1.- Paradigma Cliente/Servidor.

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; 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

1.1.- Características básicas.

Las características básicas de una arquitectura Cliente/Servidor son:

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 y 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.

1.2.- Ventajas y desventajas.

Ventajas del esquema Cliente/Servidor destacan:

Utilización de clientes ligeros (con pocos requisitos hardware) ya que el servidor es quien realmente realiza todo el procesamiento de la información.

Facilita la integración entre sistemas diferentes y comparte información permitiendo 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.

Desventajas del esquema Cliente/Servidor destacan:

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.

1.3.- 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:

1 capa (1-tier). El proceso cliente/servidor se encuentra en el mismo equipo y realmente no se considera un modelo cliente/servidor ya que no se realizan comunicaciones por la red.

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

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.

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.

1.4.- Programación.

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

Publicar puerto. Publica el puerto por donde se van a recibir las conexiones.

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.

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.

1.5.- Ejemplo I.

A modo de ejemplo, 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.

Ejemplo 1

servidor.java

2.- Optimización de sockets.

A la hora de utilizar los sockets es muy importante optimizar su funcionamiento y garantizar la seguridad del sistema. 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 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 del cliente finalizando la ejecución del servidor.

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();
}
```

2.2.- 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 ();

2.3.- Ejemplo II.

Si añade al código anterior la utilización de sockets, tal y como se ha visto anteriormente, por parte del servidor obtiene un servidor que permite atender múltiples peticiones de forma concurrente:

Ejemplo 3

Servidor.java

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

2.3.1.- Ejemplo II (II).

Ejemplo 4

Cliente.java

2.4.- 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). 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
```

2.5.- Ejemplo IV.

A continuación vamos a ver un ejemplo en el que se calcula el tiempo de trasmisión 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.

Ejemplo 5 Servidor.java

Ejemplo 6 Cliente.java