# UD2 – PROGRAMACIÓN DE COMUNICACIONES EN RED

## Contenido



## 1. Aspectos teóricos de la comunicación

El **modelo OSI** fue definido por la ISO (International Organization for Standardization) con el fin de promover la creación de estándares que especificaran un conjunto de protocolos independientes de cualquier fabricante. El modelo sirve como marco de referencia para describir y estudiar redes reales. Define **siete capas: física, enlace, red, transporte, sesión, presentación y aplicación** existiendo protocolos para cada una de ellas.

Esta organización jerárquica de niveles permite que cada nivel de un servicio a los niveles superiores ocultando los detalles físicos de la comunicación.

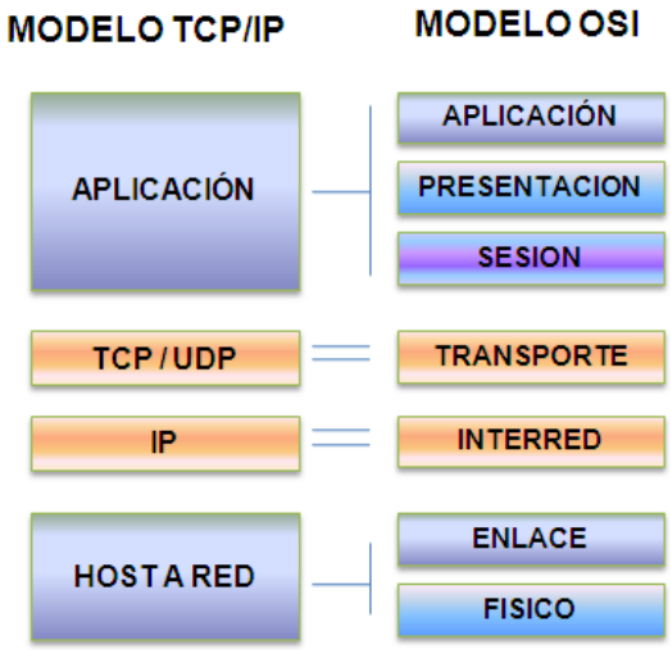
El establecimiento de **protocolos** permite que los distintos dispositivos de los distintos fabricantes puedan comunicarse y formar parte de la misma red al cumplir con las especificaciones definidas en los protocolos.

El modelo OSI es un modelo fundamentalmente teórico y el estándar de facto en las redes está definido por la **familia de protocolos TCP/IP**.

## 1.1 Modelo TCP/IP

TCPI/IP es una familia de protocolos desarrollado para permitir la comunicación entre dispositivos de cualquier red o fabricante, respetando los protocolos de cada red individual. Tiene 4 capas o niveles de abstracción:

- **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.
- **Capa de red**, tiene como propósito seleccionar la mejor ruta para enviar paquetes por la red. El protocolo principal de esta capa es el **Internet Protocol** o **IP**.
- **Capa de transporte**, suministra a las aplicaciones servicio de comunicaciones extremo a extremo utilizando los protocolos **TCP** (Transmision Cotrol Protocol) y **UDP** (User Datagram Protocol)
    - Subapartado
- **Capa de aplicación**, en este nivel se encuentran las aplicaciones disponibles para los usuarios. Algunos protocolos de esta capa son **FTP, SMTP, POP, IMAP, HTTP** etc.




Los equipos conectados a internet se comunican entre sí utilizando el protocolo **TCP** o **UDP**.

Java dispone de clases para establecer conexiones, crear servidores, enviar y recibir datos, estas operaciones junto con el uso de hilos permitirán la manipulación simultánea de múltiples conexiones.

Normalmente no es necesario preocuparse por la capa de transporte cuando programamos en Java, ya que el paquete java.net proporciona una serie de clases que nos abstraerán de las capas inferiores para centrarnos en nuestra aplicación. Sin

embargo, conviene conocer algunas de las características de los protocolos TCP y UDP y sus diferencias para poder decidir cuál es más conveniente para nuestros programas.

- **TCP.** Protocolo orientado a conexión, que garantiza que los datos enviados desde un extremo de la conexión llegan al otro extremo y en el mismo orden que fueron enviados. De lo contrario, se notifica un error.
- **UDP.** No está orientado a conexión. 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. Es un protocolo orientado a rendimiento donde la velocidad de entrega prima sobre la seguridad y consistencia de la información entregada. Por ejemplo, se utilizan para comunicaciones VoIP donde la pérdida de algunos paquetes no va a impedir la comunicación y en cambio el rendimiento es crucial para la misma.

Los protocolos TCP y UDP utilizan 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 esta conexión. Sin embargo, los datos pueden estar destinados a diferentes aplicaciones que se ejecutan en el ordenador, o incluso a distintos subprocesos de la misma aplicación (pensemos en un navegador con varias ventanas abiertas). Mediante el uso de puertos se puede asignar las comunicaciones de un ordenador a las distintas aplicaciones.

Los datos transmitidos a través de internet van acompañados además de la dirección IP (número de 32 bits en IPv4), un número de 16 bits para identificar el puerto (TCP o UDP). De manera que cada proceso conectado a red tendrá registrado un puerto para poder realizar las comunicaciones.

El paquete *java.net* contiene clases e interfaces para la implementación de aplicaciones de red. Incluyendo:

- InetAddress, que representa las direcciones de internet
- URL, que representa un puntero a un recurso en la web.
- URLConnection, para admitir operaciones más complejas en las URLs.
- ServerSocket y Socket, para dar soporte a sockets TCP-
- Las clases DatagramSocket, MulticastSocket y DatagramPacket dan soporte a las comunicaciones vía datagramas UDP.

## 1.2 Arquitectura Cliente – Servidor

La forma más clásica de comunicar dispositivos digitales es aplicando el modelo de cliente-servidor. El servidor es un dispositivo que contiene información a compartir con otros agentes llamados clientes, pero no sabe cuándo los clientes necesitarán su información. Por ello el servidor deberá estar escuchando a la espera de que algún cliente le haga una petición, pidiendo qué parte de la información necesita. La petición es emitida por un cliente y codificada de tal forma que el servidor pueda interpretar. Cuando el mensaje llega al servidor, éste detecta que la petición es para él,

la interpreta, genera la respuesta adecuada y la envía al cliente peticionario. Una vez resuelta la demanda, el servidor quedará de nuevo a la espera de nuevas peticiones.

Actualmente muchas aplicaciones siguen modelos mixtos en que los dispositivos pueden hacer de clientes y servidores a la vez, así permiten una mayor distribución de los datos y los procesos y consiguen un abaratamiento del hardware. Sin embargo, los papeles de servidor y cliente no se han perdido, sino que conviven en un mismo dispositivo, a menudo como procesos independientes ejecutados al mismo tiempo.

No se debe confundir tampoco el rol del servidor con el del receptor, ya que el servidor además de recibir las peticiones de los clientes las debe procesar, obtener una respuesta y enviarla de vuelta al cliente. El cliente por su parte debe detectar qué petición necesita realizar, hacer el envío de la petición, quedarse a la espera de la respuesta y, cuando llegue, procesarla adecuadamente.

# 2. Clases de direccionamiento en Java

## 2.1 La clase InetAddress

La clase *InetAddress* es una abstracción que nos permite gestionar de forma transparente direcciones IP de cualquiera de las dos versiones del protocolo IP. Internamente siempre se trabajará con la clase correspondiente a la versión adecuada *Inet4Address* para IPv4 e *Inet6Address* para IPv6, ambas subclases de *InetAddress*. Esto facilita mucho la codificación ya que nos permite hacer el mismo tratamiento con independencia de la versión IP usada.

Como consecuencia de esta abstracción, nunca deberíamos instanciar directamente un objeto de la jerarquía invocando la sentencia *new*, sino que hay que hacerlo a través de uno de los métodos *static* que la clase *InetAddress* pone a nuestra disposición. Los más comunes son *getByName*, *getAllByName*, *getByAddress*, *getLoopbackAddress* y *getLocalHost*. Todos ellos devuelven una instancia de *InetAddress*, internamente si la IP utilizada es de tipo IPv4 la instancia será de la clase Inet4Address y si es de tipo IPv6 será de la clase Inet6Address.

El método *getByName* necesita un parámetro de tipo *String*. El parámetro podrá tratarse de una URL, de un nombre de red identificador de un dispositivo o de una dirección IP en cualquiera de los formatos aceptados por el protocolo IP. En caso de tratarse de una cadena URL, la clase *InetAddress* hará una petición al servidor DNS por defecto, para lograr resolver el nombre suministrado y obtener una dirección IP equivalente. Si la cadena pasada contiene un nombre de red local, la clase *InetAddress* usará los protocolos estándares que permitan resolver el nombre y obtener la dirección IP asociada. Las direcciones IP obtenidas se convertirán en un vector de bytes del tamaño correspondiente a la versión de la dirección (4 o 16 bytes). También, en caso de que el parámetro pasado contenga directamente una dirección IP, se hará la conversión a vector de bytes. El vector de bytes servirá para instanciar un objeto de tipo *Inet4Address* o *Inet6Address* de acuerdo con el tamaño del vector. La instancia creada se devolverá a continuación como resultado del método. En caso de que el

sistema usado para resolver la URL o el nombre de red obtenga más de una IP se creará la instancia de una de ellas.

El método *getAllByName* recibe también un parámetro que contiene una cadena con las mismas opciones que el método *getByName* y se comportará de forma similar a la hora de resolver el nombre pasado, pero si la resolución contiene varias direcciones IP creará una instancia diferente para cada una de ellas, las agrupará en un vector de objetos *InetAddress* y las devolverá.

El método *getByAddress* espera recibir por parámetro un vector con 4 o 16 bytes. En caso de que el tamaño del vector sea de 4 bytes instanciará un objeto de la clase *Inet4Address*. Sería de la clase *Inet6Address* si el tamaño fuera de 16 bytes.

El método *getLoopbackAddress* obtiene una instancia de *InetAddress* con la dirección específica *127. 0. 0. 1*. De forma similar el método *getLocalHost* obtiene una instancia de la conexión local principal del ordenador en el que la aplicación esté ejecutado.

En caso de que el nombre del parámetro no se pueda resolver o el formato de la dirección IP sea incorrecta se lanzará una excepción del tipo *UnknownHostException*.

