Clase del día - 24/02/2021

La clase de hoy vamos a ver cómo programar un cliente y un servidor en Java.

Un cliente es un programa que **se conecta** a un programa servidor. Notar que el cliente inicia la conexión con el servidor.

Una vez que el cliente está conectado al servidor, el cliente puede enviar datos al servidor y el servidor puede mandar datos al cliente. A este tipo de comunicación se le conoce como **bi-direccional**, debido a que los datos pueden fluir en ambas direcciones.

En particular, los clientes y servidores que utilizaremos en el curso usan sockets TCP. Más adelante en el curso veremos los tipos de sockets y sus características.

Para compilar y ejecutar los programas del curso vamos a utilizar JDK8 desde la línea de comandos.

Los que quieran utilizar ambientes de desarrollo como Netbeans o Eclipse pueden hacerlo, sin embargo en general vamos a ejecutar los programas en la línea de comandos.

Cliente.java

El programa Cliente.java es un ejemplo de un cliente de sockets TCP que se conecta a un servidor y posteriormente envía y recibe datos.

Primeramente vamos a crear un socket que se conectará al servidor. En este caso el servidor se llama "localhost" (computadora local) y el puerto abierto en el servidor es el 50000. En general el nombre del servidor puede ser un nombre de dominio (como midominio.com o una dirección IP). El número de puerto es un número entero entre 0 y 65535.

```
Socket conexion = new Socket("localhost",50000);
```

En este caso declaramos una variable de tipo Socket llamada "conexión" la cual va a contener una instancia de la clase Socket.

Es importante aclarar que antes de crear el socket, el programa servidor debe estar en ejecución y esperando una conexión, de otra manera la instrucción anterior produce una excepción, la cual desde luego debería controlarse dentro de un bloque **try**.

Para enviar datos al servidor a través del socket, vamos a crear un stream de salida de la siguiente manera:

```
DataOutputStream salida = new DataOutputStream(conexion.getOutputStream());
```

De la misma forma, para leer los datos que envía el servidor a través del socket, creamos un stream de entrada:

```
DataInputStream entrada = new DataInputStream(conexion.getInputStream());
```

Ahora podemos enviar y recibir datos del servidor. Veamos algunos ejemplos.

Vamos a enviar un entero de 32 bits, en este caso el número 123, utilizando el método **writeInt**:

```
salida.writeInt(123);
```

Ahora vamos a enviar un número punto flotante de 64 bits utilizando el método **writeDouble**:

```
salida.writeDouble(1234567890.1234567890);
```

Vamos a enviar la cadena de caracteres "hola":

```
salida.write("hola".getBytes());
```

Debido a que el método **write** envía un arreglo de bytes, para enviar la cadena de caracteres "hola" es necesario convertirla a arreglo de bytes mediante el método **getBytes**.

Ahora supongamos que el servidor envía al cliente una cadena de caracteres. Para que el cliente reciba la cadena de caracteres es necesario que conozca el número de bytes que envía el servidor, en este caso el servidor envía una cadena de caracteres de 4 bytes.

Para recibir los bytes se utiliza el método **read** de la clase DataInputStream, sin embargo es importante tomar en cuenta que el método **read** podría leer solo una fracción del mensaje enviado.

Es un error muy común de los programadores creer que el método **read** siempre regresa el mensaje completo.

En realidad cuando un mensaje es largo, el método **read** debe ser invocado repetidamente hasta recibir el mensaje completo.

Para recibir el mensaje completo implementaremos un nuevo método **read** de la siguiente manera:

```
static void read(DataInputStream f,byte[] b,int posicion,int longitud) throws Exception
{
  while (longitud > 0)
  {
    int n = f.read(b,posicion,longitud);
    posicion += n;
    longitud -= n;
  }
}
```

En este caso, el método estático **read** regresará el mensaje completo en el arreglo de bytes "b". Notar que el método **read** de la clase DataInputStream regresa el número de bytes efectivamente leídos. Debido a que el método **read** de la clase DataInputStream puede producir una excepción, es necesario invocar este método dentro de un bloque try o bien. se debe utilizar la cláusula **throws** en el prototipo del método.

Para recibir la cadena de caracteres que envía el servidor, vamos a invocar el método estático **read**:

byte[] buffer = new byte[4];

read(entrada,buffer,0,4);

System.out.println(new String(buffer,"UTF-8"));

Debido a que la variable buffer contiene los bytes correspondientes a la cadena de caracteres que envió el servidor, para obtener la cadena de caracteres utilizamos el constructor de la clase String para crear una cadena de caracteres a partir del arreglo de bytes indicando la codificación, en este caso UTF-8.

Ahora veremos cómo enviar de manera eficiente un conjunto de números punto flotante de 64 bits.

Supongamos que vamos a enviar cinco números punto flotante de 64 bits.

Primero "empacaremos" los números utilizando un objeto ByteBuffer. Cinco números punto flotante de 64 bits ocupan 5x8 bytes (64 bits=8 bytes). Entonces vamos a crear un objeto de tipo ByteBuffer con una capacidad de 40 bytes:

ByteBuffer b = ByteBuffer.allocate(5*8);

Utilizamos el método **putDouble** para agregar cinco números al objeto ByteBuffer:

b.putDouble(1.1);

b.putDouble(1.2);

b.putDouble(1.3);

b.putDouble(1.4);

b.putDouble(1.5);

Para enviar el "paquete" de números, convertimos el objeto BytetBuffer a un arreglo de bytes utilizando el método **array** de la clase ByteBuffer:

byte[] a = b.array();

Entonces enviamos el arreglo de bytes utilizando el método **write**:

Para terminar el programa cerrar los streams de salida y entrada, así como la conexión con el servidor:

salida.close();

entrada.close();

conexion.close();

Servidor.java

El programa Servidor.java va a esperar una conexión del cliente, entonces recibirá los datos que envía el cliente y a su vez, enviará datos al cliente.

Primeramente vamos a crear un socket servidor que va a abrir, en este caso, el puerto 50000:

ServerSocket servidor = new ServerSocket(50000);

Notar que en Windows, por razones de seguridad el firewall solicita al usuario administrador permiso para abrir este puerto.

Ahora invocamos el método **accept** de la clase ServerSocket.

El método **accept** es bloqueante, lo que significa que el thread principal del programa quedará en estado de espera pasiva (una espera que no ocupa ciclos de CPU) hasta recibir una conexión del cliente. Cuando se recibe la conexión el método **accept** regresa un socket, en este caso vamos a declarar una variable de tipo Socket llamada "conexion":

Socket conexion = servidor.accept();

Una vez establecida la conexión con el cliente, el servidor podrá enviar y recibir datos.

Creamos un stream de salida y un stream de entrada:

DataOutputStream salida = new DataOutputStream(conexion.getOutputStream());

DataInputStream entrada = new DataInputStream(conexion.getInputStream());

Recordemos que el cliente envía un entero de 32 bits, entonces el servidor deberá recibir este dato utilizando el método **readInt**:

int n = entrada.readInt();

System.out.println(n);

Ahora el servidor recibe un número punto flotante de 64 bits utilizando el método **readDouble**:

double x = entrada.readDouble();

System.out.println(x);

El servidor recibe una cadena de cuatro caracteres:

byte[] buffer = new byte[4];

read(entrada,buffer,0,4);

System.out.println(new String(buffer,"UTF-8"));

El servidor envía una cadena de cuatro caracteres:

salida.write("HOLA".getBytes());

Ahora vamos a recibir los cinco números punto flotante empacados en un arreglo de bytes.

Recordemos que los cinco número punto flotante de 64 bits ocupan 40 bytes (5x8bytes).

```
byte[] a = new byte[5*8];

read(entrada,a,0,5*8);
```

Una vez recibido el arreglo de bytes, lo convertimos a un objeto BytetBuffer utilizando el método **wrap** de la clase ByteBuffer:

```
ByteBuffer b = ByteBuffer.wrap(a);
```

Para extraer los números punto flotante, utilizamos el método **getDouble** de la clase ByteBuffer:

```
for (int i = 0; i < 5; i++) System.out.println(b.getDouble());
```

Finalmente, cerramos los streams de salida y entrada, así como la conexión con el cliente:

```
salida.close();

entrada.close();

conexion.close();
```

¿Qué resulta más eficiente, enviar los números de manera individual mediante writeDouble o enviarlos empacados mediante ByteBuffer?

Clase del día - 25/02/2021

La clase anterior vimos el programa Servidor.java el cual invoca el método **accept** para esperar una conexión del cliente, debido a que este método es bloqueante el programa queda en espera pasiva hasta que el cliente se conecta.

Cuando el servidor recibe una conexión, el método **accept** regresa un socket. Entonces el cliente y el servidor podrán intercambiar datos. Generalmente el servidor procesa los datos que recibe del cliente y al terminar vuelve a invocar el método **accept** para esperar otra conexión.

Sin embargo, mientras el servidor procesa los datos que recibe del cliente, no puede recibir otra conexión. Para resolver este problema los servidores se construyen utilizando threads.

En la clase de hoy veremos cómo construir un servidor multithread.

Programación multithread en Java

Supongamos que tenemos una clase principal llamada **P**.

Dentro de la clase **P** definimos una clase interior (*nested class*) llamada **Worker** la cual es subclase de la clase Thread:

```
class P

{

 static class Worker extends Thread

 {
```

```
    public void run()
    {
    }
  }
  public static void main(String[] args) throws Exception
  {
  }
}
```

Podemos ver que hemos incluido en la clase **Worker** un método público llamado **run** el cual no tiene parámetros ni regresa un resultado.

En el curso de Sistemas Operativos se explicó que un thread (hilo) es una secuencia de instrucciones que ejecutan en una computadora. Si la computadora tiene un CPU *dual core*, entonces el CPU podrá ejecutar en paralelo (al mismo tiempo) dos threads, si el CPU es *quad core* entonces podrá ejecutar en paralelo cuatro threads, y así sucesivamente.

Por otra parte, si un programa crea un número de threads mayor al número de procesadores físicos (*cores*) disponibles en la computadora, entonces los threads ejecutarán en forma concurrente (por turnos).

Crear un thread e iniciar su ejecución

Para iniciar la ejecución de un thread, debemos crear una instancia de la clase **Worker** e invocar el método **start** (este método se hereda de la clase Thread):

```
Worker w = new Worker();
w.start();
```

Entonces se crea un hilo que inicia invocando el método **run** que hemos definido en la clase **Worker**.

Un thread finaliza su ejecución cuando el método **run** termina. Cuando un thread finaliza, no puede volver a ejecutarse.

El método join

Supongamos que el thread principal (el thread que invocó el método **start**) requiere esperar que el thread w termine su ejecución, entonces el thread principal deberá invocar el método **join**:

```
Worker w = new Worker();
w.start();
w.join();
```

El método **join** queda en un estado de espera pasiva mientras el thread "w" se encuentra ejecutando, cuando el thread "w" termina, el método **join** regresa, entonces el thread principal continua su ejecución.

Ahora supongamos que el thread principal requiere crear dos threads y esperar a que terminen su ejecución. Entonces creamos dos instancias de la clase **Worker** e invocamos los métodos **start** y **join** para cada thread:

```
Worker w1 = new Worker();
Worker w2 = new Worker();

w1.start();
w2.start();
w1.join();
w2.join();
```

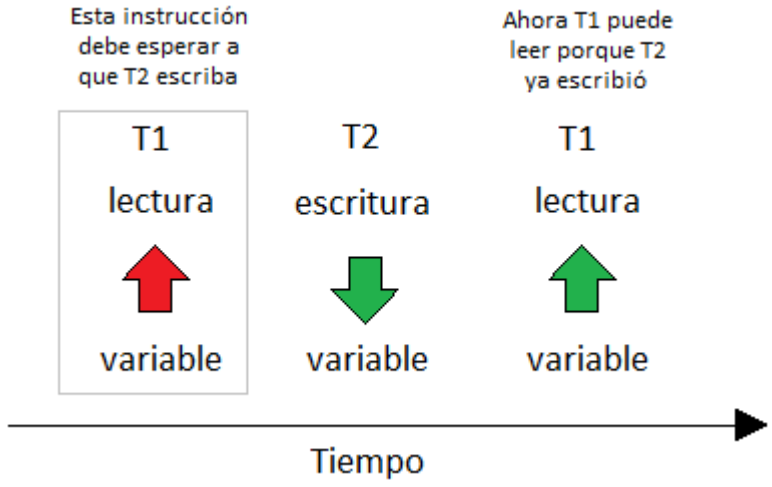
Cuando un thread (en este caso el thread principal) espera la terminación de uno o más threads para continuar su ejecución, se dice que se implementa una **barrera**. En este caso estamos implementando una barrera mediante dos métodos **join**.

Sincronización de threads

Cuándo dos o más threads acceden a una misma variable, y al menos uno de los threads modifica (escribe) la variable, entonces es necesario sincronizar el acceso de los threads a la variable.

Sincronizar el acceso de los threads significa ordenar las operaciones de escritura y de lectura que realizan los threads.

Por ejemplo, si el thread T1 va a leer una variable que escribe el thread T2, entonces el thread T1 debe esperar a que el thread T2 escriba la variable; evidentemente el thread T1 no puede leer un valor que no ha sido escrito todavía.



Para ordenar las lecturas y escrituras que realizan los threads dentro de un proceso, se utiliza la instrucción **synchronized**:

```
synchronized(objeto)
{
  instrucciones
}
```

La instrucción **synchronized** funciona de la siguiente manera:

- Primero se verifica si el lock del *objeto* está bloqueado (en Java todos los objetos tienen un lock asociado), si el lock está bloqueado entonces el thread espera a que el lock se desbloquee.
- Por otra parte, si el lock está desbloqueado entonces el thread lo bloquea (se dice que "el thread adquiere el lock") y ejecuta las instrucciones dentro del bloque.
- Al terminar de ejecutar las instrucciones el thread desbloquea el lock (se dice que "el thread libera el lock"), entonces el sistema operativo notifica a alguno de los threads que se encuentran esperando el lock para que adquiera el lock y ejecute las instrucciones dentro del bloque.
- Al terminar de ejecutar las instrucciones, el thread desbloquea el lock y nuevamente el sistema operativo notifica a alguno de los threads que esperan.

Como podemos ver, la instrucción **synchronized** evita que dos o más threads ejecuten simultáneamente un bloque de instrucciones. Al bloque de instrucciones que solo puede ser ejecutado por un thread se le llama **sección crítica**.

Veamos un ejemplo.

Supongamos que tenemos dos threads que incrementan una variable estática llamada "n" dentro de un ciclo for. La variable estática "n" es "global" a todas las instancias de la clase, por tanto los threads pueden leer y escribir esta variable.

```
class A extends Thread
{
  static long n;
  public void run()
  {
    for (int i = 0; i < 100000; i++)
        n++;
  }
  public static void main(String[] args) throws Exception
  {
    A t1 = new A();
    A t2 = new A();
    t1.start();
```

```
   t2.start();
   t1.join();
   t2.join();
   System.out.println(n);
  }
}
```

En este caso, la clase principal A es subclase de la clase Thread, por tanto hereda los métodos run, start y join (entre otros).

El programa debería desplegar 200000 ya que cada thread incrementa 100000 veces la variable "n".

¿Por qué despliega un número menor a 200000?

¿Por qué cada vez que se ejecuta el programa despliega un número diferente?

El problema es que los dos threads ejecutan al mismo tiempo la instrucción n++.

El incremento de la variable se compone de tres operaciones: la lectura a la variable, el incremento del valor y la escritura del nuevo valor. Sin embargo, los dos threads ejecutan al mismo tiempo las instrucciones de lectura y escritura sobre la misma variable, lo cual ocasiona que algunos incrementos "se pierdan" (no se escriban sobre la variable "n").

Entonces debemos impedir que ambos threads ejecuten al mismo tiempo la instrucción n++. Justamente esta instrucción es la sección crítica.

Ahora vamos a ejecutar la instrucción n++ dentro de una instrucción synchronized, Notar que utilizamos el objeto "obj" para sincronizar los threads:

```
class A extends Thread
{
  static long n;
  static Object obj = new Object();
  public void run()
  {
    for (int i = 0; i < 100000; i++)
    synchronized(obj)
    {
      n++;
    }
  }
  public static void main(String[] args) throws Exception
  {
    A t1 = new A();
    A t2 = new A();
    t1.start();
    t2.start();
```

```
   t1.join();
   t2.join();
   System.out.println(n);
  }
}
```

En este caso el programa siempre despliega 200000.

Si bien es cierto que es necesario sincronizar los threads para que el programa funcione correctamente, la sincronización hace más lento el programa, ya que obliga a que ciertas partes del programa se ejecuten en serie (una tras otra) y no en paralelo (al mismo tiempo).

Servidor2.java

Ahora vamos a implementar el servidor de sockets multithread.

La idea es que el servidor multithread espere conexiones y para cada conexión cree un thread que procese los datos que envía el cliente.

Vamos a invocar el método **accept** dentro de un ciclo, y para cada conexión vamos a crear un thread.

```
class Servidor2

{

 static class Worker extends Thread

 {

   Socket conexion;

   Worker(Socket conexion)

   {

    this.conexion = conexion;

   }

   public void run()

   {

   }

 }

 public static void main(String[] args) throws Exception

 {

   ServerSocket servidor = new ServerSocket(50000);

   for (;;)
```

```
   {
     Socket conexion = servidor.accept();
     Worker w = new Worker(conexion);
     w.start();
   }
  }
}
```

Este código será la base para los programas que desarrollaremos en el curso.

Ahora el constructor de la clase **Worker** pasa como parámetro el socket que crea el método **accept**, ya que el método **run** requiere el socket para recibir y enviar datos al cliente.

La implementación completa del servidor se puede encontrar en el programa Servidor2.java.

