Comunicación entre aplicaciones.

En Java toda la comunicación vista consiste en dos cosas

- Entrada/salida por consola: con las clases `System.in` o `System.out`.
- Lectura/escritura en ficheros: con las clases File y similares.

Se puede avanzar un paso más utilizando Java para enviar datos a través de Internet a otro programa Java remoto, que es lo que haremos en este capítulo.

Elementos de programación de aplicaciones en red. Librerías.

En Java toda la infraestructura de clases para trabajar con redes está en el paquete `java.net`.

La clase URL

La clase URL permite gestionar accesos a URLs del tipo `http://marca.com/fichero.html` y descargar cosas con bastante sencillez.

Al crear un objeto URL se debe capturar la excepción `MalformedURLException` que sucede cuando hay algún error en la URL, como por ejemplo escribir `htp://marca.com` en lugar de `http://marca.com` (obsérvese que el primero tiene un sola t en http en lugar de dos).

La clase URL nos ofrece un método `openStream` que nos devuelve un flujo básico de bytes. Podemos crear objetos más sofisticados para leer bloques como muestra el programa siguiente:

```
public void descargarArchivoOLD(String urlDescarga, String nombreArchivo) {

    InputStream is = null;
    InputStreamReader reader = null;
    BufferedReader bReader = null;
    FileWriter fWriter = null;
    try {
        System.out.println("Descargando " + urlDescarga);
        URL url = new URL(urlDescarga);
        is = url.openStream();
        reader = new InputStreamReader(is);
        bReader = new BufferedReader(reader);
        fWriter = new FileWriter(nombreArchivo);
        String linea;
        while ((linea = bReader.readLine()) != null) {
            fWriter.write(linea);
        }
        System.out.println("Done..");
    } catch (MalformedURLException e) {
        System.out.println("URL mal escrita! " + e.getMessage());
        e.printStackTrace();
        return;
    } catch (IOException e) {
        System.out.println("Fallo en la lectura del fichero " + e.getMessage());
        e.printStackTrace();
        return;
    } finally {
        close(fWriter);
        close(bReader);
        close(reader);
        close(is);
    }
}
```

A partir de la versión 20 de JAVA se deprecó el uso del constructor URL, por lo que ahora podemos hacer la misma funcionalidad, tal y como vemos en el siguiente ejemplo:

```java
public void descargarArchivo(String urlDescarga, String nombreArchivo) throws IOException {
    HttpURLConnection connection = null;
    try {
        URI uri = URI.create(urlDescarga); // Crear una URI a partir de la URL de descarga
        connection = (HttpURLConnection) uri.toURL().openConnection();

        // Configurar la conexión (puedes añadir más configuraciones si es necesario)
        connection.setRequestMethod("GET");
        connection.setConnectTimeout(5000); // Tiempo de espera para la conexión
        connection.setReadTimeout(5000); // Tiempo de espera para la lectura

        // Leer la respuesta de la conexión
        int status = connection.getResponseCode();
        System.out.println("Código de respuesta: " + status);
        FileWriter fWriter = null;
        if (status == HttpURLConnection.HTTP_OK) {
            try (BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()))) {
                // Si la respuesta es exitosa, leemos el contenido

                String inputLine;
                fWriter = new FileWriter(nombreArchivo);
                while ((inputLine = in.readLine()) != null) {
                    fWriter.write(inputLine);
                }
                System.out.println("Done..");
            } catch (IOException e) {
                e.printStackTrace();
                throw e;
            } finally {
                close(fWriter);
            }
        } else {
            System.out.println("Error: " + status);
        }

    } catch (IOException e) {
        e.printStackTrace();
        throw e;
    } finally {
        connection.disconnect(); // Cerrar la conexión
    }
}
```

Repaso de redes

En redes el protocolo IP es el responsable de dos cuestiones fundamentales:

- Establecer un sistema de direcciones universal (direcciones IP)
- Establecer los mecanismos de enrutado.

Como programadores el segundo no nos interesa, pero el primero será absolutamente fundamental para contactar con programas que estén en una ubicación remota.

Una ubicación remota *siempre* tendrá una **dirección IP** pero *solo a veces tendrá un nombre DNS*. Para nosotros no habrá diferencia ya que si es necesario el sistema operativo traducirá de nombre DNS a IP.

Otro elemento necesario en la comunicación en redes es el uso de un puerto. Un **puerto** es un número que identifica a un proceso o servicio dentro de una máquina. Los números de puerto están entre el 0 y el 65535.