Las instancias de *InetAddress*, sean de tipo *Inet4Address* o *Inet6Address* permiten todas obtener información del *host* asociado a la IP. Entre otros el método *getAddress* nos devolverá un vector de bytes con el valor de la IP asociada. De forma similar *getHostAddress* devuelve también la IP, pero en formato textual y el método *getHostName* obtiene el nombre del dispositivo asociado. Si durante la consulta el dispositivo no indicara el nombre, el método devolvería la dirección IP en forma de *String*.

Veamos el siguiente ejemplo:

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

public class TestInetAddress {
	private static final String HEADER
		= "************************************************************";

	public static void main(String[] args) {
		// acceder a localhost
		try {
			InetAddress local = InetAddress.getLocalHost();
			System.out.println(HEADER);
			System.out.println("getLocalHost : " + local);
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		// Mostrar información de distintas máquinas
		// accediendo por su nombre
		InetAddress ip = null;
		try {
			System.out.println(HEADER);
			// obtener la ip a través del nombre
			ip = InetAddress.getByName("localhost");
			test(ip);
			System.out.println(HEADER);
			// obtener la ip a través del nombre
			ip = InetAddress.getByName("www.google.es");
			test(ip);
			System.out.println(HEADER);
			// obtener la ip a través del nombre
			ip = InetAddress.getByName("www.elpais.com");
			test(ip);
			System.out.println(HEADER);
			// obtener la ip a través del nombre
			ip = InetAddress.getByName("www.heraldo.es");
			test(ip);
			System.out.println(HEADER);

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

	private static void test(InetAddress ip) {
		// Mostrar la información que devuelven algunos de los métodos de la clase
		System.out.println("\t getByName: " + ip);
		System.out.println("\t getHostName: " + ip.getHostName());
		System.out.println("\t getHostAddress: " + ip.getHostAddress());
		System.out.println("\t getCanonicalHostName: " + ip.getCanonicalHostName());
		try {
			// obtener todas las IPs asociadas a un nombre
			InetAddress[] ips = InetAddress.getAllByName(ip.getHostName());
			System.out.println("\t direcciones para " + ip.getHostName() + ":");
			for (int i = 0; i < ips.length; i++) {
				System.out.println("\t\t " + ips[i]);
			}
		} catch (UnknownHostException e1) {
			e1.printStackTrace();
		}
	}
}
```

## 2.2 La clase URL

Muchas veces las aplicaciones necesitan conseguir recursos almacenados en algún dispositivo de la red. Los recursos pueden encontrarse, por ejemplo, almacenados en forma de fichero o en una base de datos u obtenerse como resultado de un proceso.

Para identificar un recurso de la web, podemos utilizar una URL (Uniform Resource Locator). Una URL es una cadena de caracteres única para cada recurso diferente, que sigue unas determinadas reglas sintácticas y contiene información de donde se encuentra el recurso, cómo se puede localizar. Al tratarse de una cadena única para cada recurso, decimos que además de localizar el recurso también lo identifica. Por eso las URL se consideran también identificadores de recursos.

En general la sintaxis de una URL sigue el formato siguiente (las partes entre corchetes son opcionales):

```
protocol://[user[:password]@]host[:port][/path][?args]
```

- protocol. Protocolo, habitualmente https o http
- Usuario y password, habitualmente no aparecerán. Puede aparecer solo el usuario o con contraseña, si se incluye la contraseña se separa con dos puntos (:). Si aparece el usuario se separa con @ antes del nombre del host.
- host. Nombre de la máquina donde reside el recurso
- path. Ruta o directorio donde se encuentra el recurso. Si no se incluye se accede al recurso por defecto.
- args. Parámetros que se envían al servidor

La clase URL representa un puntero a un recurso en la WWW. La manera más fácil de crear un objeto de tipo *URL* es a partir de una cadena de texto con la dirección del recurso:

```
URL url = new URL("http://iesch.org/");
```

En este caso hemos creado una URL a partir de una ruta absoluta. También podemos crear un objeto URL a partir de una ruta relativa referida a otra URL.

```
URL url = new URL("http://www.iesch.org/");
URL urlLocation = new URL(url, "index.php/instituto/situacion");
```

También hay un par de constructores que permiten crear la URL a partir de sus partes

```
URL url1 = new URL("http", "docs.oracle.com", "/javase/8");
URL url2 = new URL("https", "docs.oracle.com", 443,
                   "/javase/tutorial/networking");
```

En caso de que la dirección del recurso presente caracteres especiales, podemos hacer uso de la clase URI:

```
URI uri = new URI("https", "www.google.es", "/search",
                   "q=formación profesional", "");
URL url = uri.toURL();
```

Los constructores de la clase URL lanzan la excepción MalformedURLException. Los constructores de la clase URI lanzan la excepción URISyntaxException

La clase URL proporciona varios métodos que le permiten consultar los objetos URL. Podemos obtener el protocolo, la autoridad, el nombre de host, el número de puerto, la ruta, la consulta, el nombre de archivo y la referencia de una URL utilizando estos métodos:

- *getProtocol*. Devuelve el componente identificador de protocolo de la URL.
- *getAuthority*. Devuelve el componente de autoridad de la URL.
- *getHost*. Devuelve el componente de nombre de host de la URL.
- *getPort.* Devuelve un entero que es el número de puerto. Si el puerto no está configurado, *getPort* devuelve -1.
- *getPath.* Devuelve el componente de ruta de esta URL.
- *getQuery.* Devuelve el componente de consulta de esta URL.
- *getFile.* Devuelve el componente de nombre de archivo de la URL. El método *getFile* devuelve lo mismo que *getPath*, más la concatenación del valor de *getQuery*, si la hubiera.
- *getRef.* Devuelve el componente de referencia de la URL.

Recordar que no todas las direcciones URL contienen estos componentes. La clase URL proporciona estos métodos porque las URL HTTP contienen estos componentes y son quizás las URL más utilizadas. La clase de URL está centrada en HTTP.

Veamos el siguiente ejemplo:

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

public class SampleURL {

	public static void main(String[] args) {
		URL url;
		try {
			// url absolutas
			url =
		new URL("https://es.wikipedia.org/w/index.php?sort=relevance&search=Java");
			showURL(url);

			url = new URL("https://es.wikipedia.org/wiki/Java");
			showURL(url);

			url = new URL("http://docs.oracle.com");
			showURL(url);

			// url relativas
			URL baseURL = new URL("https://docs.oracle.com");
			url = new URL(baseURL, "/javase/8/docs/api/java/net/URL.html");
			showURL(url);

			baseURL = new URL("http://www.iesch.org");
			showURL(baseURL);

			url = new URL(baseURL, "/index.php/instituto/situacion");
			showURL(url);

			// constructor con distintos parámetros
			url = new URL("http", "docs.oracle.com", "/javase/8");
			showURL(url);

			// uso de caracteres especiales
			URI uri = new URI("https", "www.google.es", "/search"
				, "q=formación profesional", "");
			url = uri.toURL();
			showURL(url);

		} catch (MalformedURLException e) {
			e.printStackTrace();
		} catch (URISyntaxException e) {
			e.printStackTrace();
		}

	}

	// muestra el resultado de distintos métodos de la clase URL
	private static void showURL(URL url) {
		System.out.println("URL 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();
	}
}
```

Después de haber creado con éxito un URL, podemos llamar al método *openStream* para obtener una secuencia desde la cual se puede leer el contenido de la URL. El método *openStream* devuelve un objeto *java.io.InputStream*, por lo que leer desde una URL es tan fácil como leer desde una secuencia de entrada.

El siguiente programa utiliza *openStream* para obtener una secuencia de entrada en la URL httpx://es.wikipedia.org/. Luego abre un *BufferedReader* en la secuencia de entrada, hace lectura de la URL y lo muestra por consola:

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

public class ReadURL {

	public static void main(String[] args) {

		try {
			// creamos el objeto url
			URL url = new URL("https://es.wikipedia.org");
			// abrimos un reader que lee directamente de la url
			BufferedReader in = new BufferedReader(
					new InputStreamReader(url.openStream())
				);
			// recorremos el reader y lo mostramos por consola
			String line;
			while ((line = in.readLine()) != null) {
				System.out.println(line);
			}
			in.close();

		} catch (MalformedURLException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

## 2.3 La clase URLConnection

Después de haber creado un objeto *URL* con éxito, podemos llamar al método *openConnection* para obtener un objeto URLConnection, o una de sus subclases específicas de protocolo, por ejemplo, java.net.HttpURLConnection

Podemos usar este objeto para configurar parámetros y propiedades de solicitud que podamos necesitar antes de conectar a un recurso. La conexión al objeto remoto representado por la URL sólo se inicia cuando llamamos al método *connect*. En ese momento se está inicializando un enlace de comunicación entre el programa Java y la URL a través de la red.

Ya hemos visto como leer directamente de una *URL*, ahora veamos cómo podemos leer a través de un objeto *URLConnection*. La conexión se abre de forma implícita al invocar el método *getInputStream*.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

public class ReadURLConnection {

	public static void main(String[] args) {
		try {
			// creamos el objeto url
			URL url = new URL("https://es.wikipedia.org");
			// creamos una conexión a la url
			URLConnection cn = url.openConnection();

			// abrimos un reader que lee directamente de la url
			BufferedReader in = new BufferedReader(
					new InputStreamReader(cn.getInputStream())
				);
			// recorremos el reader y lo mostramos por consola
			String line;
			while ((line = in.readLine()) != null) {
				System.out.println(line);
			}
			in.close();

		} catch (MalformedURLException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Este programa hace lo mismo que el anterior que leía directamente del objeto URL, sin embargo, leer desde un objeto *URLConnection* puede resultar más conveniente ya que este segundo objeto permite otras accione sobre la *URL* como, por ejemplo, escribir sobre ella.

Muchas páginas HTML contienen formularios que permiten introducir datos y enviarlos al servidor. Entonces el servidor es capaz de recibir estos datos, procesarlos y enviar una respuesta de vuelta al cliente. Muchos de estos formularios utilizan el método HTTP POST para enviar datos al servidor. Por lo tanto, llamamos hacer post (*posting)* sobre una URL a escribir sobre ella.