Podemos ver en el programa Servidor2.java que el método **run** crea los streams que se utilizarán para enviar y recibir datos del cliente. Notar que el programa Servidor2.java es completamente compatible con el programa Cliente.java

Un cliente con re-intentos de conexión

Como vimos la clase pasada, para que el cliente se conecte al servidor, es necesario que el servidor inicie su ejecución antes que el cliente, sin embargo para algunas aplicaciones el cliente debe esperar a que el servidor inicie su ejecución.

En el programa Cliente2.java podemos ver cómo implementar el re-intento de conexión cuando el servidor no está ejecutando.

```
Socket conexion = null;
for(;;)
 try
 {
   conexion = new Socket("localhost",50000);
   break;
 }
 catch (Exception e)
 {
   Thread.sleep(100);
```

}

Como podemos ver, cada vez que el cliente falla en establecer la conexión con el servidor, espera 100 milisegundos y vuelve a intentar la conexión. Cuando el cliente logra conectarse con el servidor entonces sale del ciclo for.

Actividades individuales a realizar

¿Por qué el programa sin sincronización despliega un valor incorrecto?

¿Por qué cada vez que se ejecuta el programa sin sincronización despliega un valor diferente?

¿Por qué el programa con sincronización es más lento?

Clase del día - 01/03/2021

La clase de hoy vamos a ver el tema de jerarquía de memoria y vamos a estudiar dos conceptos muy importantes relacionados con la cache: la localidad espacial y la localidad temporal.

Jerarquía de memoria

La jerarquía de memoria puede verse como una pirámide dónde cada nivel representa una capa de hardware que almacena datos.

El CPU utiliza los **registros** para realizar operaciones aritméticas, lógicas y de control.

La memoria **cache** consiste en una memoria asociativa. Las memorias asociativas son muy rápidas, pero como son costosas, suelen ser de poca capacidad. La memoria cache puede estar dividida en varios niveles L1, L2, ...

La **memoria RAM** (Random Access Memory) suele ser es una memoria dinámica, por lo que requiere tener alimentación eléctrica constante para conservar los datos. Para escribir o leer una localidad de memoria en la RAM es necesario indicar la dirección de la localidad.

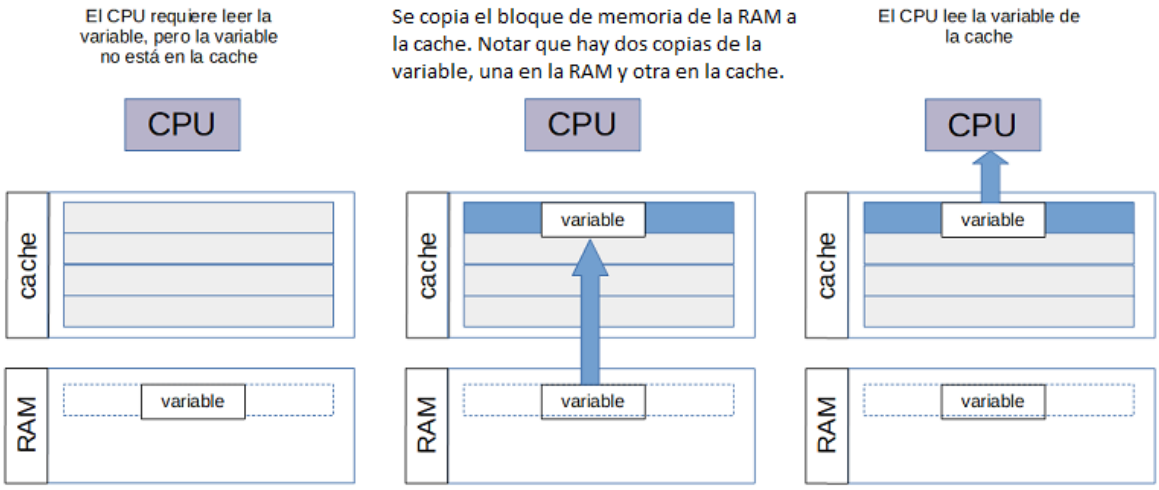
El **disco duro** almacena de manera persistente grandes cantidades de datos.

Los **respaldos** pueden ser discos duros de gran capacidad, discos ópticos, cintas, entre otros.

La memoria cache

Lectura de una variable

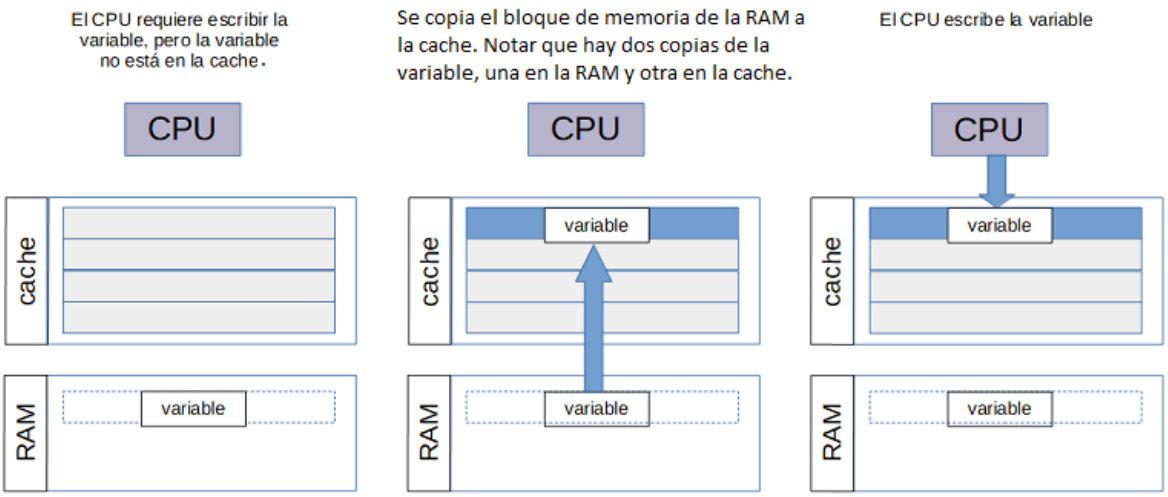
Cuando el CPU requiere leer una variable que se encuentra en la memoria RAM, busca la variable en la cache, si la variable no existe en la cache, copia el bloque de datos (que contiene a la variable) a la cache, entonces el CPU lee la variable de la cache.



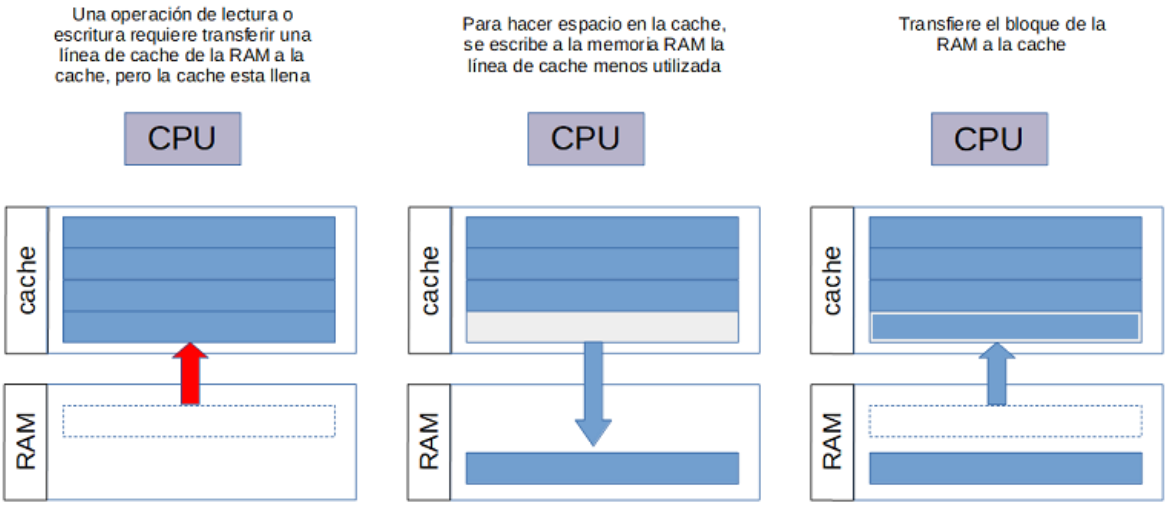
Al bloque de memoria que se transfiere de la RAM a la cache se le llama **línea de cache**. El tamaño de una línea de cache típicamente es de cientos de Kilobytes o Megabytes.

Escritura de una variable

Por otra parte, si el CPU requiere escribir una variable, busca la variable en la memoria cache, si existe, escribe el valor de la variable en la cache, si no existe, entonces copia la línea de cache (que contiene la variable) de la memoria RAM a la cache, y luego escribe el valor de la variable que está en la cache.



Debido a que la cache tiene un tamaño limitado (del orden de Megabytes), eventualmente se llenará. Para liberar líneas, la cache escribe a la memoria RAM las líneas menos utilizadas.



Como podemos ver, el CPU nunca lee o escribe datos directamente a la memoria RAM.

Así mismo, la cache nunca lee o escribe variables individuales a la memoria RAM, sino que siempre la transferencia de datos entre la cache y la memoria RAM se realiza en bloques (líneas de cache).

Localidad espacial y localidad temporal

Supongamos que el jefe de Recursos Humanos de una empresa le pide a su asistente los expedientes de algunos empleados en diferentes momentos del día.

Los expedientes se encuentran almacenados en cajas en el archivo de personal.

Cada caja contiene los expedientes organizados por apellido paterno, es decir, una caja contiene los expedientes de los empleados cuyo apellido paterno inicia con "A", otra caja contiene los expedientes de los empleados cuyo apellido paterno inicia con "B", y así sucesivamente.

La asistente puede ir al archivo de personal a traer un expediente o traer una caja completa.

En términos computacionales:

- Los expedientes representan los datos que se transfieren de la memoria RAM a la cache.
- El archivo dónde se encuentran los expedientes representa la memoria RAM.
- Una caja de expedientes representa una línea de cache.
- El jefe representa el CPU.

Consideremos tres casos:

Caso 1. La asistente obtiene expedientes individuales del archivo.

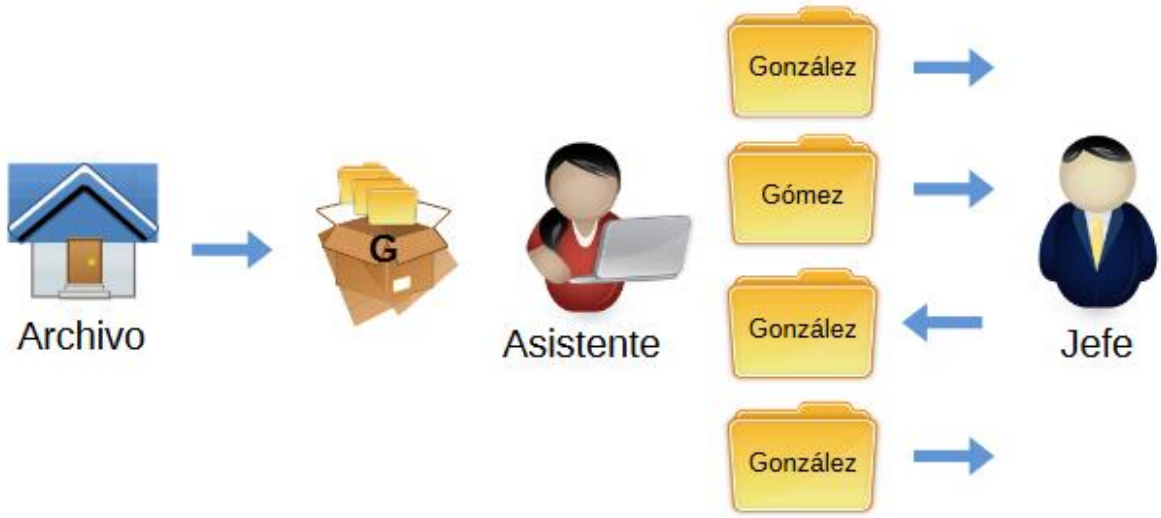
- El jefe le pide a su asistente el expediente del Sr. González.
- La asistente va al archivo a traer el expediente del Sr. González.
- La asistente le da a su jefe el expediente del Sr. González
- El jefe le pide a su asistente el expediente del Sr. Gómez.
- La asistente va al archivo a traer el expediente del Sr. Gómez.
- La asistente le da a su jefe el expediente del Sr. Gómez.
- El jefe regresa a su asistente el expediente del Sr. González.
- La asistente va al archivo a dejar el expediente del Sr. González
- El jefe le pide a su asistente el expediente del Sr. González.
- La asistente va al archivo a traer el expediente del Sr. González.
- La asistente le da a su jefe el expediente del Sr. González.

Entonces la asistente tiene que ir cuatro veces al archivo.

Caso 2. La asistente obtiene cajas de expedientes del archivo.

- El jefe le pide a su asistente el expediente del Sr. González.
- La asistente va al archivo a traer la caja correspondiente a la letra ”G”.
- La asistente le da a su jefe el expediente del Sr. González.
- El jefe le pide a su asistente el expediente del Sr. Gómez.
- La asistente le da a su jefe el expediente del Sr. Gómez.
- El jefe regresa el expediente del Sr. González.

- El jefe le pide a su asistente el expediente del Sr. González.
- La asistente le da a su jefe el expediente del Sr. González.



Entonces la asistente sólo va una vez al archivo. Los expedientes solicitados por el jefe se encuentran en la misma caja. El jefe pide más de una vez el expediente del Sr. González el mismo día.

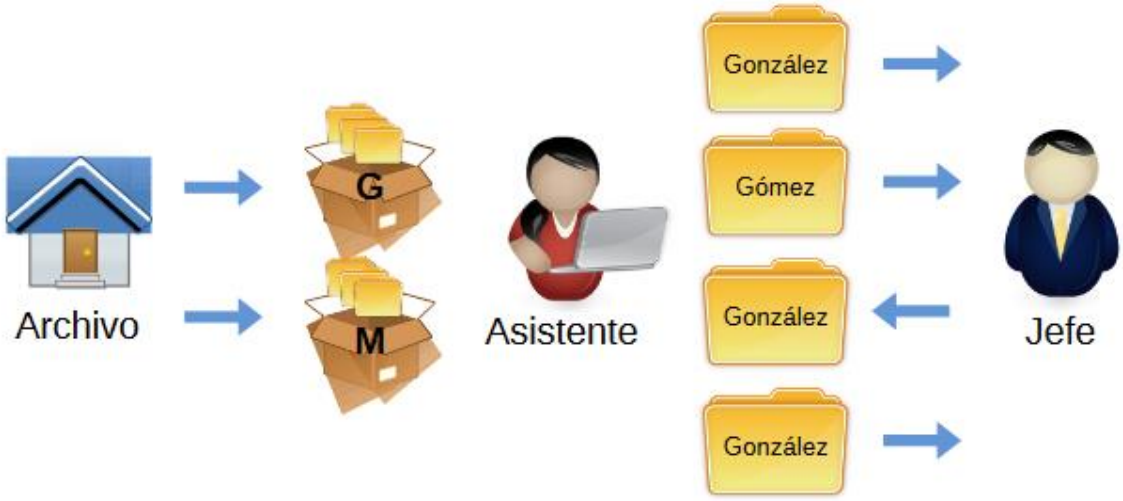
La asistente ha descubierto los conceptos de localidad espacial y localidad temporal.

- Los datos presentan **localidad espacial** si al acceder un dato existe una elevada probabilidad de que datos cercanos sean accedidos poco tiempo después. En el ejemplo, los expedientes solicitados por el jefe presentan localidad espacial ya que se encuentran en la misma caja (digamos, la misma línea de cache).
- Un dato presenta **localidad temporal** si después de acceder el dato existe una elevada probabilidad de que el mismo dato sea accedido poco tiempo después. En el ejemplo, el expediente del Sr. González presenta localidad temporal ya que el jefe lo pide más de una vez el mismo día.

Caso 3. La asistente obtiene cajas de expedientes del archivo.

- El jefe le pide a su asistente el expediente del Sr. González.
- La asistente va al archivo a traer la caja correspondiente a la letra “G”.
- La asistente le da a su jefe el expediente del Sr. González.
- El jefe le pide a su asistente el expediente del Sr. Morales.
- La asistente va al archivo a traer la caja correspondiente a la letra “M”.
- La asistente le da a su jefe el expediente del Sr. Morales.
- El jefe regresa el expediente del Sr. González.

- El jefe le pide a su asistente el expediente del Sr. González.
- La asistente le da a su jefe el expediente del Sr. González.



Entonces la asistente tiene que ir dos veces al archivo. Los expedientes del Sr. González y del Sr. Morales no presentan localidad espacial ya que se encuentran en diferentes cajas. El expediente del Sr. González presenta localidad temporal ya que el jefe lo pide más de una vez el mismo día.

Analicemos qué pasa en cada caso:

- Caso 1. En las primeras computadoras no había cache, por tanto el CPU accedia directamente los datos en la memoria. Debido a que la memoria era muy lenta, el CPU tenía que esperar mucho tiempo a que se leyera y/o escribieran los datos en la memoria RAM.
- Caso 2. La cache intercambia bloques de datos con la RAM. Dado que los datos presentan localidad espacial y localidad temporal, se reduce substancialmente los accesos a la memoria RAM, lo cual aumenta la eficiencia del programa ya que la RAM es una memoria lenta comparada con la cache.
- Caso 3. Los datos no presentan localidad espacial por tanto la cache transfiere bloques completos cada vez que se requiere leer o escribir un dato. En este caso tener la cache resulta más ineficiente que no tenerla (como sería en el caso 1).

La conclusión a la que llegamos es la siguiente: **la cache solo es de utilidad cuando los datos presentan localidad espacial y/o localidad temporal**.

Sin embargo, la cache no se puede "apagar", por tanto es necesario saber programar para la cache, o en otras palabras, es necesario que los programas presenten la máxima localidad espacial y/o localidad temporal.