Este rango se divide en tres categorías:

- 0 –1023:“Puertos bien conocidos”, reservados para los servicios especiales como ftp (21), smtp(25), HTTP (80), POP3 (110), etc
- 1024 –49151:“Puertos registrados”, reservados para servicios
- 49152 –65535:“Puertos dinámicos/privados”, no están reservados y se les puede dar cualquier uso

A partir de ahora cuando usemos un número de puerto habrá que comprobar si ese número ya está usado. Antes de usar un puerto en una aplicación comercial deberíamos consultar la lista de «IANA assigned ports».

Por ejemplo, es mala idea que nuestros servidores usen el puerto 80 para aceptar peticiones, probablemente ya esté en uso.

En líneas generales se pueden usar los puertos desde 1024 a 49151, pero deberíamos comprobar que el número que elegimos no sea un número usado por un puerto de alguna aplicación que haya en la empresa.

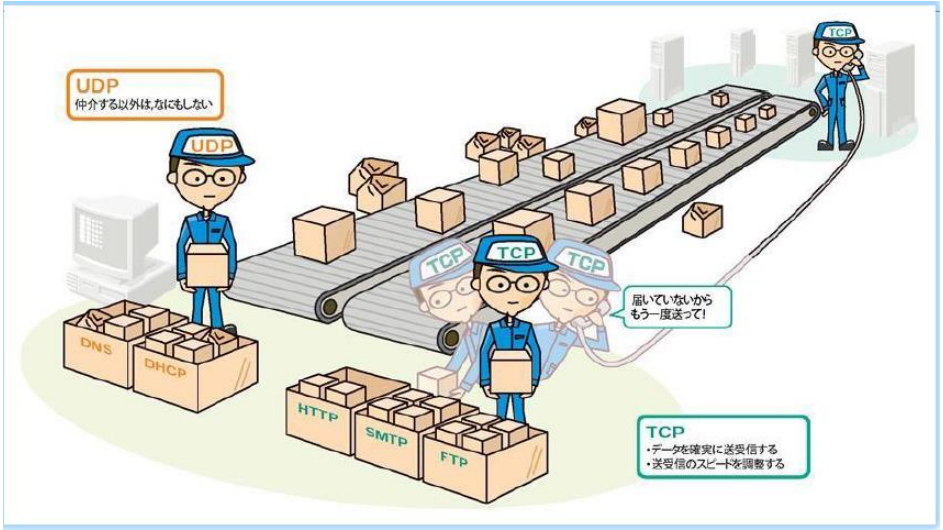
En la pila de protocolos TCP/IP, existen dos protocolos de transporte fundamentales: TCP (*Transmision Control Protocol*) y UDP (*User Datagram Protocol*).

Características de **TCP**:

- Protocolo orientado a conexión: Requiere que se establezca una conexión lógica entre los dos procesos antes de intercambiar datos. La conexión debe mantenerse durante todo el tiempo que dure la comunicación y luego debe liberarse.
- Garantiza que los datos enviados llegarán al destino y lo harán en el mismo orden en el que fueron enviados.

Características de **UDP**:

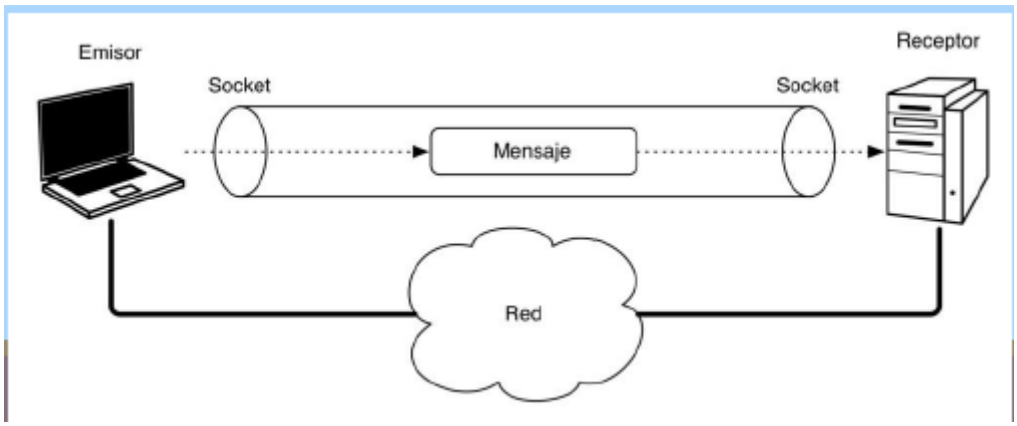
- Se trata de un protocolo no orientado a conexión
- Esto lo hace más rápido, ya que no es necesario establecer conexiones para enviar mensajes
- No garantiza que los mensajes lleguen siempre al destinatario
- No garantiza que los mensajes lleguen en el mismo orden en el que fueron enviados
- Permite enviar mensajes de 64KB como máximo
- En UDP, los mensajes se denominan “datagramas”



Sockets

La clase URL proporciona un mecanismo muy sencillo, pero por desgracia completamente atado al protocolo HTTP.