Para que un programa Java interactúe con un proceso en el lado del servidor debe ser capaz de escribir a una URL. Para hacer esto es necesario seguir los siguientes pasos:

1. Crear un objeto URL
2. Recoger la URLConnection de dicho objeto
3. Configurar la capacidad de salida de la conexión; *setDoOutput(true)*.
4. Abrir la conexión al recurso
5. Obtener un stream de salida del recurso
6. Escribir sobre este stream
7. Cerrar el stream

Para ver un ejemplo utilizaremos la siguiente dirección web:
https://httpbin.org/forms/post

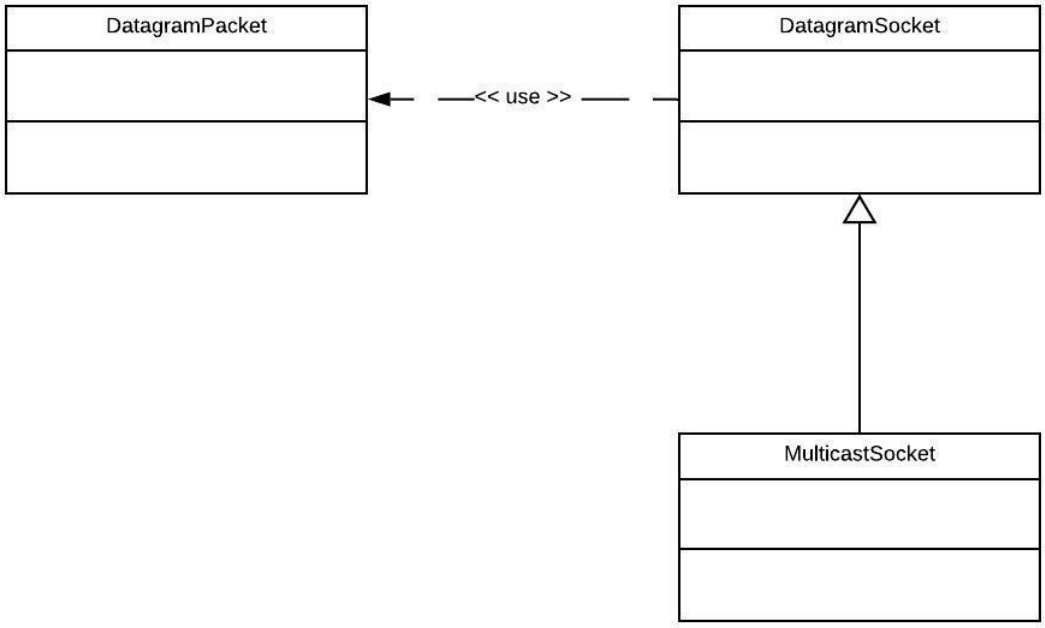
Esta dirección nos permitirá enviar información del formulario (POST), utilizaremos un objeto HttpsURLConnection para acomodarnos al tipo de recurso.

```
import java.io.*;
import java.net.*;
import javax.net.ssl.HttpsURLConnection;

public class WriteURL {

	public static void main(String[] args) {
		try {
			// dirección del recurso
			URL url = new URL("https://httpbin.org/post");
			// creamos una conexión
		 HttpsURLConnection httpClient = (HttpsURLConnection) url.openConnection();

		 // establecemos el método de la petición
		 httpClient.setRequestMethod("POST");

		 //creamos una cadena con los parámetros de la petición
		 String urlParameters = "custname=Juan Carlos"
		 		+ "&custemail=c@c.com"
		 		+ "&custtel=978978978"
		 		+ "&delivery=22:30"
		 		+ "&size=mediana"
		 		+ "&topping=extra de queso&topping=cebolla"
		 		+ "&comments=Traer bien caliente";

		 // enviamos la petición POST
		 httpClient.setDoOutput(true);
		 try (PrintWriter wr = new PrintWriter(httpClient.getOutputStream())) {
		     wr.write(urlParameters);
		     wr.close();
		 }

		 // obtenemos la respuesta del servidor
		 int responseCode = httpClient.getResponseCode();
		 System.out.println("Sending 'POST' request to URL : " + url);
		 System.out.println("Post parameters : " + urlParameters);
		 System.out.println("Response Code : " + responseCode);
		 System.out.println("Response:");
		 // imprimimos la respuesta
		 try (BufferedReader in = new BufferedReader(
		 		new InputStreamReader(httpClient.getInputStream()))) {

		     String line;
		     while ((line = in.readLine()) != null) {
		         System.out.println(line);
		     }

		 }
		 httpClient.disconnect();

		} catch (MalformedURLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

## 3. Sockets UDP

Muchas aplicaciones no van requerir un canal confiable para comunicarse a través de la red. Por el contrario, estas aplicaciones podrían beneficiarse de un modo de comunicación que ofrece paquetes de información independientes cuya llegada y orden de llegada no están garantizados.

El protocolo UDP proporciona un modo de comunicación de red mediante el cual las aplicaciones envían paquetes de datos, llamados datagramas, entre sí. Un datagrama es un mensaje independiente y autónomo enviado a través de la red cuya llegada, hora de llegada y contenido no están garantizados. Las clases *DatagramPacket* y *DatagramSocket* en el paquete java.net implementan comunicación de datagramas independiente del sistema usando UDP. La clase *MulticastSocket* deriva de *DatagramSocket* y apoya específicamente a las comunicaciones de tipo multicast.




Los clientes y servidores que se comunican a través de un canal confiable, como un socket TCP, tienen un canal punto a punto dedicado entre ellos, o al menos la ilusión de uno. Para comunicarse, establecen una conexión, transmiten los datos y luego cierran la conexión. Todos los datos enviados a través del canal se reciben en el mismo orden en que se enviaron. Esto está garantizado por el canal.

En contraste, las aplicaciones que se comunican a través de datagramas envían y reciben paquetes de información completamente independientes. Estos clientes y servidores no tienen ni necesitan un canal dedicado de punto a punto. La entrega de datagramas a sus destinos no está garantizada. Tampoco es el orden de su llegada.

Un ***datagrama*** es un mensaje independiente y autónomo enviado a través de la red cuya llegada, hora de llegada y contenido no están garantizados.

Los sockets creados usando DatagramSocket son capaces de enviar y recibir paquetes especificados por el protocolo UDP. Al crear una instancia podemos especificar un puerto concreto que el socket escuchará cuando sea necesario atender algún servicio asociado al puerto. Para los sockets temporales, en cambio, no será necesario establecer el valor del puerto. Durante la creación de la instancia la clase buscará el primer puerto libre dentro del rango disponible para atender comunicaciones temporales.

- **DatagramSocket**(). Genera una instancia de *DatagramSocket* asociada a un puerto temporal.
- **DatagramSocket**(int port). Genera una instancia de *DatagramSocket* asociada al puerto que se indica en el parámetro.
- **DatagramSocket**(int port, InetAddress). En dispositivos que tengan más de una IP se podrá especificar a través del segundo parámetro la IP que se vinculará la instancia generada.

No es necesario especificar siempre la IP en dispositivos con varias direcciones, la clase seleccionará una por defecto durante la creación de la instancia.

Los objetos de la clase DatagramPacket representan paquetes de datos de acuerdo a la especificación definida por el protocolo UDP. La clase es capaz de añadir de manera predeterminada gran parte de los campos obligatorios del paquete. Sin embargo, hay que indicar los bytes de datos a enviar, la longitud de las mismas y también la dirección y el puerto destino.

- **DatagramPacket**(byte[] buf, int length, InetAddress address, int port). Genera una instancia de *DatagramPacket* que tendrá como datos los bytes contenidos en el vector del primer parámetro en la cantidad que indique el segundo. Obviamente la longitud especificada no podrá superar nunca la cantidad de bytes contenidos en el vector. El segundo y tercer parámetros permitirán definir el destino de los datos.

Los objetos *DatagramSocket* disponen del método *send* para enviar un objeto *DatagramPacket* a la dirección establecida en el propio paquete.

Veamos un ejemplo sencillo, el servidor espera la recepción de un datagrama, muestra la información recibida y cierra. El cliente envía un datagrama al servidor y cierra.

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

public class UDPSvr1 {

	public static void main(String[] args) {
		int port;
		// obtener el puerto del servidor de los parámetros de entrada
		try {
			port = Integer.parseInt(args[0]);
		} catch (Exception e) {
			System.out.println("Missing server info");
			return;
		}

		// creamos un socket UDP para el servidor
		// el socket se cerrará al salir del bloque try
		try (DatagramSocket socket = new DatagramSocket(port);) {

			System.out.println("Waiting on port "
				+ socket.getLocalPort() + "....");
			// array de bytes para recoger los datos del datagrama
			byte[] buffer = new byte[1024];
			// objeto datagrama para la recepción del paquete del cliente
			DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
			// esperamos la recepción de un mensaje del cliente
			socket.receive(packet);
			// tamaño del dato
			int size = packet.getLength();
			// obtenemos el dato recibido
			String msg = new String(packet.getData());
			System.out.println("Bytes received: " + size);
			System.out.println("Info received: " + msg.trim());
			System.out.println("IP:Port orig: "
				+ packet.getAddress() + ":" + packet.getPort());

		} catch (SocketException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

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

public class UDPCli1 {

	public static void main(String[] args) {
		String svrIP;
		int svrPort;

		// obtener el puerto del servidor de los parámetros de entrada
		try {
			svrIP = args[0];
			svrPort = Integer.parseInt(args[1]);
		} catch (Exception e) {
			System.out.println("Missing server info");
			return;
		}

		// creamos un objeto socket
		// como es el cliente no determinamos el puerto
		// y el sistema le dará un puerto libre
		// el socket se cerrará al salir del bloque try
		try (DatagramSocket socket = new DatagramSocket();) {
			// dirección y puertos de destino (servidor)
			InetAddress dest = InetAddress.getByName(svrIP);

			// mensaje, será un array de bytes
			byte[] msg = "Hello server!!".getBytes();
			// creamos el datagrama con el mensaje, la longitud
			// , ip y puerto de destino
			DatagramPacket packet = new DatagramPacket(msg
				, msg.length, dest, svrPort);
			System.out.println("Sending datagram to the server...");
			// enviar el mensaje
			socket.send(packet);
			System.out.println("Closing connection...");
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (SocketException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		System.out.println("Terminated");
	}
}
```

Veamos otro ejemplo, en este caso se hace una implementación para el servidor con una estructura más genérica. La clase implementa la interfaz Runnable de manera que se creará un hilo para la ejecución del servidor separado del hilo principal de la aplicación. El servidor procesa continuamente peticiones y devuelve una respuesta hasta que recibe una orden de parada.