Ahora veremos un ejemplo de cómo programar tomando en cuenta la cache.

Caso de estudio: Multiplicación de matrices

Como se explicó anteriormente, la cache acelera el acceso a los datos que presentan localidad espacial y/o localidad temporal, sin embargo no siempre los algoritmos están diseñados para acceder a los datos de manera que se privilegie el acceso a la memoria en forma secuencia (localidad espacial) Vs. el acceso a la memoria en forma dispersa.

El siguiente programa multiplica dos matrices cuadradas A y B utilizando el algoritmo estándar (renglón por columna), en este caso las matrices tienen un tamaño de 1000x1000:

```
class MultiplicaMatriz
{
  static int N = 1000;
  static int[][] A = new int[N][N];
  static int[][] B = new int[N][N];
  static int[][] C = new int[N][N];

  public static void main(String[] args)
  {
    long t1 = System.currentTimeMillis();

    // inicializa las matrices A y B

    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
      {
        A[i][j] = 2 * i - j;
        B[i][j] = i + 2 * j;
        C[i][j] = 0;
      }

    // multiplica la matriz A y la matriz B, el resultado queda en la matriz C

    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
          C[i][j] += A[i][k] * B[k][j];

    long t2 = System.currentTimeMillis();

    System.out.println("Tiempo: " + (t2 - t1) + "ms");
```

Es necesario tomar en cuenta que Java almacena las matrices en la memoria como renglones, por lo que el acceso a la matriz B (por columna) es muy ineficiente si las matrices son muy grandes, ya que cada vez que se accede un elemento de la matriz B, se transfiere una línea de cache completa de la RAM a la cache.

El acceso a la matriz A es muy eficiente debido a que los elementos de la matriz A se leen secuencialmente, es decir, el acceso es por renglón, tal como la matriz se encuentra almacenada en la memoria.

Ahora vamos a modificar el algoritmo de multiplicación de matrices de manera que incrementemos la localidad espacial haciendo que el acceso a la matriz B sea por renglones y no por columnas.

El cambio es muy simple, solamente necesitamos intercambiar los índices que usamos para acceder los elementos de la matriz B, la cual previamente hemos transpuesto (es necesario transponer la matriz B para que el algoritmo siga calculando el producto de las matrices).

```
class MultiplicaMatriz_2
{
  static int N = 1000;
  static int[][] A = new int[N][N];
  static int[][] B = new int[N][N];
  static int[][] C = new int[N][N];

  public static void main(String[] args)
  {
    long t1 = System.currentTimeMillis();

    // inicializa las matrices A y B

    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
      {
        A[i][j] = 2 * i - j;
        B[i][j] = i + 2 * j;
        C[i][j] = 0;
      }

    // transpone la matriz B, la matriz traspuesta queda en B

    for (int i = 0; i < N; i++)
      for (int j = 0; j < i; j++)
      {
        int x = B[i][j];
        B[i][j] = B[j][i];
        B[j][i] = x;
      }

    // multiplica la matriz A y la matriz B, el resultado queda en la matriz C
    // notar que los indices de la matriz B se han intercambiado

    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
          C[i][j] += A[i][k] * B[j][k];

    long t2 = System.currentTimeMillis();
    System.out.println("Tiempo: " + (t2 - t1) + "ms");
```

```
  }
}
```

El resultado es un acceso más eficiente a los elementos de la matriz B, debido a que ahora se leen los elementos de B en forma secuencial, lo cual aumenta la localidad espacial y temporal de los datos.

Al ejecutar los programas MultiplicaMatriz.java y MultiplicaMatriz_2.java para diferentes tamaños de las matrices, se puede observar que el algoritmo que accede ambas matrices por renglones (el segundo programa) es mucho más eficiente, ya que en éste algoritmo la localidad espacial y la localidad temporal de los datos es mayor debido a que las matrices A y B son accedidas por renglones, tal como se almacenan en la memoria por Java.

¿Por qué el segundo programa es más rápido que el primero?

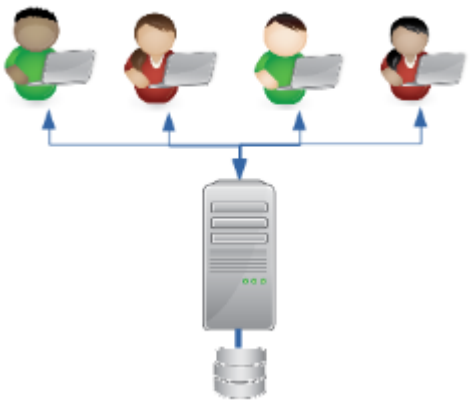
¿Podría plantear otro programa dónde el aumento de la localidad espacial y/o temporal hace más eficiente la ejecución?

Clase del día - 04/03/2021

La clase de hoy vamos a ver los conceptos de sistema centralizado y sistema distribuido.

Sistema centralizado

Un sistema centralizado es aquel dónde el código y los datos residen en una sola computadora.



Un sistema centralizado tiene las siguientes ventajas:

Facilidad de programación. Los sistemas centralizados son fáciles de programar, ya que no existe el problema de comunicar diferentes procesos en diferentes computadoras, tampoco es un problema la consistencia de los datos debido a que todos los procesos ejecutan en una misma computadora con una sola memoria.

Facilidad de instalación. Es fácil instalar un sistema central. Basta con instalar un solo *site* el cual va a requerir una acometida de energía eléctrica, un sistema de enfriamiento (generalmente por agua), conexión a la red de datos y comunicación por voz. Más adelante en el curso veremos cómo el cómputo en la nube está cambiando la idea de instalación física en pos de sistemas virtuales en la nube.

Facilidad de operación. Es fácil operar un sistema central, ya que la administración la realiza un solo equipo de operadores, incluyendo las tareas de respaldos, mantenimiento preventivo y correctivo, actualización de versiones, entre otras.

Seguridad. Es fácil garantizar la seguridad física y lógica de un sistema centralizado. La seguridad física se implementa mediante sistemas CCTV, controles de cerraduras electrónicas, biométricos, etc. La seguridad lógica se implementa mediante un esquema de permisos a los diferentes recursos como son el sistema operativo, los archivos, las bases de datos.

Bajo costo. Dados los factores anteriores, instalar un sistema centralizado resulta más barato que un sistema distribuido ya que solo se pagan licencias para un servidor, sólo se instala un *site*, se tiene un solo equipo de operadores.

Por otra parte, un sistema centralizado tiene las siguientes desventajas:

El procesamiento es limitado. El sistema centralizado cuenta con un número limitado de procesadores, por tanto a medida que incrementamos el número de procesos en ejecución, cada proceso ejecutará más lentamente. Por ejemplo, en Windows podemos ejecutar el Administrador de Tareas para ver el porcentaje de CPU que utiliza cada proceso en ejecución, si la computadora ha llegado a su límite, entonces veremos que el porcentaje de uso del CPU es 100%.

El almacenamiento es limitado. Un sistema centralizado cuenta con un número limitado de unidades de almacenamiento (discos duros). Cuando un sistema llega al límite del almacenamiento se detiene, ya que no es posible agregar datos a los archivos ni realizar *swap*.

El ancho de banda es limitado. Un sistema centralizado puede llegar al límite en el ancho de banda de entrada y/o de salida, en estas condiciones la comunicación con los usuarios se va a alentar.

El número de usuarios es limitado. Un sistema centralizado tiene un máximo de usuarios que se pueden conectar o que pueden consumir los servicios. Por ejemplo, por razones de licenciamiento los manejadores de bases de datos tienen un máximo de usuarios que pueden conectarse, así mismo, el sistema operativo tiene un límite en el número de *descriptores de archivos* que puede crear. Recordemos que cada vez que se abre un archivo y cada vez que se crea un socket se ocupa un descriptor de archivo.

Baja tolerancia a fallas. En un sistema centralizada una falla suele ser catastrófica, ya que sólo se tiene una computadora y una memoria. Cualquier falla suele producir la inhabilitación del sistema completo.

Ejemplos de sistemas centralizados

Un servidor Web centralizado

Actualmente los servidores Web suelen ser distribuidos, ya que resulta muy sencillo redirigir las peticiones a múltiples servidores utilizando un balanceador de carga. Sin embargo, todavía los sitios Web pequeños utilizan un servidor centralizado debido a su bajo costo.

Un DBMS centralizado

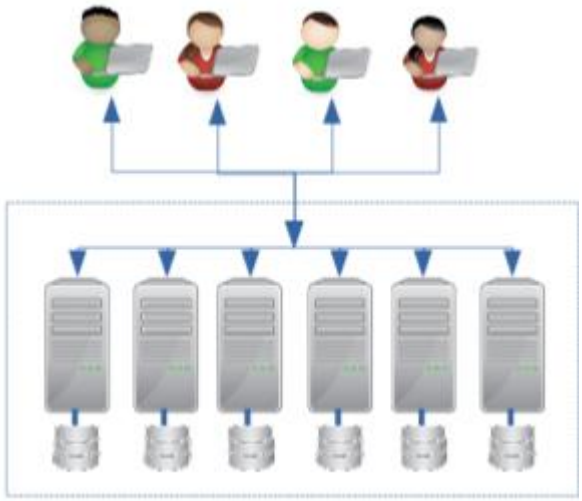
Generalmente los sistemas manejadores de bases de datos (DBMS) son centralizados debido a que resulta más fácil programar sistemas que accedan los datos que se encuentran en una base de datos central. Sin embargo, las plataformas de alcance mundial como Facebook, Twitter, Uber, etc. requieren distribuir los datos en diferentes localizaciones por razones de rendimiento.

Una computadora stand-alone

Una computadora *stand-alone* se refiere a un sistema único integrado. Generalmente entendemos una computadora personal como un sistema stand-alone ya que integra el CPU con un teclado, un monitor, una impresora, etc. Un sistema único es por antonomasia un sistema centralizado.

Sistema distribuido

"*Un sistema distribuido es una colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente.*" Andrew S. Tanenbaum



Esta definición de sistema distribuido implica que el usuario de un sistema distribuido tiene la impresión de estar utilizando un sistema central no obstante el sistema estaría compuesto de múltiples servidores interconectados.

La definición anterior tiene importantes implicaciones desde el punto de vista técnico. El hacer que una colección de computadoras se comporten como un sistema único requiere implementar mecanismos de memoria compartida distribuida, migración de procesos, sistemas de archivos distribuidos, entre muchas tecnologías.

De alguna forma, los sistemas distribuidos son antónimos de los sistemas centralizados, de manera que las desventajas de un sistema central son ventajas en un sistema distribuido y viceversa.

Las ventajas de un sistema distribuido son, entre otras:

El procesamiento es (casi) ilimitado. Un sistema distribuido puede tener un número casi ilimitado de CPUs ya que siempre será posible agregar más servidores, por tanto a medida que incrementamos el número de CPUs podemos esperar que los procesos ejecuten más rápido

debido a que los procesos ejecutarán en paralelo en diferentes CPUs. El límite del paralelismo queda definido por la **ley de Amdahl**.

El almacenamiento es (casi) ilimitado. Un sistema distribuido cuenta con un número casi ilimitado de unidades de almacenamiento (discos duros). Siempre es posible conectar más servidores de almacenamiento.

El ancho de banda es (casi) ilimitado. En un sistema distribuido cada computadora aporta su ancho de banda, esto es, en la medida que agregamos servidores podemos enviar y recibir una mayor cantidad de datos por unidad de tiempo (es decir, aumentamos el ancho de banda).

El número de usuarios es (casi) ilimitado. El número de usuarios que pueden conectarse a un sistema distribuido aumenta en la medida que agregamos servidores. Si bien es cierto que cada servidor tiene un límite en el número de *descriptores de archivos*, y con ello un límite al número de conexiones que puede abrir, cada servidor en el sistema distribuido agrega descriptores (conexiones).

Alta tolerancia a fallas. En un sistema distribuido la falla de un servidor no es catastrófica, ya que el sistema está diseñado para retomar el trabajo que realizaba el servidor que falla. Más adelante en el curso veremos las estrategias que se utilizan en los sistemas distribuidos para la replicación de datos y la replicación del sistema completo.

Las desventajas de los sistemas distribuidos son:

Dificultad de programación. La definición que hace Tanenbaum de los sistemas distribuidos implica que los usuarios del sistema tienen la impresión de utilizar un sistema único, esto incluye a los programadores. Sin embargo, en la realidad actual los sistemas distribuidos son difíciles de programar ya que el programador es quienel que tiene que implementar la comunicación entre los diferentes componentes del sistema.

Dificultad de instalación. Es complicado instalar un sistema distribuido. Es necesario interconectar múltiples computadoras, lo cual implica la necesidad de una red de alta velocidad.

Dificultad de operación. Es complicado operar un sistema distribuido, ya que se requiere un equipo de administración por cada *site*. Los equipos deberán coordinarse para realizar las tareas de respaldos, mantenimiento preventivo y correctivo, actualización de versiones, entre otras.

Seguridad. Es complicado garantizar la seguridad física y lógica de un sistema distribuido. Tanto la seguridad física como la seguridad lógica requieren la coordinación de múltiples equipos dedicados a la seguridad del sistema. La interconexión remota de los diferentes servidores implica el riesgo de ataques al sistema a través de los puertos de comunicación.

Alto costo. Instalar un sistema distribuido resulta más costoso que un sistema centralizado ya que será necesario pagar licencias para cada servidor, para cada *site* se requiere un equipo de operadores, así mismo, cada *site* requiere su propia acometida de energía, un sistema de seguridad física, infraestructura de refrigeración, etc.

Tipos de distribución

Distribución del procesamiento

La distribución del procesamiento permite repartir el cómputo entre diferentes servidores. La distribución del procesamiento se utiliza para el cómputo de alto rendimiento (*HPC: High Performance Computing*), para la implementación de sistemas tolerantes a fallas y para el balance de carga

En el **cómputo de alto rendimiento** los programas se ejecutan en forma distribuida, dividiendo el problema en componentes los cuales se ejecutan en paralelo en diferentes servidores. La clave para obtener rendimientos superiores es que los servidores se conecten mediante una red de alta velocidad.

La distribución del procesamiento permite implementar **sistemas tolerantes a fallas**. Algunos ejemplos de sistemas tolerantes a fallas son los programas que ejecutan en un avión o en una central nuclear. En estos casos los procesos se replican en diferentes computadoras, si una computadora falla entonces el proceso sigue ejecutando en otra computadora.

En el caso de los servidores Web el procesamientos de las peticiones se distribuye con el propósito de **balancear la carga** y evitar que un servidor se sature.

Distribución de los datos

La distribución de los datos aumenta la **confiabilidad** del sistema, ya que si falla el acceso a una parte o copia de los datos es posible seguir trabajando con otra parte o copia de los datos.

La distribución de datos también mejora el **rendimiento** de un sistema distribuido que requiere escalar en tamaño y geografía. Es una buena práctica distribuir los catálogos de los sistemas (por ejemplo, los catálogos de clientes, de productos, de cuentas, etc.) ya que se trata de datos que se modifican poco, por tanto en un sistema distribuido resulta más rápido el acceso a estos datos si los tiene cerca.

Ejemplos de sistemas distribuidos

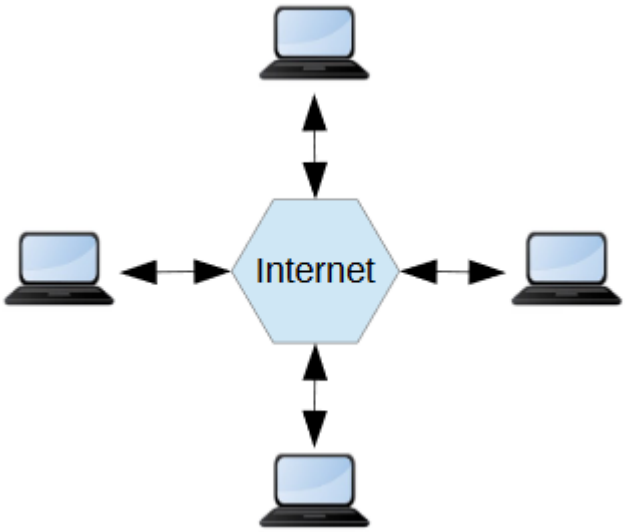
World Wide Web

La web es un sistema distribuido compuesto por servidores (web) y clientes (navegadores) que se conectan a los servidores.

La web permite la distribución a nivel mundial de documentos hipertexto (páginas web) escritos en lenguaje HTML (*Hypertext Markup Language*).

En la web un URL (*Uniform Resource Locator*) permite identificar de manera única a nivel mundial un recurso (página web, imagen, video, etc.).

El protocolo que utiliza el cliente y servidor para comunicarse es HTTP (*Hypertext Transfer Protocol*) el cual funciona sobre el protocolo TCP (*Transfer Control Protocol*).



Cómputo en la nube

En 2006 aparece en la revista Wired el artículo The Information Factories de George Gilder que describe un nuevo modelo de arquitectura basado en una infraestructura de cómputo ofrecida como servicios virtuales a nivel masivo, a este nuevo modelo se le llamó *cloud computing* (cómputo en la nube).

El concepto clave en el cómputo en la nube es el "servicio":

Infrastructure as a Service (**IaaS**): infraestructura virtual, sistema operativo y red.

Platform as a Service (**PaaS**): DBMS, plataformas de desarrollo y pruebas como servicio.

Software as a Service (**SaaS**): aplicaciones de software como servicio

SETI

Seach for Extra-Terrestrial Intelligence (SETI) es un proyecto de la Universidad de Berkeley que integra al rededor de 290,000 (2009) computadoras buscando patrones "inteligentes" en señales obtenidas de radiotelescopios. El sistema alcanza los 617 TFlop/s (1 TFlop/s = 10^{12} operaciones de punto flotante por segundo).