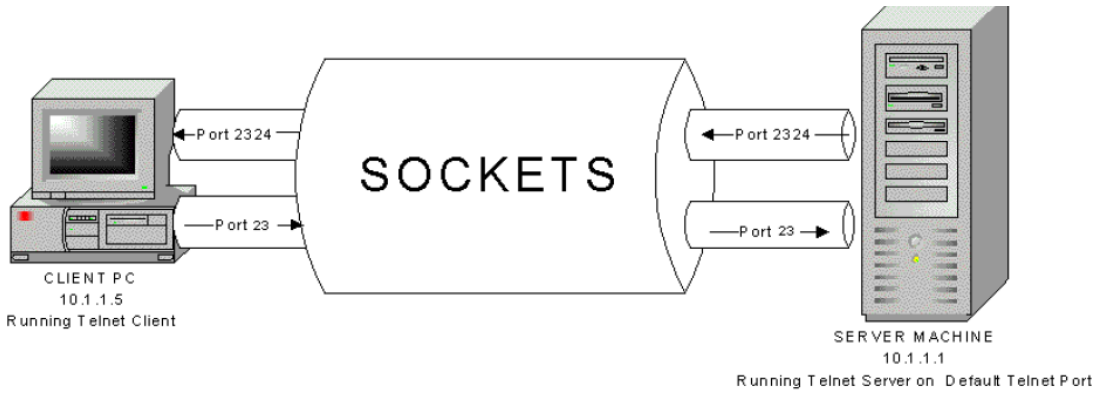
Java ofrece otros objetos que permiten tener un mayor control sobre lo que se envía o recibe a través de la red.



Los sockets son el mecanismo de comunicación básico fundamental que se usa para realizar transferencias de información entre aplicaciones:

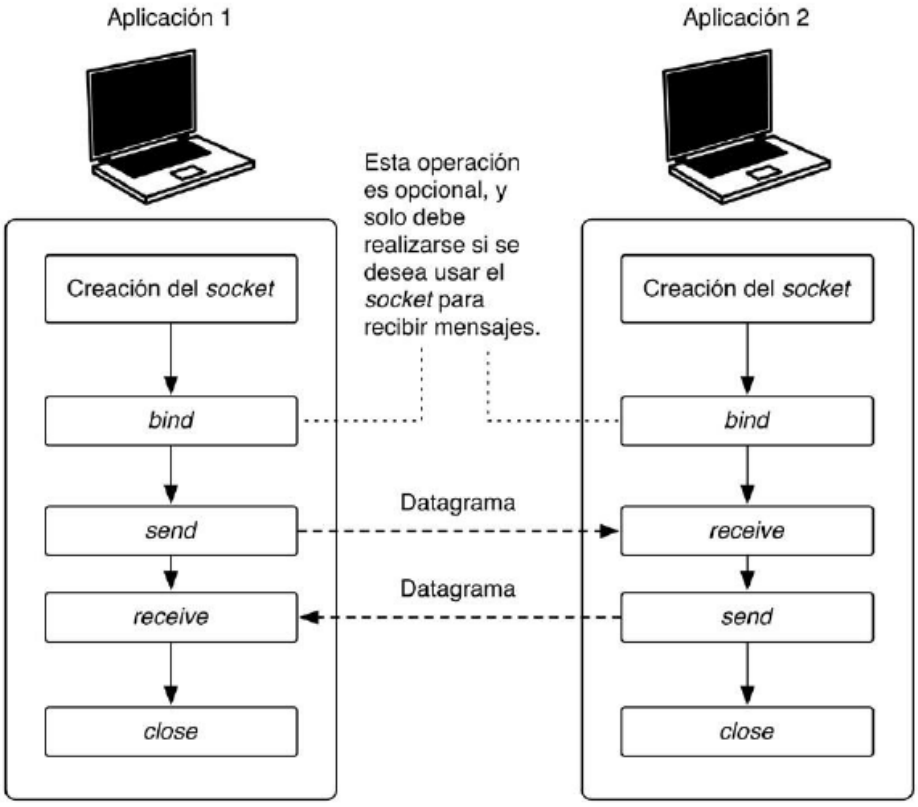
- Proporcionan una abstracción de la pila de protocolos
- Un socket (en inglés, literalmente, un "enchufe") representa el extremo de un canal de comunicación establecido entre un emisor y un receptor.