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.SocketTimeoutException;
import java.time.LocalDateTime;
import java.util.Scanner;

public class UDPSvr2 implements Runnable {
	// atributo que indica que el server se ha iniciado
	private static volatile boolean started = false;
	// atributo que indica que hay un petición de parada
	// para el servidor
	private static volatile boolean stop = false;
	// puerto en el que escucha el servidor
	private static int port;

	@Override
	public void run() {

		// creamos un socket UDP para el servidor
		// el socket se cerrará al salir del bloque try
		try (DatagramSocket socket = new DatagramSocket(port);) {

			System.out.println("Server started on port "
				+ socket.getLocalPort() + " at " + LocalDateTime.now());
			// indicamos que el server está iniciado
			started = true;
			// se indica un timeout de 3s para la escucha de peticiones
			socket.setSoTimeout(3000);

			// el servidor atiende peticiones mientras no haya una
			// petición de parada
			while (!stop) {
				// array de bytes para recoger los datos del datagrama
				byte[] buffer = new byte[1024];
				// objeto datagrama para la recepción del paquete del cliente
				DatagramPacket packet
					= new DatagramPacket(buffer, buffer.length);
				// esperamos la recepción de un mensaje del cliente
				// arriba se ha indicado un timeout de 3s
				try {
					socket.receive(packet);
				} catch (SocketTimeoutException e) {
					// tras 3s de espera se lanza la interrupción y
					// se inicia de nuevo el bucle
					// esto permitirá aplicar la condición de parada
					continue;
				}
				// procesar el dato recibido y generar una respuesta
				byte[] response
					= processData(packet.getData(), packet.getLength());
				// obtener origen del mensaje
				InetAddress origAddress = packet.getAddress();
				int origPort = packet.getPort();
				// crear datagrama de respuesta
				DatagramPacket sendPacket = new DatagramPacket(response
					, response.length, origAddress, origPort);
				// enviar mensaje
				socket.send(sendPacket);
			}
		} catch (SocketException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
```

El método *processData* permite procesar los datos recibidos por el servidor y generar una respuesta. Con distintas implementaciones de este método podemos crear diferentes servidores.

```
	// este método permitirá recibir un mensaje
	// procesarlo y generar una respuesta
	private byte[] processData(byte[] data, int length) {
		// proceso diferente para cada aplicación
		// en nuestro caso contar las aes
		int count = 0;
		for (int i = 0; i < length; i++) {
			if (data[i] == (byte) 'a') {
				count++;
			}
		}
		// generar una respuesta
		String response = "Letter a apears " + count + " time(s)";
		// devoler la respuesta en forma de array de bytes
		return response.getBytes();
	}
```

El método principal lanzará el hilo que actuará como servidor y atenderá la petición de parada del usuario.

```
	public static void main(String[] args) {
		// obtener el puerto del servidor de los parámetros de entrada
		try {
			port = Integer.parseInt(args[0]);
		} catch (Exception e) {
			System.out.println("Missing server info");
			return;
		}
		// lanzamos un hilo que implementará el servidor
		Thread t = new Thread(new UDPSvr2());
		t.start();
		// esperar un poco a mostrar el mensaje
		while (!started) {
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		System.out.println("Press ENTER to stop the server...");
		// posibilitar la parada del servidor
		Scanner sc = new Scanner(System.in);
		sc.hasNextLine();
		// Cuando el usuario interactúe con la consola
		// se ordenará la parada del servidor
		stop = true;
		sc.close();
		// esperamos a la finalización del servidor
		try {
			t.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("Execution terminated");
	}
}
```

El cliente hace peticiones en bucle al servidor y muestra la respuesta hasta que el usuario decide salir.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Scanner;

public class UDPCli2 {
	private static final String BYE = ".";
	public static void main(String[] args) {
		String svrIP;
		int svrPort;
		try {// obtener el puerto del servidor de los parámetros de entrada
			svrIP = args[0];
			svrPort = Integer.parseInt(args[1]);
		} catch (Exception e) {
			System.out.println("Missing server info");
			return;
		}
		System.out.println("Server will count number of letter a appearances");
		// creamos un objeto socket, como es el cliente no determinamos el puerto
		// y el sistema le dará un puerto libre
		// el socket se cerrará al salir del bloque try
		try (DatagramSocket socket = new DatagramSocket();
				Scanner sc = new Scanner(System.in);) {
			// dirección y puertos de destino (servidor)
			InetAddress dest = InetAddress.getByName(svrIP);
			String s;
			// entramos en bucle para permitir al usuario
			// realizar varias consultas consecutivas al servidor
			do {
				System.out.println("Enter phrase (\".\" to quit)");
				s = sc.nextLine();

				// mensaje, será un array de bytes
				byte[] msg = s.getBytes();
				// creamos el datagrama con el mensaje, la longitud,
				// ip y puerto de destino
				DatagramPacket packet
					= new DatagramPacket(msg, msg.length, dest, svrPort);
				// enviar el mensaje
				socket.send(packet);
				// recibir respuesta
				byte[] buf = new byte[1024];
				DatagramPacket res = new DatagramPacket(buf, buf.length);
				socket.receive(res);
				// mostrar la respuesta
				System.out.println(new String(res.getData()));
				// habrá una condición de salida que permita al usuario
				// detener el cliente cuando no necesite más consultas
			} while (!s.contentEquals(BYE));
			System.out.println("Closing connection...");
		} catch (UnknownHostException e) {
			e.printStackTrace();
		} catch (SocketException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		System.out.println("Execution terminated");
	}
}
```

## 3.1 Difusión a múltiples clientes

Además de *DatagramSocket*, que permite que los programas se envíen paquetes entre sí, *java.net* incluye una clase llamada MulticastSocket. Este tipo de socket se usa en el lado del cliente para escuchar los paquetes que el servidor transmite a varios clientes.

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 paquete a un grupo multicast, todos los clientes que pertenezcan a ese grupo recibirán el mensaje. La pertenencia al grupo es transparente para el emisor, es decir el emisor no conoce el número de miembros del grupo ni sus direcciones IP.

Un grupo multicast se especifica con una dirección IP de clase D y un número de puerto UDP estándar. Las direcciones de clase D van de 224.0.0.0 a 239.255.255.255. La dirección 224.0.0.0 está reservada y no se debe utilizar.

En el ejemplo el socket se une al grupo 225.1.2.3 y escucha por el puerto 54321

```
MulticastSocket socket = new MulticastSocket(54321);
InetAddress group = InetAddress.getByName("225.1.2.3");
socket.joinGroup(group);

DatagramPacket packet;
for (int i = 0; i < 5; i++) {
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);

    String received = new String(packet.getData());
    System.out.println("Quote of the Moment: " + received);
}

socket.leaveGroup(group);
socket.close();
```

Para enviar datagramas a un grupo necesitaremos especificar la dirección multicast y el puerto en la creación del paquete:

```
InetAddress group = InetAddress.getByName("225.1.2.3");
DatagramPacket packet;
packet = new DatagramPacket(buf, buf.length, group, 54321);
socket.send(packet);
```

Hay que tener en cuenta que para hacer una difusión de un datagrama a una dirección multicast es necesario que la información de dicha dirección multicast esté correctamente configurada en el objeto *DatagramPacket*, el socket que envía este paquete puede ser de tipo *DatagramSocket* o *MulticastSocket*. La recepción de una difusión sólo es posible con un objeto de tipo *MulticastSocket* tras haberse unido (joinGroup) al grupo correspondiente y escuchar en el puerto adecuado.

## 3.2 Envío de objetos en sockets UDP

Para poder enviar objetos a través de un socket es necesario que estos objetos sean serializables.

```
class Foo implements java.io.Serializable{
     …
}
```

Las clases de flujos de datos ObjectOutputStream y ObjectInputStream permiten la escritura de objetos en un stream:

```
ObjectOutputStream out = new ObjectOutputStream(outStream);
out.writeObject(foo);
```

Y la lectura de un objeto proveniente de un stream.

```
ObjectInputStream in = new ObjectInputStream(inStream);
Foo foo = (Foo)in.readObject();
```

Para intercambiar objetos en sockets UDP utilizaremos las clases ByteArrayOutputStream y ByteArrayInputStream que manera que los streams de entrada y salida que utilizaremos serán instancias de estas clases. Veamos la escritura de objetos:

```
ByteArrayOutputStream outStream = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(outStream);
out.writeObject(foo);
out.close();
byte[] bytes = outStream.toByteArray();
DatagramPacket packet
     = new DatagramPacket(bytes, bytes.length, group, port);
```

