

        if (!datos.isJuega())
            cadena = "*";

        while(datos.isJuega() && !cadena.trim().equals("*")) {
            System.out.print("Intento: "+(datos.getIntentos() +1)+"
                " => Introduce número: ");
            cadena = sc.nextLine();
            Datos d = new Datos();

            if(!validarCadena(cadena)) continue; //comprobar si es nº
            d.setCadena(cadena);

            //ENVIAR DATOS AL SERVIDOR, el número leído por teclado
            fsalida.reset();
            fsalida.writeObject(d);

            //RECIBIR DATOS DEL SERVIDOR
            datos = (Datos) fentrada.readObject();
            System.out.println("\t"+datos.getCadena());

            if (datos.getIntentos()>=5){
                System.out.println("\t<<JUEGO FINALIZADO, NO HAY MÁS
```

```

        INTENTOS>>");

        cadena="*";
    }
    if (datos.isGana()) {
        System.out.println("<<HAS GANADO!! EL JUEGO HA
TERMINADO>>");
        cadena="*";
    } else
    if ( !(datos.isJuega()) ){
        System.out.println("<<EL JUEGO HA TERMINADO, HAN
ADIVINADO EL NUMERO>>");
        cadena="*";
    }
}//fin while
fsalida.close();
fentrada.close();
System.out.println("Fin de proceso... ");
sc.close();
Cliente.close();

}//main

private static boolean validarCadena(String cadena) {
    //comprueba si la cadena es numérica
    boolean valor = false;
    try{
        Integer.parseInt(cadena);
        valor = true;
    } catch (NumberFormatException e){ }
    return valor;
}//validarCadena

}//JugadorAdivina..

```

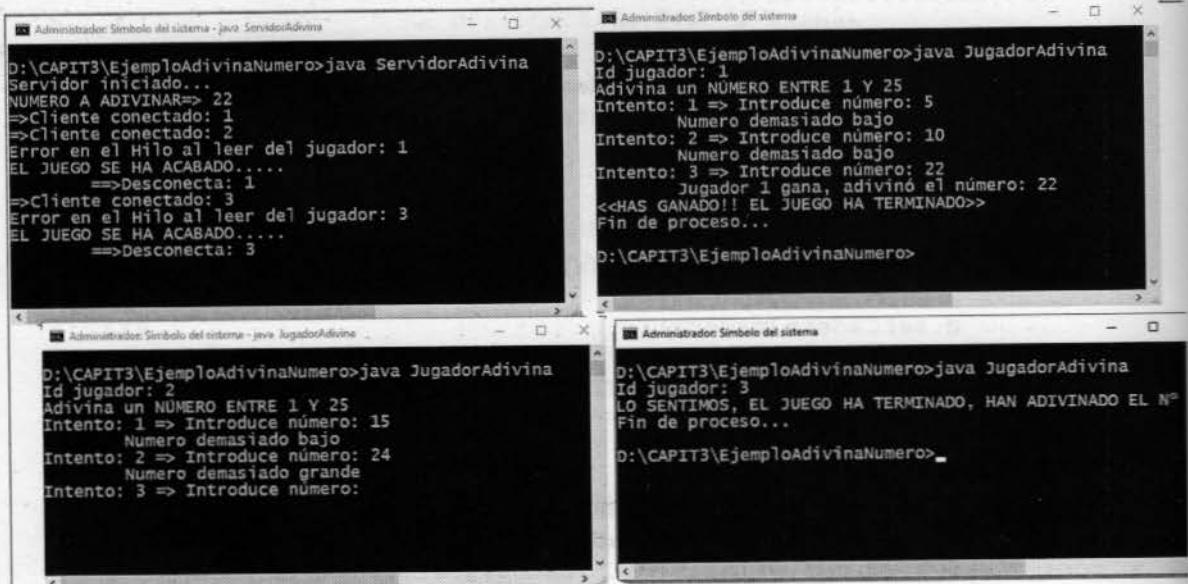


Figura 3.13. Servidor y clientes adivinando un número.

### ACTIVIDAD 3.10

Adapta el ejemplo anterior para que la entrada de datos del cliente se realice usando una pantalla gráfica. El aspecto de la pantalla se muestra en la Figura 3.14. Se debe mostrar el identificador del jugador y el número de intentos que lleva, inicialmente es 0. El botón *Enviar* realiza el envío del número introducido al servidor y el servidor le devolverá el estado del juego, se visualizará en la pantalla el mensaje que indica lo que ha ocurrido en el juego.

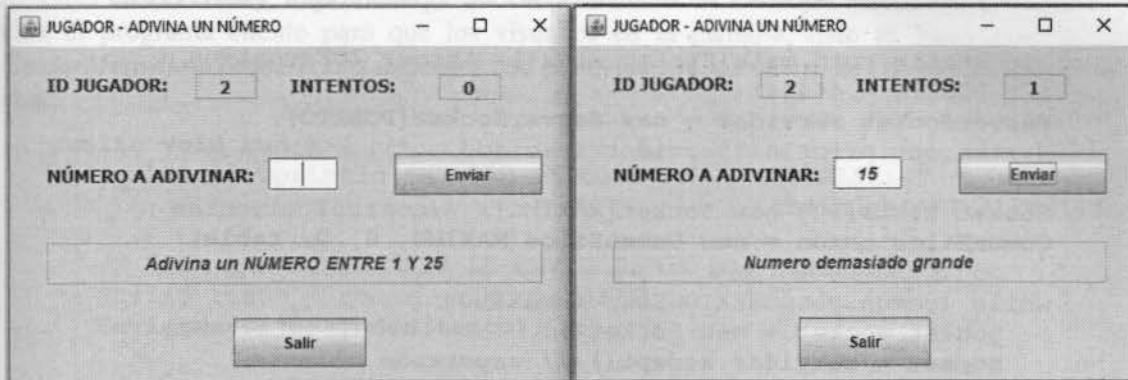


Figura 3.14. Pantalla del cliente Actividad 3.10

#### 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 de miembros del chat.

El programa servidor define el número máximo de conexiones que admite e irá controlando los clientes que actualmente estén conectados, para ello utiliza un objeto de la clase *ComunHilos* que será compartido por todos los hilos. Este objeto contiene los siguientes atributos:

- *int CONEXIONES*: Almacena el número de conexiones de clientes. Cada vez que se conecta un cliente sumamos 1 a este atributo y lo usamos como índice para ir llenando el array de sockets con los clientes que se van conectando. El máximo de conexiones permitidas lo indica el atributo *MAXIMO*.
- *int ACTUALES*: Almacena el número de clientes conectados en este momento. Cada vez que se desconecta un cliente se resta 1 a este atributo.
- *int MAXIMO*: Atributo que indica el número máximo de clientes que se pueden conectar.
- *Socket tabla[] = new Socket[MAXIMO]*: Array que almacena los sockets de los clientes que se conectan. Usaremos el array para tener control de los clientes y así poder enviarles la conversación del chat cada vez que uno envía algún mensaje.
- *String mensajes*: Contiene los mensajes del chat.

El **programa servidor** define una variable con el máximo número de conexiones permitidas, en el ejemplo se definen 10, crea el *ServerSocket*, crea un array para llevar el control de los clientes conectados y crea un objeto de tipo *ComunHilos* donde se inicializan todas las variables comentadas anteriormente. 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. El socket creado se almacena en el array, se incrementa el número de conexiones y las conexiones actuales

y se lanza el hilo para gestionar los mensajes del cliente que se acaba de conectar. El código es el siguiente:

```

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

public class ServidorChat {
    static final int MAXIMO = 10; //MÁXIMO DE CONEXIONES PERMITIDAS

    public static void main(String args[]) throws IOException {
        int PUERTO = 44444;
        ServerSocket servidor = new ServerSocket(PUERTO);
        System.out.println("Servidor iniciado...");

        Socket tabla[] = new Socket[MAXIMO]; //control clientes
        ComunHilos comun = new ComunHilos(MAXIMO, 0, 0, tabla);

        while (comun.getCONEXIONES() < MAXIMO) {
            Socket socket = new Socket();
            socket = servidor.accept(); // esperando cliente

            //OBJETO COMPARTIDO POR LOS HILOS
            comun.addTabla(socket, comun.getCONEXIONES());
            comun.setACTUALES(comun.getACTUALES() + 1);
            comun.setCONEXIONES(comun.getCONEXIONES() + 1);

            HiloServidorChat hilo = new HiloServidorChat(socket, comun);
            hilo.start();
        }
        servidor.close();
    } //main
} //ServidorChat..

```

Al lanzar el hilo se envía al constructor el socket creado y el objeto *ComunHilos* compartido por todos los hilos. Al llegar al máximo de conexiones se cierra el *ServerSocket* y finaliza el proceso servidor, los clientes que estaban conectados seguirán funcionando.

El hilo **HiloServidorChat** se encarga de recibir y enviar los mensajes a los clientes de chat. En el constructor, se recibe el socket creado y el objeto compartido por todos los hilos. 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 HiloServidorChat extends Thread {
    DataInputStream fentrada;
    Socket socket = null;
    ComunHilos comun;

    public HiloServidorChat(Socket s, ComunHilos comun) {
        this.socket = s;
        this.comun = comun;
        try {
            // CREO FLUJO DE ENTRADA PARA LEER LOS MENSAGES
            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 a Todos()`. Los mensajes que se envían son los que se hay en este momento en el chat:

```

public void run() {
    System.out.println("NÚMERO DE CONEXIONES ACTUALES: " +
                       comun.getACTUALES());

    // NADA MAS CONECTARSE LE ENVIO TODOS LOS MENSAJES
    String texto = comun.getMensajes();
    EnviarMensajes a Todos(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();
        if (cadena.trim().equals("*")) { //CLIENTE DESCONECTA
            comun.setACTUALES(comun.getACTUALES() - 1);
            System.out.println("NUMERO DE CONEXIONES ACTUALES: "
                               + comun.getACTUALES());
            break; //sale del bucle
        }
    }
}

```

El texto que el cliente escribe en su chat, se añade al atributo *mensajes* del objeto compartido para poder enviar la conversación a todos los clientes, el método `EnviarMensajes a Todos()` se encargará de ello. Después del bucle while se cierra el socket del cliente:

```

comun.setMensajes(comun.getMensajes() + cadena + "\n");
EnviarMensajes a Todos(comun.getMensajes());
} catch (Exception e) {
    e.printStackTrace();
    break;
}

} // fin while

// se cierra el socket del cliente
try {
    socket.close();
} catch (IOException e) {
    e.printStackTrace();
}

```

