Paradigma Cliente/Servidor

1

En el mundo de las comunicaciones entre equipos 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 modelo **`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, *se puede definir el modelo **`Cliente/servidor`** como 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 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. En definitiva, se comporta como la *interfaz* (**front-end**) que utiliza el usuario para interactuar con el servidor. Las funciones que lleva a cabo el proceso cliente se resumen en los siguientes puntos:

- Interactuar con el usuario.
- Procesar las peticiones para ver si son válidas y evitar peticiones maliciosas al servidor.
- Recibir los resultados del servidor.
- Formatear y mostrar los resultados.

`Servidor`. Es el proceso encargado de recibir y procesar las peticiones de los clientes para permitir el acceso a algún recurso (**back-end**). Las funciones del servidor 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.

La forma más habitual de utilizar el modelo **`Cliente/servidor`** es mediante la utilización de equipos a través de interfaces gráficas(**`Clientes`**); mientras que la administración de datos y su seguridad e integridad se deja a cargo del **`servidor`**. 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*.

1. *Características básicas arquitectura Cliente/Servidor.*

- ✓ Combinación de un cliente que interactúa con el usuario, y un servidor que interactúa con los recursos compartidos. *El proceso del cliente proporciona la interfaz de usuario y el proceso del servidor permite el acceso al recurso compartido*.
- ✓ Las tareas del cliente y del servidor tienen diferentes requerimientos en cuanto al procesamiento; todo el trabajo de procesamiento lo realiza el servidor, mientras que el cliente interactúa con el usuario.

- ✓ Se establece una relación entre distintos procesos, las cuales se pueden ejecutar en uno o varios equipos distribuidos a lo largo de la red.
- ✓ Existe una clara distinción de funciones basada en el concepto de "servicio", que se establece entre clientes y servidores.
- ✓ *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.
- ✓ Los ***clientes*** corresponden a ***procesos activos*** ya que realizan las peticiones de servicios a los ***servidores***. Estos últimos tienen un carácter ***pasivo*** ya que esperan las peticiones de los 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.

2. *Ventajas y desventajas.*

Principales ventajas:

- ✓ Utilización de clientes ligeros (con pocos requisitos hardware). Porque el servidor realiza todo el procesamiento.
- ✓ Facilita la integración entre sistemas y comparte información. Interfaces amigables al usuario.
- ✓ Se favorece la utilización de interfaces gráficas interactivas para los clientes para interactuar con el servidor. El uso de interfaces gráficas en el modelo **Cliente/Servidor** presenta la ventaja, con respecto a un sistema centralizado, de que normalmente sólo transmite los datos por lo que se aprovecha mejor el ancho de banda de la red.
- ✓ El ***mantenimiento y desarrollo*** de aplicaciones resulta ***rápido*** utilizando las herramientas existentes.
- ✓ La estructura inherentemente modular ***facilita*** además ***la integración de nuevas tecnologías y el crecimiento de la infraestructura computacional***, favoreciendo así la escalabilidad de las soluciones.
- ✓ Contribuye a proporcionar a los diferentes departamentos de una organización, soluciones locales, pero permitiendo la integración de la información relevante a nivel global.
- ✓ El **acceso a los recursos** se encuentra **centralizado**.
- ✓ *Los clientes acceden de forma simultánea a los datos compartiendo información entre sí*.

Principales desventajas:

- ✘ El ***mantenimiento*** de los sistemas es más ***difícil*** pues implica la interacción de diferentes partes de hardware y de software lo cual dificulta el diagnóstico de errores.
- ✘ Hay que tener estrategias para el manejo de errores del sistema.
- ✘ Es **importante mantener la seguridad del sistema**.
- ✘ Hay que **garantizar la consistencia de la información**. Como es posible que varios clientes operen con los mismos datos de forma simultánea, *es necesario utilizar mecanismos de sincronización para evitar que un cliente modifique datos sin que lo sepan los demás clientes*.

