Aplicaciones cliente-servidor.

# 1.- Paradigma Cliente/Servidor.

El modelo de comunicación **más utilizado** es el modelo Cliente/servidor ya que ofrece una gran **flexibilidad, interoperabilidad y estabilidad** para acceder a recursos de forma **centralizada**.

El **término Cliente/Servidor**: se acuñó por primera vez en los años 80 para explicar un sencillo paradigma: un equipo cliente requiere un servicio de un equipo servidor.

**Desde el punto de vista funcional**: este modelo es una arquitectura distribuida que permite a los **usuarios finales** obtener **acceso a recursos de forma transparente en entornos multiplataforma**. Normalmente, los recursos que suele ofrecer el servidor son **datos**, pero también puede permitir el acceso a dispositivos hardware, tiempo de procesamiento, etc.

Los **elementos que componen el modelo** son:

* **Cliente**: es el proceso que permite interactuar con el usuario, realizar las peticiones, enviarlas al servidor y mostrar los datos al cliente. Es el **front-end** (se comporta como la **interfaz** (normalmente **gráfica**) que utiliza el usuario para interactuar con el servidor). **Sus funciones son**:
  + **Interactuar con el usuario**.
  + **Procesar las peticiones**. Verificación para evitar enviar peticiones maliciosas al servidor.
  + **Recibir los resultados del servidor**.
  + **Formatear y mostrar los resultados**.
* **Servidor**: se encarga de recibir y procesar peticiones de los clientes para permitir el acceso a algún recurso. Es el **backend**. **Sus funciones son**:
  + **Aceptar las peticiones** de los clientes.
  + **Procesar** las **peticiones**.
  + **Formatear y enviar el resultado** a los clientes.
  + **Procesar la lógica de la aplicación** **y** realizar **validaciones** de datos.
  + **Asegurar la consistencia** de la información.
  + **Evitar que las peticiones de los clientes interfieran entre sí**.
  + **Mantener la seguridad** del sistema.

La idea es tratar el **servidor** como una entidad que realiza un determinado conjunto de tareas y que **las** ofrece como servicio a los clientes.

Normalmente, el trabajo pesado lo realiza el servidor y los procesos clientes sólo se encargan de interactuar con el usuario.

En otras palabras, **el modelo Cliente/Servidor es una extensión de programación modular** en la que se divide la funcionalidad del software en dos módulos con el fin de hacer más fácil el desarrollo y mejorar su mantenimiento.

Texto

Descripción generada automáticamente

## 1.1.- Características básicas.

|  |  |
| --- | --- |
| **Cliente** | **Servidor** |
| Proporciona la interfaz de usuario | Permite el acceso al recurso compartido |
| El cliente interactúa con el usuario | Todo el trabajo de procesamiento lo realiza el servidor |
| Los procesos activos son los clientes | Los servidores tienen un carácter pasivo (esperan peticiones) |

* Como se puede apreciar, existe una clara **distinción de funciones basada en el concepto de "servicio"**, que se establece **entre clientes y servidores**.
* Se establece una relación entre distintos **procesos**, los cuales **se pueden ejecutar en uno o varios equipos distribuidos a lo largo de la red**.
  + La **relación** establecida puede ser **de muchos a uno**, en la que **un servidor puede dar servicio a muchos clientes**, regulando el acceso a los recursos compartidos.
  + También puede ser de muchos a muchos (muchos servidores relacionados entre sí dan servicio a muchos clientes).
* Las **comunicaciones** se realizan estrictamente a través del **intercambio de mensajes**.
* Los **clientes** **pueden** utilizar **sistemas heterogéneos** ya que permite conectar clientes y servidores **independientemente de sus plataformas**.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## 1.2.- Ventajas y desventajas.

|  |  |
| --- | --- |
| **Ventajas** | * Utilización de clientes ligeros (con pocos requisitos hardware) -> El servidor es quien realmente hace todo el trabajo. * Facilita la **integración entre sistemas diferentes** y permite presentar la información compartida con interfaces amigables. * Se favorece **el uso de interfaces gráficas interactivas** para los clientes para interactuar con el servidor.   + Debido a que lo único que se transmite entre cliente y servidor son datos, aprovechando mejor el ancho de banda de la red. * **Mantenimiento** y **desarrollo** de aplicaciones **rápido**. * **Estructura modular**, facilitando la integración de nuevas tecnologías y la escalabilidad. * Proporciona **soluciones locales** a los departamentos de una organización, pero con la **integración de la información relevante** a nivel global. * El **acceso** a los recursos está **centralizado**. * **Clientes simultáneos**, **compartiendo información** entre sí. |
| **Desventajas** | * **Mantenimiento más difícil**: implica interacción de diferentes partes de hardware y de software, lo que dificulta el diagnóstico de errores. * Planear **estrategias para el manejo de errores** del sistema. * Importante mantener la seguridad del sistema. * **Garantizar la consistencia de la información aún cuando haya varios clientes simultáneos** operando con los mismos datos -> usar **mecanismos de sincronización** para evitar que un cliente modifique datos sin que lo sepan los demás clientes. |

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

## 1.3.- Modelos

Los modelos Cliente/Servidor **se clasifican principalmente según el nº de capas** (tiers) que tiene la infraestructura del sistema.