Y la lectura de objetos

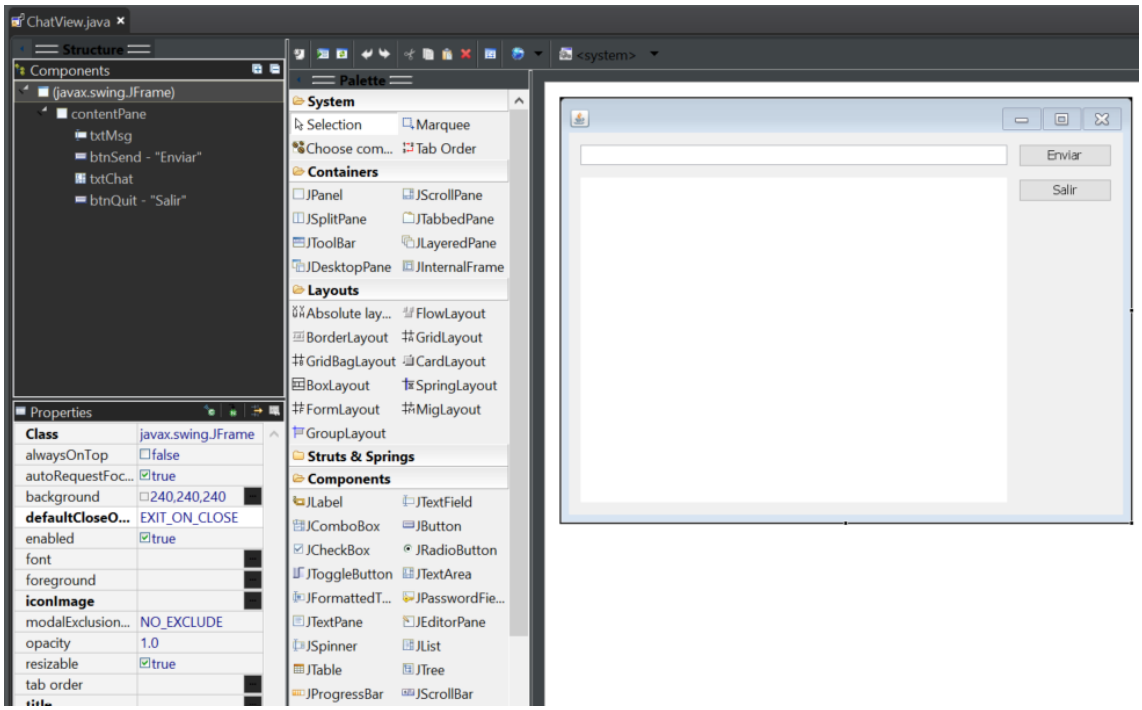
```
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
ByteArrayInputStream inStream = new ByteArrayInputStream(buf);
ObjectInputStream in = new ObjectInputStream(inStream);
Foo foo = (Foo)in.readObject();
in.close();
```

A continuación, veamos un ejemplo simple de un chat implementado con paquetes UDP que utiliza multidifusión y envío de objetos.

La clase ChatMsg corresponde con los mensajes enviados al chat, es serializable:

```
package ud2.chat;

import java.io.Serializable;

public class ChatMsg implements Serializable {
	// tipos de mensajes
	public enum MsgType {
		ENTER, MSG, QUIT
	}

	private String Name; // nombre o nick de usuario
	private MsgType type; // tipo
	private String msg; // contenido del mensaje
	// constructor con todos los atributos

	public ChatMsg(String name, MsgType type, String msg) {
		super();
		Name = name;
		this.type = type;
		this.msg = msg;
	}

	// constructor para entrada y salida del chat
	public ChatMsg(String name, MsgType type) {
		super();
		if (type.equals(MsgType.MSG)) {
			throw new IllegalArgumentException(MsgType.MSG
				+ " type is not valid with empty message");
		}
		Name = name;
		this.type = type;
		this.msg = "";
	}

	// constructor para el tipo mensaje
	public ChatMsg(String name, String msg) {
		super();
		Name = name;
		this.type = MsgType.MSG;
		this.msg = msg;
	}

	public String getName() {
		return Name;
	}

	public MsgType getType() {
		return type;
	}

	public String getMsg() {
		return msg;
	}

}
```

Crearemos una aplicación con interfaz gráfica siguiendo el patrón Modelo-Vista-Controlador. La vista tendrá el siguiente aspecto gráfico:




Para no complicarnos aplicaremos el layout Absolute layout al JFrame. El código es el siguiente. Añadiremos al código generado por WindowBuilder el método addController para gestionar la ventada desde el controlador.

```
import java.awt.EventQueue;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JTextArea;

public class ChatView extends JFrame {
	public final static String SEND = "SEND";
	public final static String QUIT = "QUIT";
	private JPanel contentPane;
	protected JTextField txtMsg;
	private JButton btnSend;
	private JButton btnQuit;
	protected JTextArea txtChat;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					ChatView frame = new ChatView();
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}
	/**
	 * Create the frame.
	 */
	public ChatView() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 739, 550);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);
		txtMsg = new JTextField();
		txtMsg.setBounds(15, 16, 552, 26);
		contentPane.add(txtMsg);
		txtMsg.setColumns(10);
		btnSend = new JButton("Enviar");
		btnSend.setBounds(582, 15, 120, 29);
		contentPane.add(btnSend);
		txtChat = new JTextArea();
		txtChat.setBounds(15, 58, 552, 420);
		contentPane.add(txtChat);
		btnQuit = new JButton("Salir");
		btnQuit.setBounds(582, 60, 120, 29);
		contentPane.add(btnQuit);
	}
	// método para asignar el controlador a los botones
	public void addController(ActionListener controller) {
		btnSend.addActionListener(controller);
		btnSend.setActionCommand(SEND);
		btnQuit.addActionListener(controller);
		btnQuit.setActionCommand(QUIT);
		// el boton Send responderá a la tecla ENTER
		this.getRootPane().setDefaultButton(btnSend);
	}
}
```

La clase Chat será el "modelo" y se encargará de enviar y recibir los mensajes. Añadimos un evento a cada recepción de mensaje para notificar al controlador los mensajes recibidos y mostrarlos en pantalla.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.net.SocketTimeoutException;

import ud2.chat.ChatMsg.MsgType;

public class Chat implements Runnable {
	//acción para el evento
	public final static String RECEIVE = "RECEIVE";

	//atributos
	private String name; //nick o nombre
	private MulticastSocket socket; //socket udp multifisión
	private int port; // puerto UDP
	private InetAddress group; // dirección del grupo de multidifusión
	private ActionListener controller = null; //controlador
	private boolean stop = false; // orden de parada
	private volatile boolean stopped = false; // ejecución de la parada
	private byte[] buf = new byte[1024]; //buffer para los mensajes

	//constructror con la información para el chat
	public Chat(String name, String ip, int port) throws Exception {
		super();
		this.name = name; //nic
		// socket info
		this.group = InetAddress.getByName(ip);
		this.port = port;
		//creación del socket
		socket = new MulticastSocket(port);
		// unir el socket al grupo multidifusión
		socket.joinGroup(group);
		// time out de espera por mensajes a 1s
		socket.setSoTimeout(1000);
	}
```

```
	//crearemos un hilo que escuchará para recibir nuevos mensajes
	@Override
	public void run() {
		//buscamos mensajes nuevos mientras no reciba orden de parada
		while (!stop) {
			try {
				//prepara un datagrama para recibir mensaje
				DatagramPacket packet
					= new DatagramPacket(buf, buf.length);
				socket.receive(packet);
				// una vez recibido el datagrama se obtiene
				// el objeto ChatMsg con el mensaje recibido
				ByteArrayInputStream bs
					= new ByteArrayInputStream(buf);
				ObjectInputStream in = new ObjectInputStream(bs);
				ChatMsg msg = (ChatMsg) in.readObject();
				in.close();
				//preparamos el texto a mostrar por pantalla
				String text;
				if(msg.getType().equals(MsgType.MSG)) {
					text = msg.getName() + " >> "
						+ msg.getMsg();
				} else {
					text = ">> " + msg.getType() + " "
						+ msg.getName();
				}
				//lanzamos un evento para que el controlador
				//se encargue de mostrar el texto por pantalla
				controller.actionPerformed(
					new ActionEvent(
						text
						, ActionEvent.ACTION_PERFORMED
						, RECEIVE
					)
				);
			} catch (SocketTimeoutException timeoutex) {
				// no hacer nada
				// el timeout permite no esperar indefinidamente
				// y responder a la orden de parada
			} catch (IOException e) {
				e.printStackTrace();
			} catch (ClassNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
		//cierre del socket
		socket.close();
		// informar a la clase de que el hilo de escucha ha parado
		stopped = true;
	}
```

```
        //enviar mensaje de entrada al chat
        public void enter() {
                ChatMsg message = new ChatMsg(name, MsgType.ENTER);
                send(message);
        }

        //enviar mensaje de salida al chat
        public void send(String msg) {
                ChatMsg message = new ChatMsg(name, MsgType.MSG, msg);
                send(message);
        }

        //enviar mensaje al chat
        private void send(ChatMsg message) {
                if (!stop) { //validar que no se ha dado orden de parada
                        try {
                //preparar stream para enviar el objeto message serializado
                                ByteArrayOutputStream bs
                                        = new ByteArrayOutputStream();
                                ObjectOutputStream out
                                        = new ObjectOutputStream(bs);
                                out.writeObject(message);
                                out.close();
                                byte[] bytes = bs.toByteArray();
                                //crear datagrama con destino
                                // a la dirección multicast y puerto definidos
                                DatagramPacket packet
                        = new DatagramPacket(bytes, bytes.length, group, port);
                                //enviar el datagrama
                                socket.send(packet);

                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }

        //añadimos el controlador que escuchará
        // los eventos de recepción de mensajes
        public void addController(ActionListener controller) {
                this.controller = controller;
        }

        //parada del servicio
        public void stopService() {
                //enviar mensaje de salida del chat
                ChatMsg message = new ChatMsg(name, MsgType.QUIT);
                send(message);
                //orden de parada
                this.stop = true;
                //esperar a la finalización del hilo de escucha
                while (!stopped) {
                        try {
                                Thread.sleep(200);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

El controlador debe responder a los eventos de la vista y el modelo interactuando con vista y modelo:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

// controlador
public class ChatController implements ActionListener {
	// atributos vista y model
	private ChatView view;
	private Chat model;

	public ChatController(ChatView view, Chat model) {
		super();
		this.view = view;
		this.model = model;
	}