3. *Modelos.*

La principal forma de clasificar los modelos **`Cliente/Servidor`** es a partir del número de capas (tiers) que tiene la infraestructura del sistema. De ésta forma podemos tener los siguientes modelos:

a) **1 capa (1-tier)**. El proceso **`Cliente/Servidor`** se encuentra en el mismo equipo y realmente no se considera como tal modelo ya que no se realizan comunicaciones por la red.

b) **2 capas (2-tiers)**. Es el modelo tradicional en el que existe un *servidor* y *unos clientes* bien diferenciados. El principal problema es que este modelo ***no permite escalabilidad*** del sistema y ***puede sobrecargarse*** con un número alto de peticiones por parte de los clientes.

c) **3 capas (3-tiers)**. Para mejorar el rendimiento del sistema en el modelo de dos capas se añade una nueva capa de servidores. En este caso se dispone de:

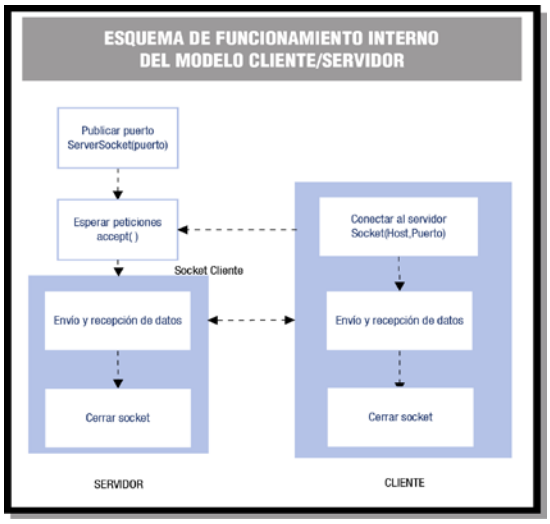
- **Servidor de aplicación**. Es el encargado de interactuar con los diferentes clientes y enviar las peticiones de procesamiento al servidor de datos.
- **Servidor de datos**. Recibe las peticiones del servidor de aplicación, las procesa y le devuelve su resultado al servidor de aplicación para que éste los envíe al cliente. Para mejorar el rendimiento del sistema, *es posible añadir los servidores de datos que sean necesarios*.

d) **n capas (n-tiers)**. A partir del modelo anterior, se pueden añadir capas adiciones de servidores con el objetivo de separar la funcionalidad de cada servidor y de mejorar el rendimiento del sistema.

4. *Programación.*

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

1. **Publicar puerto**. Publica el puerto por donde se van a recibir las conexiones.
2. **Esperar peticiones**. En este momento el servidor 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.
3. **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.
4. Una vez finalizada la comunicación **`se cierra el socket del cliente`**.



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

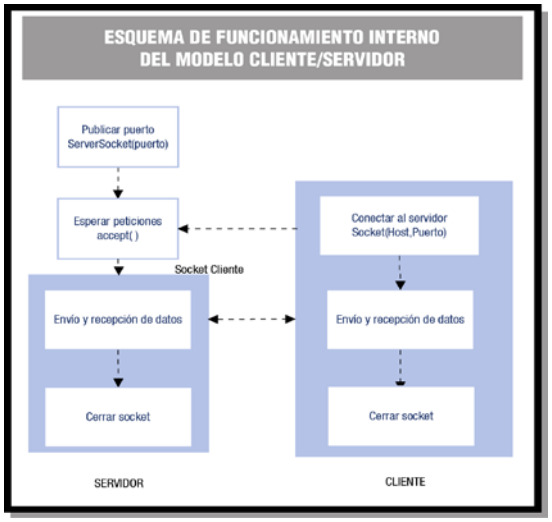
1. **`Conectarse con el servidor`**. El cliente se conecta con un determinado servidor a un puerto específico. Una vez realizada la conexión se crea el socket por donde se realizará la comunicación.
2. **`Envío y recepción de datos`**. Para poder recibir/enviar datos es necesario crear un flujo (***`stream`***)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

Optimización de sockets.