* **1 capa (1-tier)**: cliente y servidor se encuentran el mismo equipo. No se considera un modelo cliente/servidor, ya que no se realizan comunicaciones por la red.
* **2 capas (2-tiers)**: cliente y servidor se diferencian. Se ejecutan en equipos diferentes.
  + **Problema**: no permite escalabilidad del sistema, y puede sobrecargarse con un alto nº de peticiones por parte de los clientes.
* **3 capas (3-tiers)**: mejora el rendimiento del sistema respecto al modelo de 2 capas. En este caso se dispone de una nueva capa de servidor:
  + **Servidor de aplicación**: interactúa con diferentes clientes y envia las peticiones de procesamiento al servidor de datos.
  + **Servidor de datos**: recibe las peticiones del servidor de aplicación, las procesa y le devuelve su resultado a dicho servidor para que este los envíe al cliente.
    - Para mejorar el rendimiento del sistema, es posible añadir los servidores de datos que sean necesarios.
* **N capas (n-tiers)**: a partir del modelo anterior, se pueden añadir capas adicionales de servidores con el objetivo de separar la funcionalidad de cada servidor y de mejorar el rendimiento del sistema.

Texto

Descripción generada automáticamente

## 1.4.- Programación.

De forma interna, los **pasos que realiza el servidor para realizar una comunicación** son:solyd

* **Publicar puerto**. Publica el puerto por donde se van a recibir las conexiones.
* **Esperar peticiones**. Queda a la espera a que se conecte un cliente. Una vez que se conecte un cliente se crea el socket del cliente por donde se envían y reciben los datos.
* **Envío y recepción de datos**. Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida. Cuando el servidor recibe una petición, éste la procesa y le envía el resultado al cliente.
* Una vez finalizada la comunicación **se cierra el socket del cliente**.

**Los pasos que realiza el cliente para realizar una comunicación son**:

* **Conectarse con el servidor**. El cliente se conecta con un determinado servidor a un puerto específico. Una vez realizada la conexión se crea el socket por donde se realizará la comunicación.
* **Envío y recepción de datos**. Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida.
* Una vez finalizada la comunicación **se cierra el socket**.