	// escucha los eventos recibidos de la vista
	// y del modelo y responde a estos eventos
	@Override
	public void actionPerformed(ActionEvent arg0) {
		String command = arg0.getActionCommand();

		switch (command) {
		// eventos de la vista
		// botón enviar
		case ChatView.SEND:
			// enviar mensaje al chat
			model.send(view.txtMsg.getText());
			// limpiar la caja de texto
			// del mensaje una vez enviado
			view.txtMsg.setText("");
			break;
		// salir
		case ChatView.QUIT:
			// parar el servicio
			model.stopService();
			// cerrar la vista
			view.dispose();
			break;
		// mensaje recibido del chat
		case Chat.RECEIVE:
			// muestra el texto recibido del chat
			view.txtChat.append((String) arg0.getSource() + "\n");
		default:
			break;
		}
	}
}
```

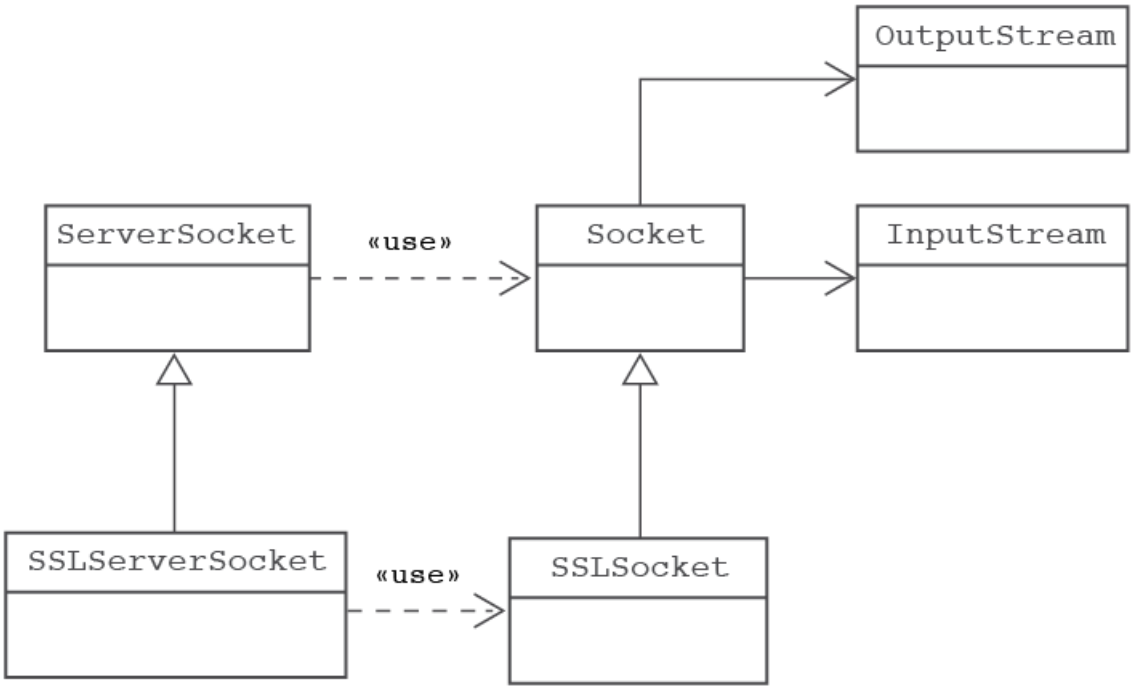
Por último, el programa ChatApp lanzará la vista y el chat

```
import javax.swing.JOptionPane;

//programa que lanza la vista y el chat
public class ChatApp {

	public static void main(String[] args) throws Exception {
		//obtener la configuración del chat (ip y puerto)
		// de la linea de comandos
		String ip;
		int port;
		try {
			ip = args[0];
			port = Integer.parseInt(args[1]);
		} catch (Exception e) {
			System.out.println("Missing server info");
			return;
		}
		//lanza un dialogo para escribir el nick
		// con el que entramos al chat
		String name = JOptionPane.showInputDialog("Enter your nick:");
		if (name.trim().length() == 0) {
			System.out.println("Empty name");
		} else {
			//crear la vista
			ChatView view = new ChatView();
			// crear el modelo con la info para el socket del chat
			Chat model = new Chat(name, ip, port);
			// crear hilo para escuchar los mensajes recibidos
			Thread server = new Thread(model);
			server.start();
			// inicializar controlador
			ChatController controller
				= new ChatController(view, model);
			// añadir controladro a vista y modelo
			model.addController(controller);
			view.addController(controller);
			// mostrar la vista
			view.setVisible(true);
			// enviar mensaje al chat indicando
			// que hemos entrado
			model.enter();
		}
	}
}
```

## 4. Sockets TCP

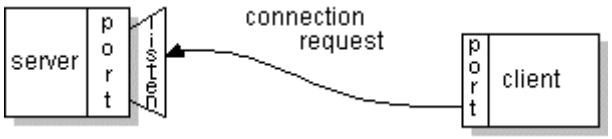
Un socket es un end-point de un enlace de comunicación bidireccional entre dos programas que se ejecutan en la red. Un socket está vinculado a un número de puerto para que la capa TCP pueda identificar la aplicación a la que están destinados los datos para ser enviados.

Las clases *Socket* representan la conexión entre un programa cliente y un programa servidor. El paquete java.net proporciona las clases Socket y ServerSocket que implementan el lado del cliente y el lado del servidor de la conexión respectivamente. La siguiente figura muestra la jerarquía de clases. Las clases *SSLSocket* y *SSLServerSocket* trabajan con los protocolos SSL y TLS permitiendo comunicaciones seguras entre los programas.

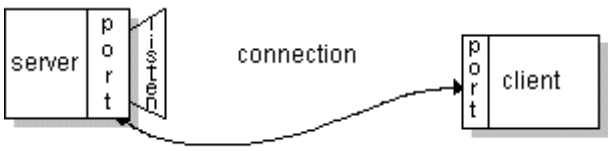



Normalmente, un servidor se ejecuta en una computadora específica y tiene un socket vinculado a un número de puerto específico. El servidor solo espera, escuchando el socket para que un cliente haga una solicitud de conexión.

En el lado del cliente: el cliente conoce el nombre de host de la máquina en la que se ejecuta el servidor y el número de puerto en el que el servidor está escuchando. Para realizar una solicitud de conexión, el cliente intenta encontrarse con el servidor en la máquina y el puerto del servidor. El cliente también necesita identificarse con el servidor para que se una a un número de puerto local que utilizará durante esta conexión. Esto generalmente lo asigna el sistema.




Si todo va bien, el servidor acepta la conexión. Una vez aceptado, el servidor obtiene un nuevo socket vinculado al mismo puerto local y también tiene su punto final (end-point) remoto establecido en la dirección y el puerto del cliente. Necesita un nuevo socket para que pueda continuar escuchando el socket original para solicitudes de conexión mientras atiende las necesidades del cliente conectado.



Un end-point es una combinación de una dirección IP y un número de puerto. Cada conexión TCP se puede identificar de forma exclusiva por sus dos puntos finales. De esa manera, puede tener múltiples conexiones entre su host y el servidor.

La clase Socket oculta los detalles de implementación de manera que los programas Java pueden usar esta clase y comunicarse entre sí manera independiente de la plataforma.

En el siguiente ejemplo el servidor calcula el cubo de un número, el cliente se conectará, pedirá al usuario un número, se lo envía al servidor, el servidor calcula el cubo del número y lo devuelve al cliente. El cliente recoge el resultado, lo muestra y termina.

Es posible hacer parar al servidor enviándole una cadena que sólo contiene un punto "."

Veamos el servidor:

Se crea el socket del servidor que permitirá escuchar las peticiones de los clientes:

```
ServerSocket svr = new ServerSocket(port);
```

Una vez que se recibe una petición de un cliente se crea un socket para la comunicación con este cliente, de esta forma el ServerSocket queda libre para recibir otras peticiones por parte de clientes:

```
Socket cli = svr.accept();
```

Al declarar los objetos de los sockets y los streams dentro de bloques try con sentencias estos objetos se cierran automáticamente al salir del bloque try.

Los objetos *PrintWriter* y *Scanner* gestionan los streams de entrada y salida del socket.

```
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.text.NumberFormat;
import java.time.LocalDateTime;
import java.util.Locale;
import java.util.Scanner;

public class CubeSvr {
	private final static String STOP = ".";
	public static void main(String[] args) {
		try {
			// obtener el puerto de la linea de comando
			Integer port = Integer.parseInt(args[0]);
			// arrancar el server
			try (ServerSocket svr = new ServerSocket(port);) {
				System.out.println("Server running at " + LocalDateTime.now());
				System.out.println("Server port: " + svr.getLocalPort());

				// aceptar peticiones en bucle
				while (true) {
					// abrir socket de cliente
					// y los stream de entrada y salida con el cliente
					// acceptará un cliente a la vez
					// al incluir los objetos en el try se cerrarán al
					// finalizar el bloque
					try (Socket cli = svr.accept();
					     PrintWriter out
							= new PrintWriter(cli.getOutputStream(), true);
					     Scanner in
							= new Scanner(cli.getInputStream());
					) {
					  // leemos el número del canal de entrada
					  if (in.hasNextDouble()) {
						double num = in.nextDouble();
						in.nextLine();
						// calculamos el cubo
						double cube = num * num * num;
						// formateamos el número ya que Scanner
						// trabaja con la configuración regional
						//de la máquina
						NumberFormat nf
					  = NumberFormat.getInstance(new Locale("es", "ES"));
						// escribimos el resultado
						// en el canal de salida
						out.println(nf.format(cube));
					  } else {
						String s = in.nextLine();
						if (s.equals(STOP)) {
							break;
						} else {
							System.out.println("Bad input received."
							+ cli.getInetAddress());
						}

					  }

					} catch (Exception e) {
						e.printStackTrace();
					}
				}
			}
			System.out.println("Server stopped");
		} catch (Exception e) {
			System.out.println("Port information missing");
		}
	}
}
```

Es necesario establecer un protocolo de comunicación entre cliente y servidor, de otra forma la comunicación sería imposible, en este caso el servidor espera a que el cliente le envíe la información; un número para calcular su cubo y enviar el resultado de vuelta. Se ha incluido la posibilidad de enviar una cadena que permite parar el servidor cuando se recibe. El servidor es la aplicación encargada de definir el protocolo de comunicación.

En este caso el cliente es muy sencillo al tener un protocolo de comunicación muy simple; el cliente envía un número al servidor y éste devuelve el valor de elevarlo al cubo.

Para inicializar un socket cliente es necesario pasarle el host y el puerto dónde encontrar al servidor.

```
Socket cli = new Socket(host, port);
```

De igual manera que en el servidor necesitamos gestionar los stream de entrada y salida para la comunicación; en este caso se vuelven a utilizar las clases *PrintWriter* y *Scanner*.

Los conceptos básicos para la creación de aplicaciones cliente son muy similares a los de este programa:

1. Abrir un socket.
2. Abrir los streams de entrada y de salida del socket.
3. Leer y escribir en los streams de acuerdo con el protocolo del servidor.
4. Cerrar los streams.
5. Cerrar el socket.

Solo el paso 3 diferirá de un cliente a otro, dependiendo del servidor. Los otros pasos serán en gran medida los mismos.