Como la información reside en el servidor y existen múltiples clientes que realizan peticiones es totalmente indispensable permitir que la aplicación cliente/servidor cuente con las siguientes características:

1. **Atender múltiples peticiones simultáneamente**.
2. **Seguridad**. 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.
3. 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.

1. *Atender múltiples peticiones simultáneas.*



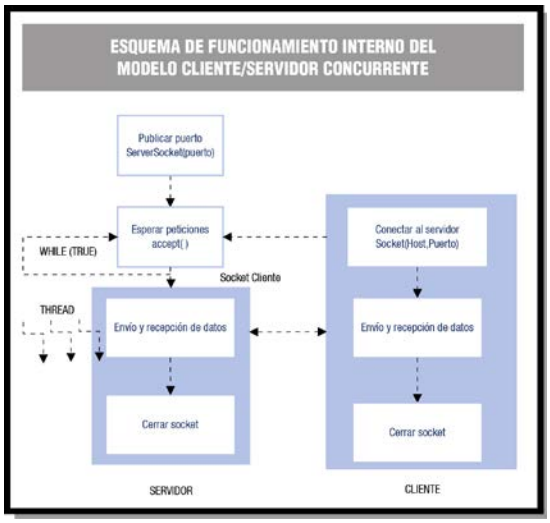
Como el objetivo es permitir que múltiples clientes utilicen el servidor de forma simultánea es necesario que la parte que atiende al cliente (zona coloreada de azul) se atienda de forma independiente para cada uno de los clientes.

Para ello, en vez de ejecutar todo el código del servidor de forma secuencial, vamos a tener un bucle ***while*** para que cada vez que se realice la conexión de un cliente se cree una hebra de ejecución (***thread***) que será la encargada de atender al cliente. De ésta forma, tendremos tantas hebras de ejecución como clientes se conecten a nuestro servidor de forma simultánea.

De forma resumida, el código necesario es:

```
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:



2. *Threads.*

Para crear una 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
  }
}
```

donde:

La función `public Servidor` permite inicializar los valores iniciales que recibe la hebra.
La función ***run***`()` es la encargada de realizar las tareas de la hebra.
Para iniciar la hebra se crea el objeto `Servidor` y se inicia:

```
new Servidor().start ();
```

Ejemplo:

Servidor.java

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

class Servidor extends Thread{
    Socket skCliente;
    static final int Puerto=2000;

    public Servidor(Socket sCliente) {
        skCliente=sCliente;
    }

public static void main( String[] arg ) {
    try{
        // Inicio el servidor en el puerto
        ServerSocket skServidor = new ServerSocket(Puerto);
        System.out.println("Escucho el puerto " + Puerto);

        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();
        }
    } catch (Exception e) {;}
}


public void run(){
        try {
               // Creo los flujos de entrada y salida
              DataInputStream flujo_entrada = new DataInputStream(skCliente.getInputStream());
              DataOutputStream flujo_salida= new DataOutputStream(skCliente.getOutputStream());

              // ATENDER PETICIÓN DEL CLIENTE
              flujo_salida.writeUTF("Se ha conectado el cliente de forma correcta");

              // Se cierra la conexión
              skCliente.close();
              System.out.println("Cliente desconectado");

     } catch( Exception e ) {
         System.out.println( e.getMessage() );
     }
    }
}
```

Lógicamente, el funcionamiento del cliente no cambia ya que la concurrencia la realiza el servidor.

Cliente.java

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

class Cliente {
    static final String HOST = "localhost";
    static final int Puerto =2000;

    public Cliente( ) {
        try{
            Socket sCliente = new Socket( HOST , Puerto);
            // Creo los flujos de entrada y salida
            DataInputStream flujo_entrada = new DataInputStream(skCliente.getInputStream());
            DataOutputStream flujo_salida= new DataOutputStream(skCliente.getOutputStream());

            // TAREAS QUE REALIZA EL CLIENTE
            String datos=flujo_entrada.readUTF();
            System.out.println(datos);

            sCliente.close();
        } catch( Exception e ) {
     System.out.println( e.getMessage() );
        }
    }

    public static void main( String[] arg ) {
     new Cliente();
    }
}
```

3. *Pérdida de información.*

La pérdida de paquetes en las comunicaciones de red es un factor muy importante que hay que tener en cuenta ya que, por ejemplo, si se envía un fichero la pérdida de un único paquete produce que el fichero no se reciba correctamente.