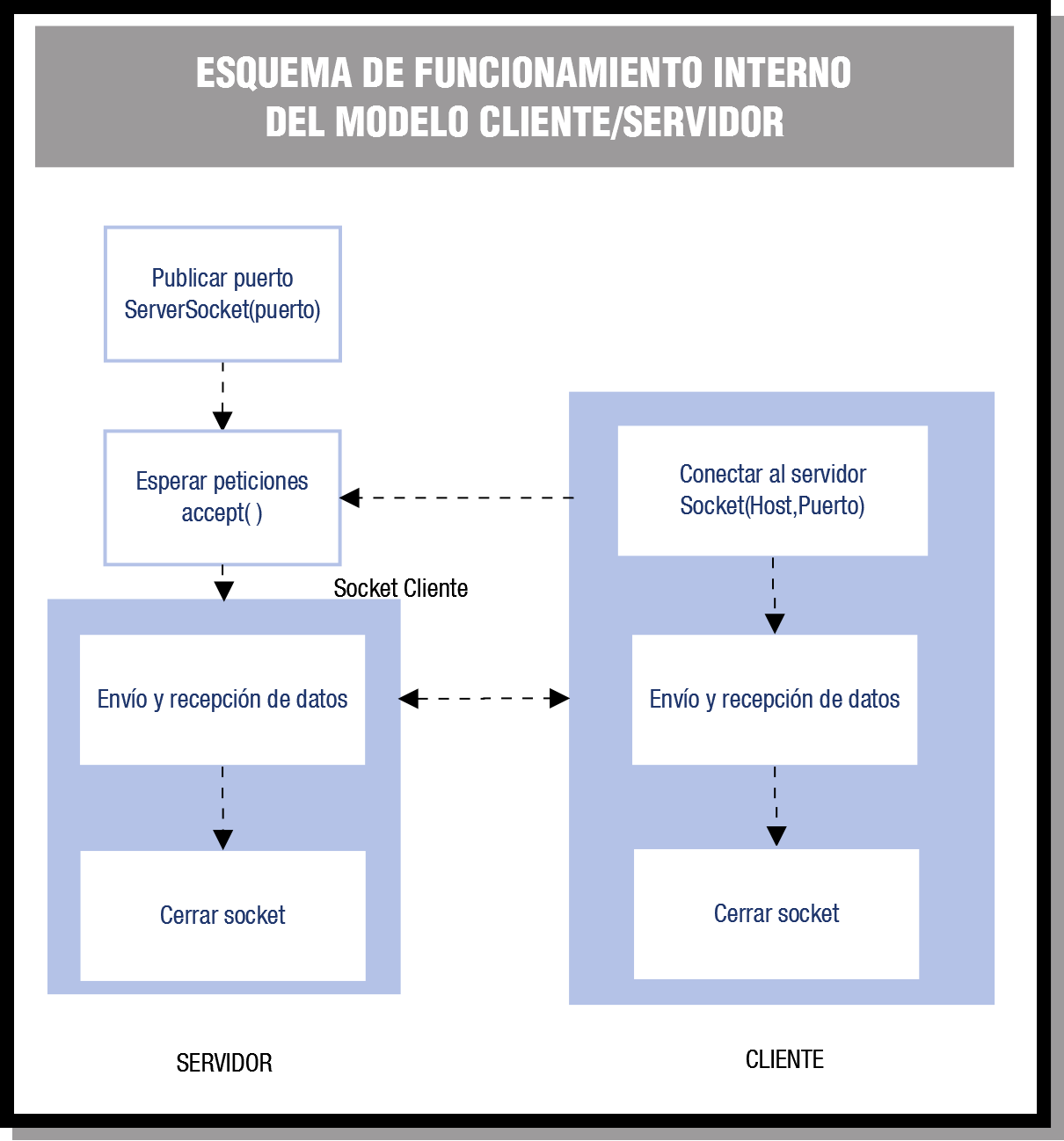


Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

## 1.5.- Ejemplo I.

Vamos a ver un ejemplo sencillo en el que **el servidor va a aceptar tres clientes** (de forma secuencial no concurrente) y le va a indicar el número de cliente que es.

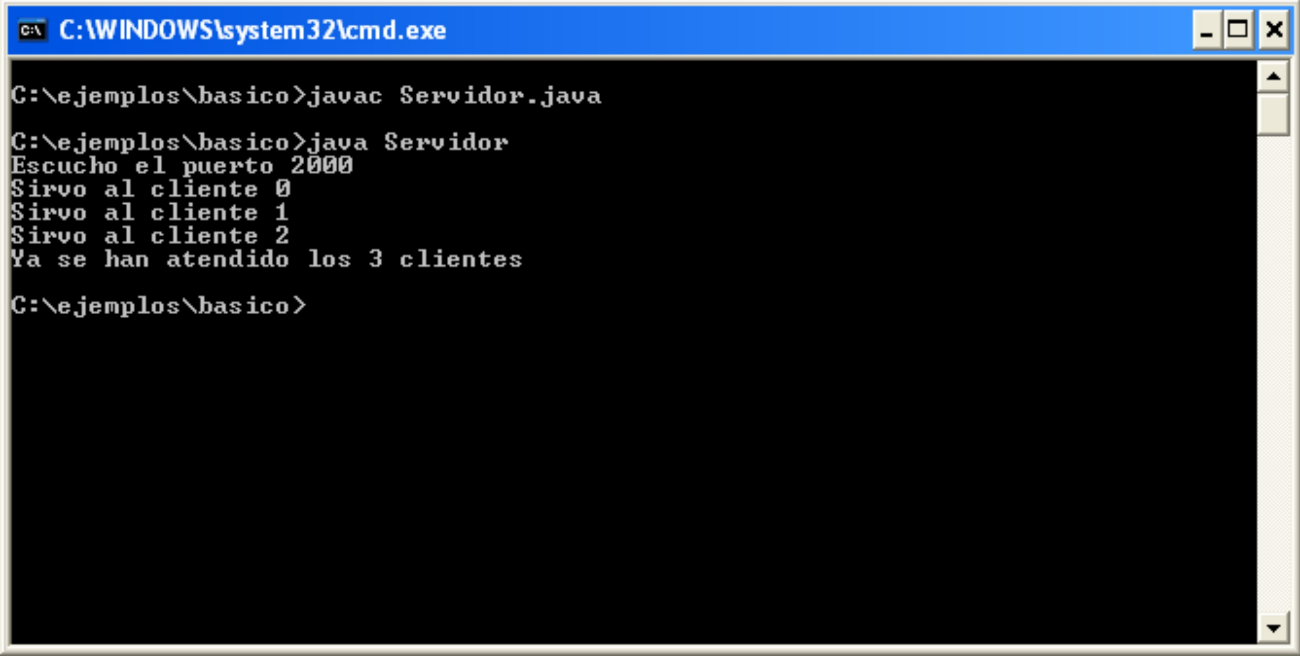
[Ver el ejemplo](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%204/Apuntes/PSP04/WEB/HTML/15_EJEMP.HTM).