En la práctica se puede ver un socket como un conjunto: IP:puerto.



Por un lado, tenemos, **socket datagram**, que se utilizan para enviar mensajes a multitud de receptores:

- Se crea un canal temporal para cada envío que se realiza.
- Son no orientados a conexión, emplean UDP
- No existe diferencia entre el proceso servidor y el proceso cliente, es decir, se realizan los mismos pasos para enviar mensajes
- Proceso cliente y servidor:
    - Creación del socket
    - Asignación de dirección y puerto (bind)
    - Cliente (Envío y recepción de mensajes) | Servidor(Recepción y envío de mensajes)
    - Cierre de la conexión(close)



Ejemplo implementación de un cliente/servidor usando socket datagram (envío UDP). Ejecutaremos primero el servidor (*ReceptorDatagram)* que se quedará esperando hasta que el cliente(*EmisorDatagram)* le envíe el mensaje.

ReceptorDatagram.java

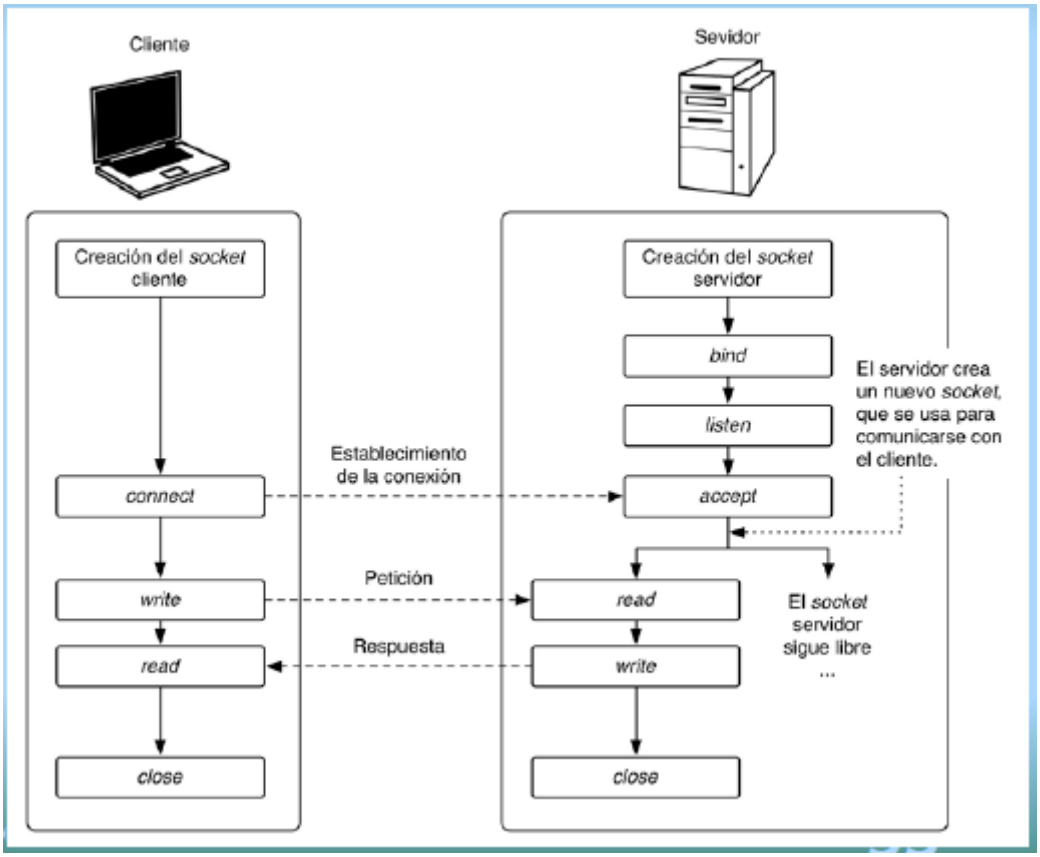
```java
package socketDatagram;

import java.io.IOException;

public class ReceptorDatagram {

    public static void main(String[] args) throws IOException {
        try {
            System.out.println("Creando socket datagram");

            InetSocketAddress addr = new InetSocketAddress ("localhost", 5555);
            DatagramSocket datagramSocket = new DatagramSocket(addr);

            System.out.println("Recibiendo mensaje");

            byte[] mensaje = new byte[25];
            DatagramPacket datagrama1 = new DatagramPacket (mensaje,25);
            datagramSocket.receive(datagrama1);//se bloquea hasta que recibe un mensaje

            System.out.println("Mensaje recibido: " + new String(mensaje));

            System.out.println("Enviando mensaje");

            InetAddress addr2 = InetAddress.getByName("localhost");
            DatagramPacket datagrama2 = new DatagramPacket (mensaje,mensaje.length,addr2,5556);
            datagramSocket.send(datagrama2);

            System.out.println("Mensaje enviado");
            System.out.println("Cerrando el socket datagrama");
            datagramSocket.close();

            System.out.println("Terminado");

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

EmisorDatagram.java