TOP500

Actualmente la computadora más grande del mundo tiene 7,630,848 procesadores con Linux Red Hat; no se trata de una computadora centralizada sino de un sistema distribuido, alcanzando un rendimiento pico de 537,212.0 TFlop/s

.

Investigar la ley de Amdahl.

Considere la lista del TOP500 ¿En qué lugar aparece la primera computadora con Windows?
ver: TOP500 List Statistics

Clase del día - 09/03/2021

Objetivos de los sistemas distribuidos

Como vimos la clase anterior los sistemas distribuidos tienen grandes ventajas sobre los sistemas centralizados.

Sin embargo, los sistemas distribuidos también tienen algunas desventajas que podemos resumir en su alta complejidad y costo. Por esta razón, es muy importante establecer claramente los objetivos de un sistema distribuido antes de su implementación.

En general, un sistema distribuido deberá cumplir los siguientes objetivos:

- Facilidad en el acceso a los recursos.
- Transparencia.
- Apertura.
- Escalabilidad.

1. Facilidad en el acceso a los recursos

Es de la mayor importancia en un sistema distribuido facilitar a los usuarios y a las aplicaciones el acceso a los recursos remotos. Entendemos como recurso el CPU, la memoria RAM, las unidades de almacenamiento, las impresoras, los DBMS, los archivos, o cualquier otra entidad lógica o física que preste un servicio en el sistema distribuido.

En un sistema distribuido se comparten los recursos por razones técnicas y por razones económicas.

En el primer caso, se comparten recursos por **razones técnicas** cuando tenemos procesos que ejecutan en forma distribuida utilizando datos que se encuentran distribuidos geográficamente, o bien, procesos que requieren la distribución del cálculo en diferentes CPUs, o procesos de facturación que envían la impresión de facturas a múltiples impresoras.

En el segundo caso, se comparten recursos por **razones económicas** debido a su alto costo.

Por ejemplo, la virtualización permite compartir los recursos de una computadora como son el CPU, la memoria, y las unidades de almacenamiento, creando entornos de ejecución llamados máquinas virtuales. La virtualización aumenta el porcentaje de utilización de los recursos de la computadora y con ello se obtiene un mayor beneficio dado el costo de los recursos.

Sin embargo, compartir recursos conlleva un compromiso en la seguridad, ya que será necesario implementar mecanismos de **comunicación segura** (SSL, TLS o HTTPS), esquemas para la confirmación de la identidad (**autenticación**) y esquemas de permisos para el acceso a los recursos (**autorización**).

2. Transparencia

La transparencia es la capacidad de un sistema distribuido de presentarse ante los usuarios y aplicaciones como una sola computadora.

Tipos de transparencia

Podemos dividir la transparencia de un sistema distribuido en siete categorías:

2.1 Transparencia en el acceso a los datos. Un sistema distribuido deberá proveer de una capa que permita a los usuario y aplicaciones acceder a los datos de manera estandarizada. Por ejemplo, un servicio que permita acceder a los archivos que residen en computadoras con diferentes sistemas operativos mediante nombres estandarizados, independientemente del tipos de nomenclatura implementada en cada sistema operativo.

2.2 Transparencia de ubicación. En un sistema distribuido los usuarios acceden a los recursos independientemente de su localización física. Por ejemplo, una URL identifica un recurso en la Web de manera única. Así, https://m4gm.com/moodle/curso.txt es la URL del archivo "curso.txt" localizado en el directorio "moodle" de la computadora cuyo dominio es "m4gm.com". Adicionalmente, la URL indica el protocolo que utilizará para acceder el recurso, en este caso el protocolo es HTTPS.

2.3 Transparencia de migración. En alguno sistemas distribuidos es posible migrar recursos de un sitio a otro. Si la migración del recurso no afecta la forma en que se accede el recurso, se dice que el sistema soporta la transparencia de migración. Por ejemplo, si un sistema permite la migración transparente de procesos de una computadora a otra como una estrategia de tolerancia de fallas, los usuarios y procesos no se verán afectados ante la migración del proceso que acceden.

2.4 Transparencia de re-ubicación. La transparencia de re-ubicación se refiere a la capacidad del sistema distribuido de cambiar la ubicación de un recurso mientras está en uso, sin que el usuario que accede el recurso se vea afectado. Por ejemplo, en UNIX (Linux) para cambiar la ubicación de un proceso en ejecución, primero se le envía al proceso un signal SIGSTOP en la ubicación de origen, el proceso se migra a la ubicación de destino, se envía al proceso un signal SIGCONT en la ubicación de destino, entonces el proceso sigue ejecutando desde el punto en que se quedó.

2.5 Transparencia de replicación. La transparencia de replicación es la capacidad del sistema distribuido de ocultar la existencia de recursos replicados. Por ejemplo, la repilcación de los datos es una estrategia que permite aumentar la confiabilidad y la rendimiento en los sistemas distribuidos.

2.6 Transparencia de concurrencia. En una computadora todos los recursos son compartidos. La transparencia de concurrencia se refiere a la capacidad de un sistema de ocultar el hecho de que varios usuarios y procesos comparten los diferentes recursos. Por ejemplo, un sistema operativo multi-tarea oculta el hecho de que varios procesos utilizan de manera concurrente el CPU, la memoria, los discos duros, etc. Por otra parte, un sistema operativo multi-usuario oculta el hecho de que la computadora es utilizada por múltiples usuarios de manera concurrente.

2.7 Transparencia ante fallas. La transparencia ante fallas es la capacidad del sistema distribuido de ocultar una falla. Como vimos anteriormente, la distribución del procesamiento permite implementar sistemas tolerantes a fallas. Por ejemplo, en un sistema que se encuentra totalmente

replicado, si el sistema principal falla entonces el usuario accederá de manera transparente a la réplica del sistema. Más adelante en el curso veremos cómo replicar un sistema completo en la nube.

3. Apertura

Un sistema abierto es aquel que ofrece servicios a través de reglas de sintaxis y semántica estándares.

Las reglas de sintaxis generalmente se definen mediante un **lenguaje de definición de interfaz**, en el cual se especifica los nombres de las operaciones del servicio, nombre y tipo de los parámetros, valores de retorno, posibles excepciones, entre otros elementos que sean de utilidad para automatizar la comunicación entre el cliente del servicio y el servidor..

La semántica (funcionalidad) de las operaciones de un servicio se define generalmente de manera informal utilizando lenguaje natural.

Los sistemas abiertos exhiben tres características que los hacen más populares que los sistemas propietarios, estas características son: interoperabilidad, portabilidad y extensibilidad.

La definición de las reglas de sintaxis permite que diferentes sistemas puedan interaccionar. A la capacidad de sistemas diferentes de trabajar de manera interactiva se le llama **interoperabilidad**. Por ejemplo, un servicio web escrito en Java, Python o en C# puede ser utilizado indistintamente por un cliente escrito en JavaScript, Java, Python, o C#.

La **portabilidad** (*cross-platform*) de un programa se refiere a la posibilidad de ejecutar el programa en diferentes plataformas sin la necesidad de hacer cambios al programa. Por ejemplo un programa escrito en Java puede ser ejecutado sin cambios en cualquier plataforma que tenga instalado el JRE (Java Runtime Environment). En 1995 Sun Microsystems explicó la portabilidad de los programas escritos en Java con la siguiente frase: "*Write once, run everywhere*".

La **extensibilidad** se refiere a la capacidad de los sistemas de crecer mediante la incorporación de componentes fáciles de reemplazar y adaptar. Más adelante en el curso veremos cómo desarrollar sistemas extensibles mediante objetos de Java distribuidos.

4. Escalabilidad

La **escalabilidad** es la capacidad de un sistema de crecer sin reducir su calidad.

Un sistema puede escalar en tres aspectos principales: tamaño, geografía y administración.

4.1 Escalar en tamaño

Cuando un sistema requiere atender más usuarios o ejecutar procesos más demandantes, es necesario agregar más CPUs, más memoria, mas unidades de almacenamiento o incrementar el ancho de banda de la red. Es decir, el sistema requiere escalar en tamaño.

Sin embargo, un sistema centralizado solo puede crecer hasta alcanzar el número máximo de CPUs, la cantidad máxima de memoria RAM, el número máximo de controladores de disco duro y el máximo ancho de banda que puede ofertar el ISP (*Internet Service Provider*).

4.2 Escalar geográficamente

En la actualidad las empresas globales requieren operar sus sistemas en múltiples regiones geográficas. Si la empresa cuenta solamente con un sistema central, los usuarios tendrán que conectarse desde ubicaciones remotas por lo que se incrementará los tiempos de respuesta debido a la latencia de la red.

Entonces surge la necesidad de escalar geográficamente los sistemas, por tanto será necesario instalar servidores en diferentes ubicaciones estratégicamente localizadas con el fin de reducir los tiempos de respuesta. Generalmente se divide el mundo en regiones (América del Norte, América del Sur, Europa, Asia, África) y se instala un centro de datos en cada región. Si la región es de alta demanda (como es el caso de América del Norte y Europa) se suele instalar más centros de datos en la misma región.

4.3 Escalar la administración

Cuando un sistema crece en tamaño y geografía, también aumenta la complejidad en la administración del sistema.

Un sistema más grande implica más computadoras, más CPUs, más tarjetas de memoria RAM, más unidades de almacenamiento, más concentradores de red, en suma, más componentes que pueden fallar, más información que se tiene que respaldar, más usuarios, más permisos que controlar, etc. En resumen, para crecer el sistema se requiere escalar así mismo la administración.

Técnicas de escalamiento

Ahora veremos brevemente algunas técnicas utilizadas para escalar los sistemas.

1. Ocultar la latencia en las comunicaciones

La latencia en las comunicaciones es el tiempo que tarda un mensaje en ir del origen al destino. Existen múltiples factores que influyen en la latencia de las comunicaciones, como son el tamaño de los mensajes, la capacidad de los enrutadores, la distancia, la hora del día, la época del año, etc.

La latencia en las comunicaciones aumenta el tiempo de espera cuando se hace una petición a un servidor remoto.

Una estrategia que se utiliza para ocultar la latencia en las comunicaciones, es el uso de **peticiones asíncronas**.

Supongamos que una aplicación realiza una petición a un servidor cuándo el usuario presiona un botón, si la petición es sincrona el usuario debe esperar a que el servidor envíe la respuesta, ya que la aplicación no puede ejecutar otra tarea mientras espera.

Por otra parte, si la petición es asíncrona, la aplicación puede ejecutar otras tareas. Por ejemplo, en Android todas las peticiones que se realizan a los servidores deben ser asíncronas, lo cual garantiza que las aplicaciones siguen respondiendo al usuario mientras esperan la respuesta del servidor.

2. Distribución

Una técnica muy utilizada para escalar un sistema es la distribución. Para distribuir un sistema se divide en partes más pequeñas las cuales se ejecutan en diferentes servidores.

Por ejemplo, supongamos que una empresa tiene una plataforma de comercio electrónico en la Web, cuando la empresa comienza a tener operaciones globales surge la necesidad de escalar la plataforma de comercio electrónico, para ello se puede distribuir el sistema en distintos servidores.

3. Replicación

Otra técnica utilizada para escalar un sistema es la replicación de los procesos y de los datos.

Replicar los procesos en diferentes computadoras permite liberar de trabajo las computadoras más saturadas, es decir, balancear la carga en el sistema.

Replicar los datos en diferentes computadoras permite acceder a los datos más rápidamente, debido a que con ello se evitan los cuellos de botella en los servidores.

Para replicar los datos se puede utilizar caches que aprovechen la localidad espacial y temporal de los datos.

Por ejemplo, si un archivo se utiliza con frecuencia, es conveniente tener una copia en una cache local. En el caso de que el archivo sea modificado en el servidor, entonces éste enviará un mensaje de **invalidación de cache**, lo que significa que el archivo deberá ser eliminado de la cache local. Si posteriormente el cliente requiere el archivo, deberá solicitarlo al servidor y con ello contará con el archivo actualizado.

4. Elasticidad

Posiblemente la técnica más interesante para escalar un sistema es la elasticidad en la nube.
La **elasticidad** es la adaptación a los cambios en la carga mediante aprovisionamiento y des-aprovisionamiento de recursos en forma automática.

Supongamos un servicio de *streaming* bajo demanda, como es el caso de Netflix. En este tipo de servicio la demanda crece los fines de semana y decrece los días entre semana. Si el proveedor de servicio no aprovisiona los recursos suficientes para atender la demanda del fin de semana, entonces muchos usuarios se quedarán sin servicio.

Por otra parte, si el proveedor del servicio aprovisiona los recursos necesarios para atender a sus usuarios el fin de semana, estos recursos estarán sub-utilizados los días entre semana, lo cual resulta en pérdidas económicas.

Entonces la solución es utilizar la posibilidad que les ofrece la nube para crecer y decrecer los recursos aprovisionados en forma automática.

Más adelante en el curso veremos cómo utilizar la elasticidad en la nube.

Clase del día - 10/03/2021

La clase de hoy vamos a ver los requisitos de diseño y los tipos de los sistemas distribuidos.

Requisitos de diseño

El diseño de un sistema consiste en la definición de la **arquitectura** del sistema, la especificación detallada de sus componentes y la especificación del entorno tecnológico que soportará al sistema.

La arquitectura de un sistema puede verse como el "plano" dónde aparecen los componentes de software y hardware del sistema y sus interacciones. A partir de la arquitectura se establecen las especificaciones de construcción del sistema.

En la arquitectura se incluye la forma en que se particiona físicamente el sistema, la organización del sistema en sub-sistemas de diseño, la especificación del entorno tecnológico, los requisitos de operación, administración, seguridad, control de acceso, así como los requisitos de calidad, esto es, las características que el sistema debe cumplir.

A continuación veremos algunos de los requisitos de diseño de los sistemas distribuidos, también conocidos como requisitos arquitectónicos o requerimientos no funcionales.

Calidad de Servicio (QoS)

Los requisitos de **calidad de servicio** (QoS) son aquellos que describen las características de calidad que los servidores debe cumplir, como son los tiempos de respuesta, la tasa de errores permitida, la disponibilidad del servicio, el volumen de peticiones, seguridad, entre otras.

Balance de carga

Los sistemas distribuidos distribuyen procesamiento y datos. Para que un sistema distribuido sea eficiente, es necesario balancear la carga del procesamiento y del acceso a los datos, con la finalidad de evitar que uno o más computadoras se conviertan en un cuello de botella que ralentice el sistema completo.

Por tanto, es importante definir los **requisitos de balance de carga** del sistema, esto es, qué criterios se utilizarán para balancear la carga de procesamiento y de acceso a los datos.

Más adelante en el curso veremos cómo implementar el balance de carga en la nube.

Tolerancia a fallas

Como vimos anteriormente, un sistema distribuido es más tolerante a las fallas que un sistema centralizado, debido a que la falla en un componente de un sistema distribuido no necesariamente implica la falla del sistema completo, como es el caso de un sistema centralizado.

Los requisitos de tolerancia a fallas de un sistema distribuido definen las estrategias que el sistema implementará para soportar la falla en determinados componentes, algunas estrategias empleadas para la tolerancia a fallas son la replicación de datos y la replicación de código.

Seguridad

Posiblemente el requisito de diseño más importante es la seguridad, debido a las amenazas a las que se expone un sistema que se conecta a Internet.

Además de las vulnerabilidades del sistema operativo y del hardware, los sistemas introducen vulnerabilidades propias.

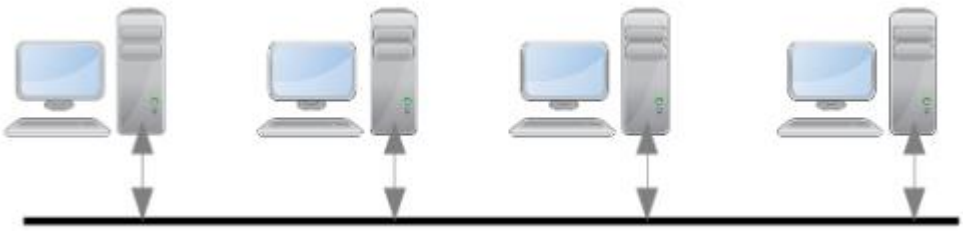
Por tanto, es muy importante definir los **requisitos de seguridad** para el sistema, entre otros: seguridad física del sistema, comunicación encriptada (SSL, TLS, HTTPS), utilización de usuarios no administrativos, configuración detallada de los permisos, programar para la prevención de ataques (p.e. SQL injection), seguridad en el proceso de desarrollo, etc.

Tipos de sistemas distribuidos

En clases anteriores vimos que podemos dividir los sistemas distribuidos en sistemas que distribuyen el procesamiento (cómputo) y sistemas que distribuyen los datos.

Los sistemas distribuidos de cómputo pueden a su vez dividirse en sistemas que ejecutan sobre un **cluster** y sistemas que ejecutan sobre una **malla** (*grid*).

Un cluster es un conjunto de computadoras homogéneas con el mismo sistema operativo conectadas mediante una red local (LAN) de alta velocidad.

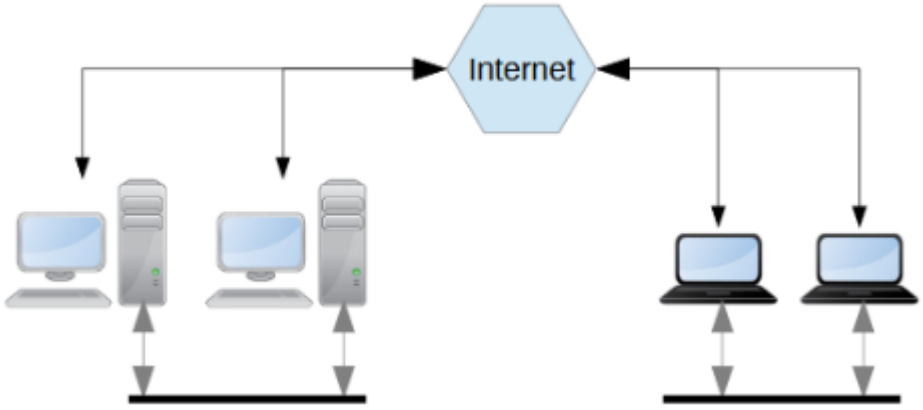


Los clusters se utilizan para el cómputo de alto rendimiento, dónde los programas se distribuyen entre los

diferentes nodos del cluster, con la finalidad de lograr rendimientos superiores.