```
// run
```

El método **EnviarMensajes a Todos()** envía el texto del atributo *mensajes* del objeto compartido a todos los sockets conectados que no hayan cerrado su conexión con el servidor, para ello se usa el array 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 CHAT A LOS CLIENTES
private void EnviarMensajes a Todos(String texto) {
    int i;
    // recorremos tabla de sockets para enviarles los mensajes
    for (i = 0; i < comun.getCONEXIONES(); i++) {
        Socket s1 = comun.getElementoTabla(i);
        if (!s1.isClosed()) {
            try {
                DataOutputStream fsalida2 = new
                    DataOutputStream(s1.getOutputStream());
                fsalida2.writeUTF(texto);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    } // for
} // EnviarMensajes a Todos
} // ..HiloServidorChat
```

Desde el hilo servidor se muestra en consola los clientes que actualmente hay conectados, por ejemplo, se muestra esta salida cuando hay 3 clientes conectados:

```
Servidor iniciado...
NÚMERO DE CONEXIONES ACTUALES: 1
NÚMERO DE CONEXIONES ACTUALES: 2
NÚMERO DE CONEXIONES ACTUALES: 3
```

La clase **ComunHilos** compartida por todos los hilos tiene los atributos comentados anteriormente y métodos para dar valor y obtener el valor de los atributos. Los métodos definidos como **synchronized** permitirán que dos o más hilos no interfieran en el estado de los atributos:

```
import java.net.Socket;

public class ComunHilos {
    int CONEXIONES; // N° DE CONEXIONES TOTALES, OCUPADAS EN EL ARRAY
    int ACTUALES; // NÚMERO DE CONEXIONES ACTUALES
    int MAXIMO; // MÁXIMO DE CONEXIONES PERMITIDAS
    Socket tabla[] = new Socket[MAXIMO]; // SOCKETS CONECTADOS
    String mensajes; // MENSAJES DEL CHAT

    public ComunHilos(int maximo, int actuales, int conexiones,
                      Socket[] tabla) {
        MAXIMO = maximo;
        ACTUALES = actuales;
        CONEXIONES = conexiones;
        this.tabla = tabla;
```

```

        mensajes="";
    }

public ComunHilos() { super(); }

public int getMAXIMO() { return MAXIMO; }
public void setMAXIMO(int maximo) { MAXIMO = maximo; }

public int getCONEXIONES() { return CONEXIONES; }
public synchronized void setCONEXIONES(int conexiones) {
    CONEXIONES = conexiones;
}

public String getMensajes() { return mensajes; }
public synchronized void setMensajes(String mensajes) {
    this.mensajes = mensajes;
}

public int getACTUALES() { return ACTUALES; }
public synchronized void setACTUALES(int actuales) {
    ACTUALES = actuales;
}
//añadir socket al array de sockets
public synchronized void addTabla(Socket s, int i) {
    tabla[i] = s;
}
public Socket getElementoTabla(int i) { return tabla[i]; }

}//ComunHilos

```

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.

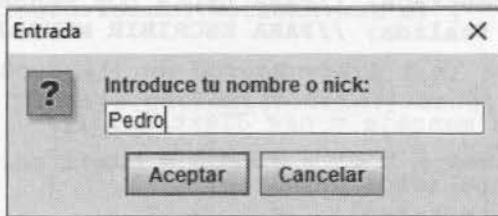


Figura 3.15. Identificación del cliente de chat.

- 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.
- 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 al chat, enviará un \* al servidor para que este sepa que va a finalizar la conexión.

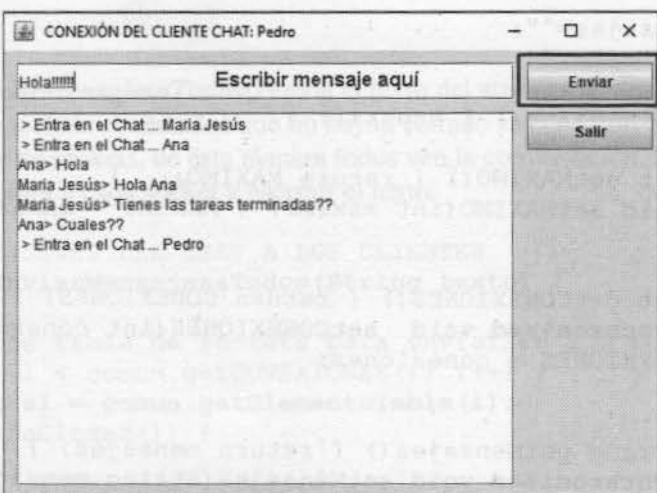


Figura 3.16. Entrada del cliente al chat.

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. La clase extiende **JFrame** que nos permite añadir la interfaz gráfica; e implementa **ActionListener**, para controlar la acción de los botones, y **Runnable** para añadir la funcionalidad de hilo a la pantalla, será necesario añadir el método **run()** con el proceso a realizar por el cliente:

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

public class ClienteChat extends JFrame
    implements ActionListener, Runnable {
    private static final long serialVersionUID = 1L;
    Socket socket = null;

    // streams
    DataInputStream fentrada; //PARA LEER LOS MENSAJES
    DataOutputStream fsalida; //PARA ESCRIBIR MENSAJES