Para compilar el programa, ejecute:

javac Servidor.java

Y para ejecutarlo, ejecute:

java Servidor



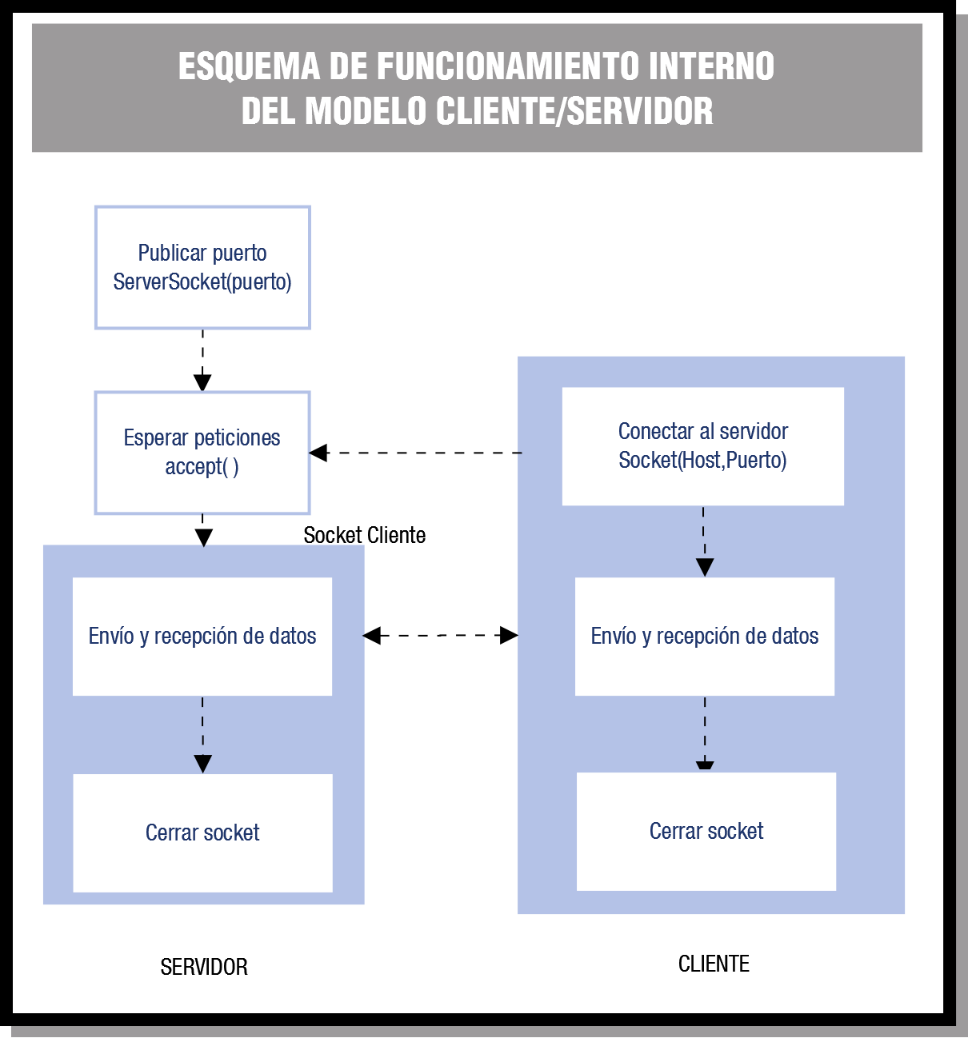
# 2.- Optimización de sockets.

A la hora de utilizar los sockets es muy importante **optimizar su funcionamiento** y **garantizar la** **seguridad del sistema**. Para lograrlo, debe cumplir estas **tres características** que veremos más adelante:

* **Atender múltiples peticiones simultáneamente**. El servidor debe permitir el acceso de forma simultánea al servidor para acceder a los recursos o servicios que éste ofrece.
* **Seguridad**. Para asegurar el sistema, como mínimo, el servidor debe ser capaz de **evitar la pérdida de información**, filtrar las peticiones de los clientes para asegurar que éstas están bien formadas y llevar un **control sobre** las diferentes **transacciones** de los clientes.
* Por último, es necesario dotar a nuestro sistema de mecanismos para **monitorizar los tiempos de respuesta** de los clientes para ver el comportamiento del sistema.

## 2.1.- Atender múltiples peticiones simultáneas.

Cuando un **servidor recibe la conexión** del cliente (**accept**) **se crea el socket** del cliente, se **realiza el envío y recepción** de datos y se **cierra** el socket.



Como el **objetivo** es **permitir que múltiples clientes utilicen el servidor de forma simultánea** es necesario que la parte de color azul en el gráfico se atienda de forma independiente para cada uno de los clientes.

Vamos a tener un **bucle while** para que **cada vez que se realice la conexión de un cliente se cree un hilo de ejecución (thread)** que será la encargada de atender al cliente. Este sistema permite tener tantos hilos como clientes se conecten a nuestro servidor de forma simultánea.

while(true){

// Se conecta un cliente

Socket skCliente = skServidor.accept();