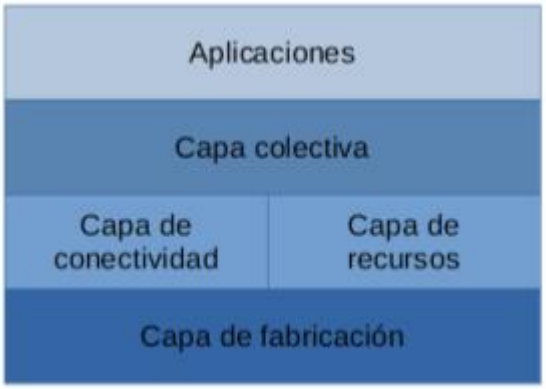
En el TOP500 474 sistemas son clusters, mientras que sólo 40 sistemas son MPP (*Massively Parallel Processing*). Un ejemplo de sistema MPP es la malla (grid).

Una malla es un conjunto de computadoras generalmente heterogéneas (hardware, sistema operativo, redes, etc.) agrupadas en organizaciones virtuales.



Una organización virtual es un conjunto de recursos (servidores, clusters, bases de datos, etc.) y los usuarios que los utilizan.

La arquitectura de una malla se puede dividir en cuatro capas:



La **capa de fabricación** está constituida por interfaces para los recursos locales de una ubicación. En esta capa se implementan funciones que permiten el intercambio de recursos dentro de la organización virtual, tales como consulta del estado del recurso, la capacidad del recurso, así como funciones administrativas para iniciar el recurso, apagar el recurso o bloquear el recurso.

La **capa de conectividad** incluye los protocolos de comunicación que utilizan los recursos para comunicarse, así como autenticación de usuarios y procesos.

La **capa de recursos** permite administrar recursos individuales incluyendo el control de acceso a los recursos (autorización).

La **capa colectiva** permite el acceso a múltiples recursos, incluyendo el descubrimiento de recursos, ubicación de recursos, planificación de tareas en los recursos, protocolos especializados.

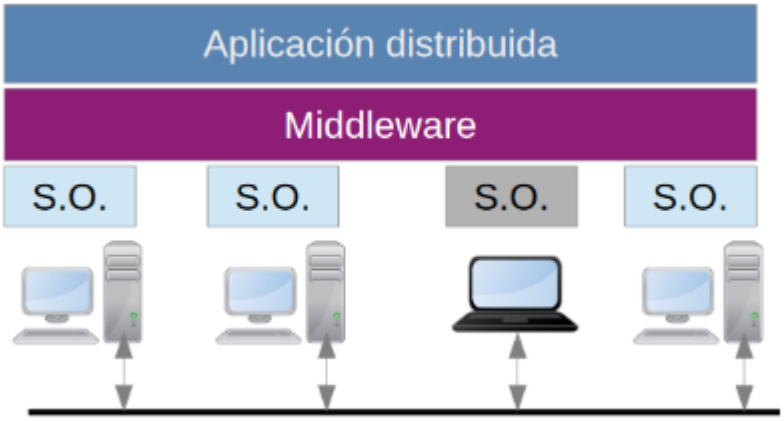
La **capa de aplicaciones** está compuesta por las aplicaciones que ejecutan dentro de la organización virtual.

Middleware

Un middleware (software enmedio) es una capa de software distribuido que actúa como "puente" entre las aplicaciones y el sistema operativo. Ofrece la vista de un sistema único en un ambiente de computadoras y/o redes heterogéneas.

La transparencia (datos, ubicación, migración, re-ubicación, replicación, concurrencia, fallas) de un sistema distribuido se implementa mediante middleware.

El middleware se distribuye entre las diversas máquinas ofreciendo a las aplicaciones una misma interfaz, no obstante las computadoras podrían ejecutar deferentes sistemas operativos.



2. Sincronización y coordinación

Clase del día - 17/03/2021

En la clase de hoy veremos el tema de sincronización en sistemas distribuidos.

¿Cuándo se requiere sincronizar?

El tiempo es una referencia que utilizan los sistemas distribuidos en varias situaciones.

Supongamos una plataforma de comercio electrónico que funciona a nivel global, en cada país se tiene un servidor con una base de datos dónde se registran las compras, incluyendo la fecha y hora en la que se realiza cada compra.

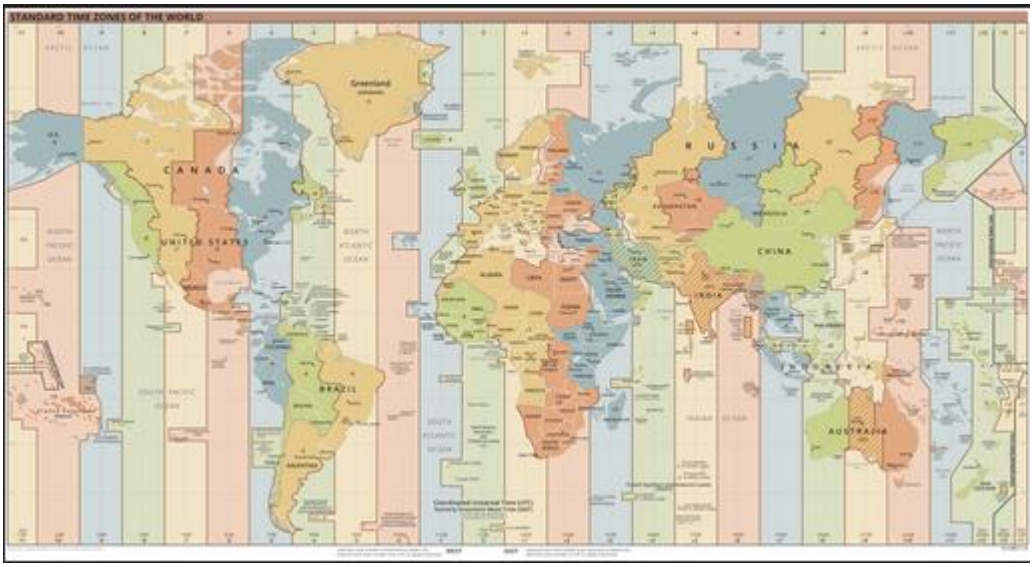
Para consolidar las compras a nivel mundial cada servidor debe enviar los datos a un servidor central. Sin embargo, no es posible ordenar las compras por fecha debido a dos situaciones:

- Cada compra se ha registrado con la fecha y hora local, y
- No es posible garantizar que los relojes de los servidores funcionen a la misma velocidad.

Para ilustrar este problema supongamos que un cliente en México realiza una compra a las 8 PM, y un cliente en España realiza una compra a las 2 AM del día siguiente ¿quién compró primero?

Aparentemente el cliente en México realizó la compra antes que el cliente en España, debido a que la fecha de la compra del cliente en México es un día anterior a la fecha de la compra del

cliente en España. Sin embargo, en realidad el cliente en España realizó la compra una hora antes que el cliente en México, debido a que la diferencia horaria entre México y España es de 7 horas.



Mapa de los husos horarios oficiales vigentes (dominio público)

La solución a este problema es registrar en las bases de datos una **fecha y hora global** en lugar de una fecha y hora local. Además, los servidores deberán sincronizar sus relojes internos a una misma hora.

Por otra parte, si los servidores no requieren consolidar las compras, tampoco será necesario que exista un acuerdo en los tiempos que marcan sus relojes.

El ejemplo anterior ilustra una regla muy importante de los sistemas distribuidos, la cual podemos enunciar de la siguiente manera: *si dos computadoras no están conectadas, entonces no requieren sincronizar sus tiempos*.

Sincronización de relojes

Sincronizar dos o más relojes significa que los servidores se ponen de acuerdo en una misma hora. Notar que un grupo de servidores pueden ponerse de acuerdo en una hora y otro grupo de servidores puede ponerse de acuerdo en otra hora; solo si ambos grupos de servidores se conectan entonces ambos grupos deberán acordar una hora.

Como se dijo anteriormente, **el tiempo es una referencia para establecer un orden** en una secuencia de eventos (como serían las compras en una plataforma de comercio electrónico).

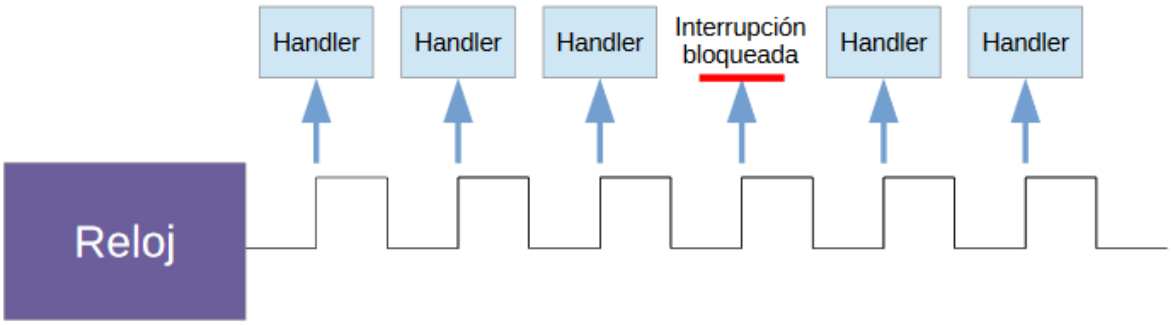
Más adelante veremos que éste orden puede establecerse utilizando relojes físicos (mecanismos que marcan el tiempo real) o bien relojes lógicos (contadores).

Relojes físicos

En los sistemas digitales, un reloj físico es un circuito que genera pulsos con un periodo “constante”.

En una computadora cada pulso de reloj produce una interrupción en el CPU para que se actualice un contador de “ticks”. Dado que el pulso tiene un periodo “constante” el número de ticks es una medida del tiempo transcurrido desde que se encendió la computadora.

El siguiente diagrama muestra un reloj físico el cual genera pulsos regulares. Cuando la señal cambia de 0 volts a 5 volts se produce una interrupción en el CPU, entonces se invoca una rutina llamada manejador de interrupción (*handler*) la cual incrementa el contador de "ticks".



El contador de "ticks" de una computadora no es un reloj preciso, dado que:

- Los relojes físicos se construyen utilizando cristales de cuarzo con la finalidad de tener un periodo de oscilación constante, sin embargo los cambios en la temperatura modifican el periodo del pulso, lo que ocasiona que el reloj se adelante o se atrase.
- Cuando se produce la interrupción al CPU, el sistema podría estar ejecutando una rutina de mayor prioridad, por tanto la rutina que incrementa los "ticks" se bloquea lo que provoca que algunos pulsos de reloj no incrementen la cuenta de "ticks".

Segundos solares

El concepto de tiempo que utilizamos en la práctica se basa en la percepción que tenemos del día. Un día es un período de luz y obscuridad debido a la rotación de la tierra sobre su eje.

Dividimos convencionalmente el día en 24 horas, cada hora en 60 minutos y cada minuto en 60 segundos. Por tanto, la tierra tarda 86,400 segundos en dar una vuelta sobre su eje, en términos de velocidad angular estamos hablando de 360/86400=0.00416 grados/segundo. Así, a la fracción 1/86400 de día le llamamos **segundo solar**.

Sin embargo la velocidad angular de la tierra no es constante, debido a que la rotación de la tierra se está deteniendo muy lentamente.

Segundos atómicos

Una forma más precisa de medir el tiempo es utilizar un reloj atómico de Cesio 133.

En un reloj atómico se aplica microondas con diferentes frecuencias a átomos de Cesio 133, entonces los electrones del átomo de Cesio 133 absorben energía y cambian de estado; posteriormente los átomos regresan a su estado basal emitiendo fotones.

A la frecuencia que produce más cambios de estado en los electrones del átomo de Cesio 133 se le llama *frecuencia natural de resonancia*.

La frecuencia natural de resonancia del Cesio 133 es de 9,192,631,770 ciclos/segundo, es decir, el átomo de Cesio 133 muestra un máximo de absorción de energía cuando se le aplica microondas con una frecuencia de 9,192,631,770 Hertzios.

Entonces se define el **segundo atómico** como el recíproco de la frecuencia natural de resonancia del Cesio 133 (recordar que el periodo de una onda es el recíproco de su frecuencia).

Los relojes atómicos de Cesio 133 son extremadamente precisos, ya que independientemente de las condiciones ambientales (temperatura, presión, etc.), se adelantan o atrasan un segundo cada 300 millones de años.

Los relojes atómicos son tan precisos que se han utilizado para probar los postulados de la teoría general de la relatividad, la cual predice la dilatación del tiempo debidos a la distorsión que causa la gravedad al espacio-tiempo.

Por ejemplo, utilizando un reloj atómico de Cesio 133 se ha demostrado que el tiempo no transcurre a la misma velocidad a diferentes altitudes, ya que al nivel del mar, dónde la gravedad es mayor, el tiempo se dilata (transcurre más lentamente) con respecto al tiempo medido en una montaña elevada dónde la gravedad es menor. A este fenómeno se le conoce como *dilatación gravitacional del tiempo*.

Tiempo atómico internacional TAI

Se define el tiempo atómico internacional (TAI) como el promedio de los segundos atómicos transcurridos desde el 1 de enero de 1958, dicho promedio obtenido de casi 70 relojes de Cesio 133 al rededor del mundo.

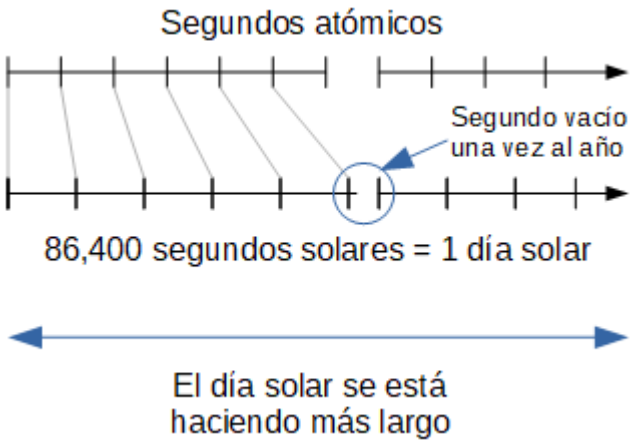
Tiempo universal coordinado UTC

El tiempo universal coordinado UTC (*Coordinated Universal Time*, CUT) es el estándar de tiempo que regula actualmente el tiempo de los relojes a nivel internacional.

El tiempo UTC ha reemplazado el tiempo tiempo medio de Greenwich GMT.

El tiempo GMT toma como referencia la posición del sol a medio día. Tanto el tiempo GMT como el tiempo UTC consideran el día solar compuesto por 86400 segundos solares.

Debido a que nuestro planeta disminuye su velocidad angular lentamente, el segundo solar dura más que el segundo atómico. Para sincronizar los segundos UTC con los segundos TAI, el tiempo UTC se debe “atrasar” con respecto al tiempo TAI, para esto “se salta” un segundo UTC una vez al año; se dice entonces que se introducen **segundos vacíos** en el tiempo UTC.



Los proveedores de nube han adoptado el uso del tiempo UTC para los relojes en las máquinas virtuales, por ejemplo cuando se ejecuta el comando **date** en una máquinas virtual con Ubuntu en Azure, se obtiene la fecha y hora UTC.

Sincronización de relojes físicos

En un sistema centralizado el tiempo se obtiene del reloj central, por tanto todos los procesos se sincronizan mediante un sólo reloj.

En un sistema distribuido cada nodo tiene un reloj que se atrasa o adelanta dependiendo de diversos factores físicos. A la diferencia en los valores de tiempo de un conjunto de computadoras se le llama **distorsión del reloj**.

¿Cómo se puede garantizar un orden temporal en un sistema distribuido?

Existen algoritmos centralizados y distribuidos los cuales se utilizan para sincronizar los relojes en un sistema distribuido.

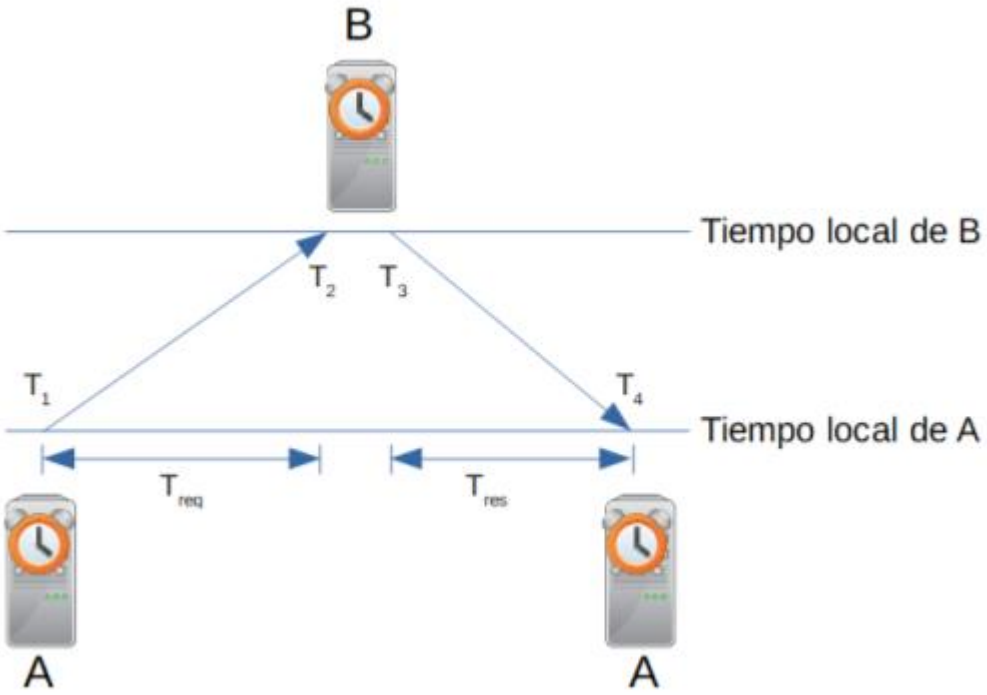
Network Time Protocol NTP

El protocolo de tiempo de red (*Network Time Protocol*, *NTP*) define un procedimiento centralizado para la sincronización de relojes. En este procedimiento los clientes consultan un servidor de tiempo, el cual podría contar con un reloj atómico o estar sicronizado con una computadora que tenga un reloj atómico.