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

En el caso que el mensaje no llegue correctamente al receptor el paquete de confirmación no se envía nunca. El emisor cuando transcurre un determinado tiempo considera que en el paquete se ha producido un error y vuelve a enviar el mismo.

Este método, aunque efectivo, resulta bastante lento ya que para enviar un nuevo paquete debe esperar el ***ACK*** del paquete anterior por lo que se produce un retardo en las comunicaciones.

Una mejora importante del método anterior consiste en permitir al emisor que envíe múltiples paquetes sin necesidad de esperar los ***ACK***. De esta forma el emisor puede enviar N paquetes de forma simultánea y así mejorar las comunicaciones.

Como una de las características de las redes es la posibilidad que los paquetes no lleguen ordenados, ni los paquetes ***ACK*** lleguen ordenados o, simplemente que se pierda algún mensaje en el camino, *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. Lógicamente, como los paquetes pueden llegar de forma desordenada, perderse paquetes,... es posible encontrar en el vector de configuración múltiples combinaciones como la siguiente:

Vector de ACK (estado inicial)										
Mensaje	0	1	2	3	4	5	6	7	8	9
ACK	1	1	0	0	1	1	0	0	0	0

Como puede ver en el ejemplo anterior, los mensajes 0, 1, 4, y 5 han llegado correctamente. Por lo tanto para poder retransmitir más mensajes se desplaza el vector de derecha a izquierda con los mensajes enviados correctamente 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:

Vector de ACK (desplazado)										
Mensaje	2	3	4	5	6	7	8	9	10	11
ACK	0	0	1	1	0	0	0	0	0	0

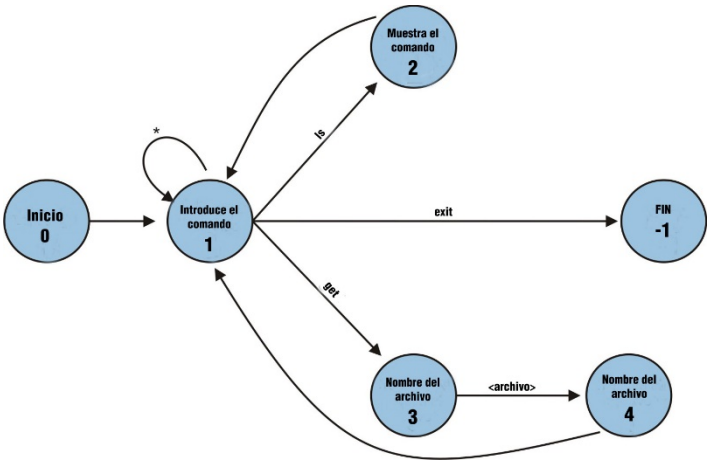
Ahora el sistema ya puede enviar los mensajes 10 y 11 mientras que espera la confirmación de los demás mensajes.

Como ha podido observar, el tamaño del vector influye muy estrechamente en el rendimiento del sistema ya que *cuanto mayor sea el vector más mensajes se pueden enviar de forma concurrente*. Lógicamente, *existe la limitación de la memoria RAM* que ocupa el vector.

4. *Transacciones.*

Uno de los principales fallos de seguridad que se producen en la programas clientes/servidor es que el cliente pueda realizar:

- ✓ **Operaciones no autorizadas**. El servidor no puede procesar una orden a la que el cliente no tiene acceso. Por ejemplo, que el cliente realice una solicitud de información a la que no tiene acceso.
- ✓ **Mensajes mal formados**. Es posible que el cliente envíe al servidor mensajes mal formados o con información incompleta que produzca un error de procesamiento del sistema llegando incluso a dejar "colgado" el servidor.