    String nombre;
    static JTextField mensaje = new JTextField();

    private JScrollPane scrollpanel;
    static JTextArea textareal;
    JButton botonEnviar = new JButton("Enviar");
    JButton botonSalir = 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 y se crean los flujos de entrada y salida. A continuación se escribe en el flujo de salida un mensaje indicando que el usuario ha entrado en el chat. Este mensaje lo recibe el hilo (**HiloServidorChat**) y se lo manda a todos los clientes conectados:

```
// 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);

botonEnviar.setBounds(420, 10, 100, 30);
add(botonEnviar);
botonSalir.setBounds(420, 50, 100, 30);
add(botonSalir);

textareal.setEditable(false);
botonEnviar.addActionListener(this);
botonSalir.addActionListener(this);
setDefaultCloseOperation(JFrame.DO NOTHING_ON_CLOSE);

socket = s;
this.nombre = nombre;
try {
    fentrada = new DataInputStream(socket.getInputStream());
    fsalida = new DataOutputStream(socket.getOutputStream());
    String texto = "> Entra en el Chat ... " + nombre;
    fsalida.writeUTF(texto);
} 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, si no se escribe nada en el mensaje, éste no se envía:

```

// accion cuando pulsamos botones
public void actionPerformed(ActionEvent e) {

    if (e.getSource() == botonEnviar) { // SE PULSA ENVIAR
        if (mensaje.getText().trim().length() == 0)
            return;
        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() == botonSalir) { // SE PULSA BOTON SALIR
    String texto = "> Abandona el Chat ... " + nombre;
}

```

```

        try {
            fsalida.writeUTF(texto);
            fsalida.writeUTF("*");
            repetir = false; //para salir del bucle
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
} // FIN acción de los botones

```

Dentro del método *run()*, el cliente lee lo que el hilo servidor 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 run() {
    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
        }
    } // fin while

    try {
        socket.close(); //cerrar socket
        System.exit(0);
    } catch (IOException e) {
        e.printStackTrace();
    }
} // fin run

```

En el método *main()* se pide el nombre de usuario, se realice la conexión al servidor, se crea un objeto **ClienteChat**, se muestra la pantalla y se lanza el hilo cliente:

```

public static void main(String args[]) {
    int puerto = 44444;
    Socket s = null;

    String nombre = JOptionPane.showInputDialog
        ("Introduce tu nombre o nick:");

    if (nombre.trim().length() == 0) {
        System.out.println("El nombre está vacío....");
        return;
    }

    try {
        s = new Socket("localhost", puerto);
    }
}

```

```

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

    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
            "IMPOSIBLE CONECTAR CON EL SERVIDOR\n" +
            e.getMessage(), "<<MENSAJE DE ERROR:>>", 
            JOptionPane.ERROR_MESSAGE);
    }

} // main

}// ..ClienteChat

```

Para ejecutar el servidor de chat se necesita que las clases java **ServidorChat** e **HiloServidorChat** y **ComunHilos** estén en la misma carpeta. El programa cliente **ClienteChat** puede estar en cualquier otra carpeta. Primero se ejecuta el programa servidor, y después se ejecutan los clientes.

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 3.11

Prueba los programas cliente y servidor de chat desde diferentes máquinas. El programa *ServidorChat*, *HiloServidor* y *ComunHilos* 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, *UDPMultiChat* 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 (véase Figura 3.17).

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("")) {
    UDPMultiChat server = new UDPMultiChat (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

```

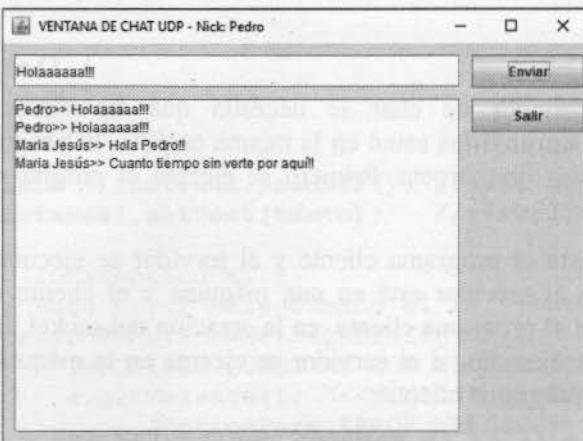


Figura 3.17. Chat UDP.

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 botón enviar
}

```

El botón *Salir* envía 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(); }
    }//fin botón salir

}//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 UDPMultiChat 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 *UDPMultiChat* en las máquinas que quieran participar en el chat.

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 los recursos del capítulo se muestra un ejemplo de servidor que utiliza una base de datos **Db4o** de empleados y departamentos para servir las peticiones de consulta a dicha base de datos de los clientes que se conectan. Los ejemplos se encuentran en la carpeta Carpeta EJEMPLOS\_DB4O.

## COMPRUEBA TU APRENDIZAJE

1º) Usando sockets TCP realiza un programa cliente que introduzca cadenas por teclado hasta introducir un asterisco. Las cadenas se enviarán a un programa servidor. El programa servidor aceptará la conexión de un único cliente y por cada cadena recibida le devolverá al cliente el número de caracteres de la misma. El programa servidor finalizará cuando reciba un asterisco como cadena.

2º) Realiza un servidor multicast usando sockets UDP. El servidor debe mostrar una pantalla inicial como la mostrada en la Figura 3.18.

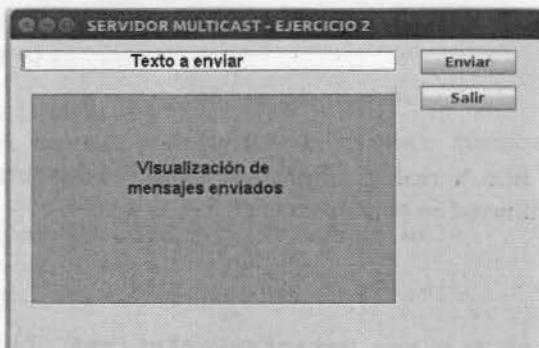


Figura 3.18. Pantalla inicial del servidor multicast.