```java
import java.io.PrintWriter;
import java.net.Socket;
import java.text.NumberFormat;
import java.util.Locale;
import java.util.Scanner;

public class CubeCli {

	private final static String STOP = ".";

	public static void main(String[] args) {
		try {
			//obtener los datos de la conexión
			// al servidor de la línea de comandos
			String host = args[0];
			int port = Integer.parseInt(args[1]);

			// solicitamos conexión al servidor
			// abrimos los stream de entrada y salida
			try(Socket cli = new Socket(host, port);
			PrintWriter out = new PrintWriter(cli.getOutputStream(), true);
				Scanner in = new Scanner(cli.getInputStream());
				Scanner sc = new Scanner(System.in)
			){
				// mostrar información de la conexión
				System.out.println("Connected to "
					+ cli.getInetAddress() + " : " + cli.getPort());
				System.out.println("Enter number (double)");

				// validar la entrada del usuario
				while(!sc.hasNextDouble()) {
					String s = sc.nextLine();
					// un punto es la señal para que el servidor se pare
					if(s.equals(STOP)) {
						out.println(STOP);
						System.out.println("Server ordered to stop");
						System.out.println("Disconnected");
						return;
					} else {
						// si no pedir un número para calcular el cubo
						System.out.println("Enter number (double)");
					}
				}
				// recoger el número
				double number = sc.nextDouble();
				sc.nextLine();
				// formatear el número para enviarlo al servidor
				// Scanner trabaja con el local de la máquina
				NumberFormat nf = NumberFormat.getInstance(new Locale("es", "ES"));
				// enviar al servidor el número correctamente formateado
				out.println(nf.format(number));
				// recoger la información devuelta por el servidor
				Double cube = in.nextDouble();
				// mostrar el resultado por consola y terminar
				System.out.println("Cube of " + number + " is " + cube);
			} catch (Exception e) {
				e.printStackTrace();
			}
			System.out.println("Disconnected");

		} catch (Exception e) {
			System.out.println("Missing server info");
		}
	}
}
```

Veamos ahora el siguiente ejemplo, el servidor está implementado por la clase EchoServer y no hace otra cosa que hacer eco continuo de los datos enviados por el cliente; EchoClient. El cliente podría funcionar con cualquier servidor que implemente el protocolo Echo Protocol.

Una parte interesante del programa *EchoClient* es el bucle while. El bucle lee una línea a la vez desde la secuencia de entrada estándar y la envía inmediatamente al servidor escribiéndola en el conector conectado al socket *PrintWriter*:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

La última instrucción en el bucle while lee una línea de información desde el *BufferedReader* conectado al socket. El método *readLine* espera hasta que el servidor repita la información. Cuando el servidor devuelve la linea, *EchoClient* imprime la información a la salida estándar.

El ciclo while continúa hasta que el usuario escribe un carácter de fin de entrada ***Ctrl + C.*** Los streams de E/S deben cerrarse antes que el socket, incluir estos objetos en las sentencias del try garantiza el cierre en orden ya que se cierran en orden inverso a su declaración.

## 4.1 Conexión de múltiples clientes

Por simplificar el ejemplo la clase CubeSvr se ha diseñado para gestionar peticiones de una única conexión. Sin embargo, un objeto ServerSocket puede atender peticiones de múltiples clientes. Las peticiones de conexión de los clientes son encoladas en el puerto abierto por el servidor, y éste aceptará las conexiones de forma secuencial. Una vez aceptadas, el servidor puede atender a los diferentes clientes de forma simultánea haciendo uso de treads (hilos), un hilo o thread por cada conexión.

El flujo básico de la lógica en un servidor será

```
while (true) {
    accept a connection;
    create a thread to deal with the client;
}
```

Entonces cada hilo leerá y escribirá en y de la conexión del cliente cuando proceda.

Veamos cómo cambia el ejemplo CubeSvr para atender a múltiples clientes; desarrollamos la clase CubeMultiSvr que atenderá aceptará las múltiples clientes y lanzará un hilo (CubeMultiSvrThread) por cada uno, encargado de procesar las peticiones de cada cliente en concreto.

```
import java.net.ServerSocket;
import java.net.SocketTimeoutException;
import java.time.LocalDateTime;

public class CubeMultiSvr {

	final static String STOP = ".";

	public static void main(String[] args) {

		try {
			boolean[] stop = {false};
			// obtener el puerto de la linea de comando
			Integer port = Integer.parseInt(args[0]);

			// arrancar el server
			try (ServerSocket svr = new ServerSocket(port);) {
				//añadimos timeout para responder orden de parada
				svr.setSoTimeout(1000);
				System.out.println("Server running at " + LocalDateTime.now());
				System.out.println("Server port: " + svr.getLocalPort());

				// aceptar peticiones en bucle mientras
				// no llegue orden de parada
				while (!stop[0]) {
					// crear nuevo thread y
					// asociarle el socket de cliente
					try {
						new Thread(new CubeMultiSvrThread(
								svr.accept(), stop
							)).start();
					} catch (SocketTimeoutException e) {
						// time out
					}
				}
			}
			System.out.println("Server stopped");
		} catch (Exception e) {
			System.out.println("Port information missing");
		}
	}
}
```

La clase que responderá al socket del cliente en un hilo queda así:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.text.NumberFormat;
import java.util.Locale;
import java.util.Scanner;

public class CubeMultiSvrThread implements Runnable {
	// socket que atendará al cliente
	private Socket socket;
	private boolean[] stop;

	public CubeMultiSvrThread(Socket socket, boolean[] stop) {
		super();
		this.socket = socket;
		this.stop = stop;
	}

	@Override
	public void run() {

		try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
				Scanner in = new Scanner(socket.getInputStream());) {
			// leemos el número del canal de entrada
			if (in.hasNextDouble()) {
				double num = in.nextDouble();
				in.nextLine();
				// calculamos el cubo
				double cube = num * num * num;
				// formateamos el número ya que Scanner
				// trabaja con la configuración regional de la máquina
				NumberFormat nf = NumberFormat.getInstance(new Locale("es", "ES"));
				// escribimos el resultado en el canal de salida
				out.println(nf.format(cube));
			} else {
				String s = in.nextLine();
				if (s.equals(CubeMultiSvr.STOP)) {
					this.stop[0] = true;
				} else {
					System.out.println("Bad input received. "
						+ socket.getInetAddress());
				}
			}

		} catch (Exception e) {
			e.printStackTrace();
		}
		// cerrar el socket
		try {
			socket.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

## 4.2 Envío de objetos mediante sockets

De igual manera que para los sockets UDP para enviar objetos a través de objetos TCP es necesario que los objetos sean serializables.

Las clases de flujos de datos ObjectOutputStream y ObjectInputStream permiten la escritura de objetos en el stream de salida del socket:

```
ObjectOutputStream out
     = new ObjectOutputStream(socket.getOutputStream);
out.reset(); // recomendado cuando se envían objetos en bucle
out.writeObject(foo);
```

Y la lectura de un objeto proveniente del stream de entrada del socket.

```
ObjectInputStream in
     = new ObjectInputStream(socket.getInputStream);
Foo foo = (Foo)in.readObject();
```

En el siguiente ejemplo vamos a desarrollar un servicio de calculadora. El servidor aceptaré peticiones de tipo *CalcRequest*, esta clase es inmutable por lo que los objetos no podrán cambiar una vez creados. La clase es serializable, lo que permitirá enviarla a través de los streams de los sockets.

```
import java.io.Serializable;

// representa las peticiones al servidor
// está definida de manera que los objetos serán inmutables
public class CalcRequest implements Serializable {

	public enum RequestType {// enum con los tipos de petición
		ENTER(-1), ADD(1), SUBSTRACT(2), MULTIPLY(3), DIVIDE(4),
POW(5), QUIT(0), SHUTDOWN(-99);

		// cada tipo tiene asociado un valor numérico
		public final int value;

		private RequestType(int value) {
			this.value = value;
		}

		// obtenemos un tipo utilizando el número correspondiente
		public static RequestType getRequestType(int option) {
			for (RequestType type : values()) {
				if (type.value==option) {
					return type;
				}
			}
			return null;
		}
	}
	// Cadena para las opciones disponibles
	public static final String MENU = "\nChoose an option\n"
			+ RequestType.ADD.value + " - Add\n"
			+ RequestType.SUBSTRACT.value + " - Substract\n"
			+ RequestType.MULTIPLY.value + " - Mulitply\n"
			+ RequestType.DIVIDE.value + " - Divide\n"
			+ RequestType.POW.value + " - Pow\n"
			+ RequestType.QUIT.value + " - Quit";

	// version UID para la serialización
	private static final long serialVersionUID = -7452638214958109616L;

	//tipo de petición
	private final RequestType requestType;
	//array de operandos
	private final Double[] operators;

	// constructor con el tipo de peticion y los operandos
	public CalcRequest(RequestType requestType, Double[] operators) {
		super();
		this.requestType = requestType;
		this.operators = operators;
	}
	// acceso al tipo de petición
	public RequestType getRequestType() {
		return requestType;
	}
	// obtencion de los operandos
	public Double[] getOperators() {
		return operators;
	}
}
```

Las respuestas del servidor se representan con la clase CalcResponse, también serializable e inmutable:

```
import java.io.Serializable;

/**
 *
 * CalcResponse. Clase serializable e inmutable que representa las
respuestas
 * del servidor
 *
 */
public class CalcResponse implements Serializable {

	/**
	 * version UID para la serialización
	 */
	private static final long serialVersionUID = -7696847334236454058L;

	// atributos de la respuesta:
	// petición
	private final CalcRequest request;
	// status de la respuesta
	private final boolean ok;
	// valor del cálculo realizado
	private final double result;
	// mensaje de la respuesta
	private final String msg;

	// constructor
	public CalcResponse(CalcRequest request, boolean ok, double result,
String msg) {
		super();
		this.request = request;
		this.ok = ok;
		this.result = result;
		this.msg = msg;
	}