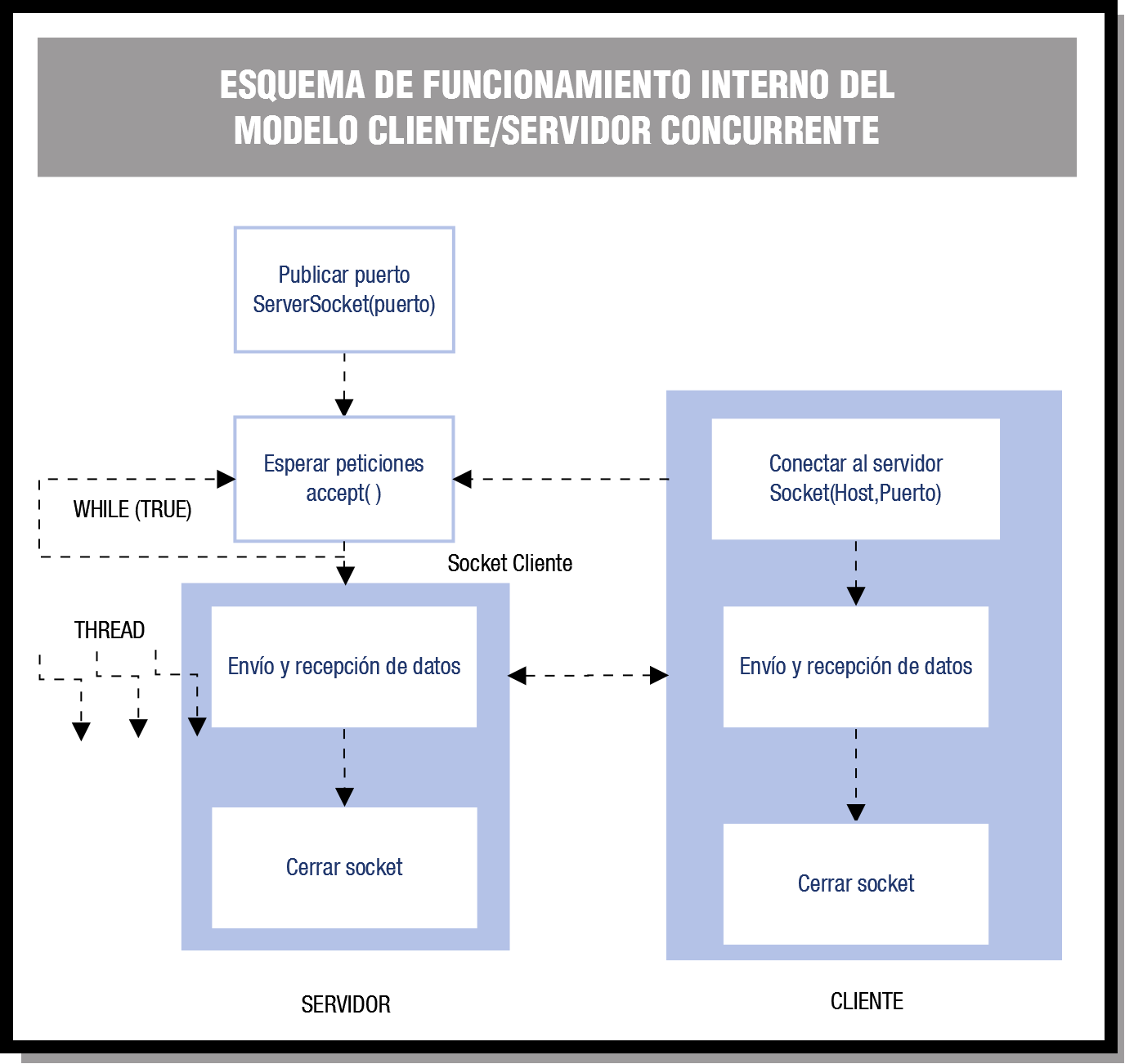
System.out.println("Cliente conectado");

// Atiendo al cliente mediante un thread

new Servidor(skCliente).start();

}

Su representación de forma gráfica sería la siguiente:



Captura de pantalla de computadora

Descripción generada automáticamente con confianza media

**Para saber más**: sobre el uso de threads y sockets en Java, [visita la siguiente presentación](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%204/Apuntes/PSP04/WEB/MP4/SOCKET_Y.MP4). O si prefieres verlo en [versión PDF](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%204/Recursos/Presentacion%20explicacion%20Sockets/Sockets%20y%20Threads.pdf).

## 2.2.- Threads.

Para crear un thread, hay que definir la clase que extienda de Threads:

class Servidor extends Thread{

public Servidor() {

// Inicialización de la hebra

}

public static void main( String[] arg ) {

new Servidor().start();

}

public void run(){

//tareas que realiza la hebra

}

}

**Para iniciar el hilo**, se crea el objeto Servidor y se inicia de la siguiente manera:

new Servidor().start();

**Para saber más**: [Thread (Java SE 13 & JDK 13 ) (oracle.com)](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Thread.html)

## 2.3.- Ejemplo II.

Si añadimos al código anterior (visto en el apartado 2.2) la parte de la utilización de sockets, obtendremos un servidor que permite atender múltiples peticiones de forma concurrente.

[Ver el ejemplo](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%204/Apuntes/PSP04/WEB/HTML/23_EJEMP.HTM).

Lógicamente, el funcionamiento del cliente no cambia ya que la concurrencia la realiza el servidor. A continuación puede ver un ejemplo básico de un cliente.