El protocolo NTP estima el tiempo que tarda en llegar al servidor de tiempo la petición del cliente T_{req} y el tiempo que tarda en llegar al cliente la respuesta del servidor T_{res}.

Supongamos que al tiempo local T1 el cliente A envía una petición al servidor B, la petición llega al servidor al tiempo local T2. El servidor B procesa el requerimiento y al tiempo local T3 envía la respuesta al cliente A, la respuesta llega al cliente al tiempo local T4.

Si el servidor B envía T3 y T2 a la computadora A y suponemos $T_{req}=T_{res}$, entonces la computadora A puede estimar T_{res} ya que conoce $T4-T1=T_{req}+T_{res}+(T3-T2)=2T_{res}+(T3-T2)$, por tanto la computadora A podrá modificar su tiempo local a $T3+T_{res}$.



Debido a que los relojes atómicos son recursos muy costosos y con el fin de evitar la saturación del servidor que cuenta con un reloj atómico, se suele implementar el mismo procedimiento sobre una topología de árbol.

Los servidores de los estratos superiores del árbol son más exactos que los servidores de estratos inferiores, de tal manera que el servidor en la raíz, llamado **servidor de estrato 1**, contará con un reloj atómico (llamado reloj de referencia).

Para instalar NTP en Ubuntu, se debe ejecutar los siguientes comandos:

sudo apt-get update

sudo apt-get install ntp

Algoritmo de sincronización de relojes de Berkeley

En el algoritmo NTP el servidor es pasivo, ya que espera recibir las peticiones de los cliente.

El algoritmo de sincronización de relojoes de Berkeley es un procedimiento descentralizado dónde el servidor tiene una función activa, ya que cada cierto tiempo inicia la sincronización de un grupo de computadoras.

El algoritmo de Berkeley se basa en el principio que enunciamos al principio: *si dos computadoras no están conectadas, entonces no requieren sincronizar sus tiempos*.

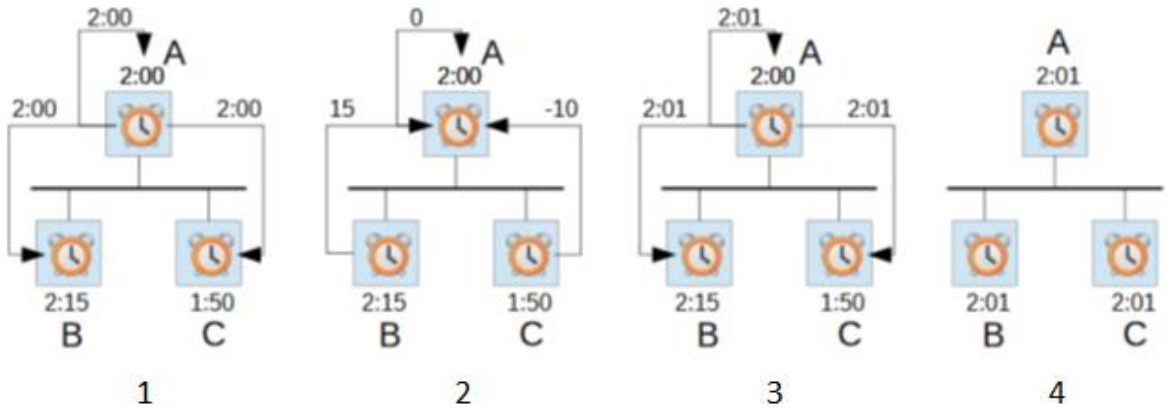
Por tanto, si un grupo de computadoras no se conectan con otras computadoras, es suficiente sincronizar los tiempos de las computadoras en el grupo, aún si el tiempo sincronizado no corresponde al tiempo real (ya que no hay una comunicación con otras computadoras).

En la práctica no hay computadoras aisladas del mundo real, de manera que el algoritmo de Berkeley se puede utilizar para sincronizar las computadoras de una red local, mientras que alguna de las computadoras se podría sincronizar con un servidor de tiempo utilizando NTP.

El algoritmo de Berkeley es el siguiente:

- El nodo A (servidor) le envía a los nodos A, B y C su tiempo.
- Los nodos A, B y C les envían al nodo A las diferencias entre sus tiempos y el tiempo en el nodo A.
- El nodo A calcula el promedio de las diferencias. El nodo A envía a los nodos A, B y C la corrección de tiempo.
- Los nodos A, B y C modifican sus tiempos locales.

A continuación se muestra un ejemplo:



El tiempo es una referencia que se utiliza para establecer un orden en una secuencia de eventos.

Como vimos anteriormente, si dos computadoras no interactúan entonces no es necesario que sus relojes estén sincronizados.

Por otra parte, si dos computadoras interactúan, en general no es importante que coincidan en el tiempo real sino en el orden en que ocurren los eventos.

Happens-before

- Si A y B son eventos del mismo proceso y A ocurre antes que B, entonces A → B
- Si A es el envío de un mensaje y B la recepción del mensaje, entonces A → B

La relación happens-before tiene las siguientes propiedades:

Transitiva: Si A → B y B → C entonces A → C

Anti-simétrica: Si A → B entonces no(B → A)

Irreflexiva: no(A → A) para cada evento A

Relojes lógicos

Se define un **reloj lógico C_i** para un procesador P_i como una función $C_i(A)$ la cual asigna un número al evento A.

Un reloj lógico se implementa como un contador sin una relación directa con un reloj físico, como es el caso de los contadores de "ticks" de las computadoras digitales.

Dados los eventos A y B, si el evento A ocurre antes que el evento B, entonces $C_i(A)<C_i(B)$, por tanto:

Si A → B entonces $C_i(A)<C_i(B)$

Esto significa que si A happens-before B, entonces el evento A ocurre en un tiempo lógico menor al tiempo lógico en que ocurre el evento B.

Algoritmo de sincronización de relojes lógicos de Lamport

Ahora utilizaremos la relación happends-before definida por Lamport para sincronizar relojes lógicos en diferentes procesadores (computadoras).

Supongamos que tenemos los procesadores P_1, P_2 y P_3. Cada procesador tiene un reloj lógico (contador) que se incrementa periódicamente mediante un thread.

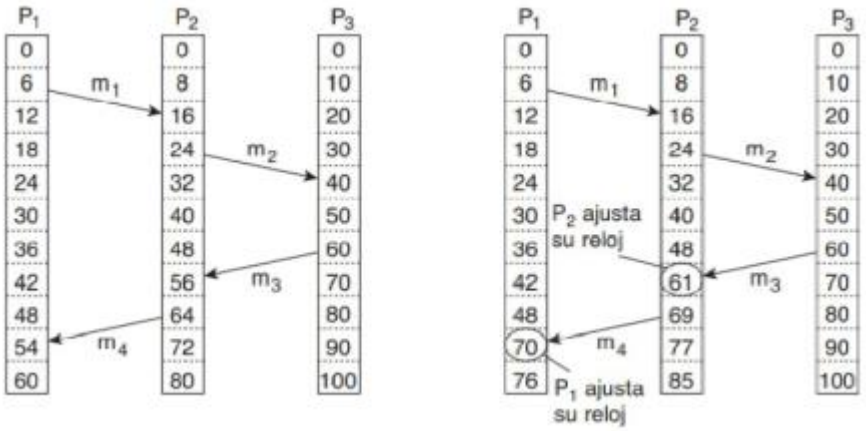
El reloj lógico del procesador P_1 se incrementa en 6, el reloj lógico del procesador P_2 se incrementa en 8 y el reloj lógico del procesador P_3 se incrementa en 10.

¿Cómo sincronizar los relojes lógicos de los tres procesadores de manera que los eventos que ocurren en los procesadores puedan ordenarse?

Lamport resuelve este problema utilizando la relación happends-before para sincronizar los relojes lógicos de diferentes procesadores. Explicaremos el algoritmo de sincronización de relojes lógicos de Lamport con un ejemplo.

Supongamos que al tiempo **6** el procesador P_1 envía el mensaje m_1 al procesador P_2, este procesador recibe el mensaje al tiempo **16**. Al tiempo **24** el procesador P_2 envía el mensaje m_2 al procesador P_3, este procesador recibe el mensaje al tiempo **40**.

Hasta ahora todo es correcto, debido a que el mensaje m_1 es enviado al tiempo 6 y recibido al tiempo 16, y el mensaje m_2 es enviado al tiempo 24 y recibido al tiempo 40, es decir, el tiempo de envío es menor al tiempo de recepción.



Fuente: Sistemas Distribuidos Principios y Paradigmas 2a. Ed. Andrew S. Tanenbaum

Al tiempo **60** el procesador P_3 envía el mensaje m_3 al procesador P_2, este procesador recibe el mensaje al tiempo **56**. Lo anterior contradice la definición de la relación happends-before, ya que la recepción de un mensaje debe ocurrir después del envío del mismo mensaje.

Entonces lo que se hace es ajustar el reloj lógico del procesador P_2, asignando el tiempo lógico del procesador P_3 cuando envía el mensaje m_3 más uno, es decir, se modifica el reloj lógico del procesador P_2 a 61 cuando recibe el mensaje m_3.

Al tiempo 69, el procesador P_2 envía el mensaje m_4 al procesador P_1, este procesador recibe el mensaje al tiempo 54, lo cual contradice la relación happends-before.

Entonces se aplica el mismo procedimiento para el ajuste del reloj lógico del procesador P_1, por tanto el reloj lógico de este procesador se modifica 70 cuando recibe el mensaje m_4.

Clase del día - 18/03/2021

La clase de hoy veremos algunos algoritmos que resuelven el problema de **exclusión mutua**, el cual se presenta cuando dos o más procesadores requieren acceder simultáneamente un recurso compartido (impresora, memoria, CPU, archivo, etc.).

Bloqueos

Existen dos tipos de bloqueos, los **bloqueos exclusivos** y **bloqueos compartidos**. Dependiendo del recurso en particular, se puede utilizar solo bloqueos exclusivos o bien, bloqueos exclusivos y compartidos.

Si un procesador bloquea un recurso de manera exclusiva, ningún procesador puede bloquear el recurso de manera exclusiva o compartida.

Si un procesador bloquea un recurso de manera compartida, otros procesadores pueden bloquear el mismo recurso de manera compartida, es decir, múltiples bloqueos compartidos sobre el mismo recurso pueden co-existir.

Si un recurso tiene uno o más bloqueos compartidos, ningún procesador puede obtener un bloqueo exclusivo sobre el mismo recurso.

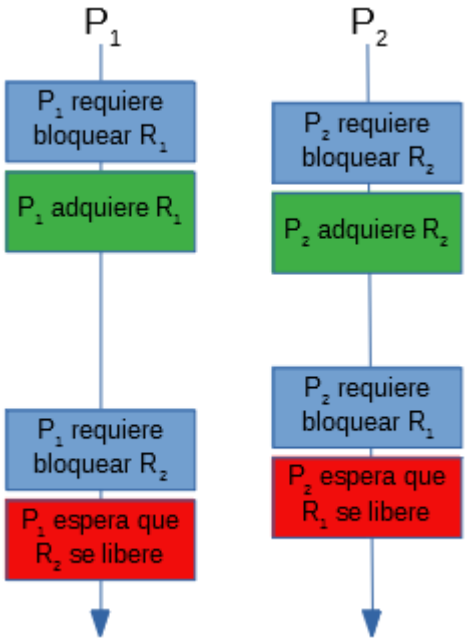
Los bloqueos exclusivos pueden utilizarse para controlar, por ejemplo, el uso de impresoras. Para el acceso a dispositivos de almacenamiento (memorias, discos, etc.), los bloqueos exclusivos se utilizan para proteger operaciones de escritura, mientras que los bloqueos compartidos se utilizan para proteger lecturas.

Por ejemplo, un bloqueo compartido sobre un archivo en el disco permite que varios procesadores puedan leer el archivo, pero no permite que ningún procesador escriba el archivo. Por otra parte, un bloqueo exclusivo sobre el archivo garantiza que solo un procesador pueda escribir y ningún otro procesador pueda leer o escribir el archivo.

En un sistema distribuido las computadoras compiten por adquirir el bloqueo sobre un recurso. En una situación de competencia existe la posibilidad de que una o varias computadoras nunca adquieran el recurso debido a deficiencias en el algoritmo de exclusión. Cuando una computadora no puede adquirir un bloqueo se dice que se presenta **inanición**.

Otro problema que se puede presentar en un algoritmo de exclusión es el **inter-bloqueo** (*dead-lock*). El inter-bloqueo es una situación en la que dos o más procesadores esperan la liberación de un bloqueo, sin que está liberación se pueda realizar.

Para ilustrar una situación de inter-bloqueo, supongamos dos procesadores P_1 y P_2 que acceden a dos recursos R_1 y R_2. Por simplicidad asumimos que los bloqueos sobre los recursos son exclusivos.



Podemos ver que el procesador P_1 requiere bloquear el recurso R_1, debido a que R_1 está desbloqueado el procesador P_1 adquiere el recurso R_1. De la misma manera el procesador P_2 requiere bloquear el recurso R_2, debido a que R_2 está desbloqueado el procesador P_2 adquiere el recurso R_2.

Cuando el procesador P_1 requiere bloquear el recurso R_2, no puede hacerlo ya que el procesador P_2 lo tiene bloqueado, por tanto queda esperando a que R_2 se libere. Así mismo, cuando el

procesador P2 requiere bloquear el recurso R_1, no puede hacerlo ya que el procesador P1 lo tiene bloqueado. Por lo tanto, ambos procesadores quedan bloqueados permanentemente.

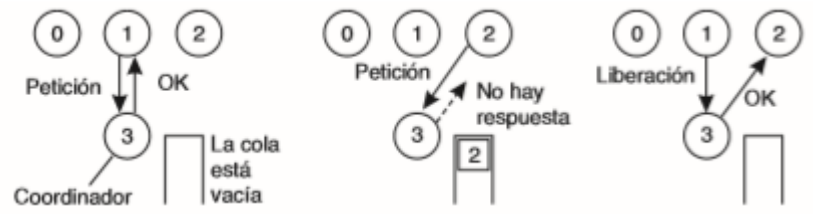
Para evitar que los procesos se bloqueen, los manejadores de bases de datos (p.e. MySQL) detectan el inter-bloqueo como un error, de manera que los programadores puedan controlar la situación.

Algoritmo centralizado para exclusión mutua

Veremos un algoritmo centralizado para implementar la exclusión mutua en un sistema distribuido.

Primeramente necesitamos un nodo que haga las funciones de coordinador, este nodo deberá tener una cola dónde se formaran los nodos que esperan por el recurso.

Explicaremos el algoritmo con un ejemplo. Supongamos que tenemos cuatro nodos. El nodo 3 hace las funciones de coordinador. En un momento dado, el nodo 1 envía una petición al coordinador, debido a que el recurso esta desbloqueado, el coordinador regresa al nodo 1 el mensaje "OK", lo que significa que el nodo 1 ha adquirido el recurso.



Fuente: Sistemas Distribuidos Principios y Paradigmas 2a. Ed. Andrew S. Tanenbaum

Después el nodo 2 envía una petición al coordinador, como el recurso está bloqueado por el nodo 1 el coordinador agrega el nodo 2 a la cola de espera; mientras tanto el nodo 2 queda esperando la respuesta del coordinador.

Cuando el nodo 1 desbloquea el recurso, envía un mensaje de liberación al coordinador, el coordinador extrae el primer nodo de la cola de espera, en este caso el nodo 2, y envía el mensaje "OK" al nodo 2. Entonces el nodo 2 continua con la ejecución de su proceso.

Algoritmo distribuido para exclusión mutua

La desventaja del algoritmo centralizado es que el coordinador puede saturarse si recibe muchas peticiones, por esta razón es mejor implementar un algoritmo descentralizado.

- Cuando un nodo requiere acceso al recurso, envía un mensaje de petición a todos los nodos (incluso a sí mismo) y espera hasta recibir el mensaje "OK" de cada nodo. El mensaje incluye el ID del recurso, el número de nodo y el tiempo lógico.
- Cuando un nodo recibe el mensaje de petición:
    - Si el nodo no esta esperando por el recurso envía un mensaje "OK" al emisor del mensaje de petición.