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 que forman parte del grupo multicast y el botón *Sair* finaliza la ejecución del servidor.

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.

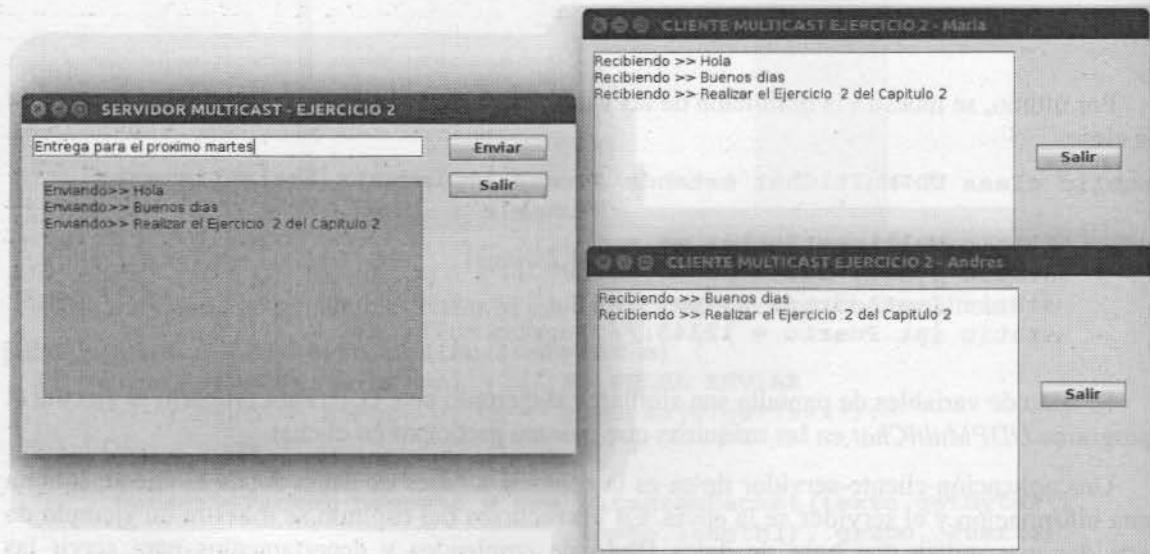


Figura 3.19. Ejecución del Ejercicio 2.

3º) Crea una clase de nombre *Curso*, con los siguientes atributos:

```
String id;
String descripcion;
```

Crea otra clase de nombre *Alumno*, con los siguientes atributos:

```
String idalumno;
String nombre;
Curso curso;
int nota;
```

Crea en las clases anteriores los constructores y métodos get y set necesarios.

Utilizando sockets UDP crea un programa servidor que inicialice un array de 5 objetos de tipo *Alumno*. Invéntate los datos, cada objeto *Alumno* deberá tener un *idalumno* distinto, igualmente

cada curso tiene su id. El servidor se ejecutará en un bucle infinito, recibirá del cliente un *idalumno* y le devolverá el objeto *Alumno* que corresponda con ese identificador. El servidor debe visualizar el identificador solicitado por el cliente.

Crea un programa cliente en el que se introduzca por teclado el *idalumno* que se desea consultar (el programa realizará la lectura en un proceso repetitivo hasta que el *idalumno* leído por teclado sea \*). Se enviará al servidor el *idalumno* a consultar. El servidor le devolverá un objeto *Alumno* con los datos solicitados. Si el alumno no existe, también le devolverá un objeto *Alumno* con datos que indiquen que el alumno no existe. El cliente debe visualizar todos los datos recibidos incluidos el curso del alumno.

**4º) Crea las siguientes clases con los siguientes atributos, los constructores y los métodos get y set necesarios:**

Clase Asignatura:	Clase Especialidad:	Clase Profesor:
int id; String nombreasig;	int id; String nombreespe;	int idprofesor; String nombre; Asignatura [] asignaturas; Especialidad espe;

Supongamos que un profesor puede impartir hasta 3 asignaturas.

Utilizando Sockets TCP crea un programa servidor que inicialice un array de 5 objetos de tipo *Profesor* (no repetir el identificador en ningún objeto). El servidor se ejecutará en un bucle infinito. Cada vez que se conecte un cliente, el servidor le asignará un identificador, este empezará en 1 y se incrementará en 1 según se van aceptando conexiones de clientes. Nada más conectarse el cliente, el servidor le enviará el identificador que le ha correspondido. El servidor recibirá del cliente un *idprofesor* y le devolverá el objeto *Profesor* que corresponda con ese identificador. El servidor debe visualizar el identificador solicitado por el cliente. Este servidor puede atender a múltiples clientes.

Crea un programa cliente que una vez conectado al servidor muestre el identificador que le ha correspondido. El cliente introducirá por teclado el *idprofesor* que desea consultar, se realizará la lectura en un proceso repetitivo hasta que el *idprofesor* leído por teclado sea \*. Se enviará al servidor el *idprofesor* a consultar. El servidor le devolverá un objeto *Profesor* con los datos solicitados. Si el profesor no existe, también le devolverá un objeto *Profesor* con datos que indiquen que el profesor no existe. El cliente debe visualizar todos los datos recibidos incluidos la especialidad y las asignaturas del profesor con sus identificadores.