	// metodos de acceso (sólo lectura)
	// a los atributos

	public CalcRequest getRequest() {
		return request;
	}

	public boolean isOk() {
		return ok;
	}

	public double getResult() {
		return result;
	}

	public String getMsg() {
		return msg;
	}
}
```

La aplicación cliente genera peticiones al servidor mientras el usuario no decida salir. La comunicación con el socket del servidor se hacer intercambiando objetos CalcRequest y CalcResponse:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Scanner;

import ud2.CalcRequest.RequestType;
```

```
public class CalcCli {

	// atributo que indica la parada del cliente
	private static boolean stop = false;

	// método principal
	public static void main(String[] args) {

		// información del servidor
		String host;
		int port;

		try {
			// obtener los datos de la conexión
			// al servidor de la línea de comandos
			host = args[0];
			port = Integer.parseInt(args[1]);

		} catch (Exception e) {
			System.out.println("Missing server info");
			return;
		}

		// solicitamos conexión al servidor
		// abrimos los stream de entrada y salida
		try (Socket cli = new Socket(host, port);
				ObjectOutputStream out
					= new ObjectOutputStream(cli.getOutputStream());
				ObjectInputStream in
					= new ObjectInputStream(cli.getInputStream());
				Scanner sc = new Scanner(System.in)) {

			// mostrar información de la conexión
			System.out.println("Connected to " + cli.getInetAddress()
				+ " : " + cli.getPort());
			// petición de entrada
			CalcRequest req = new CalcRequest(RequestType.ENTER, null);
			// envío de la petición
			out.reset();
			out.writeObject(req);
			// obtención de la respuesta
			CalcResponse response = (CalcResponse) in.readObject();
			if (processReponse(response, sc)) {

				// mientras no haya una petición de parada
				// se generará una nueva petición
				// y se procesará la respuesta
				while (!stop) {
					//nueva petición
					req = newRequest(sc);
					// envío
					out.reset();
					out.writeObject(req);
					// rececpción de la respuesta
					response = (CalcResponse) in.readObject();
					// procesamiento de la respuesta
					processReponse(response, sc);
				}
			}

		} catch (UnknownHostException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		System.out.println("Disconnected");

	}
```

```
	// creación de una nueva petición al servidor
	private static CalcRequest newRequest(Scanner sc) {
		//variable para el tipo de petición
		RequestType requestType = null;
		// operandos para la petición
		Double[] operators = null;

		// mostrar el menu de opciones
		System.out.println(CalcRequest.MENU);
		// obtener el tipo de petición desde la
		// entrada por consola
		// aseguramos que el dato introducido
		// es correcto
		while (requestType == null) {
			while (!sc.hasNextInt()) {
				sc.nextInt();
				System.out.println(CalcRequest.MENU);
			}
			requestType = RequestType.getRequestType(sc.nextInt());
			if (requestType.equals(RequestType.ENTER)) {
				requestType = null;
			}

			sc.nextLine();
		}
		// las peticiones de salida y parada del servidor no requieren operandos
		if (!(requestType.equals(RequestType.QUIT)
			|| requestType.equals(RequestType.SHUTDOWN))) {
			// entrada de los operandos por consola
			// como no sabemos cuantos introduce el usuario
			// se usa un ArrayList
			ArrayList<Double> ops = new ArrayList<Double>();
			while (ops.size() == 0) {
				System.out.println("Enter operators separated by space");

				String line = sc.nextLine();
				String[] parts = line.split(" ");
				for (int i = 0; i < parts.length; i++) {
					try {
						ops.add(Double.parseDouble(parts[i]));
					} catch (Exception e) {}
				}
			}
			// se convierte el arraylist en array
			operators = ops.toArray(new Double[0]);
		}
		// retorno de la nueva petición al servidor
		return new CalcRequest(requestType, operators);

	}
```

```
        // procesar la respuesta
        private static boolean processReponse(
                CalcResponse response, Scanner sc) {
                // evaluar si la petición se procesó
                //correctamente en el servidor
                if (response.isOk()) {
                        RequestType prevType = response.getRequest().getRequestType();

                        if (prevType.equals(RequestType.ENTER)) {
                                // la petición de entrada únicamente
                                //muestra el mensaje recibido
                                System.out.println(response.getMsg());
                        } else if (prevType.equals(RequestType.QUIT)
                                || prevType.equals(RequestType.SHUTDOWN)) {
                                // la salida y la parada del servidor
                                // mostrarán el mensaje
                                // y proceden a parar el cliente
                                System.out.println(response.getMsg());
                                stop = true;
                        } else {
                                // las peticiones de cálculo muestran el resultado
                                // y esperan la interacción del usuario
                                System.out.println("Result:" + response.getResult());
                                System.out.println("Press ENTER to continue...");
                                sc.nextLine();
                        }
                } else {
                        // mostrar mensaje si la petición fue mal
                        System.out.println(response.getMsg());
                }
                // devolvemos el resultado de la respuesta
                return response.isOk();
        }
}
```

El servidor tiene un ServerSocket para atender a los clientes y generará un hilo por cada cliente que se conecte, de esta forma el servidor podrá atender a los clientes que se vayan conectado.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.SocketTimeoutException;
import java.time.LocalDateTime;
// aplicación servidor
public class CalcSvr {

	public static void main(String[] args) {
		// atributo para contener la orden de parada
		boolean[] shutdown = { false };
		// puerto por el que escuchará la conexión de los clientes
		int port;
		// obtener el puerto de la línea de comando
		try {
			// obtener el puerto de la linea de comando
			port = Integer.parseInt(args[0]);

		} catch (Exception e) {
			System.out.println("Port information missing");
			return;
		}

		// arrancar el server
		try (ServerSocket svr = new ServerSocket(port);) {
			// añadimos timeout para responder orden de parada
			svr.setSoTimeout(3000);
			System.out.println("Server running at " + LocalDateTime.now());
			System.out.println("Server port: " + svr.getLocalPort());

			// aceptar peticiones en bucle mientras no llegue orden de parada
			while (!shutdown[0]) {
				// crear nuevo thread y
				// asociarle el socket de cliente
				try {
					new Thread(new CalcSvrThread(
							svr.accept(), shutdown)
					).start();
				} catch (SocketTimeoutException e) {
					// time out
				}
			}
		} catch (IOException ioe) {
			ioe.printStackTrace();
		}
		System.out.println("Server shutdown at " + LocalDateTime.now());
	}
}
```

La clase CalcSvrThread es la encargada de atender realmente las peticiones del cliente:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class CalcSvrThread implements Runnable {

	// mensajes
	private static final String SERVER_SHUTDOWN = "Server shutdown";
	private static final String CLOSING_CONNECTION = "Closing connection";
	private static final String CONNECTED = "Connected to Calculator service";

	// socket que atendará al cliente
	private Socket socket;
	// orden de parada del servicio
	private boolean[] shutdown;
	// orde de parada de la conexión
	private boolean stop = false;

	// constructor
	public CalcSvrThread(Socket socket, boolean[] shutdown) {
		super();
		this.socket = socket;
		this.shutdown = shutdown;
	}

	@Override
	public void run() {
		try (ObjectOutputStream out
				= new ObjectOutputStream(socket.getOutputStream());
			ObjectInputStream in
				= new ObjectInputStream(socket.getInputStream());) {

			while (!stop && !shutdown[0]) {
				// leer la petición del cliente
				CalcRequest req = (CalcRequest) in.readObject();
				// procesar petición
				CalcResponse resp = processRequest(req);
				out.reset(); // limpiar stream
				// enviar respuesta
				out.writeObject(resp);
			}

		} catch (Exception e) {
			e.printStackTrace();
		}
		// cerrar el socket
		try {
			socket.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
```

```
	// método de procesamiento de la petición
	private CalcResponse processRequest(CalcRequest req) {
		boolean ok = true;
		double result = Double.NaN;
		String msg = "";
		// selección del tipo de petición
		switch (req.getRequestType()) {
		// únicamente devolver mensaje de conexión
		case ENTER:
			msg = CONNECTED;
			break;
		// cálculo correspondiente al tipo de petición
		case ADD:
			try {
				result = add(req.getOperators());
			} catch (Exception e) {
				ok = false;
				msg = "Error: " + e.getMessage();
			}
			break;
		case SUBSTRACT:
			try {
				result = substract(req.getOperators());
			} catch (Exception e) {
				ok = false;
				msg = "Error: " + e.getMessage();
			}
			break;
		case MULTIPLY:
			try {
				result = multiply(req.getOperators());
			} catch (Exception e) {
				ok = false;
				msg = "Error: " + e.getMessage();
			}
			break;
		case DIVIDE:
			try {
				result = divide(req.getOperators());
			} catch (Exception e) {
				ok = false;
				msg = "Error: " + e.getMessage();
			}
			break;
		case POW:
			try {
				result = pow(req.getOperators());
			} catch (Exception e) {
				ok = false;
				msg = "Error: " + e.getMessage();
			}
			break;
		// parada del hilo y devolución del mensaje
		case QUIT:
			stop = true;
			msg = CLOSING_CONNECTION;
			break;
		// parada del servicio y devolución del mensaje
		case SHUTDOWN:
			shutdown[0] = true;
			msg = SERVER_SHUTDOWN;
			break;
		default:
			break;
		}
		// retorno de la respuesta
		return new CalcResponse(req, ok, result, msg);
	}
```

```
        // Cálculo de la Suma
        private double add(Double[] operators) {
                double result = operators[0];
                for (int i = 1; i < operators.length; i++) {
                        result += operators[i];
                }
                return result;
        }

        // Cálculo de la Resta
        private double substract(Double[] operators) {
                double result = operators[0];
                for (int i = 1; i < operators.length; i++) {
                        result -= operators[i];
                }
                return result;
        }

        // Cálculo de la Multiplicación
        private double multiply(Double[] operators) {
                double result = operators[0];
                for (int i = 1; i < operators.length; i++) {
                        result *= operators[i];
                }
                return result;
        }

        // Cálculo de la División
        private double divide(Double[] operators) {
                double result = operators[0];
                for (int i = 1; i < operators.length; i++) {
                        result /= operators[i];
                }
                return result;
        }

        // Cálculo de la Potencia
        private double pow(Double[] operators) {
                double result = operators[0];
                for (int i = 1; i < operators.length; i++) {
                        result = Math.pow(result, operators[i]);
                }
                return result;
        }
}
```