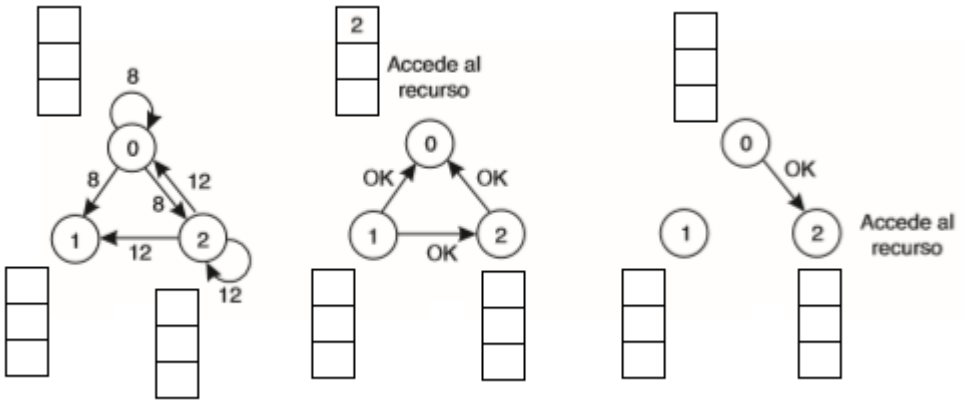
- Si el nodo posee el recurso, coloca en la cola de espera el número de nodo que viene en el mensaje de petición.
- Si el nodo está esperando por el recurso, compara el tiempo lógico T1 del mensaje de petición que recibió con el tiempo lógico T2 del mensaje de petición que previamente envió. Si T1<T2 entonces envía el mensaje "OK" al nodo que envío el mensaje de petición. Si T2<T1, entonces coloca en la cola de espera el número de nodo del mensaje de petición recibido.

- Cuando el nodo libera el recurso:
  - Extrae todos los nodos de la cola de espera y envía el mensaje "OK" a los nodos extraídos de la cola.

El algoritmo anterior requiere que los nodos cuenten con relojes lógicos sincronizados (algoritmo de Lamport) y que cada nodo cuente con **una cola de espera para cada recurso** (recordar que el mensaje de petición incluye el ID del recurso).

Ilustraremos el algoritmo con el siguiente ejemplo. Supongamos que tenemos tres nodos, cada nodo tiene una cola de espera.

El nodo 0 requiere acceso a un recurso, por tanto envía un mensaje de petición a todos los nodos; el mensaje incluye el tiempo lógico 8. Al mismo tiempo el nodo 2 requiere acceso al mismo recurso, entonces el nodo 2 envía un mensaje de petición a todos los nodos, incluyendo el tiempo lógico 12.



Fuente: Sistemas Distribuidos Principios y Paradigmas 2a. Ed. Andrew S. Tanenbaum

El nodo 1 recibe los mensajes de petición de los nodos 0 y 2, debido a que el nodo 1 no está esperando por el recurso, envía sendos mensajes "OK" a los nodos 0 y 2.

El nodo 0 recibe el mensaje de petición que envió el nodo 2. Compara el tiempo lógico T1 del mensaje de petición que recibió con el tiempo lógico T2 del mensaje de petición que previamente envió, en este caso T1=12 y T2=8. Como T2<T1 coloca el nodo 2 en la cola de espera.

El nodo 2 recibe el mensaje de petición que envió el nodo 0. Compara el tiempo lógico T1 del mensaje de petición que recibió con el tiempo lógico T2 del mensaje de petición que previamente envió, en este caso T1=8 y T2=12. Como T1<T2 entonces envía el mensaje "OK" al nodo 0.

Debido a que el nodo 0 recibió "OK" de todos los nodos, adquiere el recurso. Cuando el nodo 0 desbloquea el recurso extrae el primer nodo de la cola de espera, en este caso el nodo 2, y envía el mensaje "OK" al nodo 2, entonces el nodo adquiere el recurso.

Cuando el nodo 2 desbloquea el recurso revisa si tiene algún nodo en la cola de espera, si es así extrae el nodo de la cola y le envía el mensaje "OK". En este caso no hay nodos esperando en la cola, por tanto el nodo 2 continua su proceso sin más.

Algoritmo de token en anillo (token ring)

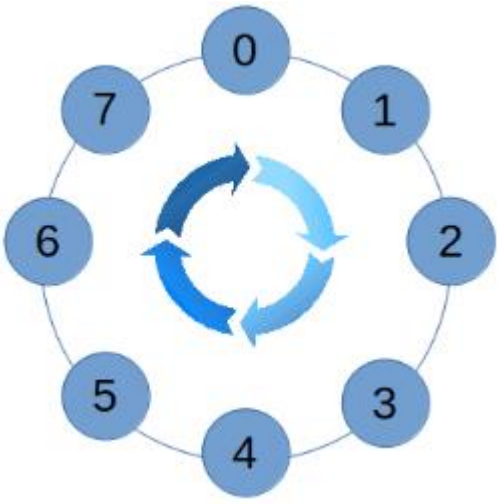
El algoritmo de token en anillo permite implementar la exclusión mutua en un sistema distribuido de manera muy simple, sin embargo tiene la desventaja de que requiere tener una conexión estable entre los nodos, por lo tanto, este algoritmo generalmente se implementa utilizando conexiones físicas.

Supongamos que tenemos ocho nodos conectados en una topología de anillo.

El nodo 0 envía un token (un dato) al nodo 1, el nodo 1 envía el token al nodo 2, y así sucesivamente.

El algoritmo de exclusión mutua utilizando un token en anillo es el siguiente:

- Cuándo un nodo requiere acceso al recurso compartido:
  - Espera recibir el token.
  - El nodo adquiere el bloqueo cuando recibe el token.
  - Cuando el nodo desbloquea el recurso, envía el token al siguiente nodo.
- Si un nodo no requiere acceso al recurso, simplemente pasa el token al siguiente nodo en el anillo.



Actividades individuales a realizar

Considere el algoritmo distribuido para exclusión mutua de Ricart y Agrawala.

1. ¿Este algoritmo puede producir inanición?

2. Suponga que tiene un sistema con múltiples procesadores y múltiples recursos ¿el algoritmo podría producir un inter-bloqueo (dos procesadores que bloquean dos recursos)?

Clase del día - 24/03/2021

En el ámbito de los sistemas distribuidos, la **elección** se refiere al acuerdo al que llegan los nodos para que uno de ellos actúe como coordinador.

El objetivo de un algoritmo de elección es garantizar que todos los nodos lleguen a un acuerdo en el nodo que será el coordinador.

Algoritmo de elección del abusón

Primeramente el algoritmo supone que los nodos están ordenados por número de nodo.

Cuando algún nodo P se da cuenta que el coordinador no responde, se inicia un proceso de elección:

- P envía un mensaje de elección a los nodos con mayor número de nodo.
- Si ningún nodo responde, P se convierte en el coordinador. Entonces P envía un mensaje de coordinador a todos los nodos.
- Si uno de los nodos superiores responde OK, entonces ese nodo inicia una nueva elección y P termina.

El algoritmo se llama del abusón, debido a que el nodo que "gana" la coordinación es el nodo "más fuerte", es decir, el nodo con el mayor número de nodo.

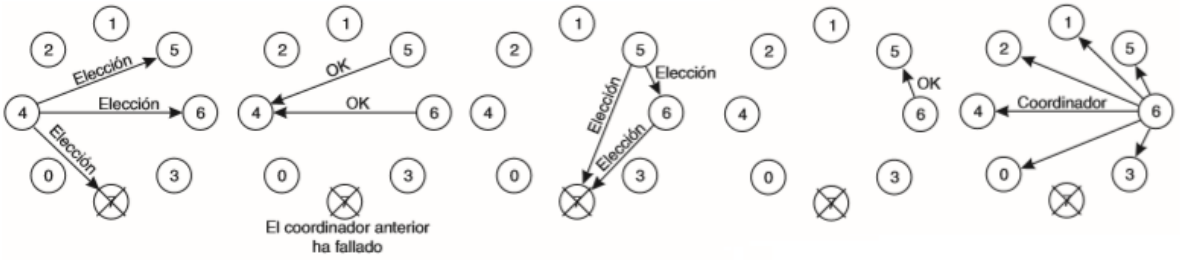
Veamos un ejemplo. Supongamos que tenemos ocho nodos, numerados del 0 al 7.

En un momento dado, el nodo 4 se da cuenta que el coordinador no responde (en este caso, el nodo 7), entonces el nodo 4 inicia un proceso de elección.

El nodo 4 envía un mensaje de elección a los nodos 5, 6 y 7. Entonces los nodos 5 y 6 responden con un mensaje "OK". El nodo 7 no responde.

El nodo 5 envía sendos mensajes de elección a los nodos 6 y 7. El nodo 6 responde con un mensaje "OK" y el nodo 7 no responde.

El nodo 6 envía un mensaje de elección al nodo 7, sin embargo este nodo no responde, por lo tanto el nodo 6 se erige el coordinador, entonces el nodo 6 envía un mensaje de coordinador a todos los nodos, excepto al nodo 7 ya que este nodo no responde.



Fuente: Sistemas Distribuidos Principios y Paradigmas 2a. Ed. Andrew S. Tanenbaum

Algoritmo de elección en anillo

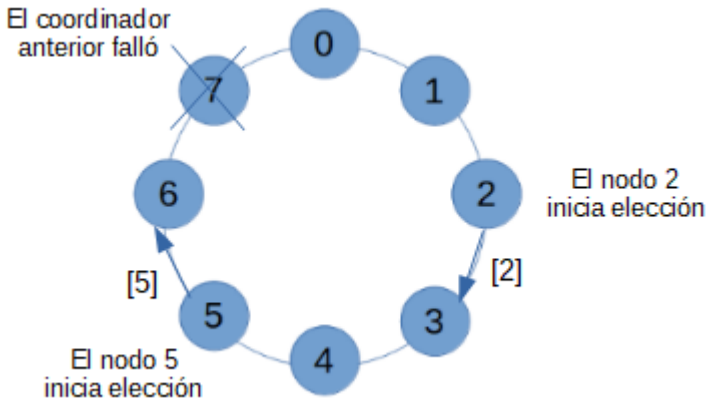
En el algoritmo de anillo se supone que los nodos están conectados en una **topología lógica de anillo** ordenados por número de nodo, de menor a mayor.

Cuando algún nodo P_n se da cuenta que el coordinador no responde, inicia un proceso de elección:

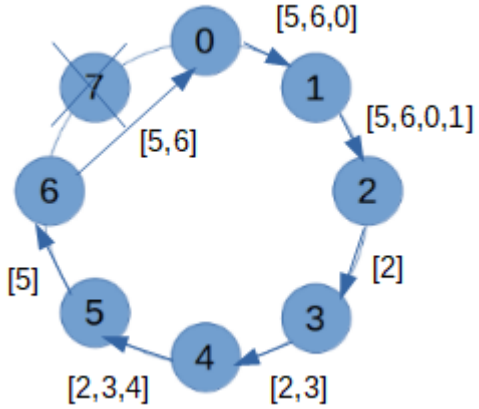
- P_n envía un mensaje de elección al nodo P_n+1 agregando al mensaje su número de nodo n. Si el nodo P_n+1 no responde, entonces el nodo P_n envía el mensaje al nodo P_n+2 y así sucesivamente hasta encontrar un nodo que responda.
- Cuándo un nodo P_m recibe un mensaje de elección:
    - Si el mensaje contiene el número de nodo m y éste es el mayor nodo en el mensaje, el nodo m se hace el coordinador. Entonces el nodo P_m quita su número de nodo de la lista y envía el mensaje de coordinador al siguiente nodo en la lista.

Supongamos que tenemos ocho nodos conectados en una topología de anillo. El nodo 7 es el coordinador actual, pero falló. Los nodos 2 y 5 se comunican con el coordinador, pero éste no responde, por tanto ambos nodos inician un proceso de elección.

El nodo 2 envía un mensaje de elección al nodo 3, incluyendo en el mensaje su número de nodo. El nodo 5 envía un mensaje de elección al nodo 6 incluyendo su número de nodo.

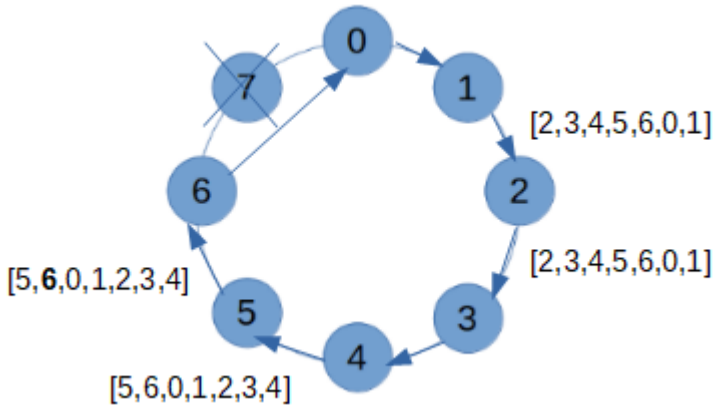


El paso 1 del algoritmo se repite, por tanto cada nodo envía un mensaje de elección al nodo siguiente incluyendo su número de nodo en el mensaje.

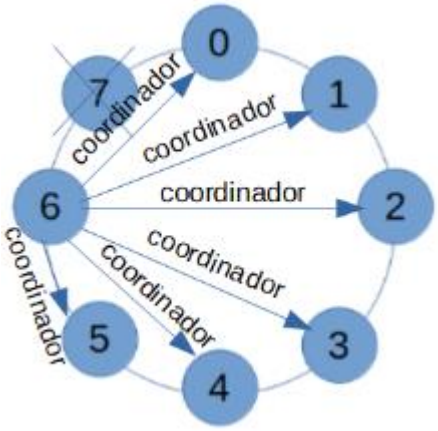


El nodo 2 recibe el mensaje [2,3,4,5,6,0,1], debido a que el nodo 2 no es el mayor nodo en el mensaje, entonces el nodo 2 re-envía el mensaje al nodo 3.

Finalmente, el nodo 5 recibe el mensaje [5,6,0,1,2,3,4], debido a que el nodo 5 no es el mayor nodo en el mensaje, entonces el nodo 5 re-envía el mensaje al nodo 6. Cuando el nodo 6 recibe el mensaje [5,**6**,0,1,2,3,4] encuentra que es el mayor nodo, por tanto se erige como coordinador.



El nodo 6 envía un mensaje de coordinador a todos los nodos.



Actividades individuales a realizar

1. Considere el ejemplo que vimos para el algoritmo del abusón, suponga que tanto el nodo 6 como el nodo 7 no responden, si el nodo 0 y el nodo 1 inician un proceso de elección ¿Qué nodo se erigirá coordinador?

2. Considere el ejemplo que vimos para el algoritmo de elección en anillo, suponga que tanto el nodo 6 como el nodo 7 no responden, si el nodo 0 y el nodo 1 inician un proceso de elección ¿Qué nodo se erigirá coordinador?

Clase del día - 25/03/2021

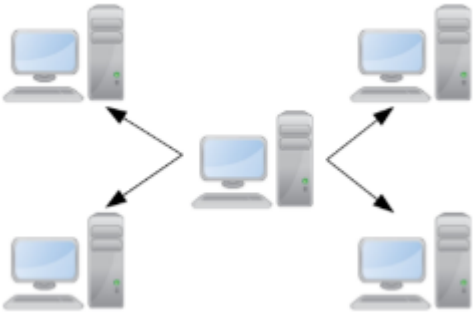
La clase de hoy veremos el tema de Comunicación en un grupo confiable.

Tipos de comunicaciones

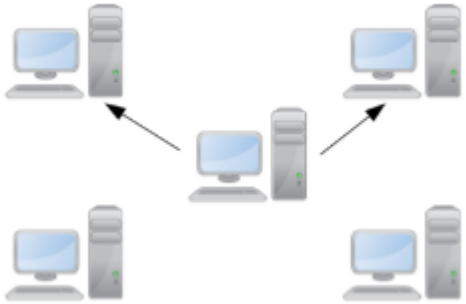
Los tipos de comunicaciones entre computadoras son los siguientes:

Unicast. El unicast es una comunicación punto a punto dónde una computadora envía mensajes a otra computadora.

Broadcast. El broadcast es un tipo de multi-transmisión en la cual una computadora envía mensajes a todas las computadoras en una red.



Multicast. El multicast también es un tipo de multi-transmisión en la cual una computadora envía mensajes a una o más computadoras en una red. El broadcast es un caso particular de multicast, cuando la computadora envía mensajes a todas las computadoras de la red.



Tolerancia a fallas

Un sistema distribuido es tolerante a las fallas si tiene la capacidad de proveer sus servicios incluso ante la presencia de fallas, es decir, el sistema continua operando con normalidad ante las fallas.

Fiabilidad de un sistema

En la medida que un sistema es tolerante a las fallas es un sistema fiable.

La **fiabilidad** de un sistema es un requerimiento no funcional, el cual a su vez se compone de los siguientes sub-requerimientos no funcionales (recordar que en Ingeniería de Software los requerimientos funcionales y no funcionales se pueden dividir en sub-requerimientos):

Disponibilidad. La disponibilidad es la capacidad que tiene un sistema de ser utilizado al momento, es decir, la probabilidad de que el sistema funcione correctamente siempre.

Confiabilidad. La confiabilidad es la capacidad de un sistema de funcionar continuamente sin fallar. La confiabilidad se define en términos de un intervalo de tiempo de funcionamiento continuo, a diferencia de la disponibilidad la cual se refiere al funcionamiento del sistema en un momento dado.

Por ejemplo, si un sistemas se cae un segundo cada día, se dice que tiene una disponibilidad de 1-1/(24x60x60)=99.998%, sin embargo no es un sistema confiable si consideramos un proceso que puede tardar más de un día y el proceso no puede terminar debido a las caídas del sistema.

Seguridad. La seguridad, desde el punto de vista de la tolerancia a fallas, se refiere a la propiedad que tiene el sistema de no causar un evento catastrófico cuándo falla. Por ejemplo, un sistema de conducción autónoma no es seguro si al fallar el automóvil choca y provoca daños a los pasajeros.

Mantenimiento. El mantenimiento se refiere a la capacidad que tiene el sistema de ser reparado cuando falla.

Clasificación de las fallas de un sistema

Las fallas en un sistema se pueden clasificar en cinco categorías:

Falla de congelación. La falla de congelación se presenta cuando el sistema estaba funcionando normalmente y de pronto se detiene.

Falla de omisión. Una falla de omisión se presenta cuando el sistema no recibe los mensajes (*omisión de recepción*) o no envía los mensajes (*omisión de envío*).