[Ver el ejemplo](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%204/Apuntes/PSP04/WEB/HTML/231_EJEM.HTM).

Texto

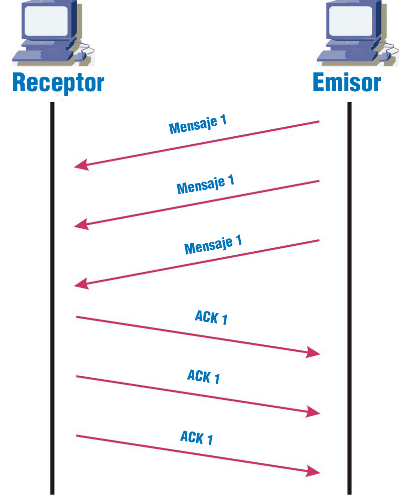
Descripción generada automáticamente

## 2.4.- Pérdida de información.

En las comunicaciones de red, la pérdida de paquetes es un factor muy importante que hay que tener en cuenta. Un solo paquete perdido produce un fichero recibido incorrectamente.

Para evitar la pérdida de paquetes, cada vez que se envía un paquete, **el receptor envía al emisor un paquete de confirmación ACK (acknowledgement)**.

**En el caso de que el mensaje no llegue correctamente al receptor, el paquete de confirmación no se envía nunca**. El emisor, pasado un tiempo, considerará que se ha producido un error y volverá a enviar el paquete.



Una **mejora** importante del método anterior consiste en **permitir al emisor que envíe múltiples paquetes de forma sin necesidad de esperar los paquetes de confirmación**:

Sin embargo, con este sistema, no está asegurado ni que los paquetes lleguen ordenados, ni que lleguen (se pierdan por el camino). Y lo mismo aplica para los paquetes ACK. Por tanto, **es necesario llevar un control sobre los paquetes enviados y los confirmados**.

**Para** **llevar un control** de los paquetes enviados **se utiliza un vector** en el que se indica si un determinado paquete se ha enviado correctamente o no.

**Por ejemplo**: supongamos un vector de ACK en un estado inicial como el siguiente:

Texto

Descripción generada automáticamente

Como puede ver en el ejemplo anterior, se han enviado 9 mensajes, de los cuales, solo el 0, 1, 4, y 5 han llegado correctamente. Por lo tanto, **para poder retransmitir más mensajes se desplaza el vector de derecha a izquierda** **hasta llegar al primer mensaje no enviado correctamente** (en el ejemplo el 2). De esta forma siguiendo el ejemplo propuesto el vector queda de la siguiente forma:

Imagen que contiene calculadora, teclado, naranja, cuarto

Descripción generada automáticamente

Ahora el sistema, **ya tiene constancia de los paquetes 10 y 11, por lo que los podría enviar mientras espera la confirmación de los demás mensajes**.

Como se puede observar, **el tamaño del vector influye en el rendimiento del sistema**: cuanto mayor sea el vector, más mensajes se pueden enviar de forma concurrente. **Sin embargo, existe la limitación de la memoria RAM** que ocupa el vector.

Todo esto ocurre de forma transparente para el usuario. El programador no debe hacer nada especial.