Para evitar cualquier problema de seguridad es muy importante modelar el flujo de información y el comportamiento del servidor con un **`diagrama de estados o autómata`**. Por ejemplo, en la imagen 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 va al estado 2 mostrando el contenido del directorio y vuelve automáticamente al estado 1.

`get` que le lleva al estado 3 donde le solicita al cliente el nombre del archivo a mostrar. Al introducir el nombre del archivo se desplaza al estado 4 donde muestra el contenido del archivo y vuelve automáticamente al estado 1.

`exit` que le 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*.

Ejemplo:

```
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);
```

5. *Monitorizar tiempos de respuesta.*

Un aspecto muy importante para ver el comportamiento de nuestra aplicación Cliente/Servidor son los tiempos de respuesta del 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.
- **Tiempo de transmisión**. Es el *tiempo que transcurre para que los mensajes viajen* a través de los diferentes dispositivos de la red hasta llegar a su destino.

Para medir el tiempo de procesamiento tan sólo se necesitar medir el tiempo que transcurre en que el servidor procese la solicitud del cliente. *Para medir el tiempo en milisegundos necesario para procesar la petición de un cliente puede utilizar el siguiente código*:

```
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");
```

Para medir el tiempo de transmisión es necesario enviar a través de un mensaje el tiempo del sistema y el receptor comparar su tiempo de respuesta 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, **`Network Time Protocol`**). En equipos Windows la sincronización de los relojes se realiza automáticamente y *en equipos GNU/Linux se realiza ejecutando el siguiente comando*:

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

Otra forma de calcular el tiempo de transmisión es utilizar el comando `ping`, desde la consola `cmd` de `Windows`.

Ejemplo:

Servidor.java

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

class Servidor {
    static final int Puerto=2000;

    public Servidor( ) {

        try {
            // Inicio el servidor en el puerto
            ServerSocket sServidor = new ServerSocket(Puerto);
            System.out.println("Escucho el puerto " + Puerto );

            // Se conecta un cliente
            Socket sCliente = sServidor.accept(); // Crea objeto
            System.out.println("Cliente conectado");

            // Creo los flujos de entrada y salida
            DataInputStream flujo_entrada = new DataInputStream( sCliente.getInputStream());
            DataOutputStream flujo_salida= new DataOutputStream(sCliente.getOutputStream());

            // CUERPO DEL ALGORITMO
            long tiempo1=(new Date()).getTime();
            flujo_salida.writeUTF(Long.toString(tiempo1));

     // Se cierra la conexión
     sCliente.close();
     System.out.println("Cliente desconectado");

        } catch( Exception e ) {
     System.out.println( e.getMessage() );
        }
}

public static void main( String[] arg ) {
    new Servidor();
}
}
```

Cliente.java

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

class Cliente {
    static final String HOST = "localhost";
    static final int Puerto=2000;

    public Cliente( ) {
     String datos=new String();
     String num_cliente=new String();
     // para leer del teclado
     BufferedReader reader=new BufferedReader(new InputStreamReader(System.in));

     try{
         // Me conecto al puerto
         Socket sCliente = new Socket( HOST , Puerto );

         // Creo los flujos de entrada y salida
         DataInputStream flujo_entrada = new DataInputStream(sCliente.getInputStream());
         DataOutputStream flujo_salida= new DataOutputStream(sCliente.getOutputStream());

         // CUERPO DEL ALGORITMO
         datos=flujo_entrada.readUTF();
         long tiempo1=Long.valueOf(datos);
         long tiempo2=(new Date()).getTime();
         System.out.println("\n El tiempo es:"+(tiempo2-tiempo1));

         // Se cierra la conexión
         sCliente.close();
     } catch( Exception e ) {
         System.out.println( e.getMessage() );
     }
        }

    public static void main( String[] arg ) {
        new Cliente();
    }
}
```