Falla de tiempo. Una falla de tiempo se produce cuando el tiempo de respuesta del sistema es mayor al especificado en los requisitos no funcionales.

Falla de respuesta. El sistema presenta una falla de respuesta cuando se produce un valor incorrecto en la respuesta (*falla de valor*) o una respuesta incorrecta debido a una desviación en la ejecución del algoritmo (*falla de transición de estado*).

Falla arbitraria. El sistema presenta una falla arbitraria cuandoproduce respuestas arbitrarias, en cualquier momento y con tiempos de repuesta arbitrarios.

En un sistema distribuido pueden fallar los servidores y/o las comunicaciones.

Las fallas que presenta un canal de comunicación pueden ser fallas por congelación, omisión, tiempo y fallas arbitrarias. Veremos más adelante que las fallas que reciben especial atención son las fallas por congelación y omisión.

Un canal de comunicación confiable es aquel que oculta las fallas en la comunicación.

Socket, dirección IP y puerto

Un socket es un punto final (*endpoint*) de un enlace de dos vías que comunica dos procesos que ejecutan en la red. Un *endpoint* es la combinación de una dirección IP y un puerto.

Una dirección IP versión 4 es un número de 32 bits dividido en cuatro bytes, cada byte puede tener un valor entre 0 y 255. El puerto es un número entre 0 y 65,535

Clases de dirección IP v4

Las direcciones IP versión 4 se dividen en cinco clases o rangos, a saber: Clase A, Clase B, Clase C, Clase D y Clase E. Cada clase se define por un rango de valores que puede tomar el primer byte de la dirección IP, así las clases A, B y C son utilizadas para la comunicación unicast, mientras que la clase D es utilizada exclusivamente para la comunicación multicast. La clase E está reservada para propósitos experimentales.

La siguiente tabla muestra los bytes que identifican a las redes y a los hosts en cada clase (Rango del primer byte), así como la máscara de subred, número de redes y número de hosts por red en cada clase.

Clase	Rango del primer byte	Red(N) Host(H)	Máscara de subred	Número de redes	Hosts por red
A	1-126	N.H.H.H	255.0.0.0	126	16,777,214
B	128-191	N.N.H.H	255.255.0.0	16,382	65,534
C	192-223	N.N.N.H	255.255.255.0	2,097,150	254
D	224-239	Usado para multicast			
E	240-254	Reservado para propósitos experimentales			

Las direcciones 127.X.X.X (*loopback address*) son utilizadas para identificar a la computadora local (localhost).

La dirección 255.255.255.255 es usada para broadcast a todos los hosts en la LAN. Las direcciones 224.0.0.1 y 224.0.0.255 están reservadas.

Protocolos TCP y UDP

Un protocolo define la estructura de los paquetes de datos. Existen dos tipos de sockets:

Socket stream. Socket orientado a conexión:

- Se establece una conexión virtual uno-a-uno mediante *handshaking*.
- Los paquetes de datos son enviados sin interrupciones a través del canal virtual.
- El protocolo TCP (*Transmission Control Protocol*) es el más utilizado para sockets orientados a conexión.

Las características de los sockets conectados son las siguientes:

- Comunicación altamente confiable.
- Un canal dedicado de comunicación punto-a-punto entre dos computadoras.
- Los paquetes son enviados y recibidos en el mismo orden.
- El canal está ocupado aunque no se esté transmitiendo.
- Recuperación tardada de datos perdidos en el camino.
- Cuándo los datos son enviados se espera un acuse de recibo (*acknowledgement*).

- Si los datos no son recibidos correctamente se retransmiten.
- No se utilizan para broadcast ni multicast, ya que los sockets stream establecen solo una conexión entre dos endpoints.
- Se implementan mayormente usando protocolo TCP.

Socket datagrama. Socket desconectado:

- Los datos son enviados en un paquete a la vez.
- No se requiere establecer una conexión.
- El protocolo UDP (*User Datagram Protocol*) es el más utilizado para sockets desconectados.

Las características de los sockets desconectados son las siguientes:

- No requieren un canal dedicado de comunicación.
- La comunicación no es 100% confiable.
- Los paquetes son enviados y recibidos en diferente orden.
- Rápida recuperación de datos perdidos en el camino.
- No hay *acknowledgement* ni re-envío.
- Utilizados para broadcast y multicast.
- Utilizados para la transmisión de audio y video en tiempo real.
- Se implementan mayormente usando el protocolo UDP.

Comunicación unicast confiable

Para establecer la comunicación unicast confiable se utiliza generalmente el protocolo TCP, el cual implementa la retransmisión de mensajes para ocultar las fallas por omisión.

Las fallas por congelación (cuando se produce la desconexión) no son ocultadas por el protocolo TCP, sin embargo el cliente es informado de la falla de manera que pueda re-conectarse (ver el programa Cliente2.java).

Comunicación multicast confiable

Anteriormente hemos hablado de la posibilidad de replicar los datos y los servidores con el propósito de tolerar las fallas en un sistema distribuido. Sin embargo, la replicación implica la multi-transmisión de los mensajes a las réplicas, lo cual requiere contar con una comunicación multicast confiable.

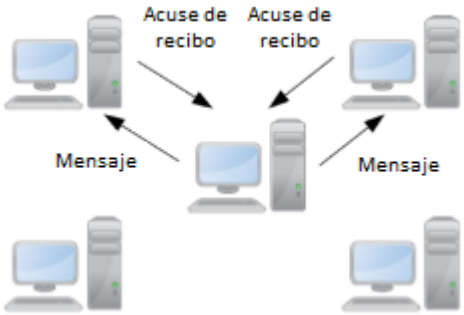
Definimos la comunicación multicast confiable como los mecanismos que garantizan que todos los miembros de un grupo reciben los mensajes transmitidos, sin importar el orden en que reciben los mensajes.

La comunicación punto a punto confiable se implementa con relativa facilidad mediante TCP, sin embargo la comunicación multicast confiable resulta mucho más complicada.

Una primera aproximación para implementar la comunicación multicast confiable es la utilización de múltiples conexiones punto a punto, sin embargo esta solución resulta poco eficiente debido a las características del protocolo TCP.

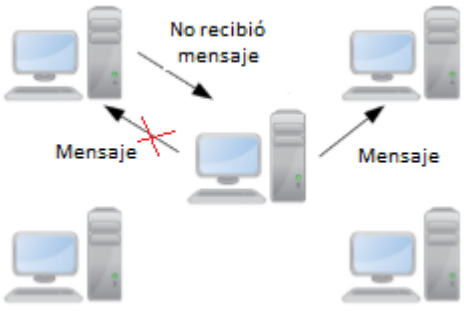
Como vimos anteriormente, la comunicación multicast basada en sockets datagrama no es 100% confiable, debido a que es posible que algunos paquetes se pierdan en el camino, además, los paquetes no son recibidos en el orden en que son enviados.

Una segunda aproximación para la implementación de la comunicación multicast confiable es la utilización de sockets desconectados. En este caso, para garantizar que todos los procesos reciben todos los mensajes, cada proceso receptor deberá enviar un mensaje de acuse de recibo (*acknowledgement*), si el acuse no se recibe en un tiempo determinado, entonces el transmisor deberá retransmitir el mensaje al proceso faltante.



La solución anterior tiene una desventaja, y es que cada mensaje enviado por el transmisor produce una multitud de acuses de recibo, lo cual degrada al transmisor.

Una tercera aproximación es el envío de acuse de recibo "negativo", esto es, si un receptor no recibe un mensaje en un tiempo determinado, entonces envía al transmisor un mensaje indicando que no ha recibido el mensaje. Desde luego, el receptor deberá tener información sobre los mensajes que recibirá, esto se puede lograr agregando al mensaje actual metadatos del siguiente mensaje.



Multicast atómico

El multicast atómico se refiere a la garantía de que un mensaje llegue a todos a destinatarios o a ninguno.

La atomicidad en la comunicación multicast es de utilidad para la implementación de los requerimientos no funcionales que tienen que ver con la consistencia de los datos.

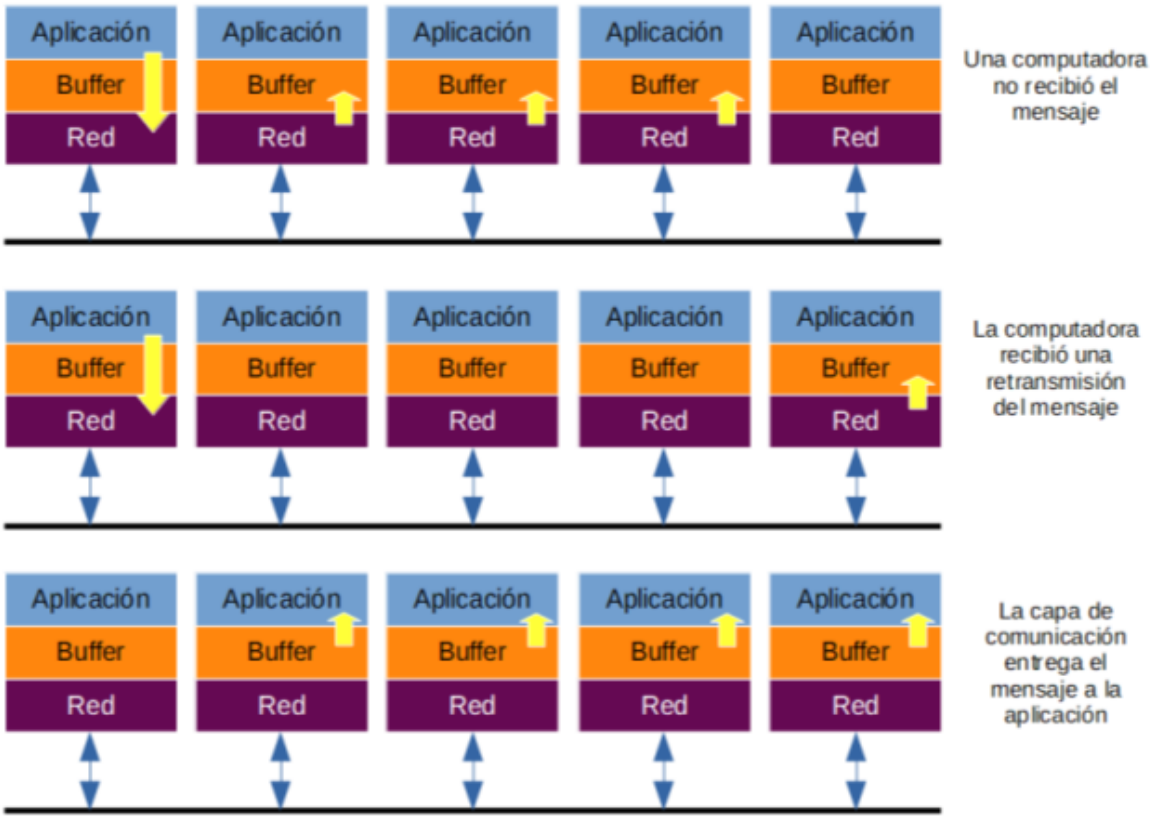
Por ejemplo, si un archivo es replicado en un grupo de computadoras, los cambios que se realizan al archivo deben ser replicados en **todas** las computadoras que forman parte del grupo. Si bien la consistencia del archivo es un requerimiento no funcional del sistema de archivos distribuido, este requerimiento puede ser satisfecho por la capa de comunicaciones si ésta ofrece el multicast atómico.

Existe una variedad de soluciones al multicast atómico, aquí estudiaremos una aproximación basada en la comunicación multicast confiable.

Como vimos anteriormente, la comunicación multicast confiable garantiza que todos los mensajes son recibidos por todos los receptores. Entonces, para contar con el multicast atómico será necesario garantizar que todos los miembros de un grupo reciben los mensajes. Por tanto hay que distinguir entre "recibir el mensaje" y "entregar el mensaje".

Supongamos que una computadora miembro de un grupo envía un mensaje al resto de computadoras en el grupo, sin embargo, por alguna razón, el mensaje no llega a una de las computadoras. Desde luego, el resto de computadoras "recibieron" el mensaje, sin embargo la capa de comunicaciones no puede entregar el mensaje a las aplicaciones antes de confirmar que todos los miembros del grupo en efecto recibieron el mensaje.

Entonces los mensajes entrantes deberán permanecer en un almacén temporal (buffer) en la capa de comunicaciones, y solo en el caso de que todas las computadoras confirmen la recepción del mensaje, entonces y solo entonces el mensaje será entregado a las aplicaciones.



Ahora vamos a ver cómo programar un cliente y un servidor multicast.

En el caso de la comunicación multicast, el servidor es el programa que envía mensajes a los clientes, por esta razón es necesario que los clientes invoquen la función **receive** antes que el servidor ejecute la función **send**.

La comunicación multicast se implementa mediante sockets desconectados, por tanto no se requiere que se establezca una conexión dedicada entre el servidor y el cliente.

Para recibir un mensaje del servidor, los clientes se "unen" a un grupo de manera que el servidor envía mensajes al grupo sin conocer el número de clientes ni sus direcciones IP.

Un grupo multicast se identifica mediante una dirección IP de clase D. Un grupo multicast se crea cuando se une el primer cliente y deja de existir cuando el último cliente abandona el grupo.

El programa ServidorMulticast.java es un ejemplo de un servidor que utiliza sockets UDP para enviar mensajes a un grupo de clientes.

Primeramente vamos a implementar el método envia_mensaje() el cual recibe como parámetros un arreglo de bytes (el mensaje), la dirección IP clase D que identifica el grupo al cual se enviará el mensaje, y el número de puerto.

```
static void envia_mensaje(byte[] buffer,String ip,int puerto) throws IOException
{
```

```
  DatagramSocket socket = new DatagramSocket();
  InetAddress grupo = InetAddress.getByName(ip);
  DatagramPacket paquete = new DatagramPacket(buffer,buffer.length,grupo,puerto);
  socket.send(paquete);
  socket.close();
}
```

Notar que el método envia_mensaje() puede producir excepciones de tipo IOException.

En este caso declaramos una variable de tipo DatagramSocket la cual va a contener una instancia de la clase DatagramSocket.

Obtenemos el grupo correspondiente a la IP, invocando el método estático getByName() de la clase InetAddress.

Para crear un paquete con el mensaje creamos una instancia de la clase DatagramPacket. Entonces enviamos el paquete utilizando el método send() de la clase DatagramSocket.

Finalmente cerramos el socket invocando el método close().

Ahora vamos a enviar la cadena de caracteres "hola", en este caso se envía el mensaje al grupo identificado por la IP 230.0.0.0:

```
envia_mensaje("hola".getBytes(),"230.0.0.0",50000);
```

Vamos a enviar cinco números punto flotante de 64 bits.

Primero "empacaremos" los números utilizando un objeto ByteBuffer. Cinco números punto flotante de 64 bits ocupan 5x8 bytes (64 bits=8 bytes). Entonces vamos a crear un objeto de tipo ByteBuffer con una capacidad de 40 bytes:

```
ByteBuffer b = ByteBuffer.allocate(5*8);
```

Utilizamos el método **putDouble** para agregar cinco números al objeto ByteBuffer:

```
b.putDouble(1.1);
b.putDouble(1.2);
b.putDouble(1.3);
b.putDouble(1.4);
b.putDouble(1.5);
```

Para enviar el paquete de números, convertimos el objeto BytetBuffer a un arreglo de bytes utilizando el método **array()** de la clase ByteBuffer. Entonces enviamos el arreglo de bytes utilizando el método envia_mensaje(), en este caso el mensaje se envía al grupo identificado por la dirección IP 230.0.0.0 a través del puerto 50000:

```
envia_mensaje(b.array(),"230.0.0.0",50000);
```

Vamos a implementar el método recibe_mensaje() al cual pasamos como parámetros un socket de tipo MulticastSocket y la longitud del mensaje a recibir (número de bytes).

```
static byte[] recibe_mensaje(MulticastSocket socket,int longitud_mensaje) throws IOException
{
  byte[] buffer = new byte[longitud_mensaje];
  DatagramPacket paquete = new DatagramPacket(buffer,buffer.length);
  socket.receive(paquete);
  return paquete.getData();
}
```

Notar que el método recibe_mensaje() puede producir una excepción de tipo IOException.

Creamos un paquete vacío como una instancia de la clase DatagramPacket; pasamos como parámetros un arreglo de bytes vacío y el tamaño del arreglo.

Para recibir el paquete invocamos el método recive() de la clase MulticastSocket. El método recibe_mensaje() regresa el mensaje recibido.

Ahora vamos a recibir mensajes utilizando el método recibe_mensaje().

Para obtener el grupo invocamos el método getByName() de la clase InetAddress, en este caso se obtiene el grupo identificado por la IP 230.0.0.0:

```
InetAddress grupo = InetAddress.getByName("230.0.0.0");
```

Luego obtenemos un socket asociado al puerto 50000, creando una instancia de la clase MulticastSocket:

```
MulticastSocket socket = new MulticastSocket(50000);
```

Para que el cliente pueda recibir los mensajes enviados al grupo 230.0.0.0 unimos el socket al grupo utilizando el método joinGroup() de la clase MulticastSocket:

Entonces el cliente puede recibir los mensajes enviados al grupo por el servidor.

Primeramente vamos a recibir una cadena de caracteres:

```
byte[] a = recibe_mensaje(socket,4);
System.out.println(new String(a,"UTF-8"));
```

Ahora vamos a recibir cinco números punto flotante de 64 bits empacados como arreglo de bytes:

```
byte[] buffer = recibe_mensaje(socket,5*8);
ByteBuffer b = ByteBuffer.wrap(buffer);
```

```
for (int i = 0; i < 5; i++)
  System.out.println(b.getDouble());
```

Finalmente, invocamos el método leaveGroup() para que el socket abandone el grupo y cerramos el socket:

```
socket.leaveGroup(grupo);

socket.close();
```