Controlar todos los posibles errores y cuando un cliente se desconecte mostrar también un mensaje indicándolo. Ejemplo de salida al ejecutar el programa servidor:

```

Servidor iniciado...
Cliente 1 conectado
    Consultando id: 1, solicitado por cliente: 1
Cliente 2 conectado
    Consultando id: 2, solicitado por cliente: 2
Cliente 3 conectado
    Consultando id: 1, solicitado por cliente: 3
EL CLIENTE 2 HA FINALIZADO
FIN CON: Socket[addr=/127.0.0.1, port=50785, localport=6000] DEL CLIENTE:
2

```

Ejemplo de salida al ejecutar el programa cliente:

```
PROGRAMA CLIENTE INICIADO....  
SOY EL CLIENTE: 3  
=====  
Introduce identificador a consultar: 1  
Nombre: María Jesús, Especialidad: 1 - INFORMÁTICA  
Asignatura: 2 - ADAT  
Asignatura: 3 - PSP  
Asignatura: 4 - PMD  
=====  
Introduce identificador a consultar: _
```

5º) Realiza un programa servidor que escuche en el puerto 44444. Cada vez que se conecte un cliente se creará un nuevo hilo para atenderlo. Se mostrará en la consola del servidor la dirección IP y el puerto remoto del cliente que se conecta y cuando el cliente se desconecte se debe mostrar un mensaje indicando que se ha desconectado. Ejemplo de salida en el servidor:

```
Servidor iniciado...  
=>Conecta IP /127.0.0.1, Puerto remoto: 54444  
=>Conecta IP /127.0.0.1, Puerto remoto: 54455  
=>Conecta IP /127.0.0.1, Puerto remoto: 54461  
=>Desconecta IP /127.0.0.1, Puerto remoto: 54455  
=>Desconecta IP /127.0.0.1, Puerto remoto: 54444  
=>Conecta IP /127.0.0.1, Puerto remoto: 54579
```

En el hilo que atiende al cliente se recibe una cadena de caracteres, 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 3.20. 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.

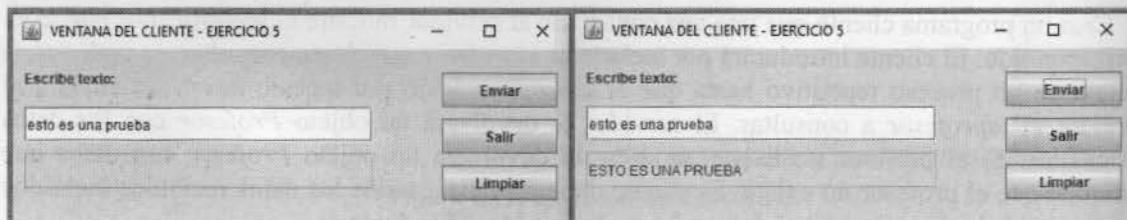


Figura 3.20. Pantalla del cliente Ejercicio 5.

6º) Usando sockets TCP. Se trata de simular en un entorno cliente-servidor un juego donde los clientes buscarán premios en un tablero de 3 filas y 4 columnas. Enviarán la fila y la columna al servidor y el servidor les dirá si hay premio o no en la posición indicada en la fila y la columna. Todos los clientes compartirán el tablero, puede ocurrir que un cliente se conecte y los premios ya se hayan dado, en ese caso el cliente finaliza y no jugará. El servidor se ejecutará siempre (bucle infinito).

#### FUNCIONALIDAD DEL PROGRAMA SERVIDOR:

El servidor debe inicializar un tablero (matriz de 3 x 4) con 4 premios en cuatro posiciones. Por ejemplo, el siguiente tablero muestra premios en las posiciones: [1,1], [2,3], [3,1], [3,4]:

Crucero			
		Entradas	
Masaje			1000€

```
//Colocando los premios en tablero
tableroinitial[0][0] = "Crucero";
tableroinitial[1][2] = "Entradas";
tableroinitial[2][0] = "Masaje";
tableroinitial[2][3] = "1000€";
```

Este tablero se comparte por todos los cliente, si un cliente acierta un premio, el premio ya no estará disponible para nadie más. Al iniciarse el servidor debe mostrar el mensaje:

```
Servidor iniciado...
Posiciones con premio: [1,1], [2,3], [3,1], [3,4]
```

#### Consideraciones:

- ✓ Cada vez que se conecte un cliente se debe mostrar mensaje en la consola del servidor donde aparezca el ID del cliente. Ejemplo:  
Cliente conectado => 1
- ✓ Cuando finalice la conexión con el cliente también se mostrará mensaje. Ejemplo:  
Cliente cerrado => 1  
==>Desconecta IDcliente: 1
- ✓ Nada más conectarse el cliente, el servidor le debe enviar su ID. Este IDentificador empieza en 1 y según se conectan clientes se va incrementando en 1.
- ✓ En ese primer envío el cliente necesita saber si el juego continúa o no, es decir si aún quedan premios en el tablero. Si todos los premios se han dado, el cliente no podrá jugar y se cerrará.
- ✓ Para cada cliente se creará un **Hilo**. En el hilo se controlará el número de jugadas o intentos que lleva el cliente y el número de premios que ha ganado.
- ✓ Cada envío del cliente al servidor es un intento o jugada.
- ✓ El cliente tiene **4 intentos** para conseguir premio.
- ✓ Si un premio ya ha sido concedido a otro cliente no se le contabilizará como premio y sí como intento.
- ✓ El cliente será desconectado si los intentos han llegado a 4 o si los premios se han agotado (antes hay que enviarle la notificación al cliente).