```java
package socketDatagram;

import java.io.IOException;

public class EmisorDatagram {

    public static void main(String[] args) throws IOException {
        try {
            System.out.println("Creando socket datagram");
            DatagramSocket datagramsocket = new DatagramSocket();
            System.out.println("Enviando mensaje");

            String mensaje = "mensaje desde el emisor";

            InetAddress addr = InetAddress.getByName("localhost");
            DatagramPacket datagrama_ = new DatagramPacket (mensaje.getBytes(),
                    mensaje.getBytes().length,addr,5555);

            datagramsocket.send(datagrama_);
            System.out.println("Mensaje enviado");
            System.out.println("Cerrando el socket datagrama");
            datagramsocket.close();

            System.out.println("Terminado");

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

Por otro lado, tenemos **socket stream**:

- Se utiliza para comunicarse siempre con el mismo receptor, manteniendo el canal de comunicación abierto entre ambas partes hasta que se termina la conexión
- Son orientados a conexión, emplean TCP
- Una parte ejerce la función de proceso cliente y otra de proceso servidor
- Proceso cliente:
  - Creación del socket
  - Conexión del socket(connect)
  - Envío y recepción de mensajes
  - Cierre de la conexión(close)
- Proceso servidor:
  - Creación del socket
  - Asignación de dirección y puerto(bind)
  - Escucha(listen)
  - Aceptación de conexiones (accept). Esta operación implica la creación de un nuevo socket, que se usa para comunicarse con el cliente que se ha conectado
  - Envío y recepción de mensajes
  - Cierre de la conexión(close)



En este caso, Java ofrece dos elementos fundamentales para crear programas que usen conexiones con los protocolos TCP (orientados a conexión) como son conexiones a FTP (21), Telnet(23), HTTP(80) . A estas conexiones les llamamos **Stream Sockets**:

- Sockets
- ServerSockets

Un ***Socket*** es un objeto Java que nos permite contactar con un programa o servidor remoto. Dicho objeto nos proporcionará flujos de entrada y/o salida y podremos comunicarnos con dicho programa.

Los ***ServerSocket*** se utilizan para crear programas que acepten conexiones o peticiones.

Todos los objetos mencionados en este tema están en el paquete `java.net`.

En el siguiente código puede verse el proceso básico de creación de un socket:

- Para poder crear un socket primero necesitamos una dirección con la que contactar. Toda dirección está formada por dirección IP (o DNS) y un puerto.
- El paso crítico para iniciar la comunicación es llamar al método `connect`. Este método puede disparar una excepción del tipo `IOException` que puede significar dos cosas:
    - La conexión no se pudo establecer.
    - Aunque la conexión se estableció no fue posible leer o escribir datos.

La clase `Socket` tiene dos métodos llamados `getInputStream` y `getOutputSream` que nos permiten obtener *flujos orientados a bytes*. Es posible crear nuestros propios flujos, con más métodos que ofrecen más comodidad.

Podemos contactar con un programa cualquiera escrito en cualquier lenguaje y enviar las peticiones de acuerdo al protocolo utilizado. Nuestro programa podrá leer las respuestas independientemente de cómo fuera el servidor.

Ejemplo implementación de un cliente/servidor usando streams sockets (envío TCP). Ejecutaremos primero el servidor (*ServidorSocketStream)* que se quedará esperando hasta recibir una conexión del cliente (*ClienteSocketStream)* le envíe el mensaje.