**Para saber más**: [sobre los mensajes ACK](https://es.wikipedia.org/wiki/ACK).

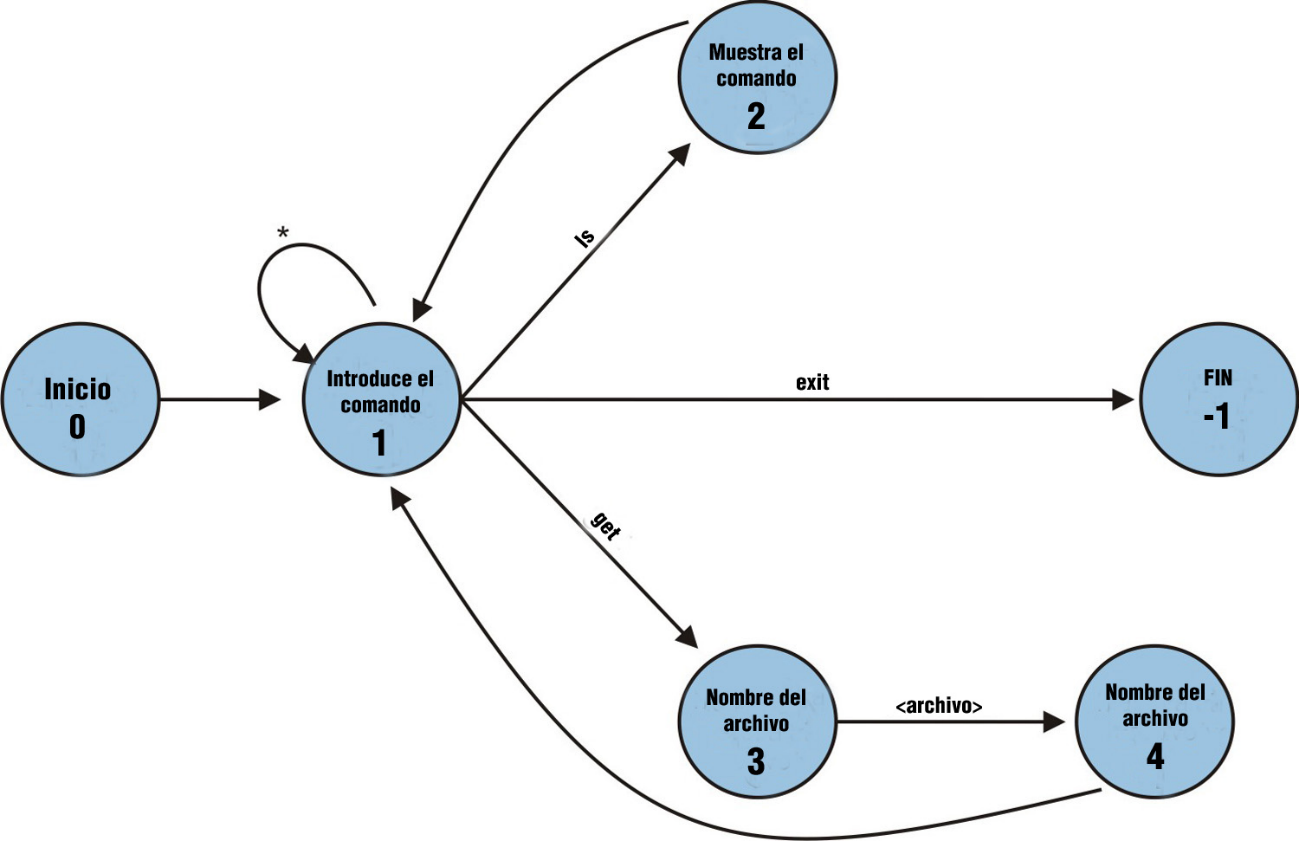
## 2.5.- Transacciones.

Uno de los principales fallos de seguridad es que el cliente pueda realizar:

* **Operaciones no autorizadas**: por ejemplo, un cliente hace una solicitud de información a la que no tiene permitido el acceso.
* **Mensajes mal formados**: por ejemplo, enviar información incompleta que produzca un error de procesamiento del sistema, llegando incluso a dejar “colgado” el servidor.

Para evitarlos, se debe **modelar el flujo de información y el comportamiento del servidor con un diagrama de estados o autómata**. Por ejemplo, en la siguiente figura, se puede ver que el servidor se inicia en el estado 0 y directamente envía al cliente el mensaje “Introduce el comando”. El cliente puede enviar los comandos:

* **ls**: que muestra el contenido del directorio (estado 2) y vuelve automáticamente al estado 1.
* **get**: solicita al cliente el nombre del archivo a mostrar (estado 3). Al introducir el nombre del archivo, se muestra el contenido del archivo (estado 4), y se vuelve automáticamente al estado 1.
* **exit**: lleva directamente al estado donde finaliza la conexión del cliente (estado -1).
* **Cualquier otro comando hace que vuelva al estado 1** solicitándole al cliente que introduzca un comando válido.



Para poder seguir el comportamiento del autómata **el servidor tiene que definir dos variables *estado* y *comando***. La variable ***estado*** almacena la **posición en la que se encuentra** y la variable ***comando*** es el **comando que recibe el servidor** y el que permite la transición de un estado a otro.

Cuando se utilizan autómatas muy sencillos como es el caso del ejemplo, es posible modelar el comportamiento del autómata utilizando estructuras **case** e **if**. En el caso de utilizar autómatas grandes la mejor forma de modelar su comportamiento es mediante una tabla cuyas filas son los diferentes estados del autómata y la columna las diferentes entradas del sistema.

**Para saber más**: [sobre los autómatas finitos](https://es.wikipedia.org/wiki/Aut%C3%B3mata_finito).

Captura de pantalla de un celular

Descripción generada automáticamente

## 2.6.- Ejemplo III.

int estado=1

do{

switch(estado){

case 1:

flujo\_salida.writeUTF("Introduce comando (ls/get/exit)");

comando=flujo\_entrada.readUTF();

if(comando.equals("ls")){

System.out.println("\tEl cliente quiere ver el contenido del directorio");

// Muestro el directorio

estado=1;

break;

}else

if(comando.equals("get")){

// Voy al estado 3 para mostrar el fichero

estado=3;

break;

}else

estado=1;

break;

case 3://voy a mostrar el archivo

flujo\_salida.writeUTF("Introduce el nombre del archivo");

String fichero =flujo\_entrada.readUTF();

// Muestor el fichero

estado=1;

break;

}

if(comando.equals("exit")) estado=-1;

}while(estado!=-1);

## 2.7.- Monitorizar tiempos de respuesta.

Es importante analizar los tiempos de respuesta de nuestro servidor. **Desde que el cliente realiza una petición hasta que recibe su resultado intervienen dos tiempos**:

* **Tiempo de procesamiento**: Es el tiempo que el servidor necesita para procesar la petición del cliente y enviar los datos.
  + Para medir el tiempo de procesamiento tan sólo se necesitar medir el tiempo que transcurre en que el servidor procese la solicitud del cliente.

import java.util.Date;

long tiempo1=(new Date()).getTime();

// Procesar la petición del cliente

long tiempo2=(new Date()).getTime();

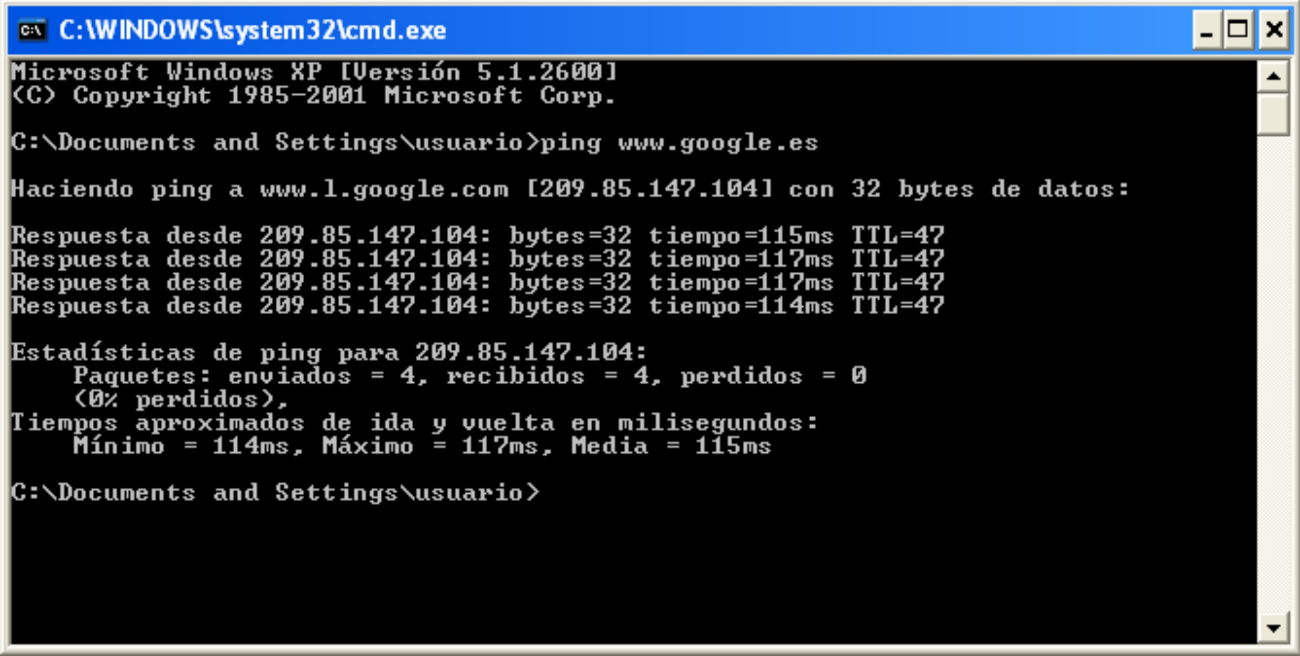
System.out.println("\t Tiempo = "+(tiempo2-tiempo1)+" ms");

* **Tiempo de transmisión**: Es el tiempo transcurrido para que el mensaje pase por todos los dispositivos de red hasta llegar a su destino.
  + Para medir el tiempo de transmisión es necesario **enviar a través de un mensaje el tiempo del sistema y el receptor comparar su tiempo con el que hay dentro del mensaje**.
  + Lógicamente, **para poder comparar los tiempos de respuesta de dos equipos es totalmente necesario que los relojes del sistema estén sincronizados** a través de cualquier servicio de tiempo (NTP).
    - En equipos Windows la sincronización de los relojes se realiza automáticamente
    - En equipos GNU/Linux se realiza ejecutando el siguiente comando:

/usr/sbin/ntpdate -u 0.centos.pool.ntp.org

**Para saber más**: [sobre el servicio NTP](https://es.wikipedia.org/wiki/Network_Time_Protocol) (Network Time Protocol) para sincronizar el reloj de dos equipos.

**Otra forma de calcular el tiempo de transmisión es utilizar el comando ping**. Por ejemplo, opdemos calcular el tiempo que el cliente tarda en acceder al servidor [www.google.es](http://www.google.es), y en este caso es de unos 115 milisegundos de media.



Texto

Descripción generada automáticamente

## 2.8.- Ejemplo IV.

Vamos a ver un ejemplo en el que se calcula el tiempo de transmisión de datos entre una aplicación Cliente y Servidor.

Para ello, el servidor le va a enviar al cliente un mensaje con el tiempo del sistema en milisegundos y el cliente cuando reciba el mensaje calculará la diferencia entre el tiempo de su sistema y el del mensaje.

[Ver ejemplo aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%204/Apuntes/PSP04/WEB/HTML/28_EJEMP.HTM).