#### FUNCIONALIDAD DEL PROGRAMA CLIENTE:

La pantalla inicial del cliente se muestra en la Figura 3.21. Contiene la siguiente información:

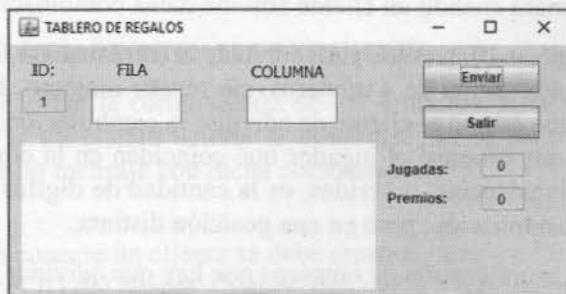


Figura 3.21. Pantalla del cliente Ejercicio 6.

- ✓ El ID del cliente. Este ID se lo manda el servidor nada más conectarse.
- ✓ 2 campos de entrada que son la posición de algún premio en el tablero. Estas posiciones se envían al servidor.
- ✓ Un textarea no editable. Se muestran todos los mensajes.
- ✓ El botón **Enviar** para enviar las posiciones al servidor. Antes de enviar las posiciones al servidor se deben comprobar si son válidas. Son posiciones válidas filas del 1 al 3 y columnas del 1 al 4).
- ✓ El botón **Salir** para finalizar la ejecución.
- ✓ Dos campos contadores no editables que muestran el número de jugadas realizadas y el número de premios conseguidos. Estos contadores se controlan en el Hilo del servidor.

#### Controlar en el cliente:

- ✓ Cuando un cliente se conecta, si no hay más premios se debe mostrar mensaje y finalizar la ejecución.
- ✓ Cuando se envían las posiciones al servidor y el servidor nos dice que no hay más premios, visualizar un mensaje y desactivar botón *Enviar*.

Una vez enviadas las posiciones al servidor, el cliente debe mostrar las jugadas que lleva, los premios conseguidos y en el textarea las posiciones enviadas y si hay o no premio, si las posiciones no son correctas, etc.

#### Ejemplos de ejecución:

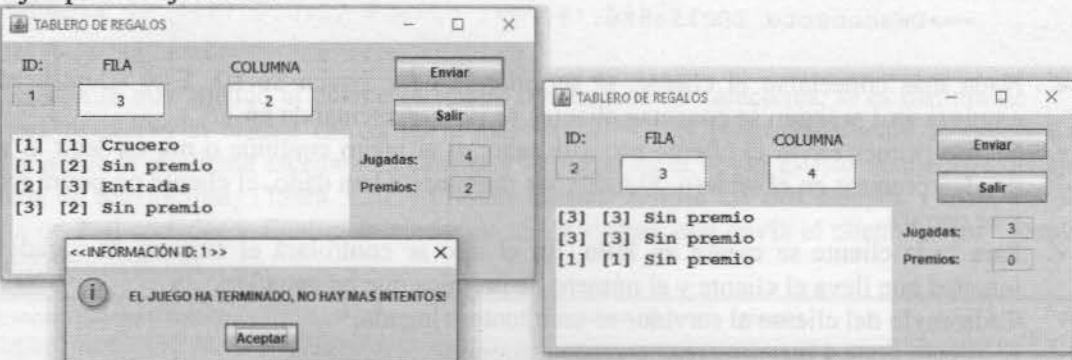


Figura 3.22. Ejecución de clientes, Ejercicio 6.

7º Usando sockets TCP. Se trata de simular en un entorno cliente-servidor el juego del Mastermind. El Mastermind es un juego que consiste en averiguar una combinación numérica de cuatro cifras. El servidor elige la combinación numérica de cuatro cifras, sin repetir ninguna, y estas cifras serán las que tienen que adivinar los clientes en la menor cantidad de intentos posibles. El proceso terminará cuando un cliente adivine dicha combinación.

El cliente (o jugador) tiene 10 intentos para adivinar la combinación, cada intento consiste en enviar al servidor una combinación de 4 números, sin repetir ninguno. El servidor le devolverá pistas acerca de lo cerca que estuvo el cliente de adivinar la combinación: los aciertos, o muertos, es la cantidad de dígitos que propuso el jugador que coinciden en la combinación y están en la misma posición; y las coincidencias, o heridos, es la cantidad de dígitos que propuso el jugador que también están en la combinación, pero en una posición distinta.

Por ejemplo, esta es la combinación de números que hay que adivinar [0, 9, 3, 6]:

- El jugador envía: [1, 2, 3, 4], el servidor le debe responder: Un acierto ya que el 3 está en la misma posición que en la combinación original y cero coincidencias, ya que el 1, 2 y 4 no coinciden con ningún número de la combinación original.
- El jugador envía: [5, 6, 3, 0], el servidor le debe responder: Un acierto ya que el 3 está en la misma posición que en la combinación original y dos coincidencias, ya que el 6 y el 0 coinciden con el 0 y el 6 de la combinación original pero no están en la misma posición.
- El jugador envía: [0, 6, 3, 9], el servidor le debe responder: Dos aciertos ya que el 0 y el 3 están en la misma posición que en la combinación original y dos coincidencias, ya que el 6 y el 9 coinciden con el 6 y el 9 de la combinación original pero no están en la misma posición.
- El jugador envía: [0, 9, 3, 7], el servidor le debe responder: Tres aciertos ya que el 0, el 9 y el 3 están en la misma posición que en la combinación original y cero coincidencias, ya que el 7 no está en la combinación original.
- El jugador envía: [0, 9, 3, 6], el servidor le debe responder: Cuatro aciertos ya que el 0, el 9, el 3 y el 6 están en la misma posición que en la combinación original.