ServidorSocketStream.java

```java
package socketStream;

import java.io.IOException;

public class ServidorSocketStream {

    public static void main(String[] args) throws IOException {
        try {
            System.out.println("Creando socket servidor");
            ServerSocket serverSocket = new ServerSocket();
            System.out.println("Realizando el bind");

            InetSocketAddress addr = new InetSocketAddress ("localhost", 5555);
            serverSocket.bind(addr);
            System.out.println("Aceptando conexiones");

            Socket newSocket = serverSocket.accept();

            System.out.println("Conexión recibida");

            InputStream is = newSocket.getInputStream();
            OutputStream os = newSocket.getOutputStream();

            byte[] mensaje = new byte[25];
            is.read(mensaje);
            System.out.println("Mensaje recibido: " + new String(mensaje));

            System.out.println("Cerrando el nuevo socket");

            newSocket.close();
            System.out.println("Cerrando el socket servidor");
            serverSocket.close();
            System.out.println("Terminado");

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

    }

}
```

ClienteSocketStream.java

```java
package socketStream;

import java.io.IOException;

public class ClienteSocketStream {

    public static void main(String[] args) throws IOException {
        try {
            System.out.println("Creando socket cliente");
            Socket clientSocket = new Socket();
            System.out.println("Estableciendo la conexión");

            InetSocketAddress addr = new InetSocketAddress ("localhost", 5555);
            clientSocket.connect(addr);

            InputStream is = clientSocket.getInputStream();
            OutputStream os = clientSocket.getOutputStream();
            System.out.println("Enviando mensaje");

            String mensaje = "hola clase!!\n";
            os.write(mensaje.getBytes());
            System.out.println("mensaje enviado");
            System.out.println("Cerrando el socket cliente");
            clientSocket.close();
            System.out.println("Terminado");

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

Programación de aplicaciones cliente y servidor.

Cuando se hacen programas Java que se comuniquen, lo habitual es que uno o varios actúen de cliente y uno o varios actúen de servidores:

- **Servidor**: espera peticiones, recibe datos de entrada y devuelve respuestas.
- **Cliente**: genera peticiones, las envía a un servidor y espera respuestas.

Al crear aplicaciones cliente y servidor puede ocurrir que tengamos que implementar varias operaciones:

- Si tenemos que programar el servidor **deberemos definir un protocolo** de acceso a ese servidor.
- Si tenemos que programar solo el cliente **necesitaremos conocer el protocolo de acceso** a ese servidor.
- Si tenemos que programar los dos tendremos que empezar por **definir el protocolo de comunicación entre ambos**.

Supongamos que se nos pide crear un servidor de operaciones de cálculo:

- Cualquier parámetro que envíe el usuario debe ir terminado en un fin de línea UNIX (`\n`).
- El usuario enviará primero un símbolo para indicar la operación a realizar: «+», «-«, «*» o «/».
- Después se puede enviar un número positivo de 1 a 8 cifras. El usuario podría equivocarse y enviar en vez de «3762» algo como «37a62». En ese caso se asume que el número enviado es 0.
- Después se envía un segundo número positivo de 1 a 8 cifras.
- Cuando el servidor haya recogido todos los parámetros contestará al cliente con el resultado de haber realizado la operación indicada, que será un número positivo de 1 a 16 cifras.

Ejemplo: El servidor está implementado en la clase: *suma.ServidorCalculo* y el cliente en la clase: *suma.ClienteCalculo*

Utilización de hilos en la programación de aplicaciones en red.

Un factor fundamental en los servidores es que tienen que ser capaces de procesar varias peticiones a la vez: **deben ser multihilo**.

Su arquitectura típica es la siguiente:

```
while (true){
        peticion=esperarPeticion();
        hiloAsociado=new Hilo();
        hiloAsociado.atender(peticion);
}
```

Por lo que, en el caso de aplicaciones que necesiten aceptar varias conexiones **habrá que mover todo el código de gestión de peticiones a una clase que implemente Runnable o extienda de Thread.**

Ahora el servidor será así:

```
while (true){
        Socket conexion=socketEscucha.accept();
        System.out.println("Conexion recibida");
        Peticion p=new Peticion(conexion);
        Thread hilo=new Thread(p);
        hilo.start();
}
```

Ejemplo: Ahora tendremos la clase *sumahilo.Petición*, que atiende cada petición que llega al servidor: *sumahilo.ServidorCalculoHilo*, que son enviadas por el cliente: *sumahilo.ClienteCalculo*