La Figura 3.23 muestra la pantalla del programa cliente con el progreso de aciertos y coincidencias al enviar las distintas combinaciones al servidor.

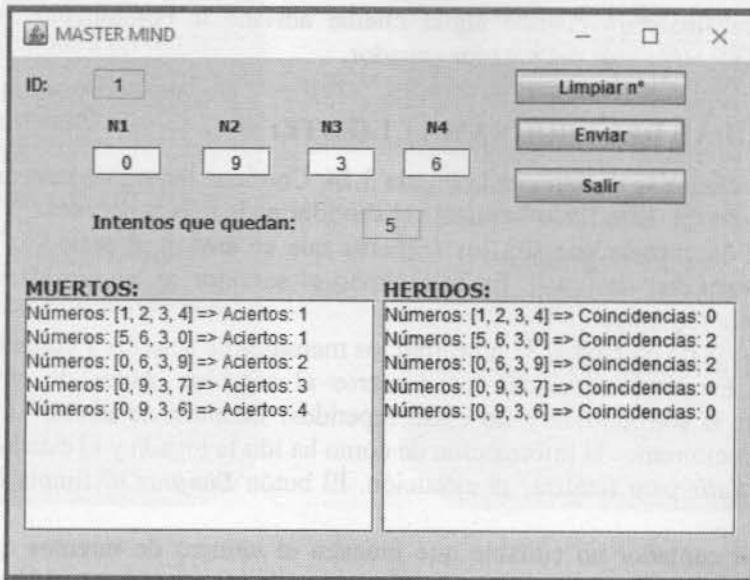


Figura 3.23. Pantalla del cliente Ejercicio 7.

#### RESUMEN DEL CAPÍTULO

### FUNCIONALIDAD DEL PROGRAMA SERVIDOR:

El servidor debe inicializar la combinación de los 4 dígitos. Esta combinación se comparte por todos los clientes, si un cliente acierta la combinación, el juego termina. Al iniciarse el servidor se debe mostrar un mensaje con dicha combinación.

#### Consideraciones:

- ✓ Cada vez que se conecte un cliente se debe mostrar mensaje en la consola del servidor donde aparezca el ID del cliente. Cuando finalice la conexión con el cliente también se mostrará mensaje. Ejemplo:

```

Servidor iniciado...
Combinación de Números: [0, 9, 3, 6]
Jugador 1 conectado
Jugador 2 conectado
Jugador 3 conectado
    ==>Desconectando a ID Jugador: 2
Jugador 4 conectado,
    ==>Desconectando a ID Jugador: 3

```

- ✓ Nada más conectarse el cliente, el servidor le debe enviar su ID. Este Identificador empieza en 1 y según se conectan clientes se va incrementando en 1.
- ✓ En ese primer envío el cliente necesita saber el estado del juego, es decir si el juego continúa o no porque han averiguado o no la combinación de números. Si algún jugador acierta la combinación el juego finalizará y el cliente que se conecta no podrá jugar y se cerrará.
- ✓ Para cada cliente se creará un **Hilo**. En el hilo se controlará el número de jugadas o intentos que lleva el cliente y se comprobará la combinación de números con la que hay que adivinar. Después de las comprobaciones se enviará al cliente la información necesaria de cómo va su juego.
- ✓ Cada envío del cliente al servidor es un intento o jugada. El cliente tiene **10 intentos** para conseguir averiguar la combinación. El cliente será desconectado si los intentos han llegado a 10 o si alguien ha acertado la combinación.
- ✓ El servidor finalizará cuando algún cliente adivine la combinación, además deberá mostrar el identificador del jugador ganador.

#### FUNCIONALIDAD DEL PROGRAMA CLIENTE:

La pantalla del cliente se muestra en la Figura 3.24. Contiene los siguientes campos:

- ✓ El ID del cliente. Este ID se lo manda el servidor nada más conectarse.
- ✓ 4 campos de entrada que son los números que se envían al servidor, los valores que pueden tomar son de 0 a 9. En cada envío al servidor se envían los 4 números y no pueden estar repetidos.
- ✓ Dos textareas no editables. Se muestran los mensajes de aciertos y coincidencias.
- ✓ El botón **Enviar** para enviar los números al servidor. Antes de enviarlos se debe comprobar si son válidos y no están repetidos. Después de enviar la combinación al servidor, recibiremos la información de cómo ha ido la jugada y el estado del juego.
- ✓ El botón **Salir** para finalizar la ejecución. El botón **Limpiar nº** limpia los 4 campos de entrada.
- ✓ Un campo contador no editable que muestra el número de intentos que le quedan al jugador para adivinar la combinación. Este contador se controla en el Hilo del servidor.

#### Controlar en el cliente:

- ✓ Cuando un cliente se conecta, si ya hay algún otro cliente que ha adivinado la combinación se debe mostrar mensaje indicándolo y finalizar la ejecución.
- ✓ Cuando el servidor nos responda que hemos acertado la combinación se mostrará mensaje y se desactivará el botón **Enviar**. Igualmente, cuando nos responda que el juego ha finalizado porque algún jugador acertó la combinación.